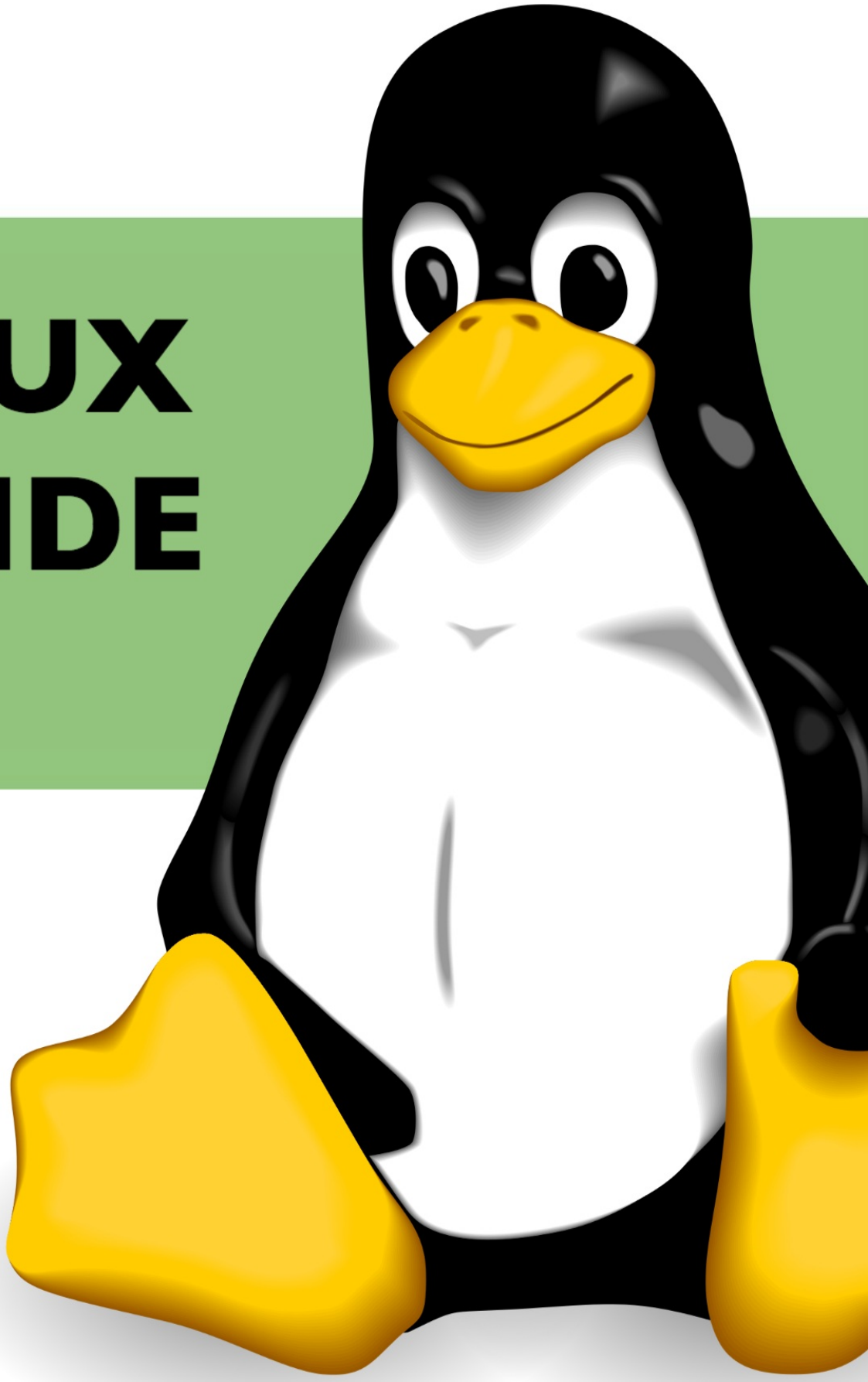
Published
with GitBook



LINUX INSIDE

By OxAX

Table of Contents

Linux

Linux

- Linux Linux

/ : issue - linux-insides issue - linux-insides-zh issue

issues PRs `linux-insides-zh`

-
-
-
-

TRANSLATION_STATUS.md

CONTRIBUTING.md TRANSLATION_NOTES.md issue

@mudongliang

@xinqiu

CONTRIBUTORS.md

@0xAX

Linux

- - ;
- - EDDIST ...
- -
- 64 - 64
- -
- - Linux

- -

Linux x86_64 Linux

- C
- AT&T

Linux 3.18

Linux CPU CPU

80386 CPUs CPU

```
IP          0xfff0
CS selector 0xf000
CS base     0xffff0000
```

8086 Intel 64 CPU x868086 200 2^20 1MB 1 KB 16 64KB 16

```
PhysicalAddress = Segment * 16 + Offset
```

`CS:IP` `0x2000:0x0010` ,

```
>>> hex((0x2000 << 4) + 0x0010)
'0x20010'
```

162 `0xffff:0xffff`

```
>>> hex((0xffff << 4) + 0xffff)
'0x10ffef'
```

1MB 65519 CPU 1MB `0x10ffef` A20 `0x00ffef`

`CS` `CS` `IP`

```
0xffff0000:0xfff0
```

EIP

```
>>> 0xffff0000 + 0xfff0
'0xfffffff0'
```

`0xfffffff0` 4GB - 16 (Reset vector) CPU jump BIOS coreboot

```
    .section ".reset", "ax", %progbits
    .code16
.globl    _start
_start:
    .byte  0xe9
    .int   _start16bit - ( . + 2 )
    ...
```

opcode - 0xe9 `_start16bit - ( . + 2)` `reset` `16` `0xfffffff0` (`src/cpu/x86/16bit/r`
CPU

```
SECTIONS {
    /* Trigger an error if I have an unuseable start address */
     _bogus = ASSERT(_start16bit >= 0xffff0000, "_start16bit too low. Please report.");
    _ROMTOP = 0xfffffff0;
    . = _ROMTOP;
    .reset . : {
        *(.reset);
        . = 15;
        BYTE(0x00);
    }
}
```

BIOS BIOS , BIOS BIOS
2

```
;
; Note: this example is written in Intel Assembly syntax
;
[BITS 16]
[ORG  0x7c00]

boot:
    mov al, '!'
    mov ah, 0x0e
    mov bh, 0x00
    mov bl, 0x07

    int 0x10
    jmp $

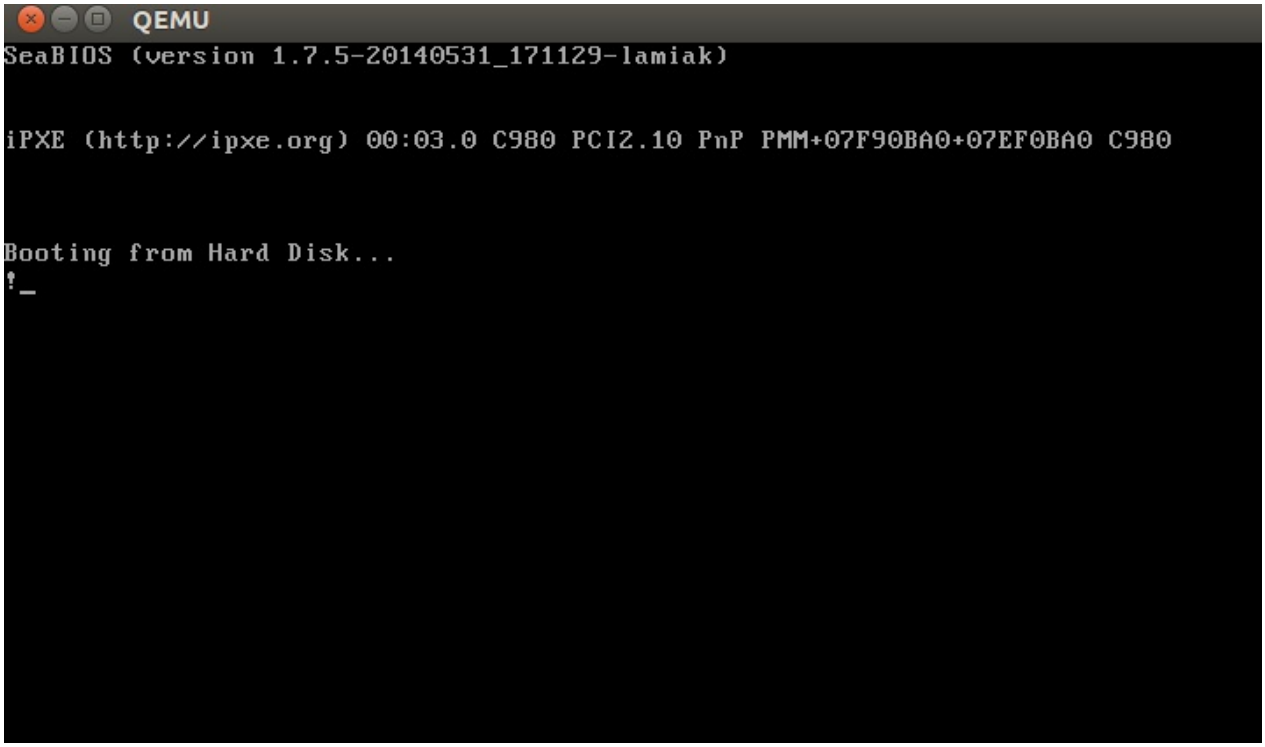
times 510-($-$$) db 0

db 0x55
db 0xaa
```

```
nasm -f bin boot.nasm && qemu-system-x86_64 boot
```

QEMU `boot` (`0x7c00` , Magic Bytes)QEMU(MBR)

:



160x7c00 0x10 `!` 0510Magic Bytes `0xaa` `0x55`

`objdump`

```
nasm -f bin boot.nasm
objdump -D -b binary -mi386 -Maddr16,data16,intel boot
```

BIOS

NOTE: CPU

```
PhysicalAddress = Segment * 16 + Offset
```

CPU 1616 `0xffff`

```
>>> hex((0xffff * 16) + 0xffff)
'0x10ffef'
```

8086 `0x0ffef`, 8086 cpu 20 `2^20 = 1MB` 1MB CPU

1MB

```
0x00000000 - 0x000003FF - Real Mode Interrupt Vector Table
0x00000400 - 0x000004FF - BIOS Data Area
0x00000500 - 0x00007BFF - Unused
0x00007C00 - 0x00007DFF - Our Bootloader
0x00007E00 - 0x0009FFFF - Unused
0x000A0000 - 0x000BFFFF - Video RAM (VRAM) Memory
0x000B0000 - 0x000B7777 - Monochrome Video Memory
0x000B8000 - 0x000BFFFF - Color Video Memory
0x000C0000 - 0x000C7FFF - Video ROM BIOS
0x000C8000 - 0x000EFFFF - BIOS Shadow Area
0x000F0000 - 0x000FFFFF - System BIOS
```

CPU `0xFFFFFFF0` `0xFFFFF` (1MB) CPU coreboot :

```
0xFFFE_0000 - 0xFFFF_FFFF: 128 kilobyte ROM mapped into address space
```

`0xFFFFFFF0` ROM CPU ROM RAM

Linux GRUB 2 syslinuxLinux Boot protocol GRUB 2

BIOS boot.img GRUB 2's core image Core image diskboot.img core image core image core image GRUB 2 grub_main

`grub_main` root grub GRUB normal `grub_normal_execute` (from `grub-core/normal/main.c`) `grub_menu_execute_entry` GRUB `boot`

kernel boot protocol kernel setup header kernel setup code `0x01f1` kernel setup header arch/x86/boot/header.S

```
    .globl hdr
hdr:
    setup_sects: .byte 0
    root_flags:  .word ROOT_RDONLY
    syssize:     .long 0
    ram_size:    .word 0
    vid_mode:    .word SVGA_MODE
    root_dev:    .word 0
    boot_flag:   .word 0xAA55
```

bootloader Linux boot protocol `write` type_of_loader kernel setup header boot protocol

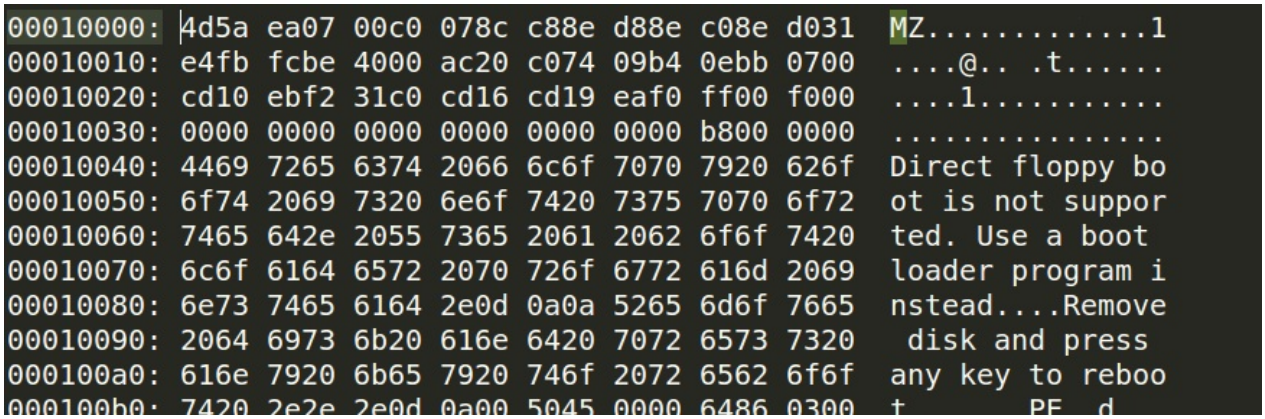
kernel boot protocol

```
         | Protected-mode kernel  |
100000   +------------------------+
         | I/O memory hole        |
0A0000   +------------------------+
         | Reserved for BIOS      | Leave as much as possible unused
         ~                        ~
         | Command line           | (Can also be below the X+10000 mark)
X+10000  +------------------------+
         | Stack/heap             | For use by the kernel real-mode code.
X+08000  +------------------------+
         | Kernel setup           | The kernel real-mode code.
         | Kernel boot sector     | The kernel legacy boot sector.
       X +------------------------+
         | Boot loader            | <- Boot sector entry point 0x7C00
001000   +------------------------+
         | Reserved for MBR/BIOS  |
000800   +------------------------+
         | Typically used by MBR  |
000600   +------------------------+
         | BIOS use only          |
000000   +------------------------+
```

bootloader kernelkernel

```
0x1000 + X + sizeof(KernelBootSector) + 1
 X + sizeof(KernelBootSector) + 1  X
```

`X` kernel bootsector `X` `0x10000` memory dump

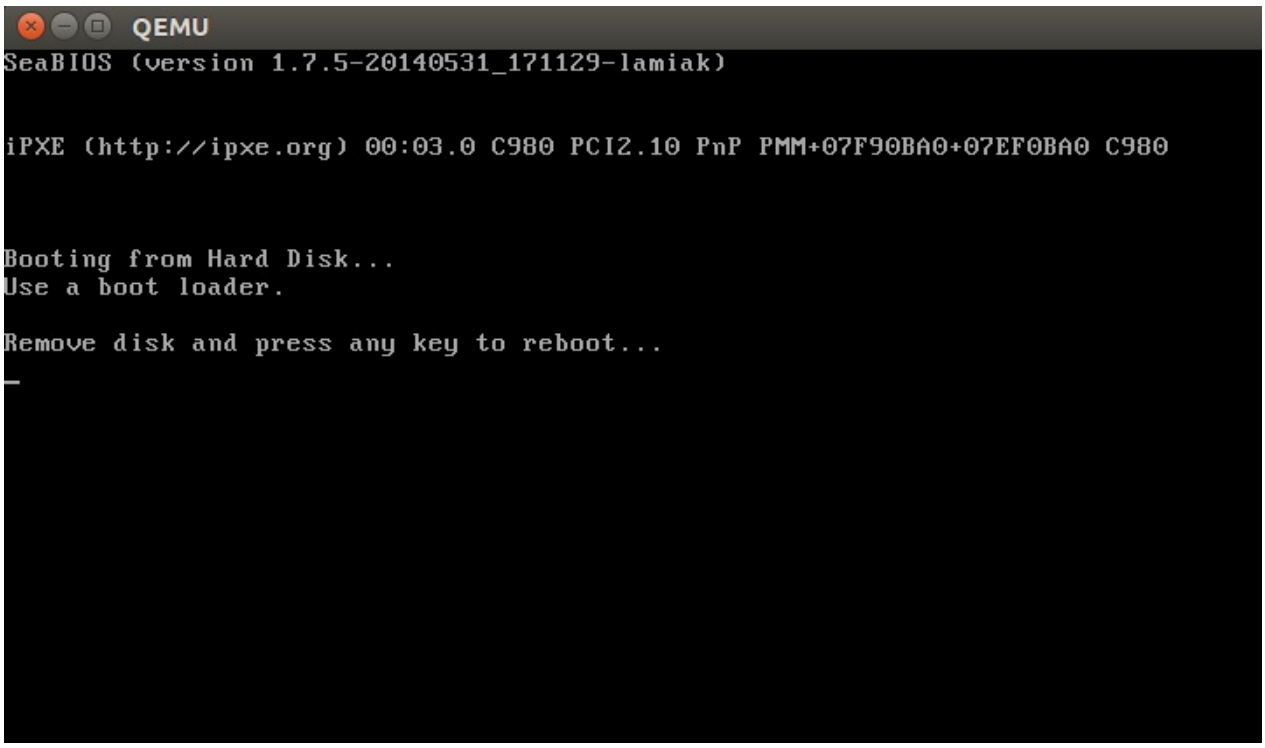


Linux kernel kernel setup code

arch/x86/boot/header.S _start `_start`

`_start` kernel bootloader bootloader Linux Linux bootloader Linux Linux

```
qemu-system-x86_64 vmlinuz-3.18-generic
```



bootloader , `header.S` [MZ] MZ , PE PE

```
#ifdef CONFIG_EFI_STUB
# "MZ", MS-DOS header
.byte 0x4d
.byte 0x5a
#endif
...
...
...
pe_header:
    .ascii "PE"
```

```
    .word 0
```

UEFI

bootloader `_start`

```
// header.S line 292
.globl _start
_start:
```

bootloader (grub2 and others) _start `MZ` `0x200` bootloader `.bstext`

```
//
// arch/x86/boot/setup.ld
//
. = 0;                    // current position
.bstext : { *(.bstext) }  // put .bstext section to position 0
.bsdata : { *(.bsdata) }
```

```
    .globl _start
_start:
    .byte 0xeb
    .byte start_of_setup-1f
1:
    //
    // rest of the header
    //
```

`_start` `jmp` `jmp` opcode `0xeb` `start_of_setup - 1f` `Nf` `N` `_start` `1` setup header `1` `start_of_setup` `.entrytext`

GRUB2 `_start` Linux `_start` `0x200` GRUB2

```
  state.gs = state.fs = state.es = state.ds = state.ss = segment;
  state.cs = segment + 0x20;
```

`0x10000` `cs = 0x1020` `cs << 4 + 0 = 0x10200` `0x10000` `0x200`

```
fs = es = ds = ss = 0x1000
cs = 0x1020
```

`start_of_setup`

-
-
- bss
- main.c

`ds` `es` `cld`

```
    movw    %ds, %ax
    movw    %ax, %es
    cld
```

`_start` grub2 `cs` `0x1020` `cs`

```
    pushw   %ds
    pushw   $6f
    lretw
```

`cs` `ds` 6 `lretw` `6` `ip` instruction pointer `ds` `cs` `ds` `cs`

setup C `ds` `es` step `ss`

```
    movw    %ss, %dx
    cmpw    %ax, %dx
    movw    %sp, %dx
    je    2f
```

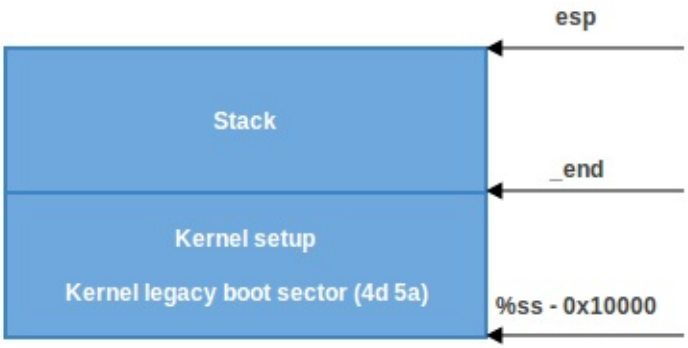
`ss`

- `ss` 0x10000 (`cs`
- `ss` 0x10000 `CAN_USE_HEAP`
- `ss` 0x10000 `CAN_USE_HEAP`

- `ss` 0x10000 `2` :

```
2:      andw    $~3, %dx
    jnz    3f
    movw    $0xfffc, %dx
3:  movw    %ax, %ss
    movzwl %dx, %esp
    sti
```

`dx` `sp` 4000 `dx` `0xfffc` 64KB40 `ax` 0x10000 `ss` `dx` `sp`



- `ss` != `ds` setup code _end `dx` `loadflags` `CAN_USE_HEAP` kernel boot protocol loadflags `Bit 7` `CAN_USE_HEAP`

```
Field name:    loadflags

  This field is a bitmask.

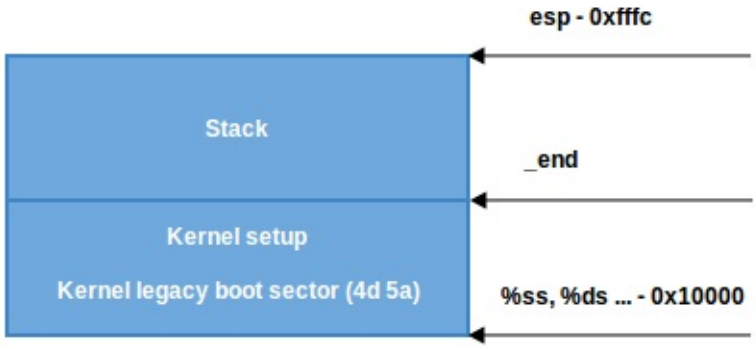
  Bit 7 (write): CAN_USE_HEAP
    Set this bit to 1 to indicate that the value entered in the
    heap_end_ptr is valid.  If this field is clear, some setup code
```

```
    functionality will be disabled.
```

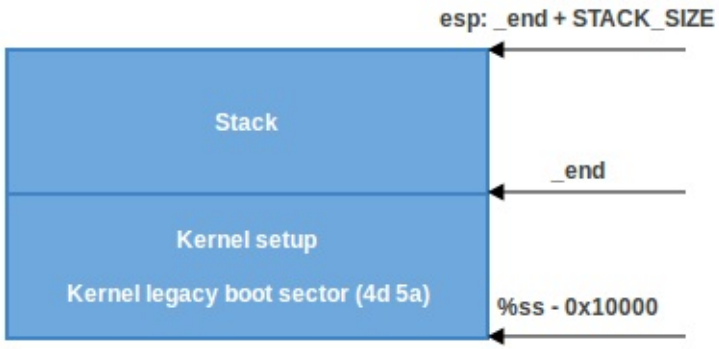
`loadflags`

```
#define LOADED_HIGH        (1<<0)
#define QUIET_FLAG         (1<<5)
#define KEEP_SEGMENTS      (1<<6)
#define CAN_USE_HEAP       (1<<7)
```

`CAN_USE_HEAP` `heap_end_ptr` `dx` `STACK_SIZE` 512 bytesCF flag `2` 1



- `CAN_USE_HEAP` `dx` `STACK_SIZE` `2`



BSS

C 21 BSS 2 `magic` `magic` setup_sig `setup_bad`

```
cmpl    $0x5a5aaa55, setup_sig
jne     setup_bad
```

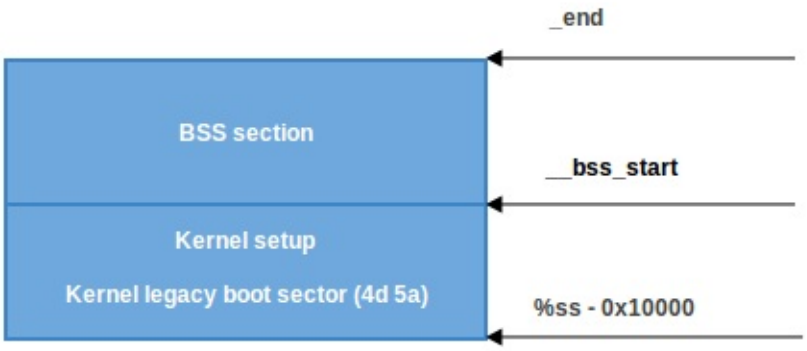
`magic` `BSS` C

BSS Linux

```
    movw    $__bss_start, %di
    movw    $_end+3, %cx
    xorl    %eax, %eax
    subw    %di, %cx
```

```
    shrw    $2, %cx
    rep; stosl
```

__bss_start `di` `_end + 3` 4 `cx` `xor` `ax` BSS `cx` - `di` `cx` `cx` 4 `rep; stosl` `ax` 0 BSS BSS :



main

BSS `main()` C

```
    call main
```

`main()` arch/x86/boot/main.c

Linux C `main()` `memset` , `memcpy` , `earlyprintk`

twitter

PR **linux-insides-zh**

- Intel 80386 programmer's reference manual 1986
- Minimal Boot Loader for Intel® Architecture
- 8086
- 80386
- Reset vector
- Real mode
- Linux kernel boot protocol
- CoreBoot developer manual
- Ralf Brown's Interrupt List
- Power supply
- Power good signal

`main` `main` C `main` arch/x86/boot/main.c

-
-
-
- cpu
-

Intel 64CPUCPU

1982Intel CPUIntel 64Intel CPU

1M640K

20324GB

2

-
-

2

-
-

2

```
PhysicalAddress = Segment * 16 + Offset
```

64K (GDT)

 `GDTR` Linux `GDTR`

```
lgdt gdt
```

`lgdt` `GDTR` `GDTR` 482:

- (16
- (32)

 64

```
31         24        19      16             7           0
-------------------------------------------------------------
|            | |B| |A|       | |   | |0|E|W|A|           |
```

```
| BASE 31:24  |G|/|L|V| LIMIT |P|DPL|S|  TYPE | BASE 23:16 | 4
|             | |D| |L| 19:16 | |   | |1|C|R|A|            |
-------------------------------------------------------------
|                             |                             |
|        BASE 15:0            |       LIMIT 15:0            | 0
|                             |                             |
-------------------------------------------------------------
```

LIMIT 15:0 015015 LIMITE 19:16 1619161920

1. Limit[20] 0-1516-19 `G` 20
   - `G` = 0, Limit = 0 1 byte
   - `G` = 1, Limit = 0, 4K bytes
   - `G` = 0Limit = 0xfffff1M bytes
   - `G` = 1Limit = 0xfffff4G bytes

   - G = 0, 1 byte (Limit11 byte)1M bytes
   - G = 1, 4K bytes (Limit14K bytes)4G bytes;
   - base_seg_length * (LIMIT + 1)
2. Base[32-bits] 0-15 32-3956-63Base
3. Type/Attribute (40-47 bits)
   - `S` 44 `S` = 0 `S` = 1

`S` = 14343 = 0

```
|           Type Field        | Descriptor Type | Description
|-----------------------------|-----------------|-----------------
| Decimal                     |                 |
|             0    E    W   A |                 |
| 0           0    0    0   0 | Data            | Read-Only
| 1           0    0    0   1 | Data            | Read-Only, accessed
| 2           0    0    1   0 | Data            | Read/Write
| 3           0    0    1   1 | Data            | Read/Write, accessed
| 4           0    1    0   0 | Data            | Read-Only, expand-down
| 5           0    1    0   1 | Data            | Read-Only, expand-down, accessed
| 6           0    1    1   0 | Data            | Read/Write, expand-down
| 7           0    1    1   1 | Data            | Read/Write, expand-down, accessed
|                  C    R   A |                 |
| 8           1    0    0   0 | Code            | Execute-Only
| 9           1    0    0   1 | Code            | Execute-Only, accessed
| 10          1    0    1   0 | Code            | Execute/Read
| 11          1    0    1   1 | Code            | Execute/Read, accessed
| 12          1    1    0   0 | Code            | Execute-Only, conforming
| 14          1    1    0   1 | Code            | Execute-Only, conforming, accessed
| 13          1    1    1   0 | Code            | Execute/Read, conforming
| 15          1    1    1   1 | Code            | Execute/Read, conforming, accessed
```

43 `0` `1` 424140(*E W A*424140(*C R A*

- `E` = 0
- `W` = 1
- ACPU
- `C` = 1 `C` = 0
- `R` = 1

1. DPL2-bits, bit 45 460-3
2. P (bit 47) - `P` = 0

3. AVL (bit 52) - Linux
4. L (bit 53) - `L` = 164
5. D/B flag(bit 54) - 321160
    - D323280161680x660x67
    - SSBBigPUSHPOPCALL32ESP016SPB
    - B0xFFFFFFFF4GB0xFFFF64KB

    16

```
----------------------------
|       Index    | TI | RPL |
----------------------------
```

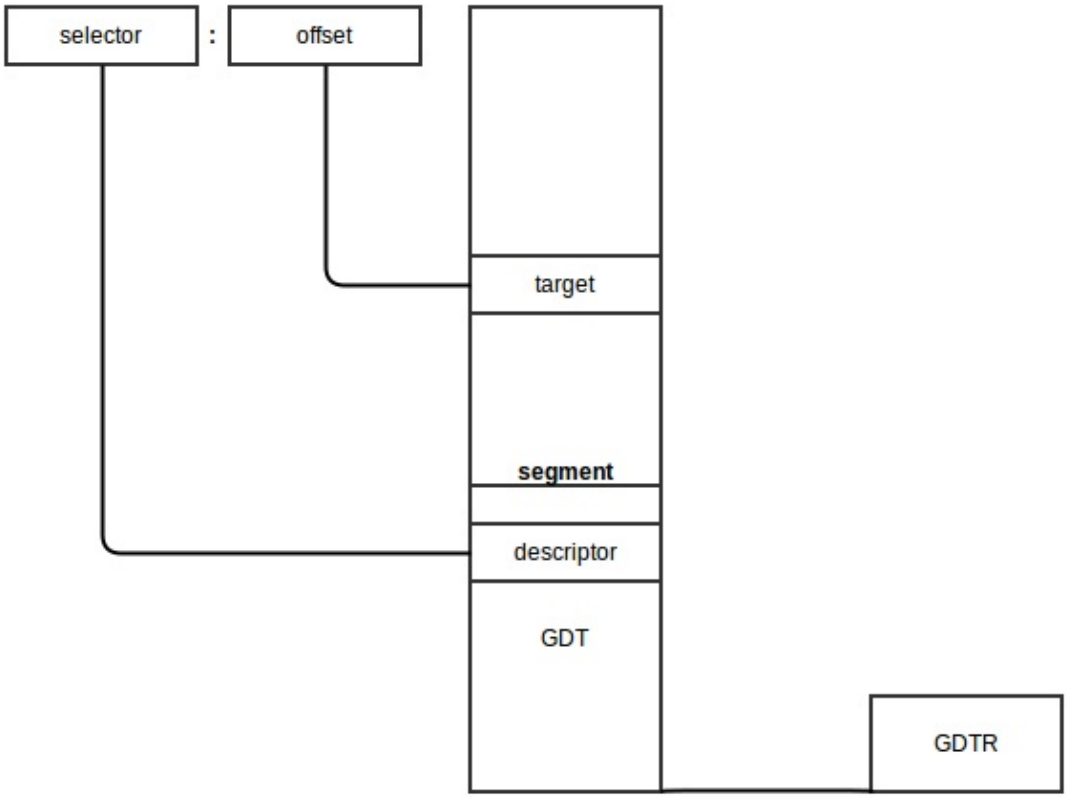
- **Index** GDT
- **TI** GDTLDT
- **RPL**

2

- -
- -

cpu

-
- CPU GDT
- limit + 1 +



-
- `lgdt` GDT `GDTR`
- CR0PE1CPU
-

Linux

C arch/x86/boot/main.c...

"zeropage"

`main` `copy_boot_params(void)`

`boot_params` arch/x86/include/uapi/asm/bootparam.h `boot_params`

`boot_params` `struct setup_header hdr` linux boot protocolboot loader `copy_boot_params`

1. header.S `hdr` `boot_params` `struct setup_header hdr`
2.

`hdr` `memcpy` C copy.S

```
GLOBAL(memcpy)
    pushw    %si          ;push si to stack
    pushw    %di          ;push di to stack
```

```
    movw    %ax, %di     ;move &boot_param.hdr to di
    movw    %dx, %si     ;move &hdr to si
    pushw    %cx          ;push cx to stack ( sizeof(hdr) )
    shrw    $2, %cx
    rep; movsl           ;copy based on 4 bytes
    popw    %cx          ;pop cx
    andw    $3, %cx      ;cx = cx % 4
    rep; movsb           ;copy based on one byte
    popw    %di
    popw    %si
    retl
ENDPROC(memcpy)
```

`copy.S` `GLOBAL` `ENDPROC`

arch/x86/include/asm/linkage.h `GLOBAL`

```
#define GLOBAL(name)    \
    .globl name;    \
    name:
```

include/linux/linkage.h `ENDPROC` `END(name)`

```
#define ENDPROC(name) \
    .type name, @function ASM_NL \
    END(name)
```

`memcpy` `si` `di` `si` `di` `memcpy` copy.s `fastcall` `ax` , `dx` , `cx` `memcpy`

```
memcpy(&boot_params.hdr, &hdr, sizeof hdr);
```

- `ax` `boot_param.hdr`
- `dx` `hdr`
- `cx` `hdr`

`memcpy` `si` `di` `boot_param.hdr` `di` `hdr` `si` `hdr` 4 `si` `di` 4 `hdr` `cx` `cx` 4 `cx` `si` `di` `si` `di`

`hdr` `boot_params.hdr` arch/x86/boot/early_serial_console.c `console_init`

`earlyprintk` `earlyprintk`

- serial,0x3f8,115200
- serial,ttyS0,115200
- ttyS0,115200

`debug`

```
if (cmdline_find_option_bool("debug"))
    puts("early console in setup code\n");
```

`puts` tty.c `putchar` `putchar`

```
void __attribute__((section(".inittext"))) putchar(int ch)
{
```

```
    if (ch == '\n')
        putchar('\r');

    bios_putchar(ch);

    if (early_serial_base != 0)
        serial_putchar(ch);
}
```

`__attribute__((section(".inittext")))` `.inittext` `.inittext` setup.ld

`\n` `putchar` `\r` `bios_putchar` bios int10

```
static void __attribute__((section(".inittext"))) bios_putchar(int ch)
{
    struct biosregs ireg;

    initregs(&ireg);
    ireg.bx = 0x0007;
    ireg.cx = 0x0001;
    ireg.ah = 0x0e;
    ireg.al = ch;
    intcall(0x10, &ireg, NULL);
}
```

`initreg` `biosregs` `memset` `biosregs` 0

```
    memset(reg, 0, sizeof *reg);
    reg->eflags |= X86_EFLAGS_CF;
    reg->ds = ds();
    reg->es = ds();
    reg->fs = fs();
    reg->gs = gs();
```

memset :

```
GLOBAL(memset)
    pushw   %di
    movw    %ax, %di
    movzbl  %dl, %eax
    imull   $0x01010101,%eax
    pushw   %cx
    shrw    $2, %cx
    rep; stosl
    popw    %cx
    andw    $3, %cx
    rep; stosb
    popw    %di
    retl
ENDPROC(memset)
```

`memset` `memcpy` `fastcall` `ax dx` `cx`

memcpy `memset` `di` `biosregs` `ax` `di` `movzbl` `dl` `ax` `ax` `di`

`imull` `eax` `0x01010101` 4 `0x7` `imull` `eax` `0x7` `imull` `eax` `0x07070707` 4 `0x7` `imull` `rep; stosl` `eax` `es:di`

`bisoregs` `initregs` `bios_putchar` 0x10 `putchar` serial_putchar

bssheader.S (part) `init_heap`

`loadflags` `CAN_USE_HEAP` :

```
    char *stack_end;

    //%P1 is (-STACK_SIZE)
    if (boot_params.hdr.loadflags & CAN_USE_HEAP) {
        asm("leal %P1(%%esp),%0"
            : "=r" (stack_end) : "i" (-STACK_SIZE));
```

`stack_end = esp - STACK_SIZE` .

```
    //heap_end = heap_end_ptr + 512
    heap_end = (char *)((size_t)boot_params.hdr.heap_end_ptr + 0x200);
```

`heap_end` `stack_end` `stack_end` `heap_end`

`GET_HEAP`

CPU

arch/x86/boot/cpu.c `validate_cpu` CPUCPU

`validate_cpu` `check_cpu` CPUCPU

```
/*from cpu.c*/
check_cpu(&cpu_level, &req_level, &err_flags);
/*after check_cpu call, req_level = req_level defined in cpucheck.c*/
if (cpu_level < req_level) {
    printf("This kernel requires an %s CPU, ", cpu_name(req_level));
    printf("but only detected an %s CPU.\n", cpu_name(cpu_level));
    return -1;
}
```

`check_cpu` 1cpucpu64cpu long mode, 2) CPUCPUAMDcpu `SSE+SSE2`

`detect_memory` `0xe820` `0xe801` `0x88` arch/x86/boot/memory.c `detect_memory_e820`

`initregs` `biosregs` `0xe820`

```
    initregs(&ireg);
    ireg.ax  = 0xe820;
    ireg.cx  = sizeof buf;
    ireg.edx = SMAP;
    ireg.di  = (size_t)&buf;
```

- `ax` `0xe820`
- `cx`
- `edx` `SMAP` `0x534d4150`
- `es:di`
- `ebx` 0.

`0x15` `ebx` `biosregs` `0x15` `0x15` eflags `X86_EFLAGS_CF` :

```
    intcall(0x15, &ireg, &oreg);
    ireg.ebx = oreg.ebx;
```

`e820entry` 3:

-
-
- reserved, usable)

`dmesg`

```
[    0.000000] e820: BIOS-provided physical RAM map:
[    0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009fbff] usable
[    0.000000] BIOS-e820: [mem 0x000000000009fc00-0x000000000009ffff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000000f0000-0x00000000000fffff] reserved
[    0.000000] BIOS-e820: [mem 0x0000000000100000-0x000000003ffdffff] usable
[    0.000000] BIOS-e820: [mem 0x000000003ffe0000-0x000000003fffffff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000fffc0000-0x00000000ffffffff] reserved
```

`keyboard_init()` `initregs` 0x16

```
    initregs(&ireg);
    ireg.ah = 0x02;     /* Get keyboard status */
    intcall(0x16, &ireg, &oreg);
    boot_params.kbd_status = oreg.al;
```

0x16

```
    ireg.ax = 0x0305;   /* Set keyboard repeat rate */
    intcall(0x16, &ireg, NULL);
```

:

query_mca 0x15BIOS

```
int query_mca(void)
{
    struct biosregs ireg, oreg;
    u16 len;

    initregs(&ireg);
    ireg.ah = 0xc0;
    intcall(0x15, &ireg, &oreg);

    if (oreg.eflags & X86_EFLAGS_CF)
        return -1;  /* No MCA present */

    set_fs(oreg.es);
    len = rdfs16(oreg.bx);

    if (len > sizeof(boot_params.sys_desc_table))
        len = sizeof(boot_params.sys_desc_table);

    copy_from_fs(&boot_params.sys_desc_table, oreg.bx, len);
    return 0;
}
```

`ah` `0xc0` `0x15` BIOS carry flagBIOS MCACF0 `ES:BX`

```
Offset  Size    Description
 00h    WORD    number of bytes following
 02h    BYTE    model (see #00515)
 03h    BYTE    submodel (see #00515)
 04h    BYTE    BIOS revision: 0 for first release, 1 for 2nd, etc.
 05h    BYTE    feature byte 1 (see #00510)
 06h    BYTE    feature byte 2 (see #00511)
 07h    BYTE    feature byte 3 (see #00512)
 08h    BYTE    feature byte 4 (see #00513)
 09h    BYTE    feature byte 5 (see #00514)
---AWARD BIOS---
 0Ah  N BYTEs   AWARD copyright notice
---Phoenix BIOS---
 0Ah    BYTE    ??? (00h)
 0Bh    BYTE    major version
 0Ch    BYTE    minor version (BCD)
 0Dh  4 BYTEs   ASCIZ string "PTL" (Phoenix Technologies Ltd)
---Quadram Quad386---
 0Ah 17 BYTEs   ASCII signature string "Quadram Quad386XT"
---Toshiba (Satellite Pro 435CDS at least)---
 0Ah  7 BYTEs   signature "TOSHIBA"
 11h    BYTE    ??? (8h)
 12h    BYTE    ??? (E7h) product ID??? (guess)
 13h  3 BYTEs   "JPN"
```

`set_fs` `es` `fs` :

```
static inline void set_fs(u16 seg)
{
    asm volatile("movw %0,%%fs" : : "rm" (seg));
}
```

boot.h `set_fs` , `set_gs`

`query_mca` `es:bx` `boot_params.sys_desc_table`

`query_ist` Intel SpeedStepCPU `0x15` `boot_params`

query_apm_bios BIOS `query_apm_bios` `0x15` `ax` `0x5300` APM `bx` `cx` `bx` `0x504d` (PM) `cx` `0x02` (0x0232)

`ax = 0x5304` `0x15` `APM` `ax = 0x5303` `0x15` 32 `APM` `ax = 0x5300` `0x15` APM `boot_params.apm_bios_info`

`CONFIG_APM` `CONFIG_APM_MODULE` `query_apm_bios`

```
#if defined(CONFIG_APM) || defined(CONFIG_APM_MODULE)
    query_apm_bios();
#endif
```

`query_edd` , BIOS `Enhanced Disk Drive` `query_edd`

edd edd `off query_edd`

EDD `query_edd` BIOSEDD

```
for (devno = 0x80; devno < 0x80+EDD_MBR_SIG_MAX; devno++) {
    if (!get_edd_info(devno, &ei) && boot_params.eddbuf_entries < EDDMAXNR) {
        memcpy(edp, &ei, sizeof ei);
        edp++;
        boot_params.eddbuf_entries++;
    }
    ...
    ...
```

```
...
```

`0x80` `EDD_MBR_SIG_MAX` 16 edd_info `get_edd_info` `0x13` `ah = 0x41`) EDDEDD `0x13` `ah = 0x48` `si` EDD `si`

twitter.

PR **linux-insides-zh**

- Protected mode
- Protected mode
- Long mode
- Nice explanation of CPU Modes with code
- How to Use Expand Down Segments on Intel 386 and Later CPUs
- earlyprintk documentation
- Kernel Parameters
- Serial console
- Intel SpeedStep
- APM
- EDD specification
- TLDP documentation for Linux Boot Process (old)
- Previous Part
- BIOS Interrupt

`set_video` main.c

-
-
-

`set_video` arch/x86/boot/video.c `boot_params.hdr`

```
u16 mode = boot_params.hdr.vid_mode;
```

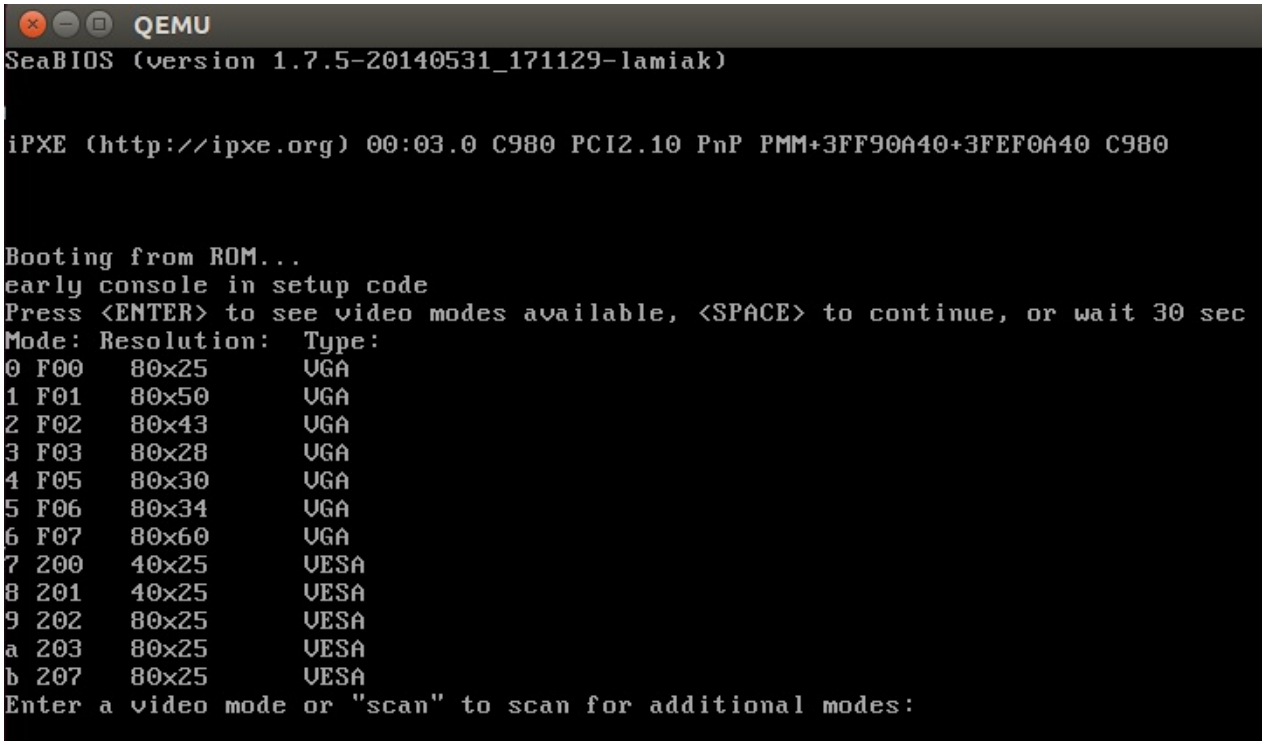
`boot_params.hdr` `copy_boot_params` `boot_params.hdr` `vid_mode` `kernel boot protocol` `vid_mode`

```
Offset    Proto    Name        Meaning
/Size
01FA/2    ALL        vid_mode    Video mode control
```

`linux kernel boot protocol` `vid_mode`

```
**** SPECIAL COMMAND LINE OPTIONS
vga=<mode>
    <mode> here is either an integer (in C notation, either
    decimal, octal, or hexadecimal) or one of the strings
    "normal" (meaning 0xFFFF), "ext" (meaning 0xFFFE) or "ask"
    (meaning 0xFFFD).  This value should be entered into the
    vid_mode field, as it is used by the kernel before the command
    line is parsed.
```

`vga` grub 0XFFFD ask2



`u16`

Type	char	short	int	long	u8	u16	u32	u64
Size	1	2	4	8	1	2	4	8

API

`set_video` `vid_mod` `RESET_HEAP` HEAP `_end` `RESET_HEAP` boot.h

```
#define RESET_HEAP() ((void *)( HEAP = _end ))
```

`init_heap` HEAP `boot.h` HEAP

```
#define RESET_HEAP() ((void *)( HEAP = _end ))
```

HEAP `_end` `_end` `boot.h` `extern char _end[];`

`GET_HEAP`

```
#define GET_HEAP(type, n) \
    ((type *)__get_heap(sizeof(type),__alignof__(type),(n)))
```

`__get_heap` `__get_heap` 3

-
- `__alignof__(type)` (gcc

- `n`

`__get_heap`

```
static inline char *__get_heap(size_t s, size_t a, size_t n)
{
    char *tmp;

    HEAP = (char *)(((size_t)HEAP+(a-1)) & ~(a-1));
    tmp = HEAP;
    HEAP += s*n;
    return tmp;
}
```

`a` `HEAP` `HEAP` `tmp` `n` `tmp`

HEAP

```
static inline bool heap_free(size_t n)
{
    return (int)(heap_end - HEAP) >= (int)n;
}
```

`heap_end - HEAP`

HEAP

HEAP `RESET_HEAP()` `set_video` `store_mode_params` `boot_params.screen_info` include/uapi/linux/screen_info.h

`store_mode_params` `store_cursor_position` `store_cursor_poistion`

`biosregs` `AH` `0x3` `0x10` BIOS `DL` `DH` 2 `boot_params.screen_info` `orig_x` `orig_y`

`store_cursor_position` `store_mode_params` `store_video_mode` `boot_params.screen_info.orig_video_mode`

`store_mode_params` `video_segment` BIOS

```
0xB000:0x0000     32 Kb     Monochrome Text Video Memory
0xB800:0x0000     32 Kb     Color Text Video Memory
```

MDA, HGC VGA `video_sgement` `0xB000` `video_segment` `0xB800` `store_mode_params` `boot_params.screen_info.orig_video_points`

```
//
set_fs(0);
font_size = rdfs16(0x485);
boot_params.screen_info.orig_video_points = font_size;
```

`set_fs` boot.h `0` `FS` `0x485` `boot_params.screen_info.orig_video_points`

```
 x = rdfs16(0x44a);
 y = (adapter == ADAPTER_CGA) ? 25 : rdfs8(0x484)+1;
```

`0x44a` `0x484` `boot_params.screen_info.orig_video_cols` `boot_params.screen_info.orig_video_lines` `store_mode_params`

`set_video` `save_screen` HEAP `saved_screen`

```
static struct saved_screen {
    int x, y;
    int curx, cury;
    u16 *data;
} saved;
```

HEAP

```
if (!heap_free(saved.x*saved.y*sizeof(u16)+512))
        return;
```

HEAP HEAP `saved_screen` HEAP

`set_video` `probe_cards(0)` arch/x86/boot/video-mode.c

```
for (card = video_cards; card < video_cards_end; card++) {
        if (card->unsafe == unsafe) {
            if (card->probe)
                card->nmodes = card->probe();
            else
                card->nmodes = 0;
        }
}
```

`video_cards` arch/x86/boot/setup.ld `.videocards`

```
    .videocards    : {
        video_cards = .;
        *(.videocards)
        video_cards_end = .;
    }
```

```
static __videocard video_vga = {
    .card_name    = "VGA",
    .probe        = vga_probe,
    .set_mode    = vga_set_mode,
};
```

`__videocard`

```
#define __videocard struct card_info __attribute__((used,section(".videocards")))
```

`__videocard` `card_info`

```
struct card_info {
    const char *card_name;
    int (*set_mode)(struct mode_info *mode);
    int (*probe)(void);
    struct mode_info *modes;
    int nmodes;
    int unsafe;
    u16 xmode_first;
    u16 xmode_n;
};
```

`.videocards` `card_info` `probe_cards` `video_cards` `card_info`

`probe_cards` `set_video` `vid_mode=ask` `vid_mod` `set_mode`

```
for (;;) {
      if (mode == ASK_VGA)
          mode = mode_menu();

      if (!set_mode(mode))
          break;

      printf("Undefined video mode number: %x\n", mode);
      mode = ASK_VGA;
  }
```

video-mode.c `set_mode`

`set_mode` `mode` `raw_set_mode` `card_info` `card_info` `set_mode` `video_vga` `card_info` `set_mode` `vga_set_mode` `vga_set_mode` vga

```
static int vga_set_mode(struct mode_info *mode)
{
    vga_set_basic_mode();

    force_x = mode->x;
    force_y = mode->y;

    switch (mode->mode) {
    case VIDEO_80x25:
        break;
    case VIDEO_8POINT:
        vga_set_8font();
        break;
    case VIDEO_80x43:
        vga_set_80x43();
        break;
    case VIDEO_80x28:
        vga_set_14font();
        break;
    case VIDEO_80x30:
        vga_set_80x30();
        break;
    case VIDEO_80x34:
        vga_set_80x34();
        break;
    case VIDEO_80x60:
        vga_set_80x60();
        break;
    }
    return 0;
}
```

`vga_set***` `0x10` BIOS

`boot_params.hdr.vid_mode`

`set_video` `vesa_store_edid` EDID (**E**xtended **D**isplay **I**dentification **D**ata) `set_video` `do_restore`

main.c `go_to_protected_mode`

`go_to_protected_mode` arch/x86/boot/pm.c

`go_to_protected_mode` `realmode_switch_hook` `realmode_switch` hook NMI NMI bootloader DOS hook boot protocol (see **ADVANCED BOOT LOADER HOOKS**) hook

```
/*
 * Invoke the realmode switch hook if present; otherwise
 * disable all interrupts.
 */
static void realmode_switch_hook(void)
{
    if (boot_params.hdr.realmode_swtch) {
        asm volatile("lcallw *%0"
                : : "m" (boot_params.hdr.realmode_swtch)
                : "eax", "ebx", "ecx", "edx");
    } else {
        asm volatile("cli");
        outb(0x80, 0x70); /* Disable NMI */
        io_delay();
    }
}
```

`realmode_switch` 16 16 NMI `realmode_swtch` hook `lcallw` hook `else` NMI

```
asm volatile("cli");
outb(0x80, 0x70);    /* Disable NMI */
io_delay();
```

`cli` `IF` NMI

CPU CPU NMI NMI

NMI `0x80` CMOS `0x70` `io_delay` I/O `io_delay`

```
static inline void io_delay(void)
{
    const u16 DELAY_PORT = 0x80;
    asm volatile("outb %%al,%0" : : "dN" (DELAY_PORT));
}
```

I/O `0x80` 1 ms `al` `io_delay` `realmode_switch_hook`

`enable_a20` A20 line arch/x86/boot/a20.c A20 `a20_test_short` `a20_test` A20

```
static int a20_test(int loops)
{
    int ok = 0;
    int saved, ctr;

    set_fs(0x0000);
    set_gs(0xffff);

    saved = ctr = rdfs32(A20_TEST_ADDR);

    while (loops--) {
        wrfs32(++ctr, A20_TEST_ADDR);
        io_delay();    /* Serialize and make delay constant */
        ok = rdgs32(A20_TEST_ADDR+0x10) ^ ctr;
        if (ok)
            break;
    }

    wrfs32(saved, A20_TEST_ADDR);
    return ok;
}
```

`0x0000` `FS` `0xffff` `GS` `rdfs32` `A20_TEST_ADDR` `saved` `ctr`

`wrfs32` `ctr` `fs:gs` 1ms `GS:A20_TEST_ADDR+0x10` 0 A20 A20 A20 BIOS `0X15` A20

`enabled_a20` `die` `die` arch/x86/boot/header.S:

```
die:
    hlt
    jmp    die
    .size    die, .-die
```

A20 `reset_coprocessor`

```
outb(0, 0xf0);
outb(0, 0xf1);
```

`0` I/O `0xf0` `0xf1`

`mask_all_interrupts`

```
outb(0xff, 0xa1);       /* Mask all interrupts on the secondary PIC */
outb(0xfb, 0x21);       /* Mask all but cascade on the primary PIC */
```

(Programmable Interrupt Controller) IRQ2IRQ2 CPU

`setup_idt` IDT

```
static void setup_idt(void)
{
    static const struct gdt_ptr null_idt = {0, 0};
    asm volatile("lidtl %0" : : "m" (null_idt));
}
```

`lidtl` `null_idt` IDT `null_idt` 0 `null_idt` `gdt_ptr`

```
struct gdt_ptr {
    u16 len;
    u32 ptr;
} __attribute__((packed));
```

16 bit 32 bit `__attribute__((packed))` 48 bit `GDTR` GDTR 48 bit

`setup_gdt` `setup_gdt` `boot_gdt` GDTR

```
//GDT_ENTRY_BOOT_CS http://lxr.free-electrons.com/source/arch/x86/include/asm/segment.h#L19 = 2
 static const u64 boot_gdt[] __attribute__((aligned(16))) = {
     [GDT_ENTRY_BOOT_CS] = GDT_ENTRY(0xc09b, 0, 0xfffff),
     [GDT_ENTRY_BOOT_DS] = GDT_ENTRY(0xc093, 0, 0xfffff),
     [GDT_ENTRY_BOOT_TSS] = GDT_ENTRY(0x0089, 4096, 103),
 };
```

`boot_gdt` TSS (Task State Segment,) `null_idt` TSS TSS Intel
`boot_gdt` `__attribute__((aligned(16)))` 16 16

```
#include <stdio.h>

struct aligned {
    int a;
}__attribute__((aligned(16)));

struct nonaligned {
    int b;
};

int main(void)
{
    struct aligned    a;
    struct nonaligned na;

    printf("Not aligned - %zu \n", sizeof(na));
    printf("Aligned - %zu \n", sizeof(a));

    return 0;
}
```

16 `int` 16

```
$ gcc test.c -o test && test
Not aligned - 4
Aligned - 16
```

`boot_gdt` `GDT_ENTRY_BOOT_CS = 2` 2 `align 16` 8*5=40 `align 16` 48

`GDT_ENTRY` 3 `GDT_ENTRY_BOOT_CS` `GDT_ENTRY` 3

- - 0
- - 0xfffff
- - 0xc09b

0 `0xfffff` 1 MB

```
1100 0000 1001 1011
```

:

- 1 - (G) 1 `0xfffff * 4kb = 4GB`
- 1 - (D) 32
- 0 - (L) long mode
- 0 - (AVL) Linux
- 0000 - 4
- 1 - (P)
- 00 - (DPL) - 0
- 1 - (S)
- 101 - /
- 1 -

Intel® 64 and IA-32 Architectures Software Developer's Manuals 3A

GDT

```
gdt.len = sizeof(boot_gdt)-1;
```

GDT gdt.ptr

```
gdt.ptr = (u32)&boot_gdt + (ds() << 4);
```

ds << 4 +

`lgdtl` GDT GDTR

```
asm volatile("lgdtl %0" : : "m" (gdt));
```

`go_to_protected_mode` IDT, GDT NMI `protected_mode_jump`

```
protected_mode_jump(boot_params.hdr.code32_start, (u32)&boot_params + (ds() << 4));
```

`protected_mode_jump` arch/x86/boot/pmjump.S2:

-
- `boot_params`

`eax` `edx`

`boot_params` `esi` `cs` `bx` `bx << 4 + 2` `bx` 2 `cx` TSS `di`

```
movw    $__BOOT_DS, %cx
movw    $__BOOT_TSS, %di
```

`GDT_ENTRY_BOOT_CS` 2 8 `cx` `2*8 = 16 di` `4*8 =32`

`CR0` CPU

```
movl    %cr0, %edx
orb     $X86_CR0_PE, %dl
movl    %edx, %cr0
```

32

```
    .byte    0x66, 0xea
2:    .long    in_pm32
    .word    __BOOT_CS ;(GDT_ENTRY_BOOT_CS*8) = 16
```

- `0x66` 16 32
- `0xea` -
- `in_pm32`
- `__BOOT_CS`

```
.code32
.section ".text32","ax"
```

`CS` :

```
GLOBAL(in_pm32)
```

```
movl    %ecx, %ds
movl    %ecx, %es
movl    %ecx, %fs
movl    %ecx, %gs
movl    %ecx, %ss
```

`$__BOOT_DS` `cx` `CS` 0

```
xorl    %ecx, %ecx
xorl    %edx, %edx
xorl    %ebx, %ebx
xorl    %ebp, %ebp
xorl    %edi, %edi
```

32

```
jmpl    *%eax ;?jmpl cs:eax?
```

32

32 long mode

twitter.

PR **linux-insides-zh**

- VGA
- VESA BIOS Extensions
- Data structure alignment
- Non-maskable interrupt
- A20
- GCC designated inits
- GCC type attributes
- Previous part

. Part 4.

64

CPU SSE

arch/x86/boot/pmjump.S 32

```
jmpl    *%eax
```

eax 32 x86 linux

```
When using bzImage, the protected-mode kernel was relocated to 0x100000
```

```
 bzImage  0x100000
```

32

```
eax            0x100000    1048576
ecx            0x0         0
edx            0x0         0
ebx            0x0         0
esp            0x1ff5c     0x1ff5c
ebp            0x0         0x0
esi            0x14470     83056
edi            0x0         0
eip            0x100000     0x100000
eflags         0x46         [ PF ZF ]
cs             0x10    16
ss             0x18    24
ds             0x18    24
es             0x18    24
fs             0x18    24
gs             0x18    24
```

cs - 0x10 eip 0x100000 0 0:0x100000 0x100000 32

32

arch/x86/boot/compressed/head_64.S 32

```
    __HEAD
    .code32
ENTRY(startup_32)
....
....
....
ENDPROC(startup_32)
```

(compressed) bzimage vmlinux + + gzip head_64.S

arch/x86/boot/compressed

- head_32.S

- head_64.S

`x86_64` `head_64.S` `head_32.S` arch/x86/boot/compressed/Makefile

```
vmlinux-objs-y := $(obj)/vmlinux.lds $(obj)/head_$(BITS).o $(obj)/misc.o \
    $(obj)/string.o $(obj)/cmdline.o \
    $(obj)/piggy.o $(obj)/cpuflags.o
```

`$(obj)/head_$(BITS).o` `$(BITS)` head_32.o head_64.o `$(BITS)` arch/x86/Makefile .config

```
ifeq ($(CONFIG_X86_32),y)
        BITS := 32
        ...
        ...
else
        BITS := 64
        ...
        ...
endif
```

arch/x86/boot/compressed/head_64.S `startup_32`

```
    __HEAD
    .code32
ENTRY(startup_32)
```

`__HEAD` include/linux/init.h

```
#define __HEAD        .section    ".head.text","ax"
```

`.head.text` `ax` arch/x86/boot/compressed/vmlinux.lds.S

```
SECTIONS
{
    . = 0;
    .head.text : {
        _head = . ;
        HEAD_TEXT
        _ehead = . ;
    }
```

`GNU LD` `.` - 0 `0`

```
Be careful parts of head_64.S assume startup_32 is at address 0.
```

```
 head_64.S  startup_32  0
```

`startup_32`

`startup_32` `cld` `DF` stos scas `esi` `edi`

`DF` `loadflags` `KEEP_SEGMENTS` `loadflags` `CAN_USE_HEAP` `KEEP_SEGMENTS` linux

```
Bit 6 (write): KEEP_SEGMENTS
  Protocol: 2.07+
  - If 0, reload the segment registers in the 32bit entry point.
  - If 1, do not reload the segment registers in the 32bit entry point.
    Assume that %cs %ds %ss %es are all set to flat segments with
    a base of 0 (or the equivalent for their environment).
```

```
 6  (): KEEP_SEGMENTS
  : 2.07+
  - 032
  - 132 %cs %ds %ss %es 0
```

`KEEP_SEGMENTS` `loadflags` `ds` , `ss` `es` `0`

```
    testb $(1 << 6), BP_loadflags(%esi)
    jnz 1f

    cli
    movl    $(__BOOT_DS), %eax
    movl    %eax, %ds
    movl    %eax, %es
    movl    %eax, %ss
```

`__BOOT_DS` `0x18` `KEEP_SEGMENTS` `1f` `1f` `__BOOT_DS`
arch/x86/boot/pmjump.S Linux 3232 `startup_32` `startup_32`

`KEEP_SEGMENTS` `setup.ld.S` `.head.text` `. = 0` `0` `objdump`

```
arch/x86/boot/compressed/vmlinux:     file format elf64-x86-64


Disassembly of section .head.text:

0000000000000000 <startup_32>:
   0:   fc                      cld
   1:   f6 86 11 02 00 00 40    testb  $0x40,0x211(%rsi)
```

`objdump` `startup_32` `0` `rip` `startup_32`

```
call label
label: pop %reg
```

Linux `startup_32`

```
    leal    (BP_scratch+4)(%esi), %esp
    call    1f
1:  popl    %ebp
    subl    $1b, %ebp
```

`esi` boot_params `bootparams` `scratch` `0x1e4` 4 `call` `scratch` 4 `esp`
`BP_scratch` `4` `x86_64` `1f` `ebp` `call` `1f`
`startup_32`

```
startup_32 (0x0)     +----------------------+
                     |                      |
                     |                      |
                     |                      |
                     |                      |
                     |                      |
                     |                      |
                     |                      |
```

```
                    |                       |
1f (0x0 + 1f offset) +-----------------------+ %ebp -
                    |                       |
                    |                       |
                    +-----------------------+
```

`startup_32` `0x0` `1f` `0x0 + 1f` `0x22` `ebp` `1f` `ebp` `1f` `startup_32`
Linux `0x100000` gdb `1f` `0x100022` `ebp` `0x100022`

```
$ gdb
(gdb)$ target remote :1234
Remote debugging using :1234
0x0000fff0 in ?? ()
(gdb)$ br *0x100022
Breakpoint 1 at 0x100022
(gdb)$ c
Continuing.

Breakpoint 1, 0x00100022 in ?? ()
(gdb)$ i r
eax            0x18    0x18
ecx            0x0    0x0
edx            0x0    0x0
ebx            0x0    0x0
esp            0x144a8    0x144a8
ebp            0x100021    0x100021
esi            0x142c0    0x142c0
edi            0x0    0x0
eip            0x100022    0x100022
eflags         0x46    [ PF ZF ]
cs             0x10    0x10
ss             0x18    0x18
ds             0x18    0x18
es             0x18    0x18
fs             0x18    0x18
gs             0x18    0x18
```

`subl $1b, %ebp`

```
nexti
...
ebp            0x100000    0x100000
...
```

`startup_32` `0x100000` `startup_32` CPU SSE

CPU

`startup_32` `esp`

```
    movl    $boot_stack_end, %eax
    addl    %ebp, %eax
    movl    %eax, %esp
```

`boots_stack_end` arch/x86/boot/compressed/head_64.S .bss

```
    .bss
    .balign 4
boot_heap:
    .fill BOOT_HEAP_SIZE, 1, 0
boot_stack:
    .fill BOOT_STACK_SIZE, 1, 0
boot_stack_end:
```

`boot_stack_end` `eax` `eax` `boot_stack_end` `0x0 + boot_stack_end` `boot_stack_end` `startup_32` `ebp` `eax` `boot_stack_end`

CPU CPU `SSE` `verify_cpu`

```
    call    verify_cpu
    testl   %eax, %eax
    jnz     no_longmode
```

arch/x86/kernel/verify_cpu.S cpuid `SSE` `eax` 01

`eax` 0 `no_longmode` `hlt` CPU

```
no_longmode:
1:
    hlt
    jmp     1b
```

`eax` 0

Linux 32 `0x100000` 32 `CONFIG_PHYSICAL_START` `0x1000000` `16 MB` kdump Linux - `CONFIG_RELOCATABLE`

```
This builds a kernel image that retains relocation information
so it can be loaded someplace besides the default 1MB.

Note: If CONFIG_RELOCATABLE=y, then the kernel runs from the address
it has been loaded at and the compile time physical address
(CONFIG_PHYSICAL_START) is used as the minimum location.
```

```
 1MB

 CONFIG_RELOCATABLE=y   (CONFIG_PHYSICAL_START)
```

Linux /arch/x86/boot/compressed/Makefile `-fPIC`

```
KBUILD_CFLAGS += -fno-strict-aliasing -fPIC
```

`startup_32` Linux `CONFIG_RELOCATABLE`

```
#ifdef CONFIG_RELOCATABLE
    movl    %ebp, %ebx
    movl    BP_kernel_alignment(%esi), %eax
    decl    %eax
    addl    %eax, %ebx
    notl    %eax
    andl    %eax, %ebx
    cmpl    $LOAD_PHYSICAL_ADDR, %ebx
    jge     1f
#endif
    movl    $LOAD_PHYSICAL_ADDR, %ebx
1:
    addl    $z_extract_offset, %ebx
```

`ebp` `startup_32` `CONFIG_RELOCATABLE` `ebx` `2M` `LOAD_PHYSICAL_ADDR` `LOAD_PHYSICAL_ADDR` arch/x86/include/asm/boot.h

```
#define LOAD_PHYSICAL_ADDR ((CONFIG_PHYSICAL_START \
                + (CONFIG_PHYSICAL_ALIGN - 1)) \
                & ~(CONFIG_PHYSICAL_ALIGN - 1))
```

`CONFIG_PHYSICAL_ALIGN` `LOAD_PHYSICAL_ADDR` `ebx` `startup_32` `CONFIG_RELOCATABLE` `z_extract_offset`

`ebp` `ebx`

64

```
    leal    gdt(%ebp), %eax
    movl    %eax, gdt+2(%ebp)
    lgdt    gdt(%ebp)
```

`ebp` `gdt` `eax` `ebp` `gdt+2` `lgdt` `gdt`

```
    .data
gdt:
    .word   gdt_end - gdt
    .long   gdt
    .word   0
    .quad   0x0000000000000000    /* NULL descriptor */
    .quad   0x00af9a000000ffff    /* __KERNEL_CS */
    .quad   0x00cf92000000ffff    /* __KERNEL_DS */
    .quad   0x0080890000000000    /* TS descriptor */
    .quad   0x0000000000000000    /* TS continued */
gdt_end:
```

`.data` 5 `null` `CS.L = 1` `CS.D = 0` `64` `gdt` `gdt_end - gdt` `gdt` 4 `gdt` `48 GDTR-`

- (16
- (32)

`gdt` `eax` `.long gdt` `gdt+2` `GDTR` `lgdt`

PAE `cr4` `eax` 51 `cr4`

```
    movl    %cr4, %eax
    orl     $X86_CR4_PAE, %eax
    movl    %eax, %cr4
```

64

x86_64 `x86_64` `x86`

`64`

- `r8` `r15` 864
- 64 - `RIP` ;
- - ;
- 64;
- RIP ().

- 64
-

`64`

- PAE;
- `cr3` ;
- `EFER.LME` ;
- ;

`cr4` `PAE` `PAE`

`64` `4G`

Linux `4` 6

- 1 `PML4` `4` 1
- 1 `PDP` 4
- 4 `2048`

`4096` `24` KB

```
    leal    pgtable(%ebx), %edi
    xorl    %eax, %eax
    movl    $((4096*6)/4), %ecx
    rep    stosl
```

`ebx` `pgtable` `edi` `eax` `ecx` `6144` `rep stosl` `eax` `edi` `edi` 4 `ecx` 4 `ecx` 0 `6144` `ecx`

`pgtable` arch/x86/boot/compressed/head_64.S

```
    .section ".pgtable","a",@nobits
    .balign 4096
pgtable:
    .fill 6*4096, 1, 0
```

`.pgtable` `24KB`

`pgtable` - `PML4`

```
    leal    pgtable + 0(%ebx), %edi
    leal    0x1007 (%edi), %eax
    movl    %eax, 0(%edi)
```

`ebx` `startup_32` `pgtable` `edi` `0x1007` `eax` `0x1007` `PML4` `4096` `7` `7` `PML4` `PRESENT+RW+USER` `PDP` `PML4`

`PDP` 3 4 `PRESENT+RW+USE` `Page Directory 2`

```
    leal    pgtable + 0x1000(%ebx), %edi
    leal    0x1007(%edi), %eax
    movl    $4, %ecx
1:  movl    %eax, 0x00(%edi)
    addl    $0x00001000, %eax
```

```
    addl    $8, %edi
    decl    %ecx
    jnz     1b
```

3 `pgtable` `4096` `0x1000` `edi` 2 `eax` `4` `ecx` `edi` `edi`
`0x7` 8 `eax` `edi` `2048` `2MB`

```
    leal    pgtable + 0x2000(%ebx), %edi
    movl    $0x00000183, %eax
    movl    $2048, %ecx
1:  movl    %eax, 0(%edi)
    addl    $0x00200000, %eax
    addl    $8, %edi
    decl    %ecx
    jnz     1b
```

- `$0x00000183` - `PRESENT + WRITE + MBZ` `2048` `2MB`

```
>>> 2048 * 0x00200000
4294967296
```

`4G` `4G` `PML4` `cr3`

```
    leal    pgtable(%ebx), %eax
    movl    %eax, %cr3
```

MSR `EFER.LME` `0xC0000080`

```
    movl    $MSR_EFER, %ecx
    rdmsr
    btsl    $_EFER_LME, %eax
    wrmsr
```

`MSR_EFER` arch/x86/include/uapi/asm/msr-index.h `ecx` `rdmsr` MSR `rdmsr`
`edx:eax` `ecx` `btsl` `EFER_LME` `wrmsr` `eax` `MSR`

GDT `startup_64` `eax`

```
    pushl    $__KERNEL_CS
    leal    startup_64(%ebp), %eax
```

`cr0` `PG` `PE`

```
    movl    $(X86_CR0_PG | X86_CR0_PE), %eax
    movl    %eax, %cr0
```

```
lret
```

`startup_64` `lret` CPU

64

```
    .code64
    .org 0x200
ENTRY(startup_64)
....
....
....
```

linux 4 twitter issue

PR **linux-insides-zh**

- Protected mode
- Intel® 64 and IA-32 Architectures Software Developer's Manual 3A
- GNU linker
- SSE
- Paging
- Model specific register
- .fill instruction
- Previous part
- Paging on osdev.org
- Paging Systems
- x86 Paging Tutorial

. Part 5.

64...

`64` ——— `startup_64` arch/x86/boot/compressed/head_64.S `startup_32` `startup_64`

```
    pushl    $__KERNEL_CS
    leal    startup_64(%ebp), %eax
    ...
    ...
    ...
    pushl    %eax
    ...
    ...
    ...
    lret
```

CPU `64` `startup_64`

```
    .code64
    .org 0x200
ENTRY(startup_64)
    xorl    %eax, %eax
    movl    %eax, %ds
    movl    %eax, %es
    movl    %eax, %ss
    movl    %eax, %fs
    movl    %eax, %gs
```

`cs`

```
#ifdef CONFIG_RELOCATABLE
    leaq    startup_32(%rip), %rbp
    movl    BP_kernel_alignment(%rsi), %eax
    decl    %eax
    addq    %rax, %rbp
    notq    %rax
    andq    %rax, %rbp
    cmpq    $LOAD_PHYSICAL_ADDR, %rbp
    jge    1f
#endif
    movq    $LOAD_PHYSICAL_ADDR, %rbp
1:
    movl    BP_init_size(%rsi), %ebx
    subl    $_end, %ebx
    addq    %rbp, %rbx
```

`rbp` `rbx` `startup_32` 64 `startup_32`

```
    leaq    boot_stack_end(%rbx), %rsp
```

```
    pushq   $0
    popfq
```

`rbx` `boot_stack_entry` `rsp` arch/x86/boot/compressed/head_64.S `boot_stack_end`

```
    .bss
    .balign 4
boot_heap:
    .fill BOOT_HEAP_SIZE, 1, 0
boot_stack:
    .fill BOOT_STACK_SIZE, 1, 0
boot_stack_end:
```

`.bss` `.pgtable` arch/x86/boot/compressed/vmlinux.lds.S `.bss` `.pgtable`

```
    pushq   %rsi
    leaq    (_bss-8)(%rip), %rsi
    leaq    (_bss-8)(%rbx), %rdi
    movq    $_bss, %rcx
    shrq    $3, %rcx
    std
    rep     movsq
    cld
    popq    %rsi
```

`rsi` `rsi` `boot_params` `boot_params` `rsi` .

`leaq _bss - 8 rip rbx rsi rdi` . `startup_32` arch/x86/boot/compressed/vmlinux.lds.S

```
    . = 0;
    .head.text : {
        _head = . ;
        HEAD_TEXT
        _ehead = . ;
    }
    .rodata..compressed : {
        *(.rodata..compressed)
    }
    .text :    {
        _text = .;     /* Text */
        *(.text)
        *(.text.*)
        _etext = . ;
    }
```

`.head.text` `startup_32` .

```
    __HEAD
    .code32
ENTRY(startup_32)
...
...
...
```

`.text`

```
    .text
relocated:
...
...
...
/*
```

```
 * Do the decompression, and jump to the new kernel..
 */
...
```

`.rodata..compressed` `rsi _bss - 8` `rdi _bss - 8` `_bss rcx` `vmlinux.lds.S` / `movsq` 8 `rsi rdi`

`std` `DF` `rsi rdi` `cld DF` `boot_params rsi` .

`.text`

```
    leaq    relocated(%rbx), %rax
    jmp    *%rax
```

`.text relocated` `.bss`

```
    xorl    %eax, %eax
    leaq    _bss(%rip), %rdi
    leaq    _ebss(%rip), %rcx
    subq    %rdi, %rcx
    shrq    $3, %rcx
    rep    stosq
```

`.bss` C `eax _bss rdi _ebss rcx rep stosq`

`extract_kernel`

```
    pushq    %rsi
    movq    %rsi, %rdi
    leaq    boot_heap(%rip), %rsi
    leaq    input_data(%rip), %rdx
    movl    $z_input_len, %ecx
    movq    %rbp, %r8
    movq    $z_output_len, %r9
    call    extract_kernel
    popq    %rsi
```

`rdi boot_params` `rsi` `extract_kernel` `extract_kernel` arch/x86/boot/compressed/misc.c

- `rmode` - boot_params `boot_params`
- `heap` - `boot_heap`
- `input_data` - `arch/x86/boot/compressed/vmlinux.bin.bz2`
- `input_len` -
- `output` -
- `output_len` -

System V Application Binary Interface

`extract_kernel` arch/x86/boot/compressed/misc.c / 3264

```
free_mem_ptr     = heap;
free_mem_end_ptr = heap + BOOT_HEAP_SIZE;
```

`heap` arch/x86/boot/compressed/head_64.S `extract_kernel`

```
leaq    boot_heap(%rip), %rsi
```

`boot_heap`

```
boot_heap:
    .fill BOOT_HEAP_SIZE, 1, 0
```

`BOOT_HEAP_SIZE 0x10000` (`bzip2 0x400000`)

arch/x86/boot/compressed/kaslr.c `choose_random_location` Linux kASLR

Linux

misc.c.

```
if ((unsigned long)output & (MIN_KERNEL_ALIGN - 1))
    error("Destination physical address inappropriately aligned");

if (virt_addr & (MIN_KERNEL_ALIGN - 1))
    error("Destination virtual address inappropriately aligned");

if (heap > 0x3fffffffffffUL)
    error("Destination address too large");

if (virt_addr + max(output_len, kernel_total_size) > KERNEL_IMAGE_SIZE)
    error("Destination virtual address is beyond the kernel mapping area");

if ((unsigned long)output != LOAD_PHYSICAL_ADDR)
    error("Destination address does not match LOAD_PHYSICAL_ADDR");

if (virt_addr != LOAD_PHYSICAL_ADDR)
    error("Destination virtual address changed when not relocatable");
```

```
Decompressing Linux...
```

`__decompress`

```
__decompress(input_data, input_len, NULL, NULL, output, output_len, NULL, error);
```

`__decompress`

```
#ifdef CONFIG_KERNEL_GZIP
#include "../../../../lib/decompress_inflate.c"
#endif

#ifdef CONFIG_KERNEL_BZIP2
#include "../../../../lib/decompress_bunzip2.c"
#endif

#ifdef CONFIG_KERNEL_LZMA
#include "../../../../lib/decompress_unlzma.c"
#endif

#ifdef CONFIG_KERNEL_XZ
#include "../../../../lib/decompress_unxz.c"
#endif

#ifdef CONFIG_KERNEL_LZO
#include "../../../../lib/decompress_unlzo.c"
```

```
#endif

#ifdef CONFIG_KERNEL_LZ4
#include "../../../../lib/decompress_unlz4.c"
#endif
```

`parse_elf` `handle_relocations` . ELF `parse_elf` `readelf`

```
readelf -l vmlinux

Elf file type is EXEC (Executable file)
Entry point 0x1000000
There are 5 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000200000 0xffffffff81000000 0x0000000001000000
                 0x0000000000893000 0x0000000000893000  R E    200000
  LOAD           0x0000000000a93000 0xffffffff81893000 0x0000000001893000
                 0x000000000016d000 0x000000000016d000  RW     200000
  LOAD           0x0000000000c00000 0x0000000000000000 0x0000000001a00000
                 0x00000000000152d8 0x00000000000152d8  RW     200000
  LOAD           0x0000000000c16000 0xffffffff81a16000 0x0000000001a16000
                 0x0000000000138000 0x000000000029b000  RWE    200000
```

`parse_elf` `choose_random_location` `output` ELF

```
Elf64_Ehdr ehdr;
Elf64_Phdr *phdrs, *phdr;

memcpy(&ehdr, output, sizeof(ehdr));

if (ehdr.e_ident[EI_MAG0] != ELFMAG0 ||
    ehdr.e_ident[EI_MAG1] != ELFMAG1 ||
    ehdr.e_ident[EI_MAG2] != ELFMAG2 ||
    ehdr.e_ident[EI_MAG3] != ELFMAG3) {
        error("Kernel is not a valid ELF file");
        return;
}
```

`ELF` `ELF`

```
    for (i = 0; i < ehdr.e_phnum; i++) {
        phdr = &phdrs[i];

        switch (phdr->p_type) {
        case PT_LOAD:
#ifdef CONFIG_RELOCATABLE
            dest = output;
            dest += (phdr->p_paddr - LOAD_PHYSICAL_ADDR);
#else
            dest = (void *)(phdr->p_paddr);
#endif
            memmove(dest, output + phdr->p_offset, phdr->p_filesz);
            break;
        default:
            break;
        }
    }
```

`parse_elf` `handle_relocations` `CONFIG_X86_NEED_RELOCS` `CONFIG_RANDOMIZE_BASE` `handle_relocations` `LOAD_PHYSICAL_ADDR`

`extract_kernel` arch/x86/boot/compressed/head_64.S.

`rax`

```
jmp    *%rax
```

twitter

PR **linux-insides-zh**

- address space layout randomization
- initrd
- long mode
- bzip2
- RDRand instruction
- Time Stamp Counter
- Programmable Interval Timers
- Previous part

ACPI

- -
- -
- - start_kernel
- - start_kernel -
- -
- -
- -
- -
- RCU - RCU
- - Linux

TODO: Need proofreading

Linux Linux Linux —— PID `1` `init`

arch/x86/kernel/head_64.S `start_kernel` init/main.c

arch/x86/boot/compressed/head_64.S jmp

```
jmp    *%rax
```

`rax` Linux `decompress_kernel` arch/x86/boot/compressed/misc.c Linux

OK `decompress_kernel` `rax` arch/x86/kernel/head_64.S

```
    __HEAD
    .code64
    .globl startup_64
startup_64:
    ...
    ...
    ...
```

`startup_64` `__HEAD` `__HEAD` `.head.text`

```
#define __HEAD        .section    ".head.text","ax"
```

arch/x86/kernel/vmlinux.lds.S

```
.text : AT(ADDR(.text) - LOAD_OFFSET) {
    _text = .;
    ...
    ...
    ...
} :text = 0x9090
```

`.text` `_text` x86_64

```
. = __START_KERNEL;
```

`__START_KERNEL` arch/x86/include/asm/page_types.h

```
#define _START_KERNEL    (__START_KERNEL_map + __PHYSICAL_START)

#define __PHYSICAL_START  ALIGN(CONFIG_PHYSICAL_START, CONFIG_PHYSICAL_ALIGN)
```

- Linux - `0x1000000` ;
- Linux - `0xffffffff81000000` .

`startup_64`

```
    leaq    _text(%rip), %rbp
    subq    $_text - __START_KERNEL_map, %rbp
```

`0x1000000` kASLR `0x1000000` RIP `rip-relative` `rbp` `$_text - __START_KERNEL_map` `_text` `0xffffffff81000000` `0x1000000` `__START_KERNEL_map` `0xffffffff80000000`

```
rbp = 0x1000000 - (0xffffffff81000000 - 0xffffffff80000000)
```

`rbp` `0` `0` Linux kASLR `startup_64`

```
    testl   $~PMD_PAGE_MASK, %ebp
    jnz     bad_address
```

`rbp` 32 `PMD_PAGE_MASK` `PMD_PAGE_MASK` `Page middle directory` paging

```
#define PMD_PAGE_MASK           (~(PMD_PAGE_SIZE-1))

#define PMD_PAGE_SIZE           (_AC(1, UL) << PMD_SHIFT)
#define PMD_SHIFT       21
```

`PMD_PAGE_SIZE` `2MB` `text` `2MB` `bad_address`

`18`

```
    leaq    _text(%rip), %rax
    shrq    $MAX_PHYSMEM_BITS, %rax
    jnz     bad_address
```

`46` 246

```
#define MAX_PHYSMEM_BITS       46
```

OK

Identity

```
    addq    %rbp, early_level4_pgt + (L4_START_KERNEL*8)(%rip)
    addq    %rbp, level3_kernel_pgt + (510*8)(%rip)
    addq    %rbp, level3_kernel_pgt + (511*8)(%rip)
    addq    %rbp, level2_fixmap_pgt + (506*8)(%rip)
```

`startup_64` `0x1000000` `early_level4_pgt` `level3_kernel_pgt` `rbp` `early_level4_pgt` `level3_kernel_pgt` `level2_fixmap_pgt`

```
NEXT_PAGE(early_level4_pgt)
    .fill   511,8,0
    .quad   level3_kernel_pgt - __START_KERNEL_map + _PAGE_TABLE

NEXT_PAGE(level3_kernel_pgt)
    .fill   L3_START_KERNEL,8,0
    .quad   level2_kernel_pgt - __START_KERNEL_map + _KERNPG_TABLE
    .quad   level2_fixmap_pgt - __START_KERNEL_map + _PAGE_TABLE
```

```
NEXT_PAGE(level2_kernel_pgt)
    PMDS(0, __PAGE_KERNEL_LARGE_EXEC,
        KERNEL_IMAGE_SIZE/PMD_SIZE)

NEXT_PAGE(level2_fixmap_pgt)
    .fill    506,8,0
    .quad    level1_fixmap_pgt - __START_KERNEL_map + _PAGE_TABLE
    .fill    5,8,0

NEXT_PAGE(level1_fixmap_pgt)
    .fill    512,8,0
```

`early_level4_pgt` (4096 - 8) `0` `511` `level3_kernel_pgt - __START_KERNEL_map + _PAGE_TABLE` `__START_KERNEL_map` `__START_KERNEL_map` `level3_kernel_pgt` `_PAGE_TABLE`

```
#define _PAGE_TABLE     (_PAGE_PRESENT | _PAGE_RW | _PAGE_USER | \
                         _PAGE_ACCESSED | _PAGE_DIRTY)
```

.

`level3_kernel_pgt` `510` `L3_START_KERNEL` `0` `L3_START_KERNEL` Page Upper Directory `__START_KERNEL_map` `510` `level2_kernel_pgt - __START_KERNEL_map + _KERNPG_TABLE` `level2_kernel_pgt`

```
#define _KERNPG_TABLE   (_PAGE_PRESENT | _PAGE_RW | _PAGE_ACCESSED | \
                         _PAGE_DIRTY)
```

`level2_fixmap_pgt` `level2_fixmap_pgt` `10` MB vsyscalls `level2_kernel_pgt` `PDMS` `__START_KERNEL_map` `.text` `512` MB `512` MB

`rbp` `startup_64` `startup_64`

```
    addq    %rbp, early_level4_pgt + (L4_START_KERNEL*8)(%rip)
    addq    %rbp, level3_kernel_pgt + (510*8)(%rip)
    addq    %rbp, level3_kernel_pgt + (511*8)(%rip)
    addq    %rbp, level2_fixmap_pgt + (506*8)(%rip)
```

`early_level4_pgt` `level3_kernel_pgt` `level3_kernel_pgt` `level2_kernel_pgt` `level2_fixmap_pgt` `level2_fixmap_pgt` 507 `level1_fixmap_pgt`

```
early_level4_pgt[511] -> level3_kernel_pgt[0]
level3_kernel_pgt[510] -> level2_kernel_pgt[0]
level3_kernel_pgt[511] -> level2_fixmap_pgt[0]
level2_kernel_pgt[0]   -> 512 MB kernel mapping
level2_fixmap_pgt[507] -> level1_fixmap_pgt
```

`early_level4_pgt`

Identity Map Paging

Identity Identity `1 : 1` `_text` `_early_level4_pgt` RIP `rdi` `rbx`

```
    leaq    _text(%rip), %rdi
    leaq    early_level4_pgt(%rip), %rbx
```

`rax` `_text` `_text` `_text` `PGDIR_SHIFT`

```
    movq    %rdi, %rax
    shrq    $PGDIR_SHIFT, %rax

    leaq    (4096 + _KERNPG_TABLE)(%rbx), %rdx
    movq    %rdx, 0(%rbx,%rax,8)
    movq    %rdx, 8(%rbx,%rax,8)
```

`PGDIR_SHIFT` `39` `PGDIR_SHIFT` mask

```
#define PGDIR_SHIFT     39
#define PUD_SHIFT       30
#define PMD_SHIFT       21
```

`level3_kernel_pgt` `rdx` `_KERNPG_TABLE` `level3_kernel_pgt` `early_level4_pgt`

`rdx` `4096` `early_level4_pgt` `rdi` `_text` `rax` `level3_kernel_pgt`

```
    addq    $4096, %rdx
    movq    %rdi, %rax
    shrq    $PUD_SHIFT, %rax
    andl    $(PTRS_PER_PUD-1), %eax
    movq    %rdx, 4096(%rbx,%rax,8)
    incl    %eax
    andl    $(PTRS_PER_PUD-1), %eax
    movq    %rdx, 4096(%rbx,%rax,8)
```

`level2_kernel_pgt` text data

```
    leaq    level2_kernel_pgt(%rip), %rdi
    leaq    4096(%rdi), %r8
1:   testq    $1, 0(%rdi)
    jz    2f
    addq    %rbp, 0(%rdi)
2:   addq    $8, %rdi
    cmp    %r8, %rdi
    jne    1b
```

`level2_kernel_pgt` `rdi` `r8` `level2_kernel_pgt` 0 `rdi` 8 `r8` `1`

`rbp` `_text` `phys_base` `early_level4_pgt` `rbp` `1`

```
    addq    %rbp, phys_base(%rip)
    movq    $(early_level4_pgt - __START_KERNEL_map), %rax
    jmp 1f
```

`phys_base` `level2_kernel_pgt` `512` MB

`1` `PAE` `PGE` Paging Global Extension `phys_base` `rax` `cr3`

```
1:
    movl    $(X86_CR4_PAE | X86_CR4_PGE), %ecx
    movq    %rcx, %cr4

    addq    phys_base(%rip), %rax
    movq    %rax, %cr3
```

CPU NX

```
    movl    $0x80000001, %eax
    cpuid
    movl    %edx,%edi
```

`0x80000001` `eax` `cpuid` `edx` `edi`

`MSR_EFER` `0xc0000080` `ecx` `rdmsr` CPUModel Specific Register (MSR)

```
    movl    $MSR_EFER, %ecx
    rdmsr
```

`edx:eax` `EFER`

```
63                                                                              32
 ---------------------------------------------------------------------------------
|                                                                                 |
|                               Reserved MBZ                                      |
|                                                                                 |
 ---------------------------------------------------------------------------------
31                            16  15      14      13   12  11   10  9  8 7  1   0
 ---------------------------------------------------------------------------------
|                               | T |       |       |    |   |   |   |   |   |   |
| Reserved MBZ                  | C | FFXSR | LMSLE |SVME|NXE|LMA|MBZ|LME|RAZ|SCE|
|                               | E |       |       |    |   |   |   |   |   |   |
 ---------------------------------------------------------------------------------
```

MSR `EFER` `edx:eax` `btsl` `_EFER_SCE` 01 `SCE` `SYSCALL` `SYSRET` `edi` `cpuid` 20 20 `NX` `EFER_SCE` MSR

```
    btsl    $_EFER_SCE, %eax
    btl       $20,%edi
    jnc     1f
    btsl    $_EFER_NX, %eax
    btsq    $_PAGE_BIT_NX,early_pmd_flags(%rip)
1:    wrmsr
```

NX `_EFER_NX` MSR NX `cr0` control register

- `X86_CR0_PE` - ;
- `X86_CR0_MP` - CR0TS WAIT/FWAIT
- `X86_CR0_ET` - 3868028780387;
- `X86_CR0_NE` - x87PCx87
- `X86_CR0_WP` - CPU0;
- `X86_CR0_AM` - AMEFLGSAC3;
- `X86_CR0_PG` - .

```
#define CR0_STATE    (X86_CR0_PE | X86_CR0_MP | X86_CR0_ET | \
             X86_CR0_NE | X86_CR0_WP | X86_CR0_AM | \
             X86_CR0_PG)
movl    $CR0_STATE, %eax
movq    %rax, %cr0
```

C FLAGS

```
movq stack_start(%rip), %rsp
pushq $0
popfq
```

`stack_start`

```
GLOBAL(stack_start)
.quad  init_thread_union+THREAD_SIZE-8
```

`GLOABL` arch/x86/include/asm/linkage.h

```
#define GLOBAL(name)    \
         .globl name;           \
         name:
```

`THREAD_SIZE` arch/x86/include/asm/page_64_types.h `KASAN_STACK_ORDER` :

```
#define THREAD_SIZE_ORDER       (2 + KASAN_STACK_ORDER)
#define THREAD_SIZE  (PAGE_SIZE << THREAD_SIZE_ORDER)
```

kasan `PAGE_SIZE` 4096 `THREAD_SIZE` `16` KB Linux `thread_info` union

```
union thread_union {
         struct thread_info thread_info;
         unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

`init_thread_union`

```
union thread_union init_thread_union __init_task_data =
    { INIT_THREAD_INFO(init_task) };
```

`INIT_THREAD_INFO` `task_struct`

```
#define INIT_THREAD_INFO(tsk)        \
{                                               \
    .task        = &tsk,                         \
    .flags        = 0,                            \
    .cpu        = 0,                            \
    .addr_limit    = KERNEL_DS,                     \
}
```

`task_struct` `thread_union`

```
+----------------------+
|                      |
|                      |
|                      |
|     Kernel stack     |
|                      |
|                      |
|                      |
|----------------------|
|                      |
|  struct thread_info  |
|                      |
+----------------------+
```

`8`

`lgdt`

```
lgdt    early_gdt_descr(%rip)
```

`early_gdt_descr`

```
early_gdt_descr:
    .word    GDT_ENTRIES*8-1
early_gdt_descr_base:
    .quad    INIT_PER_CPU_VAR(gdt_page)
```

`early_gdt_descr` 32

```
#define GDT_ENTRIES 32
```

`early_gdt_descr_base` . `gdt_page` arch/x86/include/asm/desc.h:

```
struct gdt_page {
    struct desc_struct gdt[GDT_ENTRIES];
} __attribute__((aligned(PAGE_SIZE)));
```

`desc_struct` `gdt` `desc_struct` :

```
struct desc_struct {
         union {
                 struct {
                         unsigned int a;
                         unsigned int b;
                 };
                 struct {
                         u16 limit0;
                         u16 base0;
                         unsigned base1: 8, type: 4, s: 1, dpl: 2, p: 1;
                         unsigned limit: 4, avl: 1, l: 1, d: 1, g: 1, base2: 8;
                 };
         };
 } __attribute__((packed));
```

`GDT` `gdt_page` `PAGE_SIZE` (`4096`) `gdt`

`INIT_PER_CPU_VAR` arch/x86/include/asm/percpu.h `init_per_cpu__`

```
#define INIT_PER_CPU_VAR(var) init_per_cpu__##var
```

`init_per_cpu__gdt_page` linker script :

```
#define INIT_PER_CPU(x) init_per_cpu__##x = x + __per_cpu_load
INIT_PER_CPU(gdt_page);
```

`INIT_PER_CPU` `init_per_cpu__gdt_page` `__per_cpu_load` GDT

per-CPU2.6 `per-CPU` CPU `gdt_page` per-CPUCPU `GDT` Concepts/per-cpu `per-CPU`

```
    xorl %eax,%eax
    movl %eax,%ds
    movl %eax,%ss
    movl %eax,%es
    movl %eax,%fs
    movl %eax,%gs
```

`gs` `irqstack`

```
    movl    $MSR_GS_BASE,%ecx
```

```
    movl    initial_gs(%rip),%eax
    movl    initial_gs+4(%rip),%edx
    wrmsr
```

`MSR_GS_BASE`

```
#define MSR_GS_BASE             0xc0000101
```

`MSR_GS_BASE` `ecx` `wrmsr` `eax` `edx` `initial_gs` `cs` , `fs` , `ds` `ss` 64 `fs` `gs` `fs` `gs` `cs` Model Specific Registers `0xc0000101` `gs.base` MSR `MSR_GS_BASE`

bootparam `rdi` (`rsi`)C

```
    movq    initial_code(%rip),%rax
    pushq   $0
    pushq   $__KERNEL_CS
    pushq   %rax
    lretq
```

`initial_code` `rax` `__KERNEL_CS` `initial_code` `lreq` `initial_code`

```
    .balign    8
    GLOBAL(initial_code)
    .quad    x86_64_start_kernel
    ...
    ...
    ...
```

`initial_code` `x86_64_start_kernel` arch/x86/kerne/head64.c

```
asmlinkage __visible void __init x86_64_start_kernel(char * real_mode_data) {
    ...
    ...
    ...
}
```

`real_mode_data` `rdi`

C

start_kernel

“”- init/main.cstart_kernel

`x86_64_start_kernel`

```
BUILD_BUG_ON(MODULES_VADDR < __START_KERNEL_map);
BUILD_BUG_ON(MODULES_VADDR - __START_KERNEL_map < KERNEL_IMAGE_SIZE);
BUILD_BUG_ON(MODULES_LEN + KERNEL_IMAGE_SIZE > 2*PUD_SIZE);
BUILD_BUG_ON((__START_KERNEL_map & ~PMD_MASK) != 0);
BUILD_BUG_ON((MODULES_VADDR & ~PMD_MASK) != 0);
BUILD_BUG_ON(!(MODULES_VADDR > __START_KERNEL));
BUILD_BUG_ON(!(((MODULES_END - 1) & PGDIR_MASK) == (__START_KERNEL & PGDIR_MASK)));
BUILD_BUG_ON(__fix_to_virt(__end_of_fixed_addresses) <= MODULES_END);
```

text `__START_KERNEL_map` text `BUILD_BUG_ON`

```
#define BUILD_BUG_ON(condition) ((void)sizeof(char[1 - 2*!!(condition)]))
```

`MODULES_VADDR < __START_KERNEL_map` `!!conditions` `condition != 0` `MODULES_VADDR < __START_KERNEL_map` `!!(condition)` 10 `2*!!(condition)` `2` `0`

-
-

C

start_kernel `cr4_init_shadow` CPU `cr4` Shadow Copy `cr4` CPU `cr4` `reset_early_page_tables` `cr3`

```
for (i = 0; i < PTRS_PER_PGD-1; i++)
    early_level4_pgt[i].pgd = 0;

next_early_pgt = 0;

write_cr3(__pa_nodebug(early_level4_pgt));
```

`PTRS_PER_PGD` `512` 0 `next_early_pgt` 0 `early_level4_pgt` `cr3` `__pa_nodebug`

```
((unsigned long)(x) - __START_KERNEL_map + phys_base)
```

`__bss_stop` `__bss_start` `_bss` `IDT`

Linux

twitter 0xAX issue

- Model Specific Register
- Paging
- Previous part - Kernel decompression
- NX
- ASLR

Linux

```
for (i = 0; i < NUM_EXCEPTION_VECTORS; i++)
    set_intr_gate(i, early_idt_handler_array[i]);
```

arch/x86/kernel/head64.c

CPUCPU—— (Interrupt Handler)

- - CPU
- -
- - CPU

0 255 32 32 255 NUM_EXCEPTION_VECTORS

```
#define NUM_EXCEPTION_VECTORS 32
```

CPU APIC CPU 0-31

```
-----------------------------------------------------------------------------------------
|Vector|Mnemonic|Description         |Type |Error Code|Source                    |
-----------------------------------------------------------------------------------------
|0     | #DE    |Divide Error        |Fault|NO        |DIV and IDIV              |
|----------------------------------------------------------------------------------------
|1     | #DB    |Reserved            |F/T  |NO        |                          |
|----------------------------------------------------------------------------------------
|2     | ---    |NMI                 |INT  |NO        |external NMI              |
|----------------------------------------------------------------------------------------
|3     | #BP    |Breakpoint          |Trap |NO        |INT 3                     |
|----------------------------------------------------------------------------------------
|4     | #OF    |Overflow            |Trap |NO        |INTO  instruction         |
|----------------------------------------------------------------------------------------
|5     | #BR    |Bound Range Exceeded|Fault|NO        |BOUND instruction         |
|----------------------------------------------------------------------------------------
|6     | #UD    |Invalid Opcode      |Fault|NO        |UD2 instruction           |
|----------------------------------------------------------------------------------------
|7     | #NM    |Device Not Available|Fault|NO        |Floating point or [F]WAIT |
|----------------------------------------------------------------------------------------
|8     | #DF    |Double Fault        |Abort|YES       |Ant instrctions which can generate NMI|
|----------------------------------------------------------------------------------------
|9     | ---    |Reserved            |Fault|NO        |                          |
|----------------------------------------------------------------------------------------
|10    | #TS    |Invalid TSS         |Fault|YES       |Task switch or TSS access |
|----------------------------------------------------------------------------------------
|11    | #NP    |Segment Not Present |Fault|NO        |Accessing segment register|
|----------------------------------------------------------------------------------------
|12    | #SS    |Stack-Segment Fault |Fault|YES       |Stack operations          |
|----------------------------------------------------------------------------------------
|13    | #GP    |General Protection  |Fault|YES       |Memory reference          |
|----------------------------------------------------------------------------------------
```

```
|14    | #PF    |Page fault          |Fault|YES       |Memory reference                      |
|------------------------------------------------------------------------------------------------
|15    | ---    |Reserved            |     |NO        |                                      |
|------------------------------------------------------------------------------------------------
|16    | #MF    |x87 FPU fp error    |Fault|NO        |Floating point or [F]Wait             |
|------------------------------------------------------------------------------------------------
|17    | #AC    |Alignment Check     |Fault|YES       |Data reference                        |
|------------------------------------------------------------------------------------------------
|18    | #MC    |Machine Check       |Abort|NO        |                                      |
|------------------------------------------------------------------------------------------------
|19    | #XM    |SIMD fp exception   |Fault|NO        |SSE[2,3] instructions                 |
|------------------------------------------------------------------------------------------------
|20    | #VE    |Virtualization exc. |Fault|NO        |EPT violations                        |
|------------------------------------------------------------------------------------------------
|21-31 | ---    |Reserved            |INT  |NO        |External interrupts                   |
-------------------------------------------------------------------------------------------------
```

CPU - IDTIDT 8IDT `(gate)` CPU 86416CPU `GDTR` `IDTR` `lidt`

64 IDT

```
127                                                                             96
 ---------------------------------------------------------------------------------
|                                                                                 |
|                               Reserved                                          |
|                                                                                 |
 ---------------------------------------------------------------------------------
95                                                                              64
 ---------------------------------------------------------------------------------
|                                                                                 |
|                             Offset 63..32                                       |
|                                                                                 |
 ---------------------------------------------------------------------------------
63                              48 47      46  44   42    39             34    32
 ---------------------------------------------------------------------------------
|                                 |       |  D  |   |     |      |   |   |     |
|       Offset 31..16             |   P   |  P  | 0 |Type |0 0 0 | 0 | 0 | IST |
|                                 |       |  L  |   |     |      |   |   |     |
 ---------------------------------------------------------------------------------
31                                  15 16                                       0
 ---------------------------------------------------------------------------------
|                                   |                                             |
|          Segment Selector         |                Offset 15..0                 |
|                                   |                                             |
 ---------------------------------------------------------------------------------
```

:

- `Offset` -
- `DPL` -
- `P` - Segment Present ;
- `Segment selector` - GDTLDT
- `IST` -

`Type`

-
-
-

(far) CPU `IF` CPU `IF` CPU CPU `iret` `IF`

0 CPU

- CPU `cs` IP
- `#PF` CPU

- `iret`

OK

IDT

```
for (i = 0; i < NUM_EXCEPTION_VECTORS; i++)
    set_intr_gate(i, early_idt_handler_array[i]);
```

`set_intr_gate`

- `
-

`IDT` `&idt_descr` `IDT` `early_idt_handler_array` arch/x86/include/asm/segment.h 32

```
#define EARLY_IDT_HANDLER_SIZE   9
#define NUM_EXCEPTION_VECTORS    32

extern const char early_idt_handler_array[NUM_EXCEPTION_VECTORS][EARLY_IDT_HANDLER_SIZE];
```

`early_idt_handler_array` `288` `9` 225

`IDT` 32 `early_idt_handler_array` arch/x86/kernel/head_64.S `set_intr_gate`

`set_intr_gate` arch/x86/include/asm/desc.h

```
#define set_intr_gate(n, addr)                          \
        do {                                                    \
                BUG_ON((unsigned)n > 0xFF);                     \
                _set_gate(n, GATE_INTERRUPT, (void *)addr, 0, 0,        \
                          __KERNEL_CS);                         \
                _trace_set_gate(n, GATE_INTERRUPT, (void *)trace_##addr,\
                                0, 0, __KERNEL_CS);             \
        } while (0)
```

`BUG_ON` 255 `256` `_set_gate` `IDT`

```
static inline void _set_gate(int gate, unsigned type, void *addr,
                             unsigned dpl, unsigned ist, unsigned seg)
{
        gate_desc s;
        pack_gate(&s, type, (unsigned long)addr, dpl, ist, seg);
        write_idt_entry(idt_table, gate, &s);
        write_trace_idt_entry(gate, &s);
}
```

`_set_gate` `pack_gate` `gate_desc`

```
static inline void pack_gate(gate_desc *gate, unsigned type, unsigned long func,
                             unsigned dpl, unsigned ist, unsigned seg)
{
        gate->offset_low        = PTR_LOW(func);
        gate->segment           = __KERNEL_CS;
        gate->ist               = ist;
        gate->p                 = 1;
        gate->dpl               = dpl;
        gate->zero0             = 0;
        gate->zero1             = 0;
        gate->type              = type;
```

```
        gate->offset_middle     = PTR_MIDDLE(func);
        gate->offset_high       = PTR_HIGH(func);
}
```

```
#define PTR_LOW(x) ((unsigned long long)(x) & 0xFFFF)
#define PTR_MIDDLE(x) (((unsigned long long)(x) >> 16) & 0xFFFF)
#define PTR_HIGH(x) ((unsigned long long)(x) >> 32)
```

`PTR_LOW` x `2` `PTR_MIDDLE` x `2` `PTR_HIGH` x `4` `__KERNEL_CS` `Interrupt Stack Table` 0 `GAT_INTERRUPT`

IDT `native_write_idt_entry` `IDT`

```
static inline void native_write_idt_entry(gate_desc *idt, int entry, const gate_desc *gate)
{
        memcpy(&idt[entry], gate, sizeof(*gate));
}
```

`idt_table` `gate_desc`

```
load_idt((const struct desc_ptr *)&idt_descr);
```

`idt_descr`

```
struct desc_ptr idt_descr = { NR_VECTORS * 16 - 1, (unsigned long) idt_table };
```

`load_idt` `lidt`

```
asm volatile("lidt %0"::"m" (*dtr));
```

`_trace_*` `_set_gate` `IDT` `idt_table` `trace_idt_table` tracepoint CPU

`early_idt_handler_array` `IDT` `early_idt_handler_array` arch/x86/kernel/head_64.S

```
    .globl early_idt_handler_array
early_idt_handlers:
    i = 0
    .rept NUM_EXCEPTION_VECTORS
    .if (EXCEPTION_ERRCODE_MASK >> i) & 1
    pushq $0
    .endif
    pushq $i
    jmp early_idt_handler_common
    i = i + 1
    .fill early_idt_handler_array + i*EARLY_IDT_HANDLER_SIZE - ., 1, 0xcc
    .endr
```

`32` `0` `early_idt_handler_common` `objdump`

```
$ objdump -D vmlinux
...
...
...
```

```
ffffffff81fe5000 <early_idt_handler_array>:
ffffffff81fe5000:       6a 00                   pushq  $0x0
ffffffff81fe5002:       6a 00                   pushq  $0x0
ffffffff81fe5004:       e9 17 01 00 00          jmpq   ffffffff81fe5120 <early_idt_handler_common>
ffffffff81fe5009:       6a 00                   pushq  $0x0
ffffffff81fe500b:       6a 01                   pushq  $0x1
ffffffff81fe500d:       e9 0e 01 00 00          jmpq   ffffffff81fe5120 <early_idt_handler_common>
ffffffff81fe5012:       6a 00                   pushq  $0x0
ffffffff81fe5014:       6a 02                   pushq  $0x2
...
...
...
```

CPU `CS` `RIP` `early_idt_handler`

```
|--------------------|
| %rflags            |
| %cs                |
| %rip               |
| rsp --> error code |
|--------------------|
```

`early_idt_handler_common` arch/x86/kernel/head_64.S (NMI)

```
    cmpl $2,(%rsp)
    je .Lis_nmi
```

`is_nmi` :

```
is_nmi:
    addq $16,%rsp
    INTERRUPT_RETURN
```

`INTERRUPT_RETURN` `iretq`

`NMI` `early_recursion_flag` `early_idt_handler_common`

```
    pushq %rax
    pushq %rcx
    pushq %rdx
    pushq %rsi
    pushq %rdi
    pushq %r8
    pushq %r9
    pushq %r10
    pushq %r11
```

```
    cmpl $__KERNEL_CS,96(%rsp)
    jne 11f
```

`11` `PANIC` `#PF` Page Fault `cr2` `rdi` `early_make_pgtable`

```
    cmpl $14,72(%rsp)
    jnz 10f
    GET_CR2_INTO(%rdi)
    call early_make_pgtable
    andl %eax,%eax
    jz 20f
```

`#PF`

```
    popq %r11
    popq %r10
    popq %r9
    popq %r8
    popq %rdi
    popq %rsi
    popq %rdx
    popq %rcx
    popq %rax
```

`iret`

`early_make_pgtable` `#PF` `4G` 4G `boot_params`

`early_make_pgtable` arch/x86/kernel/head64.c `cr2`

```
int __init early_make_pgtable(unsigned long address)
{
    unsigned long physaddr = address - __PAGE_OFFSET;
    unsigned long i;
    pgdval_t pgd, *pgd_p;
    pudval_t pud, *pud_p;
    pmdval_t pmd, *pmd_p;
    ...
    ...
    ...
}
```

`*val_t`

```
typedef unsigned long   pgdval_t;
```

`*_t` (val) `pgd_t` ..... arch/x86/include/asm/pgtable_types.h

```
typedef struct { pgdval_t pgd; } pgd_t;
```

```
extern pgd_t early_level4_pgt[PTRS_PER_PGD];
```

`early_level4_pgt` `pdg_t` `pgd`

`#PF` `pgd`

```
pgd_p = &early_level4_pgt[pgd_index(address)].pgd;
pgd = *pgd_p;
```

`pgd` `pud_p`

```
pud_p = (pudval_t *)((pgd & PTE_PFN_MASK) + __START_KERNEL_map - phys_base);
```

`PTE_PFN_MASK`

```
#define PTE_PFN_MASK            ((pteval_t)PHYSICAL_PAGE_MASK)
```

```
(~(PAGE_SIZE-1)) & ((1 << 46) - 1)
```

```
0b1111111111111111111111111111111111111111111111
```

46bit

`pgd` `next_early_pgt` `EARLY_DYNAMIC_PAGE_TABLES` `64` `EARLY_DYNAMIC_PAGE_TABLES` `next_early_pgt` `EARLY_DYNAMIC_PAGE_TABLES` `_KERPG_TABLE`

```
if (next_early_pgt >= EARLY_DYNAMIC_PAGE_TABLES) {
    reset_early_page_tables();
    goto again;
}

pud_p = (pudval_t *)early_dynamic_pgts[next_early_pgt++];
for (i = 0; i < PTRS_PER_PUD; i++)
    pud_p[i] = 0;
*pgd_p = (pgdval_t)pud_p - __START_KERNEL_map + phys_base + _KERNPG_TABLE;
```

```
pud_p += pud_index(address);
pud = *pud_p;
```

In the end we fix address of the page middle directory which contains maps kernel text+data virtual addresses:

```
pmd = (physaddr & PMD_MASK) + early_pmd_flags;
pmd_p[pmd_index(address)] = pmd;
```

`early_level4_pgt`

twitter 0xAX issue

`start_kernel`

- GNU assembly .rept
- APIC
- NMI
- Page table
- Interrupt handler
- Page Fault,
- Previous part

Linux Linux “”—— init/main.c `start_kernel` `start_kernel`

boot_params again

`IDTR` `copy_bootdata`

```
copy_bootdata(__va(real_mode_data));
```

—— `read_mode_data` `boot_params` arch/x86/include/uapi/asm/bootparam.h arch/x86/kernel/head_64.S `x86_64_start_kernel`

```
    /* rsi is pointer to real mode structure with interesting info.
       pass it to C */
    movq    %rsi, %rdi
```

`__va` init/main.c

```
#define __va(x)                 ((void *)((unsigned long)(x)+PAGE_OFFSET))
```

`PAGE_OFFSET` `__PAGE_OFFSET` `0xffff880000000000` `boot_params` `copy_bootdata` `real_mod_data` arch/x86/kernel/setup.h `boot_params`

```
extern struct boot_params boot_params;
```

`copy_boot_data` :

```
static void __init copy_bootdata(char *real_mode_data)
{
    char * command_line;
    unsigned long cmd_line_ptr;

    memcpy(&boot_params, real_mode_data, sizeof boot_params);
    sanitize_boot_params(&boot_params);
    cmd_line_ptr = get_cmd_line_ptr();
    if (cmd_line_ptr) {
        command_line = __va(cmd_line_ptr);
        memcpy(boot_command_line, command_line, COMMAND_LINE_SIZE);
    }
}
```

`__init`

`memcpy` `real_mode_data` `boot_params` bootloader `boot_params` `sanitize_boot_params` `ext_ramdisk_image` `get_cmd_line_ptr`

```
unsigned long cmd_line_ptr = boot_params.hdr.cmd_line_ptr;
cmd_line_ptr |= (u64)boot_params.ext_cmd_line_ptr << 32;
return cmd_line_ptr;
```

`get_cmd_line_ptr` `boot_params` 64 `cmd_line_ptr` `boot_command_line`

```
extern char __initdata boot_command_line[];
```

`boot_params` `load_ucode_bsp` microcode

`console_loglevel` `early_printk` `Kernel Alive` `early_printk` bug commit

`boot_params` `reset_early_page_tables`

```
    clear_page(init_level4_pgt);
```

`init_level4_pgt` arch/x86/kernel/head_64.S:

```
NEXT_PAGE(init_level4_pgt)
    .quad   level3_ident_pgt - __START_KERNEL_map + _KERNPG_TABLE
    .org    init_level4_pgt + L4_PAGE_OFFSET*8, 0
    .quad   level3_ident_pgt - __START_KERNEL_map + _KERNPG_TABLE
    .org    init_level4_pgt + L4_START_KERNEL*8, 0
    .quad   level3_kernel_pgt - __START_KERNEL_map + _PAGE_TABLE
```

bss 2.5G `clear_page` arch/x86/lib/clear_page_64.S

```
ENTRY(clear_page)
    CFI_STARTPROC
    xorl %eax,%eax
    movl $4096/64,%ecx
    .p2align 4
    .Lloop:
    decl    %ecx
#define PUT(x) movq %rax,x*8(%rdi)
    movq %rax,(%rdi)
    PUT(1)
    PUT(2)
    PUT(3)
    PUT(4)
    PUT(5)
    PUT(6)
    PUT(7)
    leaq 64(%rdi),%rdi
    jnz    .Lloop
    nop
    ret
    CFI_ENDPROC
    .Lclear_page_end:
    ENDPROC(clear_page)
```

`CFI_STARTPROC` `CFI_ENDPROC` GNU

```
#define CFI_STARTPROC           .cfi_startproc
#define CFI_ENDPROC             .cfi_endproc
```

`CFI_STARTPROC` `eax` `ecx` 64 `.Lloop` `ecx` `rax` 0 `rdi` `rdi` `init_level4_pgt` 7 `rdi` 8 `init_level4_pgt` 640 `rdi` 64 `ecx` 0 `init_level4_pgt`

`init_level4_pgt` 0

```
init_level4_pgt[511] = early_level4_pgt[511];
```

`reset_early_page_table` `early_level4_pgt`

`x86_64_start_kernel`

```
x86_64_start_reservations(real_mode_data);
```

`real_mode_data` `x86_64_start_reservations` `x86_64_start_kernel`

```
void __init x86_64_start_reservations(char *real_mode_data)
{
    if (!boot_params.hdr.version)
        copy_bootdata(__va(real_mode_data));

    reserve_ebda_region();

    start_kernel();
}
```

`x86_64_start_reservations` `boot_params.hdr.version`

```
if (!boot_params.hdr.version)
    copy_bootdata(__va(real_mode_data));
```

0 `copy_bootdata` `real_mode_data`

`reserve_ebda_region` arch/x86/kernel/head.c `EBDA` Extended BIOS Data AreaBIOSBIOS Conventiional Memory640K

`reserve_ebda_region`

```
if (paravirt_enabled())
    return;
```

`reserve_ebda_region` BIOS

```
lowmem = *(unsigned short *)__va(BIOS_LOWMEM_KILOBYTES);
lowmem <<= 10;
```

BIOSKB101024BIOS

```
ebda_addr = get_bios_ebda();
```

`get_bios_ebda` arch/x86/include/asm/bios_ebda.h

```
static inline unsigned int get_bios_ebda(void)
{
    unsigned int address = *(unsigned short *)phys_to_virt(0x40E);
    address <<= 4;
    return address;
}
```

`0x40E` `0x0040:0x000e` BIOS `phys_to_virt` `__va`

```
static inline void *phys_to_virt(phys_addr_t address)
```

```
{
        return __va(address);
}
```

`phys_to_virt` `phys_addr_t` `CONFIG_PHYS_ADDR_T_64BIT`

```
#ifdef CONFIG_PHYS_ADDR_T_64BIT
    typedef u64 phys_addr_t;
#else
    typedef u32 phys_addr_t;
#endif
```

`CONFIG_PHYS_ADDR_T_64BIT` BIOS4 `ebda_addr` BIOS

BIOS `INSANE_CUTOFF`

```
if (ebda_addr < INSANE_CUTOFF)
    ebda_addr = LOWMEM_CAP;

if (lowmem < INSANE_CUTOFF)
    lowmem = LOWMEM_CAP;
```

`INSANE_CUTOFF`

```
#define INSANE_CUTOFF        0x20000U
```

128 KB. BIOS `memblock_reserve` 1MBBIOS

```
lowmem = min(lowmem, ebda_addr);
lowmem = min(lowmem, LOWMEM_CAP);
memblock_reserve(lowmem, 0x100000 - lowmem);
```

`memblock_reserve` mm/block.c

-
-

`memblock_reserve` Linux

Linux

`memblock_reserve` `memblock_reserve`

```
memblock_reserve_region(base, size, MAX_NUMNODES, 0);
```

`memblock_reserve_region`

-
-
- NUMA
- flags

`memblock_reserve_region` `memblock_type`

```
struct memblock_type *_rgn = &memblock.reserved;
```

`memblock_type`

```
struct memblock_type {
        unsigned long cnt;
        unsigned long max;
        phys_addr_t total_size;
        struct memblock_region *regions;
};
```

BIOS `memblock`

```
struct memblock {
        bool bottom_up;
        phys_addr_t current_limit;
        struct memblock_type memory;
        struct memblock_type reserved;
#ifdef CONFIG_HAVE_MEMBLOCK_PHYS_MAP
        struct memblock_type physmem;
#endif
};
```

`memblock.reserved` `_rgn` `memblock`

```
struct memblock memblock __initdata_memblock = {
    .memory.regions        = memblock_memory_init_regions,
    .memory.cnt        = 1,
    .memory.max        = INIT_MEMBLOCK_REGIONS,
    .reserved.regions    = memblock_reserved_init_regions,
    .reserved.cnt        = 1,
    .reserved.max        = INIT_MEMBLOCK_REGIONS,
#ifdef CONFIG_HAVE_MEMBLOCK_PHYS_MAP
    .physmem.regions    = memblock_physmem_init_regions,
    .physmem.cnt        = 1,
    .physmem.max        = INIT_PHYSMEM_REGIONS,
#endif
    .bottom_up        = false,
    .current_limit        = MEMBLOCK_ALLOC_ANYWHERE,
};
```

`__initdata_memblock`

```
#define __initdata_memblock __meminitdata
```

`__meminit_data`

```
#define __meminitdata    __section(.meminit.data)
```

`.meminit.data` `_rgn` `memblock_dbg` `memblock=debug`

```
memblock_add_range(_rgn, base, size, nid, flags);
```

`.meminit.data` `_rgn` `&memblock.reserved` BIOS `_rgn`

```
if (type->regions[0].size == 0) {
    WARN_ON(type->cnt != 1 || type->total_size);
    type->regions[0].base = base;
    type->regions[0].size = size;
    type->regions[0].flags = flags;
    memblock_set_region_node(&type->regions[0], nid);
    type->total_size = size;
    return 0;
}
```

`memblock_set_region_node`

-
- NUMAID

`memblock_region`

```
struct memblock_region {
    phys_addr_t base;
    phys_addr_t size;
    unsigned long flags;
#ifdef CONFIG_HAVE_MEMBLOCK_NODE_MAP
    int nid;
#endif
};
```

NUMAID `MAX_NUMNODES` include/linux/numa.h

```
#define MAX_NUMNODES    (1 << NODES_SHIFT)
```

`NODES_SHIFT` `CONFIG_NODES_SHIFT`

```
#ifdef CONFIG_NODES_SHIFT
  #define NODES_SHIFT     CONFIG_NODES_SHIFT
#else
  #define NODES_SHIFT     0
#endif
```

`memblick_set_region_node` `memblock_region` `nid`

```
static inline void memblock_set_region_node(struct memblock_region *r, int nid)
{
        r->nid = nid;
}
```

`.meminit.data` BIOS `memblock reserve_ebda_region` arch/x86/kernel/head64.c

`x86_64_start_reservations` init/main.c

```
start_kernel()
```

—— `start_kernel` `init`

twitter 0xAX issue

- BIOS data area
- What is in the extended BIOS data area on a PC?
- Previous part

. Part 4.

Kernel entry point

- `init/main.c` `start_kernel` . `start_kernel` arch/ `start_kernel` 86

`start_kernel` (1) `start_kernel` ,IDcgroupsCPU VFS Cachercu,vmalloc,scheduler(),IRQs( ),ACPI()

: `Linux Kernel initialization process`

`__attribute__`

`start_kernel` init/main.c. `__init` GCC `__attribute__`

```
#define __init      __section(.init.text) __cold notrace
```

`free_initmem` sections() `__init` `__cold` `notrace` `cold` `notrace`

```
#define notrace __attribute__((no_instrument_function))
```

`no_instrument_function` ()

`start_kernel` `__visible`

```
#define __visible __attribute__((externally_visible))
```

`externally_visible` / `unusable` include/linux/init.h

start_kernel

start_kernel

```
char *command_line;
char *after_dashes;
```

`parse_args` 'name=value':

```
lockdep_init();
```

`lockdep_init` lock validator. list_head `lockdep_initialized` `1` spinlockmutex .

`set_task_stack_end_magic` `init_task` `STACK_END_MAGIC` (`0x57AC6E9D`) `init_task` ():

```
struct task_struct init_task = INIT_TASK(init_task);
```

`task_struct` include/linux/sched.h `task_sreuct` `100` `task_struct` Linux

`init_task` `INIT_TASK` include/linux/init_task.h(0)

- zero `runnable` . CPU;

- - `PF_KTHREAD` -;
- ;
- ;
- `&init_thread_info` - `init_thread_union.thread_info` `initthread_union` - `thread_union` `thread_info`:

```
union thread_union {
    struct thread_info thread_info;
    unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

`x86_64` CPU16KB or 4stack `unsigned long thread_union thread_union`

```
struct thread_info {
        struct task_struct      *task;
        struct exec_domain      *exec_domain;
        __u32                   flags;
        __u32                   status;
        __u32                   cpu;
        int                     saved_preempt_count;
        mm_segment_t            addr_limit;
        struct restart_block    restart_block;
        void __user             *sysenter_return;
        unsigned int            sig_on_uaccess_error:1;
        unsigned int            uaccess_err:1;
};
```

52 `thread_info` `X86_64 thread_union.thread_info` 16KB `thread_info` `16 kilobytes - 62 bytes = 16332 bytes`. `thread_union` union:

```
+-----------------------+
|                       |
|                       |
|        stack          |
|                       |
|_______________________|
|           |           |
|           |           |
|           |           |
|___________|___________|             +--------------------+
|                       |             |                    |
|      thread_info      |<----------->|     task_struct    |
|                       |             |                    |
+-----------------------+             +--------------------+
```

http://www.quora.com/In-Linux-kernel-Why-thread_info-structure-and-the-kernel-stack-of-a-process-binds-in-union-construct

`INIT_TASK task_struct's '` `INIT_TASK`

`set_task_stack_end_magic` kernel/fork.ccanary `init`

```
void set_task_stack_end_magic(struct task_struct *tsk)
{
    unsigned long *stackend;
    stackend = end_of_stack(tsk);
    *stackend = STACK_END_MAGIC; /* for overflow detection */
}
```

`set_task_stack_end_magic end_of_stack` `task_struct` `CONFIG_STACK_GROWSUP` x86

```
(unsigned long *)(task_thread_info(p) + 1);
```

`task_thread_info`

```
#define task_thread_info(task)  ((struct thread_info *)(task)->stack)
```

`STACK_END_MAGIC` `canary`

```
if (*end_of_stack(task) != STACK_END_MAGIC) {
        //
        // handle stack overflow here
        //
}
```

`set_task_stack_end_magic` `smp_setup_processor_id` . `x86_64`

```
void __init __weak smp_setup_processor_id(void)
{
}
```

s390 and arm64.

`debug_objects_early_init` `lockdep_init`

`debug_object_early_init` `boot_init_stack_canary` `task_struct->canary` GCC `CONFIG_CC_STACKPROTECTOR` `boot_init_stack_canary` TSC:

```
get_random_bytes(&canary, sizeof(canary));
tsc = __native_read_tsc();
canary += tsc + (tsc << 32UL);
```

, `stack_canary` `task_struct`

```
current->stack_canary = canary;
```

IRQ:

```
this_cpu_write(irq_stack_union.stack_canary, canary); // read below about this_cpu_write
```

IRQ, IRQs.canary, bootstrap CPUCPU maps. (interrupts for current CPU) `local_irq_disable` `arch_local_irq_disable` include/linux/percpu-defs.h:

```
static inline notrace void arch_local_irq_enable(void)
{
        native_irq_enable();
}
```

`native_irq_enable` `cli` `X86_64` Where `native_irq_enable` is `cli` instruction for `x86_64` .()CPU IDCPU bitmap

CPU

`start_kernel` `boot_cpu_init` CPUID

```
int cpu = smp_processor_id();
```

0. `CONFIG_DEBUG_PREEMPT` `smp_processor_id` `raw_smp_processor_id` :

```
#define raw_smp_processor_id() (this_cpu_read(cpu_number))
```

`this_cpu_read` (`this_cpu_write` , `this_cpu_add` ...) include/linux/percpu-defs.h `this_cpu` . cpu per-cpu .
`this_cpu_read` :

```
__pcpu_size_call_return(this_cpu_read_, pcp)
```

cpu `cpu_number` `this_cpu_read` `raw_smp_processor_id` `__pcpu_size_call_return`

```
#define __pcpu_size_call_return(stem, variable)                         \
({                                                                      \
        typeof(variable) pscr_ret__;                                    \
        __verify_pcpu_ptr(&(variable));                                 \
        switch(sizeof(variable)) {                                      \
        case 1: pscr_ret__ = stem##1(variable); break;                  \
        case 2: pscr_ret__ = stem##2(variable); break;                  \
        case 4: pscr_ret__ = stem##4(variable); break;                  \
        case 8: pscr_ret__ = stem##8(variable); break;                  \
        default:                                                        \
                __bad_size_call_parameter(); break;                     \
        }                                                               \
        pscr_ret__;                                                     \
})
```

`pscr_ret__` `int` int `common_cpu` cpu(per-cpu):

```
DECLARE_PER_CPU_READ_MOSTLY(int, cpu_number);
```

`__verify_pcpu_ptr` cpu `cpu_number` `pscr_ret__` `common_cpu` `int` ,4 `this_cpu_read_4(common_cpu)` cpu `pscr_ret__`
`__pcpu_size_call_return` __pcpu_size_call_return

```
#define this_cpu_read_4(pcp)       percpu_from_op("mov", pcp)
```

`percpu_from_op` `mov` cpu `percpu_from_op`

```
asm("movl %%gs:%1,%0" : "=r" (pfo_ret__) : "m" (common_cpu))
```

`gs` CPU `mov` copy `common_cpu`

```
this_cpu_read(common_cpu)
```

:

```
movl %gs:$common_cpu, $pfo_ret__
```

CPU, - CPU `zero` `smp_processor_id` .

IDCPU, `boot_cpu_init` CPU, , :

```
set_cpu_online(cpu, true);
set_cpu_active(cpu, true);
set_cpu_present(cpu, true);
set_cpu_possible(cpu, true);
```

CPU- CPU `cpumask` . `cpu_possible` CPUCPU ID. `cpu_present` CPU. `cpu_online` CPU `cpu_present` CPU. CPU
`CONFIG_HOTPLUG_CPU` `possible == present` `active == online` `true` `cpumask_set_cpu` or `cpumask_clear_cpu`

true

```
cpumask_set_cpu(cpu, to_cpumask(cpu_possible_bits));
```

`to_cpumask` `struct cpumask *` CPUCPU'sCPU1bitCPU `cpu_mask`:

```
typedef struct cpumask { DECLARE_BITMAP(bits, NR_CPUS); } cpumask_t;
```

```
#define DECLARE_BITMAP(name, bits) unsigned long name[BITS_TO_LONGS(bits)]
```

`DECLARE_BITMAP` `unsigned long` `to_cpumask`:

```
#define to_cpumask(bitmap)                                              \
        ((struct cpumask *)(1 ? (bitmap)                                \
                            : (void *)sizeof(__check_is_bitmap(bitmap))))
```

, `__check_is_bitmap` `__check_is_bitmap`

```
static inline int __check_is_bitmap(const unsigned long *bitmap)
{
        return 1;
}
```

1 `bitmap` `bitmap` `unsigned long *`, `to_cpumask` `unsigned long` `struct cpumask *` `cpumask_set_cpu` `set_bit` CPU `set_cpu_*`

`set_cpu_*` `cpumask` cpumask or documentation.

CPUstart_kernel `page_address_init`,

Linux

pr_notice

```
#define pr_notice(fmt, ...) \
    printk(KERN_NOTICE pr_fmt(fmt), ##__VA_ARGS__)
```

pr_noticeprintkLinux banner

```
pr_notice("%s", linux_banner);
```

:

```
Linux version 4.0.0-rc6+ (alex@localhost) (gcc version 4.9.1 (Ubuntu 4.9.1-16ubuntu6) ) #319 SMP
```

Linux `setup_arch` `start_kernel` `arch/` `setup_arch` arch/x86/kernel/setup.c -

`_text` `_data` `_text` `_bss_stop`(arch/x86/kernel/head_64.S) `memblock`

```
memblock_reserve(__pa_symbol(_text), (unsigned long)__bss_stop - (unsigned long)_text);
```

`memblock` Linux kernel memory management Part 1. `memblock_reserve`

- base physical address of a memory block;
- size of a memory block.

`__pa_symbol _text`

```
#define __pa_symbol(x) \
    __phys_addr_symbol(__phys_reloc_hide((unsigned long)(x)))
```

`__phys_reloc_hide` `__phys_reloc_hide x86_64` `__phys_addr_symbol` `_text`

```
#define __phys_addr_symbol(x) \
 ((unsigned long)(x) - __START_KERNEL_map + phys_base)
```

`memblock_reserve`

initrd

textdatainitrd,initrd `early_reserve_initrd` RAM DISKRAM DISKRAM DISK

```
u64 ramdisk_image = get_ramdisk_image();
u64 ramdisk_size  = get_ramdisk_size();
u64 ramdisk_end   = PAGE_ALIGN(ramdisk_image + ramdisk_size);
```

Linux Kernel Booting Process `boot_params` `boot_params` bootRAM DISK

```
Field name:    ramdisk_image
Type:        write (obligatory)
Offset/size:    0x218/4
Protocol:    2.00+

  The 32-bit linear address of the initial ramdisk or ramfs.  Leave at
  zero if there is no initial ramdisk/ramfs.
```

`boot_params` . `get_ramdisk_image` :

```
static u64 __init get_ramdisk_image(void)
{
        u64 ramdisk_image = boot_params.hdr.ramdisk_image;

        ramdisk_image |= (u64)boot_params.ext_ramdisk_image << 32;

        return ramdisk_image;
}
```

32ramdisk Documentation/x86/zero-page.txt:

```
0C0/004    ALL    ext_ramdisk_image ramdisk_image high 32bits
```

3264ramdiskbootloader ramdisk

```
if (!boot_params.hdr.type_of_loader ||
    !ramdisk_image || !ramdisk_size)
    return;
```

ramdisk

```
memblock_reserve(ramdisk_image, ramdisk_end - ramdisk_image);
```

`start_kernel` `setup_arch`

twitter

PR **linux-insides-zh**.

- GCC function attributes
- this_cpu operations
- cpumask
- lock validator
- cgroups
- stack buffer overflow
- IRQs
- initrd
- Previous part

setup_arch initrd olpc_ofw_detect One Laptop Per Child support
early_trap_init #DB - TF rflags int3 #BP x86 INT INT0 INT3
INT3 #BP

```
-----------------------------------------------------------------------------------------
|Vector|Mnemonic|Description         |Type |Error Code|Source                           |
-----------------------------------------------------------------------------------------
|3     | #BP    |Breakpoint          |Trap |NO        |INT 3                            |
-----------------------------------------------------------------------------------------
```

#DB early_trap_init arch/x86/kernel/traps.c #DB #BP IDT

```
void __init early_trap_init(void)
{
        set_intr_gate_ist(X86_TRAP_DB, &debug, DEBUG_STACK);
        set_system_intr_gate_ist(X86_TRAP_BP, &int3, DEBUG_STACK);
        load_idt(&idt_descr);
}
```

set_intr_gate set_intr_gate_ist set_system_intr_gate_ist

-
- /
- Interrupt Stack Table IST TSS x86_64 16kb CPU linux -
  Kernel stacks x86_64 Interrupt Stack Table CPU 7 IST DEBUG_STACK

set_intr_gate_ist set_system_intr_gate_ist set_intr_gate _set_gate

```
BUG_ON((unsigned)n > 0xFF);
_set_gate(n, GATE_INTERRUPT, addr, 0, ist, __KERNEL_CS);
```

set_intr_gate dpl ist 0 _set_gate set_intr_gate_ist set_system_intr_gate_ist ist
DEBUG_STACK set_system_intr_gate_ist dpl 0x3 IST cpu_init

#DB #BP idt_descr load_idt ldtr IDT linux debug

#DB

set_intr_gate_ist &debug #DB lxr.free-electorns.com linux debug
arch/x86/include/asm/traps.h debug

```
asmlinkage void debug(void);
```

asmlinkage debug assembly :) #DB arch/x86/kernel/entry_64.S idtentry

```
idtentry debug do_debug has_error_code=0 paranoid=1 shift_ist=DEBUG_STACK
```

idtentry /

-

-
-
- paranoid - 1
- shift_ist -

`idtentry` `ENTRY` `debug` `idtentry` `#DB` `INTR_FRAME` `XCPT_FRAM` `XCPT_FRAME` `INTR_FRAME` `CFI` `CFI` CFI arch/x86/kernel/entry_64.S `CFI` `dwarf2`

```
.macro idtentry sym do_sym has_error_code:req paranoid=0 shift_ist=-1
ENTRY(\sym)
    /* Sanity check */
    .if \shift_ist != -1 && \paranoid == 0
    .error "using shift_ist requires paranoid=1"
    .endif

    .if \has_error_code
    XCPT_FRAME
    .else
    INTR_FRAME
    .endif
    ...
    ...
    ...
```

/

```
    +----------------------+
    |                      |
+40 |         SS           |
+32 |         RSP          |
+24 |        RFLAGS        |
+16 |         CS           |
+8  |         RIP          |
 0  |       Error Code     | <---- rsp
    |                      |
    +----------------------+
```

`idtentry`

```
    ASM_CLAC
    PARAVIRT_ADJUST_EXCEPTION_FRAME
```

`ASM_CLAC` `CONFIG_X86_SMAP` `PARAVIRT_EXCEPTION_FRAME` `Xen` `$-1` (`x86_64` `0xffffffffffffffff`)

```
    .ifeq \has_error_code
    pushq_cfi $-1
    .endif
```

`dummy` `$ORIG_RAX-R15`

```
    subq $ORIG_RAX-R15, %rsp
```

`ORIG_RAX R15` arch/x86/include/asm/calling.h `ORIG_RAX-R15` 120 120

```
testl $3, CS(%rsp)
jnz 1f
```

`CS` `CS` `RPL` 0-3 `save_paranoid` `1` `save_paranoid` `gs` `gs`

```
movl $1,%ebx
    movl $MSR_GS_BASE,%ecx
    rdmsr
    testl %edx,%edx
    js 1f
    SWAPGS
    xorl %ebx,%ebx
1:    ret
```

`pt_regs` `rdi` `rsi` arch/x86/kernel/trap.c `do_debug` `do_debug`

- pt_regs - CPU
- error code -

`paranoid_exit` `iret` `#DB` `idtentry` `idtentry` `early_trap_init` `early_cpu_init` arch/x86/kernel/cpu/common.c `CPU`

ioremap

`ioremap`

- I/O
-

linux `outb/inb` `I/O` `CPU` `I/O` `RAM` `ioremap`

`early_ioremap_init` `ioremap` `I/O` `ioremap` `I/O` `ioremap` arch/x86/mm/ioremap.c `early_ioremap_init` `pmd_t` `pmd` `typedef struct {pmdval_t pmd; } pmd_t;` `pmdval_t` `fixmap`

```
pmd_t *pmd;
BUILD_BUG_ON((fix_to_virt(0) + PAGE_SIZE) & ((1 << PMD_SHIFT) - 1));
```

`fixmap` - `FIXADDR_START` `FIXADDR_TOP` `early_ioremap_init` mm/early_ioremap.c `early_ioremap_setup` `early_ioremap_setup` 512 `slot_virt`

```
for (i = 0; i < FIX_BTMAPS_SLOTS; i++)
    slot_virt[i] = __fix_to_virt(FIX_BTMAP_BEGIN - NR_FIX_BTMAPS*i);
```

`FIX_BTMAP_BEGIN` `pmd` `bm_pte` 0 `pmd_populate_kernel`

```
pmd = early_ioremap_pmd(fix_to_virt(FIX_BTMAP_BEGIN));
memset(bm_pte, 0, sizeof(bm_pte));
pmd_populate_kernel(&init_mm, pmd, bm_pte);
```

`ioremap` `fixmaps`

`ioremap`

```
ROOT_DEV = old_decode_dev(boot_params.hdr.root_dev);
```

`initrd` `do_mount_root` `old_decode_dev` `boot_params_structure` x86 linux

```
Field name:    root_dev
Type:        modify (optional)
```

```
Offset/size:    0x1fc/2
Protocol:    ALL

  The default root device device number.  The use of this field is
  deprecated, use the "root=" option on the command line instead
```

`old_decode_dev` `MKDEV` `dev_t`

```
static inline dev_t old_decode_dev(u16 val)
{
         return MKDEV((val >> 8) & 255, val & 255);
}
```

`dev_t` / `old` 2 8 bit 8 bit 256 256 32 bit 12 20
`new_decode_dev`

```
static inline dev_t new_decode_dev(u32 dev)
{
         unsigned major = (dev & 0xfff00) >> 8;
         unsigned minor = (dev & 0xff) | ((dev >> 12) & 0xfff00);
         return MKDEV(major, minor);
}
```

`dev` `0xffffffff` 12 `0xfff` 20 `0xfffff` `old_decode_dev` `ROOT_DEV`

Memory Map

`setup_memory_map`

```
screen_info = boot_params.screen_info;
    edid_info = boot_params.edid_info;
    saved_video_mode = boot_params.hdr.vid_mode;
    bootloader_type = boot_params.hdr.type_of_loader;
    if ((bootloader_type >> 4) == 0xe) {
        bootloader_type &= 0xf;
        bootloader_type |= (boot_params.hdr.ext_loader_type+0x10) << 4;
    }
    bootloader_version  = bootloader_type & 0xf;
    bootloader_version |= boot_params.hdr.ext_loader_ver << 4;
```

`boot_params` `I/O` `I/O` `/proc/ioports` `/proc/iomem`

- /proc/ioports -
- /proc/iomem - `/proc/iomem`

```
cat /proc/iomem
00000000-00000fff : reserved
00001000-0009d7ff : System RAM
0009d800-0009ffff : reserved
000a0000-000bffff : PCI Bus 0000:00
000c0000-000cffff : Video ROM
000d0000-000d3fff : PCI Bus 0000:00
000d4000-000d7fff : PCI Bus 0000:00
000d8000-000dbfff : PCI Bus 0000:00
000dc000-000dffff : PCI Bus 0000:00
000e0000-000fffff : reserved
  000e0000-000e3fff : PCI Bus 0000:00
  000e4000-000e7fff : PCI Bus 0000:00
  000f0000-000fffff : System ROM
```

linux API PICs I/O `resource`

```
struct resource {
        resource_size_t start;
        resource_size_t end;
        const char *name;
        unsigned long flags;
        struct resource *parent, *sibling, *child;
};
```

`start` `end` `resource_size_t` `phys_addr_t` `x86_64` `u64` `/proc/iomem`

include/linux/ioport.h

```
+-------------+      +-------------+
|             |      |             |
|    parent   |------|    sibling  |
|             |      |             |
+-------------+      +-------------+
       |
       |
+-------------+
|             |
|    child    |
|             |
+-------------+
```

`iomem` `iomem_resource`

```
struct resource iomem_resource = {
        .name   = "PCI mem",
        .start  = 0,
        .end    = -1,
        .flags  = IORESOURCE_MEM,
};
EXPORT_SYMBOL(iomem_resource);
TODO EXPORT_SYMBOL
```

`iomem_resource` `PCI mem` `IORESOURCE_MEM (0x00000200)` `io` `iomem`

```
iomem_resource.end = (1ULL << boot_cpu_data.x86_phys_bits) - 1;
```

1 `boot_cpu_data.x86_phys_bits` `boot_cpu_data` `early_cpu_init` `cpuinfo_x86` `x86_phys_bits` `iomem_resource` `EXPORT_SYMBOL` `iomem_resource` `iomem` `setup_memory_map`

```
void __init setup_memory_map(void)
{
        char *who;

        who = x86_init.resources.memory_setup();
        memcpy(&e820_saved, &e820, sizeof(struct e820map));
        printk(KERN_INFO "e820: BIOS-provided physical RAM map:\n");
        e820_print_map(who);
}
```

`x86_init.resources.memory_setup` `x86_init` `x86_init_ops` `pci` `x86_init`

arch/x86/kernel/x86_init.c

```
struct x86_init_ops x86_init __initdata = {
    .resources = {
            .probe_roms             = probe_roms,
            .reserve_resources      = reserve_standard_io_resources,
            .memory_setup           = default_machine_specific_memory_setup,
    },
    ...
```

```
    ...
    ...
}
```

`memory_setup` `default_machine_specific_memory_setup` e820 `e820map` `printk` `dmesg`

```
[    0.000000] e820: BIOS-provided physical RAM map:
[    0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009d7ff] usable
[    0.000000] BIOS-e820: [mem 0x000000000009d800-0x000000000009ffff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000000e0000-0x00000000000fffff] reserved
[    0.000000] BIOS-e820: [mem 0x0000000000100000-0x00000000be825fff] usable
[    0.000000] BIOS-e820: [mem 0x00000000be826000-0x00000000be82cfff] ACPI NVS
[    0.000000] BIOS-e820: [mem 0x00000000be82d000-0x00000000bf744fff] usable
[    0.000000] BIOS-e820: [mem 0x00000000bf745000-0x00000000bfff4fff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000bfff5000-0x00000000dc041fff] usable
[    0.000000] BIOS-e820: [mem 0x00000000dc042000-0x00000000dc0d2fff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000dc0d3000-0x00000000dc138fff] usable
[    0.000000] BIOS-e820: [mem 0x00000000dc139000-0x00000000dc27dfff] ACPI NVS
[    0.000000] BIOS-e820: [mem 0x00000000dc27e000-0x00000000deffefff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000defff000-0x00000000dfffffff] usable
...
...
...
```

BIOS

`parse_setup_data` `setup_data` `BIOS` `EDD` `setup_data` `x86`

```
Field name:    setup_data
Type:        write (special)
Offset/size:    0x250/8
Protocol:    2.09+

  The 64-bit physical pointer to NULL terminated single linked list of
  struct setup_data. This is used to define a more extensible boot
  parameters passing mechanism.
```

`blob` `EFI` `boot_params` arch/x86/boot/edd.c `BIOS` `EDD` `edd`

```
static inline void __init copy_edd(void)
{
     memcpy(edd.mbr_signature, boot_params.edd_mbr_sig_buffer,
            sizeof(edd.mbr_signature));
     memcpy(edd.edd_info, boot_params.eddbuf, sizeof(edd.edd_info));
     edd.mbr_signature_nr = boot_params.edd_mbr_sig_buf_entries;
     edd.edd_info_nr = boot_params.eddbuf_entries;
}
```

`memory descriptor` linux `mm_struct` `mm_struct` / `brk` include/linux/mm_types.h `task_struct` `mm` `active_mm` `init` `INIT_TASK` `task_struct`

```
#define INIT_TASK(tsk)  \
{
    ...
    ...
    ...
    .mm = NULL,         \
    .active_mm  = &init_mm, \
```

```
    ...
}
```

`mm` `active_mm` `brk`

```
    init_mm.start_code = (unsigned long) _text;
    init_mm.end_code = (unsigned long) _etext;
    init_mm.end_data = (unsigned long) _edata;
    init_mm.brk = _brk_end;
```

`init_mm`

```
struct mm_struct init_mm = {
    .mm_rb          = RB_ROOT,
    .pgd            = swapper_pg_dir,
    .mm_users       = ATOMIC_INIT(2),
    .mm_count       = ATOMIC_INIT(1),
    .mmap_sem       = __RWSEM_INITIALIZER(init_mm.mmap_sem),
    .page_table_lock =  __SPIN_LOCK_UNLOCKED(init_mm.page_table_lock),
    .mmlist         = LIST_HEAD_INIT(init_mm.mmlist),
    INIT_MM_CONTEXT(init_mm)
};
```

`mm_rb` `pgd` `mm_user` `mm_count` `mmap_sem` `mpx_mm_init` `Intel` // `bss`

```
code_resource.start = __pa_symbol(_text);
    code_resource.end = __pa_symbol(_etext)-1;
    data_resource.start = __pa_symbol(_etext);
    data_resource.end = __pa_symbol(_edata)-1;
    bss_resource.start = __pa_symbol(__bss_start);
    bss_resource.end = __pa_symbol(__bss_stop)-1;
```

`resource` // `bss` `/proc/iomem`

```
00100000-be825fff : System RAM
  01000000-015bb392 : Kernel code
  015bb393-01930c3f : Kernel data
  01a11000-01ac3fff : Kernel bss
 [arch/x86/kernel/setup.c](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/arch/x86/k
ernel/setup.c)
static struct resource code_resource = {
    .name    = "Kernel code",
    .start    = 0,
    .end    = 0,
    .flags    = IORESOURCE_BUSY | IORESOURCE_MEM
};
```

`NX` `NX-bit` `no-execute` 63 `EFER.NXE` 1 `no-execute` / `x86_configure_nx` `CPU` `NX-bit` `_supported_pte_mask`

```
void x86_configure_nx(void)
{
        if (cpu_has_nx && !disable_nx)
                __supported_pte_mask |= _PAGE_NX;
        else
                __supported_pte_mask &= ~_PAGE_NX;
}
```

linux `setup_arch` `setup_arch`
`pci`

twitter

PR **linux-insides**

- mm vs active_mm
- e820
- Supervisor mode access prevention
- Kernel stacks
- TSS
- IDT
- Memory mapped I/O
- CFI directives
- PDF. dwarf4 specification
- Call stack
- . Part 4.

arch/x86/kernel/setup.c(`x86_64`) `x86_configure_nx` NX bit `_PAGE_NX` , `setup_arch` `start_kernel` `x86_configure_nx` `parse_early_param` init/main.c (Documentation/kernel-parameters.txt) `earlyprintk` arch/x86/boot/cmdline.c `cmdline_find_option` `__cmdline_find_option` , `__cmdline_find_option_bool` linux

```
early_param("gbpages", parse_direct_gbpages_on);
```

`early_param` :

-
-

:

```
#define early_param(str, fn) \
        __setup_param(str, fn, fn, 1)
```

include/linux/init.h .
`early_param` `__setup_param` :

```
#define __setup_param(str, unique_id, fn, early)                \
        static const char __setup_str_##unique_id[] __initconst \
                __aligned(1) = str; \
        static struct obs_kernel_param __setup_##unique_id      \
                __used __section(.init.setup)                   \
                __attribute__((aligned((sizeof(long)))))        \
                = { __setup_str_##unique_id, fn, early }
```

`__setup_str_*_id` (`*`) `obs_kernel_param` `__setup_ *`

`obs_kernel_param` :

```
struct obs_kernel_param {
        const char *str;
        int (*setup_func)(char *);
        int early;
};
```

:

-
-
- early

`__set_param` `__section(.init.setup)` `__setup_str_ *` `.init.setup` include/asm-generic/vmlinux.lds.h `.init.setup` `__setup_start` `__setup_end` :

```
#define INIT_SETUP(initsetup_align)                \
                . = ALIGN(initsetup_align);        \
                VMLINUX_SYMBOL(__setup_start) = .; \
                *(.init.setup)                     \
                VMLINUX_SYMBOL(__setup_end) = .;
```

`parse_early_param` :

```
void __init parse_early_param(void)
{
        static int done __initdata;
        static char tmp_cmdline[COMMAND_LINE_SIZE] __initdata;

        if (done)
                return;

        /* All fall through to do_early_param. */
        strlcpy(tmp_cmdline, boot_command_line, COMMAND_LINE_SIZE);
        parse_early_options(tmp_cmdline);
        done = 1;
}
```

`parse_early_param` `done` `parse_early_param` `boot_command_line` (`tmp_cmdline`) `main.c` `parse_early_options` `parse_early_options` kernel/params.c `parse_args` , `parse_args` `do_early_param` do_early_param `__setup_start` `__setup_end` `obs_kernel_param` `early` 1 , `obs_kernel_param` `setup_func` `parse_early_param` `x86_report_nx` `x86_configure_nx` `NX-bit` arch/x86/mm/setup_nx.c `x86_report_nx` `NX` `x86_report_nx` `x86_configure_nx` `parse_early_param` : `noexec` `parse_early_param` `noexec` `x86_report_nx` :

```
noexec        [X86]
            On X86-32 available only on PAE configured kernels.
            //X86-32PAE
            noexec=on: enable non-executable mappings (default)
            //noexec=on:()
            noexec=off: disable non-executable mappings
            //noexec=off:
```

:

```
bootconsole [earlyser0] enabled
NX (Execute Disable) protection: active
SMBIOS 2.8 present.
```

:

```
    memblock_x86_reserve_range_setup_data();
```

arch/x86/kernel/setup.c `setup_data` (`setup_data` Linux kernel memory management `ioremap` and `memblock`)

:

```
    if (acpi_mps_check()) {
#ifdef CONFIG_X86_LOCAL_APIC
        disable_apic = 1;
#endif
        setup_clear_cpu_cap(X86_FEATURE_APIC);
    }
```

`acpi_mps_check` arch/x86/kernel/acpi/boot.c `CONFIG_X86_LOCAL_APIC` `CONFIG_x86_MPPARSE` :

```
int __init acpi_mps_check(void)
{
#if defined(CONFIG_X86_LOCAL_APIC) && !defined(CONFIG_X86_MPPARSE)
        /* mptable code is not built-in*/
        if (acpi_disabled || acpi_noirq) {
                printk(KERN_WARNING "MPS support code is not built-in.\n"
                       "Using acpi=off or acpi=noirq or pci=noacpi "
                       "may have problem\n");
```

```
                return 1;
        }
#endif
        return 0;
}
```

`acpi_mps_check` `MPS`) `CONFIG_X86_LOCAL_APIC` `CONFIG_x86_MPPAARSE` `acpi=off` `acpi=noirq` `pci=noacpi` `acpi_mps_check` `acpi_mps_check` 1 APIC , `setup_clear_cpu_cap` CPU `X86_FEATURE_APIC` CPU masks CPU mask)

PCI

PCI :

```
#ifdef CONFIG_PCI
    if (pci_early_dump_regs)
        early_dump_pci_devices();
#endif
```

`pci_early_dump_regs` arch/x86/pci/common.c `pci=earlydump` drivers/pci/pci.c :

```
early_param("pci", pci_setup);
```

`pci_setup` `pci=` drivers/pci/pci.c `_weak` `pcibios_setup` `_weak` `x86_64` arch/x86/pci/common.c :

```
char *__init pcibios_setup(char *str) {
        ...
        ...
        ...
        } else if (!strcmp(str, "earlydump")) {
                pci_early_dump_regs = 1;
                return NULL;
        }
        ...
        ...
        ...
}
```

`CONFIG_PCI` `pci=earlydump` arch/x86/pci/early.c `early_dump_pci_devices` pci `noearly` :

```
if (!early_pci_allowed())
        return;
```

PCI `256` 32:

```
for (bus = 0; bus < 256; bus++) {
                for (slot = 0; slot < 32; slot++) {
                        for (func = 0; func < 8; func++) {
                        ...
                        ...
                        ...
                        }
                }
}
```

`read_pci_config` `pci`

pci `pci` `Drivers/PCI`

`early_dump_pci_devices` e820 e820

```
    /* update the e820_saved too */
    e820_reserve_setup_data();
    finish_e820_parsing();
    ...
    ...
    ...
    e820_add_kernel_range();
    trim_bios_range(void);
    max_pfn = e820_end_of_ram_pfn();
    early_reserve_e820_mpc_new();
```

`e820_reserve_setup_data` `memblock_x86_reserve_range_setup_data` `e820_update_range` `e820map` `E820_RESERVED_KERN` `finish_e820_parsing`, `sanitize_e820_map` `e820map` e820 `e820_add_kernel_range` :

```
u64 start = __pa_symbol(_text);
u64 size = __pa_symbol(_end) - start;
```

`e820map` `E820RAM` `.text` `.data` `.bss` `trm_bios_range` `e820Map` 4096 `E820_RESERVED` `sanitize_e820_map` `e820map` `e820_end_of_ram_pfn` - `e820_end_of_ram_pfn` `e820_end_pfn` :

```
unsigned long __init e820_end_of_ram_pfn(void)
{
    return e820_end_pfn(MAX_ARCH_PFN);
}
```

`e820_end_pfn` (`x86_64` `MAX_ARCH_PFN` `0x400000000`) `e820_end_pfn` `e820` `e820` `E820_RAM` `E820_PRAM` `e820` :

```
for (i = 0; i < e820.nr_map; i++) {
        struct e820entry *ei = &e820.map[i];
        unsigned long start_pfn;
        unsigned long end_pfn;

        if (ei->type != E820_RAM && ei->type != E820_PRAM)
            continue;

        start_pfn = ei->addr >> PAGE_SHIFT;
        end_pfn = (ei->addr + ei->size) >> PAGE_SHIFT;

        if (start_pfn >= limit_pfn)
            continue;
        if (end_pfn > limit_pfn) {
            last_pfn = limit_pfn;
            break;
        }
        if (end_pfn > last_pfn)
            last_pfn = end_pfn;
}
```

```
    if (last_pfn > max_arch_pfn)
        last_pfn = max_arch_pfn;

    printk(KERN_INFO "e820: last_pfn = %#lx max_arch_pfn = %#lx\n",
             last_pfn, max_arch_pfn);
    return last_pfn;
```

`last_pfn` `last_pfn` (`x86_64`), `last_pfn` `dmesg` `last_pfn` :

```
...
[    0.000000] e820: last_pfn = 0x41f000 max_arch_pfn = 0x400000000
...
```

`max_low_pfn` , 4GB4GBRAM `max_low_pfn` `e820_end_of_low_ram_pfn` `e820_end_of_ram_pfn` 4GB `max_low_pfn` `max_pfn` :

```
if (max_pfn > (1UL<<(32 - PAGE_SHIFT)))
    max_low_pfn = e820_end_of_low_ram_pfn();
else
    max_low_pfn = max_pfn;

high_memory = (void *)__va(max_pfn * PAGE_SIZE - 1) + 1;
```

`__va` ()，

`e820` :

```
dmi_scan_machine();
dmi_memdev_walk();
```

drivers/firmware/dmi_scan.c `dmi_scan_machine` System Management BIOS `SMBIOS` : EFI `SMBIOS` `0xF0000` `0x10000` `dmi_scan_machine` `dmi_early_remap` `0xf0000` `0x10000` `early_ioremap` :

```
void __init dmi_scan_machine(void)
{
    char __iomem *p, *q;
    char buf[32];
    ...
    ...
    ...
    p = dmi_early_remap(0xF0000, 0x10000);
    if (p == NULL)
            goto error;
```

`DMI` `_SM_` :

```
memset(buf, 0, 16);
for (q = p; q < p + 0x10000; q += 16) {
        memcpy_fromio(buf + 16, q, 16);
        if (!dmi_smbios3_present(buf) || !dmi_present(buf)) {
            dmi_available = 1;
            dmi_early_unmap(p, 0x10000);
            goto out;
        }
        memcpy(buf, buf + 16, 16);
}
```

`_SM_` `000F0000h` `0x000FFFFF` `memcpy_fromio` `buf` 16 `memcpy` (`buf`) `dmi_smbios3_present` `dmi_present` `buf` 4 `__SM__` `SMBIOS` `_DMI_` `_DMI_` ... `dmesg` :

```
[    0.000000] SMBIOS 2.7 present.
[    0.000000] DMI: Gigabyte Technology Co., Ltd. Z97X-UD5H-BK/Z97X-UD5H-BK, BIOS F6 06/17/2014
```

`dmi_scan_machine`：

```
dmi_early_unmap(p, 0x10000);
```

- `dmi_memdev_walk`：

```
void __init dmi_memdev_walk(void)
{
    if (!dmi_available)
        return;

    if (dmi_walk_early(count_mem_devices) == 0 && dmi_memdev_nr) {
        dmi_memdev = dmi_alloc(sizeof(*dmi_memdev) * dmi_memdev_nr);
        if (dmi_memdev)
            dmi_walk_early(save_mem_devices);
    }
}
```

`DMI`（`dmi_scan_machine` `dmi_available`） `dmi_walk_early` `dmi_alloc`， `dmi_al`

```
#ifdef CONFIG_DMI
RESERVE_BRK(dmi_alloc, 65536);
#endif
```

arch/x86/include/asm/setup.h `RESERVE_BRK` `brk`：

```
init_hypervisor_platform();
x86_init.resources.probe_roms();
insert_resource(&iomem_resource, &code_resource);
insert_resource(&iomem_resource, &data_resource);
insert_resource(&iomem_resource, &bss_resource);
early_gart_iommu_check();
```

(SMP)

SMP `find_smp_config`：

```
static inline void find_smp_config(void)
{
        x86_init.mpparse.find_smp_config();
}
```

`x86_init.mpparse.find_smp_config` arch/x86/kernel/mpparse.c `default_find_smp_config`
`default_find_smp_config` `SMP`：

```
if (smp_scan_config(0x0, 0x400) ||
            smp_scan_config(639 * 0x400, 0x400) ||
            smp_scan_config(0xF0000, 0x10000))
            return;
```

`smp_scan_config`：

```
unsigned int *bp = phys_to_virt(base);
struct mpf_intel *mpf;
```

`SMP` `mpf_intel` `mpf_intel` `mpf_intel`：

```
struct mpf_intel {
        char signature[4];
        unsigned int physptr;
        unsigned char length;
        unsigned char specification;
        unsigned char checksum;
        unsigned char feature1;
        unsigned char feature2;
        unsigned char feature3;
        unsigned char feature4;
        unsigned char feature5;
};
```

- BIOSMPMP `mpf_intel` (), `smp_scan_config` `MP floating pointer structure` `SMP` `mpf->specification` 14(14):

```
while (length > 0) {
if ((*bp == SMP_MAGIC_IDENT) &&
    (mpf->length == 1) &&
    !mpf_checksum((unsigned char *)bp, 16) &&
    ((mpf->specification == 1)
    || (mpf->specification == 4))) {

        mem = virt_to_phys(mpf);
        memblock_reserve(mem, sizeof(*mpf));
        if (mpf->physptr)
            smp_reserve_memory(mpf);
    }
}
```

`memblock_reserve` MultiProcessor Specification `SMP`

`setup_arch` `early_alloc_pgt_buf` , `brk` :

```
void  __init early_alloc_pgt_buf(void)
{
        unsigned long tables = INIT_PGT_BUF_SIZE;
        phys_addr_t base;

        base = __pa(extend_brk(tables, PAGE_SIZE));

        pgt_buf_start = base >> PAGE_SHIFT;
        pgt_buf_end = pgt_buf_start;
        pgt_buf_top = pgt_buf_start + (tables >> PAGE_SHIFT);
}
```

`INIT_PGT_BUF_SIZE` linux 4.0 `(6 * PAGE_SIZE)` `extend_brk` : sizealign, `brk` linux `brk` BSS :

```
    . = ALIGN(PAGE_SIZE);
    .brk : AT(ADDR(.brk) - LOAD_OFFSET) {
        __brk_base = .;
        . += 64 * 1024;         /* 64k alignment slop space */
        *(.brk_reservation)     /* areas brk users have reserved */
        __brk_limit = .;
    }
```

`readelf` :

```
[25] .bss              NOBITS           ffffffff8199d000  00d9d000
     00000000000b4000  0000000000000000  WA       0     0     4096
[26] .brk              NOBITS           ffffffff81a51000  00d9d000
     0000000000026000  0000000000000000  WA       0     0     1
```

`_pa` `brk` `reserve_brk` `brk` :

```
static void __init reserve_brk(void)
{
    if (_brk_end > _brk_start)
        memblock_reserve(__pa_symbol(_brk_start),
                _brk_end - _brk_start);

    _brk_start = 0;
}
```

`reserve_brk` `_brk_start` 0, `brk` `cleanup_highmap` `__START_KERNEL_map` `_end - _text` `level2_kernel_pgt` `_text data` `bss` `clean_high_map` :

```
unsigned long vaddr = __START_KERNEL_map;
unsigned long end = roundup((unsigned long)_end, PMD_SIZE) - 1;
pmd_t *pmd = level2_kernel_pgt;
pmd_t *last_pmd = pmd + PTRS_PER_PMD;
```

, `_text` `end` :

```
for (; pmd < last_pmd; pmd++, vaddr += PMD_SIZE) {
        if (pmd_none(*pmd))
            continue;
        if (vaddr < (unsigned long) _text || vaddr > end)
            set_pmd(pmd, __pmd(0));
}
```

`memblock_set_current_limit` (linux `memblock`) `memblock` `ISA_END_ADDRESS` `0x100000` `memblock_x86_fill` `e820` `memblock` :

```
MEMBLOCK configuration:
 memory size = 0x1fff7ec00 reserved size = 0x1e30000
 memory.cnt  = 0x3
 memory[0x0]    [0x00000000001000-0x0000000009efff], 0x9e000 bytes flags: 0x0
 memory[0x1]    [0x00000000100000-0x000000bffdffff], 0xbfee0000 bytes flags: 0x0
 memory[0x2]    [0x00000100000000-0x0000023fffffff], 0x140000000 bytes flags: 0x0
 reserved.cnt  = 0x3
 reserved[0x0]    [0x0000000009f000-0x000000000fffff], 0x61000 bytes flags: 0x0
 reserved[0x1]    [0x00000001000000-0x00000001a57fff], 0xa58000 bytes flags: 0x0
 reserved[0x2]    [0x0000007ec89000-0x0000007fffffff], 0x1377000 bytes flags: 0x0
```

`memblock_x86_fill` : `early_reserve_e820_mpc_new` `e820map` `reserve_real_mode` - `0x0` 1M(...) `trim_platform_memory_ranges` `0x20050000` , `0x20110000` Sandy Bridge `trim_low_memory_range` `memblock` 4KB `init_mem_mapping` `PAGE_OFFSET` , `early_trap_pf_init` `#PF` (), `setup_real_mode`

`setup_arch` (`early_gart_iommu_check` mtrr ...), `setup_arch` linuxlinux, : , , , linux, , ,

linux `setup_arch` , `setup_arch`

twitter

PR **linux-insides.**

- MultiProcessor Specification
- NX bit
- Documentation/kernel-parameters.txt
- APIC
- CPU masks
- Linux kernel memory management
- PCI
- e820
- System Management BIOS
- EFI
- SMP
- MultiProcessor Specification
- BSS
- SMBIOS specification
-

Kernel initialization. Part 7.

The End of the architecture-specific initialization, almost...

This is the seventh part of the Linux Kernel initialization process which covers insides of the `setup_arch` function from the arch/x86/kernel/setup.c. As you can know from the previous parts, the `setup_arch` function does some architecture-specific (in our case it is x86_64) initialization stuff like reserving memory for kernel code/data/bss, early scanning of the Desktop Management Interface, early dump of the PCI device and many many more. If you have read the previous part, you can remember that we've finished it at the `setup_real_mode` function. In the next step, as we set limit of the memblock to the all mapped pages, we can see the call of the `setup_log_buf` function from the kernel/printk/printk.c.

The `setup_log_buf` function setups kernel cyclic buffer and its length depends on the `CONFIG_LOG_BUF_SHIFT` configuration option. As we can read from the documentation of the `CONFIG_LOG_BUF_SHIFT` it can be between `12` and `21`. In the insides, buffer defined as array of chars:

```
#define __LOG_BUF_LEN (1 << CONFIG_LOG_BUF_SHIFT)
static char __log_buf[__LOG_BUF_LEN] __aligned(LOG_ALIGN);
static char *log_buf = __log_buf;
```

Now let's look on the implementation of the `setup_log_buf` function. It starts with check that current buffer is empty (It must be empty, because we just setup it) and another check that it is early setup. If setup of the kernel log buffer is not early, we call the `log_buf_add_cpu` function which increase size of the buffer for every CPU:

```
if (log_buf != __log_buf)
    return;

if (!early && !new_log_buf_len)
    log_buf_add_cpu();
```

We will not research `log_buf_add_cpu` function, because as you can see in the `setup_arch`, we call `setup_log_buf` as:

```
setup_log_buf(1);
```

where `1` means that it is early setup. In the next step we check `new_log_buf_len` variable which is updated length of the kernel log buffer and allocate new space for the buffer with the `memblock_virt_alloc` function for it, or just return.

As kernel log buffer is ready, the next function is `reserve_initrd`. You can remember that we already called the `early_reserve_initrd` function in the fourth part of the Kernel initialization. Now, as we reconstructed direct memory mapping in the `init_mem_mapping` function, we need to move initrd into directly mapped memory. The `reserve_initrd` function starts from the definition of the base address and end address of the `initrd` and check that `initrd` is provided by a bootloader. All the same as what we saw in the `early_reserve_initrd`. But instead of the reserving place in the `memblock` area with the call of the `memblock_reserve` function, we get the mapped size of the direct memory area and check that the size of the `initrd` is not greater than this area with:

```
mapped_size = memblock_mem_size(max_pfn_mapped);
if (ramdisk_size >= (mapped_size>>1))
    panic("initrd too large to handle, "
          "disabling initrd (%lld needed, %lld available)\n",
          ramdisk_size, mapped_size>>1);
```

You can see here that we call `memblock_mem_size` function and pass the `max_pfn_mapped` to it, where `max_pfn_mapped` contains the highest direct mapped page frame number. If you do not remember what is `page frame number`, explanation is simple: First `12` bits of the virtual address represent offset in the physical page or page frame. If we right-shift out `12` bits of the virtual address, we'll

discard offset part and will get `Page Frame Number`. In the `memblock_mem_size` we go through the all memblock `mem` (not reserved) regions and calculates size of the mapped pages and return it to the `mapped_size` variable (see code above). As we got amount of the direct mapped memory, we check that size of the `initrd` is not greater than mapped pages. If it is greater we just call `panic` which halts the system and prints famous Kernel panic message. In the next step we print information about the `initrd` size. We can see the result of this in the `dmesg` output:

```
[0.000000] RAMDISK: [mem 0x36d20000-0x37687fff]
```

and relocate `initrd` to the direct mapping area with the `relocate_initrd` function. In the start of the `relocate_initrd` function we try to find a free area with the `memblock_find_in_range` function:

```
relocated_ramdisk = memblock_find_in_range(0, PFN_PHYS(max_pfn_mapped), area_size, PAGE_SIZE);

if (!relocated_ramdisk)
    panic("Cannot find place for new RAMDISK of size %lld\n",
          ramdisk_size);
```

The `memblock_find_in_range` function tries to find a free area in a given range, in our case from `0` to the maximum mapped physical address and size must equal to the aligned size of the `initrd`. If we didn't find a area with the given size, we call `panic` again. If all is good, we start to relocated RAM disk to the down of the directly mapped memory in the next step.

In the end of the `reserve_initrd` function, we free memblock memory which occupied by the ramdisk with the call of the:

```
memblock_free(ramdisk_image, ramdisk_end - ramdisk_image);
```

After we relocated `initrd` ramdisk image, the next function is `vsmp_init` from the arch/x86/kernel/vsmp_64.c. This function initializes support of the `ScaleMP vSMP`. As I already wrote in the previous parts, this chapter will not cover non-related `x86_64` initialization parts (for example as the current or `ACPI`, etc.). So we will skip implementation of this for now and will back to it in the part which cover techniques of parallel computing.

The next function is `io_delay_init` from the arch/x86/kernel/io_delay.c. This function allows to override default I/O delay `0x80` port. We already saw I/O delay in the Last preparation before transition into protected mode, now let's look on the `io_delay_init` implementation:

```
void __init io_delay_init(void)
{
    if (!io_delay_override)
        dmi_check_system(io_delay_0xed_port_dmi_table);
}
```

This function check `io_delay_override` variable and overrides I/O delay port if `io_delay_override` is set. We can set `io_delay_override` variably by passing `io_delay` option to the kernel command line. As we can read from the Documentation/kernel-parameters.txt, `io_delay` option is:

```
io_delay=    [X86] I/O delay method
    0x80
        Standard port 0x80 based delay
    0xed
        Alternate port 0xed based delay (needed on some systems)
    udelay
        Simple two microseconds delay
    none
        No delay
```

We can see `io_delay` command line parameter setup with the `early_param` macro in the arch/x86/kernel/io_delay.c

```
early_param("io_delay", io_delay_param);
```

More about `early_param` you can read in the previous part. So the `io_delay_param` function which setups `io_delay_override` variable will be called in the do_early_param function. `io_delay_param` function gets the argument of the `io_delay` kernel command line parameter and sets `io_delay_type` depends on it:

```
static int __init io_delay_param(char *s)
{
        if (!s)
                return -EINVAL;

        if (!strcmp(s, "0x80"))
                io_delay_type = CONFIG_IO_DELAY_TYPE_0X80;
        else if (!strcmp(s, "0xed"))
                io_delay_type = CONFIG_IO_DELAY_TYPE_0XED;
        else if (!strcmp(s, "udelay"))
                io_delay_type = CONFIG_IO_DELAY_TYPE_UDELAY;
        else if (!strcmp(s, "none"))
                io_delay_type = CONFIG_IO_DELAY_TYPE_NONE;
        else
                return -EINVAL;

        io_delay_override = 1;
        return 0;
}
```

The next functions are `acpi_boot_table_init` , `early_acpi_boot_init` and `initmem_init` after the `io_delay_init` , but as I wrote above we will not cover ACPI related stuff in this `Linux Kernel initialization process` chapter.

Allocate area for DMA

In the next step we need to allocate area for the Direct memory access with the `dma_contiguous_reserve` function which is defined in the drivers/base/dma-contiguous.c. `DMA` is a special mode when devices communicate with memory without CPU. Note that we pass one parameter - `max_pfn_mapped << PAGE_SHIFT` , to the `dma_contiguous_reserve` function and as you can understand from this expression, this is limit of the reserved memory. Let's look on the implementation of this function. It starts from the definition of the following variables:

```
phys_addr_t selected_size = 0;
phys_addr_t selected_base = 0;
phys_addr_t selected_limit = limit;
bool fixed = false;
```

where first represents size in bytes of the reserved area, second is base address of the reserved area, third is end address of the reserved area and the last `fixed` parameter shows where to place reserved area. If `fixed` is `1` we just reserve area with the `memblock_reserve` , if it is `0` we allocate space with the `kmemleak_alloc` . In the next step we check `size_cmdline` variable and if it is not equal to `-1` we fill all variables which you can see above with the values from the `cma` kernel command line parameter:

```
if (size_cmdline != -1) {
   ...
   ...
   ...
}
```

You can find in this source code file definition of the early parameter:

```
early_param("cma", early_cma);
```

where `cma` is:

```
cma=nn[MG]@[start[MG][-end[MG]]]
        [ARM,X86,KNL]
```

```
        Sets the size of kernel global memory area for
        contiguous memory allocations and optionally the
        placement constraint by the physical address range of
        memory allocations. A value of 0 disables CMA
        altogether. For more information, see
        include/linux/dma-contiguous.h
```

If we will not pass `cma` option to the kernel command line, `size_cmdline` will be equal to `-1`. In this way we need to calculate size of the reserved area which depends on the following kernel configuration options:

- `CONFIG_CMA_SIZE_SEL_MBYTES` - size in megabytes, default global `CMA` area, which is equal to `CMA_SIZE_MBYTES * SZ_1M` or `CONFIG_CMA_SIZE_MBYTES * 1M`;
- `CONFIG_CMA_SIZE_SEL_PERCENTAGE` - percentage of total memory;
- `CONFIG_CMA_SIZE_SEL_MIN` - use lower value;
- `CONFIG_CMA_SIZE_SEL_MAX` - use higher value.

As we calculated the size of the reserved area, we reserve area with the call of the `dma_contiguous_reserve_area` function which first of all calls:

```
ret = cma_declare_contiguous(base, size, limit, 0, 0, fixed, res_cma);
```

function. The `cma_declare_contiguous` reserves contiguous area from the given base address with given size. After we reserved area for the `DMA`, next function is the `memblock_find_dma_reserve`. As you can understand from its name, this function counts the reserved pages in the `DMA` area. This part will not cover all details of the `CMA` and `DMA`, because they are big. We will see much more details in the special part in the Linux Kernel Memory management which covers contiguous memory allocators and areas.

Initialization of the sparse memory

The next step is the call of the function - `x86_init.paging.pagetable_init`. If you try to find this function in the linux kernel source code, in the end of your search, you will see the following macro:

```
#define native_pagetable_init        paging_init
```

which expands as you can see to the call of the `paging_init` function from the arch/x86/mm/init_64.c. The `paging_init` function initializes sparse memory and zone sizes. First of all what's zones and what is it `Sparsemem`. The `Sparsemem` is a special foundation in the linux kernel memory manager which used to split memory area into different memory banks in the NUMA systems. Let's look on the implementation of the `paginig_init` function:

```
void __init paging_init(void)
{
        sparse_memory_present_with_active_regions(MAX_NUMNODES);
        sparse_init();

        node_clear_state(0, N_MEMORY);
        if (N_MEMORY != N_NORMAL_MEMORY)
                node_clear_state(0, N_NORMAL_MEMORY);

        zone_sizes_init();
}
```

As you can see there is call of the `sparse_memory_present_with_active_regions` function which records a memory area for every `NUMA` node to the array of the `mem_section` structure which contains a pointer to the structure of the array of `struct page`. The next `sparse_init` function allocates non-linear `mem_section` and `mem_map`. In the next step we clear state of the movable memory nodes and initialize sizes of zones. Every `NUMA` node is divided into a number of pieces which are called - `zones`. So, `zone_sizes_init` function from the arch/x86/mm/init.c initializes size of zones.

Again, this part and next parts do not cover this theme in full details. There will be special part about `NUMA`.

vsyscall mapping

The next step after `SparseMem` initialization is setting of the `trampoline_cr4_features` which must contain content of the `cr4` Control register. First of all we need to check that current CPU has support of the `cr4` register and if it has, we save its content to the `trampoline_cr4_features` which is storage for `cr4` in the real mode:

```
if (boot_cpu_data.cpuid_level >= 0) {
    mmu_cr4_features = __read_cr4();
    if (trampoline_cr4_features)
        *trampoline_cr4_features = mmu_cr4_features;
}
```

The next function which you can see is `map_vsyscal` from the arch/x86/kernel/vsyscall_64.c. This function maps memory space for vsyscalls and depends on `CONFIG_X86_VSYSCALL_EMULATION` kernel configuration option. Actually `vsyscall` is a special segment which provides fast access to the certain system calls like `getcpu`, etc. Let's look on implementation of this function:

```
void __init map_vsyscall(void)
{
        extern char __vsyscall_page;
        unsigned long physaddr_vsyscall = __pa_symbol(&__vsyscall_page);

        if (vsyscall_mode != NONE)
                __set_fixmap(VSYSCALL_PAGE, physaddr_vsyscall,
                             vsyscall_mode == NATIVE
                             ? PAGE_KERNEL_VSYSCALL
                             : PAGE_KERNEL_VVAR);

        BUILD_BUG_ON((unsigned long)__fix_to_virt(VSYSCALL_PAGE) !=
                     (unsigned long)VSYSCALL_ADDR);
}
```

In the beginning of the `map_vsyscall` we can see definition of two variables. The first is extern variable `__vsyscall_page`. As a extern variable, it defined somewhere in other source code file. Actually we can see definition of the `__vsyscall_page` in the arch/x86/kernel/vsyscall_emu_64.S. The `__vsyscall_page` symbol points to the aligned calls of the `vsyscalls` as `gettimeofday`, etc.:

```
    .globl __vsyscall_page
    .balign PAGE_SIZE, 0xcc
    .type __vsyscall_page, @object
__vsyscall_page:

    mov $__NR_gettimeofday, %rax
    syscall
    ret

    .balign 1024, 0xcc
    mov $__NR_time, %rax
    syscall
    ret
    ...
    ...
    ...
```

The second variable is `physaddr_vsyscall` which just stores physical address of the `__vsyscall_page` symbol. In the next step we check the `vsyscall_mode` variable, and if it is not equal to `NONE`, it is `EMULATE` by default:

```
static enum { EMULATE, NATIVE, NONE } vsyscall_mode = EMULATE;
```

And after this check we can see the call of the `__set_fixmap` function which calls `native_set_fixmap` with the same parameters:

```
void native_set_fixmap(enum fixed_addresses idx, unsigned long phys, pgprot_t flags)
```

```
{
        __native_set_fixmap(idx, pfn_pte(phys >> PAGE_SHIFT, flags));
}

void __native_set_fixmap(enum fixed_addresses idx, pte_t pte)
{
        unsigned long address = __fix_to_virt(idx);

        if (idx >= __end_of_fixed_addresses) {
                BUG();
                return;
        }
        set_pte_vaddr(address, pte);
        fixmaps_set++;
}
```

Here we can see that `native_set_fixmap` makes value of `Page Table Entry` from the given physical address (physical address of the `__vsyscall_page` symbol in our case) and calls internal function - `__native_set_fixmap` . Internal function gets the virtual address of the given `fixed_addresses` index (`VSYSCALL_PAGE` in our case) and checks that given index is not greater than end of the fix-mapped addresses. After this we set page table entry with the call of the `set_pte_vaddr` function and increase count of the fix-mapped addresses. And in the end of the `map_vsyscall` we check that virtual address of the `VSYSCALL_PAGE` (which is first index in the `fixed_addresses`) is not greater than `VSYSCALL_ADDR` which is `-10UL << 20` or `ffffffffff600000` with the `BUILD_BUG_ON` macro:

```
BUILD_BUG_ON((unsigned long)__fix_to_virt(VSYSCALL_PAGE) !=
                     (unsigned long)VSYSCALL_ADDR);
```

Now `vsyscall` area is in the `fix-mapped` area. That's all about `map_vsyscall` , if you do not know anything about fix-mapped addresses, you can read Fix-Mapped Addresses and ioremap. We will see more about `vsyscalls` in the `vsyscalls and vdso` part.

Getting the SMP configuration

You may remember how we made a search of the SMP configuration in the previous part. Now we need to get the `SMP` configuration if we found it. For this we check `smp_found_config` variable which we set in the `smp_scan_config` function (read about it the previous part) and call the `get_smp_config` function:

```
if (smp_found_config)
    get_smp_config();
```

The `get_smp_config` expands to the `x86_init.mpparse.default_get_smp_config` function which is defined in the arch/x86/kernel/mpparse.c. This function defines a pointer to the multiprocessor floating pointer structure - `mpf_intel` (you can read about it in the previous part) and does some checks:

```
struct mpf_intel *mpf = mpf_found;

if (!mpf)
    return;

if (acpi_lapic && early)
   return;
```

Here we can see that multiprocessor configuration was found in the `smp_scan_config` function or just return from the function if not. The next check is `acpi_lapic` and `early` . And as we did this checks, we start to read the `SMP` configuration. As we finished reading it, the next step is - `prefill_possible_map` function which makes preliminary filling of the possible CPU's `cpumask` (more about it you can read in the Introduction to the cpumasks).

The rest of the setup_arch

Here we are getting to the end of the `setup_arch` function. The rest of function of course is important, but details about these stuff will not will not be included in this part. We will just take a short look on these functions, because although they are important as I wrote above, but they cover non-generic kernel features related with the `NUMA`, `SMP`, `ACPI` and `APICs`, etc. First of all, the next call of the `init_apic_mappings` function. As we can understand this function sets the address of the local APIC. The next is `x86_io_apic_ops.init` and this function initializes I/O APIC. Please note that we will see all details related with `APIC` in the chapter about interrupts and exceptions handling. In the next step we reserve standard I/O resources like `DMA`, `TIMER`, `FPU`, etc., with the call of the `x86_init.resources.reserve_resources` function. Following is `mcheck_init` function initializes `Machine check Exception` and the last is `register_refined_jiffies` which registers jiffy (There will be separate chapter about timers in the kernel).

So that's all. Finally we have finished with the big `setup_arch` function in this part. Of course as I already wrote many times, we did not see full details about this function, but do not worry about it. We will be back more than once to this function from different chapters for understanding how different platform-dependent parts are initialized.

That's all, and now we can back to the `start_kernel` from the `setup_arch`.

Back to the main.c

As I wrote above, we have finished with the `setup_arch` function and now we can back to the `start_kernel` function from the init/main.c. As you may remember or saw yourself, `start_kernel` function as big as the `setup_arch`. So the couple of the next part will be dedicated to learning of this function. So, let's continue with it. After the `setup_arch` we can see the call of the `mm_init_cpumask` function. This function sets the cpumask pointer to the memory descriptor `cpumask`. We can look on its implementation:

```
static inline void mm_init_cpumask(struct mm_struct *mm)
{
#ifdef CONFIG_CPUMASK_OFFSTACK
        mm->cpu_vm_mask_var = &mm->cpumask_allocation;
#endif
        cpumask_clear(mm->cpu_vm_mask_var);
}
```

As you can see in the init/main.c, we pass memory descriptor of the init process to the `mm_init_cpumask` and depends on `CONFIG_CPUMASK_OFFSTACK` configuration option we clear TLB switch `cpumask`.

In the next step we can see the call of the following function:

```
setup_command_line(command_line);
```

This function takes pointer to the kernel command line allocates a couple of buffers to store command line. We need a couple of buffers, because one buffer used for future reference and accessing to command line and one for parameter parsing. We will allocate space for the following buffers:

- `saved_command_line` - will contain boot command line;
- `initcall_command_line` - will contain boot command line. will be used in the `do_initcall_level`;
- `static_command_line` - will contain command line for parameters parsing.

We will allocate space with the `memblock_virt_alloc` function. This function calls `memblock_virt_alloc_try_nid` which allocates boot memory block with `memblock_reserve` if slab is not available or uses `kzalloc_node` (more about it will be in the linux memory management chapter). The `memblock_virt_alloc` uses `BOOTMEM_LOW_LIMIT` (physical address of the `(PAGE_OFFSET + 0x1000000)` value) and `BOOTMEM_ALLOC_ACCESSIBLE` (equal to the current value of the `memblock.current_limit`) as minimum address of the memory region and maximum address of the memory region.

Let's look on the implementation of the `setup_command_line`:

```
static void __init setup_command_line(char *command_line)
{
        saved_command_line =
```

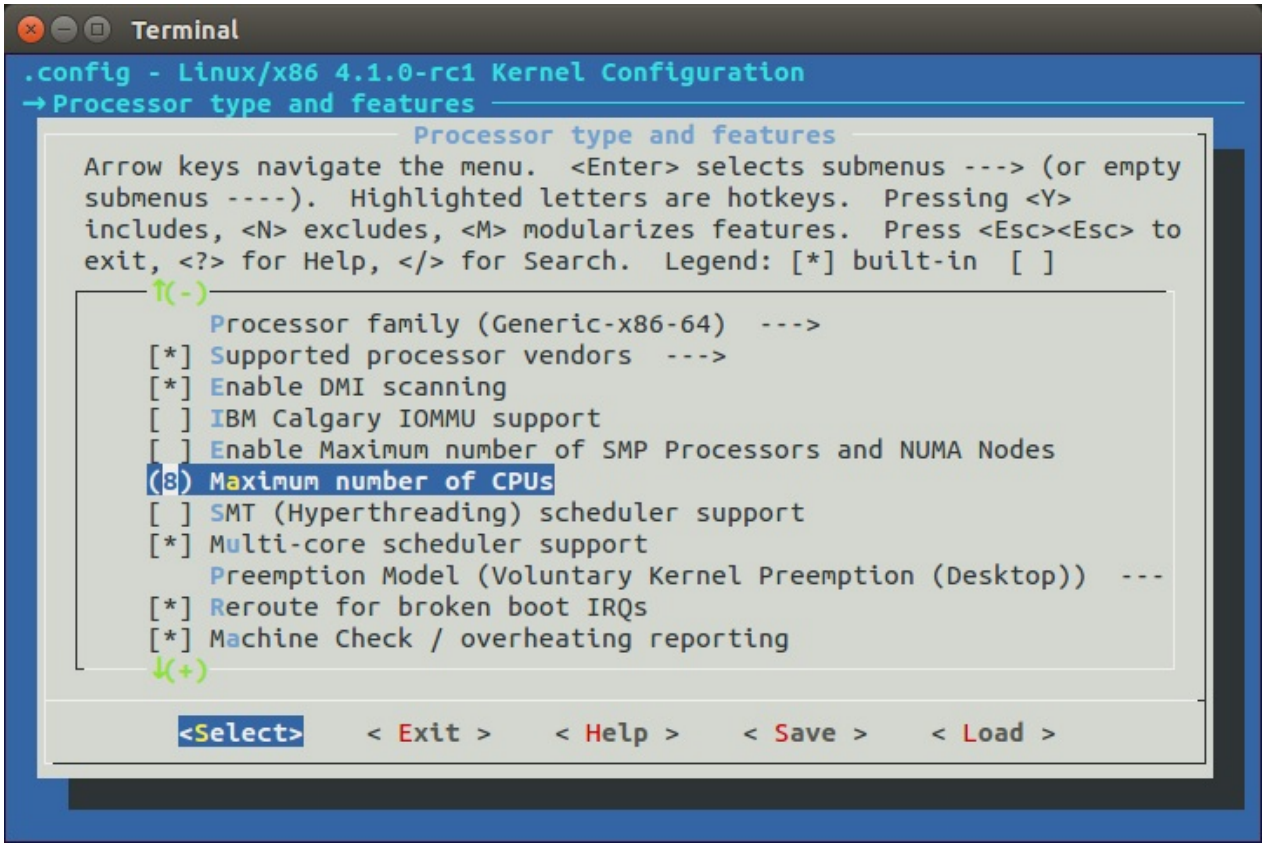
```
                memblock_virt_alloc(strlen(boot_command_line) + 1, 0);
        initcall_command_line =
                memblock_virt_alloc(strlen(boot_command_line) + 1, 0);
        static_command_line = memblock_virt_alloc(strlen(command_line) + 1, 0);
        strcpy(saved_command_line, boot_command_line);
        strcpy(static_command_line, command_line);
 }
```

Here we can see that we allocate space for the three buffers which will contain kernel command line for the different purposes (read above). And as we allocated space, we store `boot_command_line` in the `saved_command_line` and `command_line` (kernel command line from the `setup_arch`) to the `static_command_line`.

The next function after the `setup_command_line` is the `setup_nr_cpu_ids`. This function setting `nr_cpu_ids` (number of CPUs) according to the last bit in the `cpu_possible_mask` (more about it you can read in the chapter describes cpumasks concept). Let's look on its implementation:

```
void __init setup_nr_cpu_ids(void)
{
        nr_cpu_ids = find_last_bit(cpumask_bits(cpu_possible_mask),NR_CPUS) + 1;
}
```

Here `nr_cpu_ids` represents number of CPUs, `NR_CPUS` represents the maximum number of CPUs which we can set in configuration time:



Actually we need to call this function, because `NR_CPUS` can be greater than actual amount of the CPUs in the your computer. Here we can see that we call `find_last_bit` function and pass two parameters to it:

- `cpu_possible_mask` bits;
- maximum number of CPUS.

In the `setup_arch` we can find the call of the `prefill_possible_map` function which calculates and writes to the `cpu_possible_mask` actual number of the CPUs. We call the `find_last_bit` function which takes the address and maximum size to search and returns bit number of the first set bit. We passed `cpu_possible_mask` bits and maximum number of the CPUs. First of all the `find_last_bit` function splits given `unsigned long` address to the words:

```
words = size / BITS_PER_LONG;
```

where `BITS_PER_LONG` is `64` on the `x86_64`. As we got amount of words in the given size of the search data, we need to check is given size does not contain partial words with the following check:

```
if (size & (BITS_PER_LONG-1)) {
        tmp = (addr[words] & (~0UL >> (BITS_PER_LONG
                                - (size & (BITS_PER_LONG-1)))));
        if (tmp)
                goto found;
}
```

if it contains partial word, we mask the last word and check it. If the last word is not zero, it means that current word contains at least one set bit. We go to the `found` label:

```
found:
    return words * BITS_PER_LONG + __fls(tmp);
```

Here you can see `__fls` function which returns last set bit in a given word with help of the `bsr` instruction:

```
static inline unsigned long __fls(unsigned long word)
{
        asm("bsr %1,%0"
            : "=r" (word)
            : "rm" (word));
        return word;
}
```

The `bsr` instruction which scans the given operand for first bit set. If the last word is not partial we going through the all words in the given address and trying to find first set bit:

```
while (words) {
    tmp = addr[--words];
    if (tmp) {
found:
        return words * BITS_PER_LONG + __fls(tmp);
    }
}
```

Here we put the last word to the `tmp` variable and check that `tmp` contains at least one set bit. If a set bit found, we return the number of this bit. If no one words do not contains set bit we just return given size:

```
return size;
```

After this `nr_cpu_ids` will contain the correct amount of the available CPUs.

That's all.

Conclusion

It is the end of the seventh part about the linux kernel initialization process. In this part, finally we have finished with the `setup_arch` function and returned to the `start_kernel` function. In the next part we will continue to learn generic kernel code from the `start_kernel` and will continue our way to the first `init` process.

If you have any questions or suggestions write me a comment or ping me at twitter.

Please note that English is not my first language, And I am really sorry for any inconvenience. If you find any mistakes please send me PR to linux-insides.

Links

- Desktop Management Interface
- x86_64
- initrd
- Kernel panic
- Documentation/kernel-parameters.txt
- ACPI
- Direct memory access
- NUMA
- Control register
- vsyscalls
- SMP
- jiffy
- Previous part

Kernel initialization. Part 8.

Scheduler initialization

This is the eighth part of the Linux kernel initialization process and we stopped on the `setup_nr_cpu_ids` function in the previous part. The main point of the current part is scheduler initialization. But before we will start to learn initialization process of the scheduler, we need to do some stuff. The next step in the init/main.c is the `setup_per_cpu_areas` function. This function setups areas for the `percpu` variables, more about it you can read in the special part about the Per-CPU variables. After `percpu` areas is up and running, the next step is the `smp_prepare_boot_cpu` function. This function does some preparations for the SMP:

```
static inline void smp_prepare_boot_cpu(void)
{
         smp_ops.smp_prepare_boot_cpu();
}
```

where the `smp_prepare_boot_cpu` expands to the call of the `native_smp_prepare_boot_cpu` function (more about `smp_ops` will be in the special parts about `SMP`):

```
void __init native_smp_prepare_boot_cpu(void)
{
        int me = smp_processor_id();
        switch_to_new_gdt(me);
        cpumask_set_cpu(me, cpu_callout_mask);
        per_cpu(cpu_state, me) = CPU_ONLINE;
}
```

The `native_smp_prepare_boot_cpu` function gets the id of the current CPU (which is Bootstrap processor and its `id` is zero) with the `smp_processor_id` function. I will not explain how the `smp_processor_id` works, because we already saw it in the Kernel entry point part. As we got processor `id` number we reload Global Descriptor Table for the given CPU with the `switch_to_new_gdt` function:

```
void switch_to_new_gdt(int cpu)
{
        struct desc_ptr gdt_descr;

        gdt_descr.address = (long)get_cpu_gdt_table(cpu);
        gdt_descr.size = GDT_SIZE - 1;
        load_gdt(&gdt_descr);
        load_percpu_segment(cpu);
}
```

The `gdt_descr` variable represents pointer to the `GDT` descriptor here (we already saw `desc_ptr` in the Early interrupt and exception handling). We get the address and the size of the `GDT` descriptor where `GDT_SIZE` is `256` or:

```
#define GDT_SIZE (GDT_ENTRIES * 8)
```

and the address of the descriptor we will get with the `get_cpu_gdt_table` :

```
static inline struct desc_struct *get_cpu_gdt_table(unsigned int cpu)
{
        return per_cpu(gdt_page, cpu).gdt;
}
```

The `get_cpu_gdt_table` uses `per_cpu` macro for getting `gdt_page` percpu variable for the given CPU number (bootstrap processor with `id` - 0 in our case). You may ask the following question: so, if we can access `gdt_page` percpu variable, where it was defined? Actually we already saw it in this book. If you have read the first part of this chapter, you can remember that we saw definition of the

`gdt_page` in the arch/x86/kernel/head_64.S:

```
early_gdt_descr:
    .word    GDT_ENTRIES*8-1
early_gdt_descr_base:
    .quad    INIT_PER_CPU_VAR(gdt_page)
```

and if we will look on the linker file we can see that it locates after the `__per_cpu_load` symbol:

```
#define INIT_PER_CPU(x) init_per_cpu__##x = x + __per_cpu_load
INIT_PER_CPU(gdt_page);
```

and filled `gdt_page` in the arch/x86/kernel/cpu/common.c:

```
DEFINE_PER_CPU_PAGE_ALIGNED(struct gdt_page, gdt_page) = { .gdt = {
#ifdef CONFIG_X86_64
    [GDT_ENTRY_KERNEL32_CS]        = GDT_ENTRY_INIT(0xc09b, 0, 0xfffff),
    [GDT_ENTRY_KERNEL_CS]        = GDT_ENTRY_INIT(0xa09b, 0, 0xfffff),
    [GDT_ENTRY_KERNEL_DS]        = GDT_ENTRY_INIT(0xc093, 0, 0xfffff),
    [GDT_ENTRY_DEFAULT_USER32_CS]    = GDT_ENTRY_INIT(0xc0fb, 0, 0xfffff),
    [GDT_ENTRY_DEFAULT_USER_DS]    = GDT_ENTRY_INIT(0xc0f3, 0, 0xfffff),
    [GDT_ENTRY_DEFAULT_USER_CS]    = GDT_ENTRY_INIT(0xa0fb, 0, 0xfffff),
    ...
    ...
    ...
```

more about `percpu` variables you can read in the Per-CPU variables part. As we got address and size of the `GDT` descriptor we reload `GDT` with the `load_gdt` which just execute `lgdt` instruct and load `percpu_segment` with the following function:

```
void load_percpu_segment(int cpu) {
    loadsegment(gs, 0);
    wrmsrl(MSR_GS_BASE, (unsigned long)per_cpu(irq_stack_union.gs_base, cpu));
    load_stack_canary_segment();
}
```

The base address of the `percpu` area must contain `gs` register (or `fs` register for `x86`), so we are using `loadsegment` macro and pass `gs` . In the next step we writes the base address if the IRQ stack and setup stack canary (this is only for `x86_32`). After we load new `GDT` , we fill `cpu_callout_mask` bitmap with the current cpu and set cpu state as online with the setting `cpu_state` percpu variable for the current processor - `CPU_ONLINE` :

```
cpumask_set_cpu(me, cpu_callout_mask);
per_cpu(cpu_state, me) = CPU_ONLINE;
```

So, what is `cpu_callout_mask` bitmap... As we initialized bootstrap processor (processor which is booted the first on `x86`) the other processors in a multiprocessor system are known as `secondary processors` . Linux kernel uses following two bitmasks:

- `cpu_callout_mask`
- `cpu_callin_mask`

After bootstrap processor initialized, it updates the `cpu_callout_mask` to indicate which secondary processor can be initialized next. All other or secondary processors can do some initialization stuff before and check the `cpu_callout_mask` on the boostrap processor bit. Only after the bootstrap processor filled the `cpu_callout_mask` with this secondary processor, it will continue the rest of its initialization. After that the certain processor finish its initialization process, the processor sets bit in the `cpu_callin_mask` . Once the bootstrap processor finds the bit in the `cpu_callin_mask` for the current secondary processor, this processor repeats the same procedure for initialization of one of the remaining secondary processors. In a short words it works as i described, but we will see more details in the chapter about `SMP` .

That's all. We did all `SMP` boot preparation.

Build zonelists

In the next step we can see the call of the `build_all_zonelists` function. This function sets up the order of zones that allocations are preferred from. What are zones and what's order we will understand soon. For the start let's see how linux kernel considers physical memory. Physical memory is split into banks which are called - `nodes` . If you has no hardware support for `NUMA` , you will see only one node:

```
$ cat /sys/devices/system/node/node0/numastat
numa_hit 72452442
numa_miss 0
numa_foreign 0
interleave_hit 12925
local_node 72452442
other_node 0
```

Every `node` is presented by the `struct pglist_data` in the linux kernel. Each node is divided into a number of special blocks which are called - `zones` . Every zone is presented by the `zone struct` in the linux kernel and has one of the type:

- `ZONE_DMA` - 0-16M;
- `ZONE_DMA32` - used for 32 bit devices that can only do DMA areas below 4G;
- `ZONE_NORMAL` - all RAM from the 4GB on the `x86_64` ;
- `ZONE_HIGHMEM` - absent on the `x86_64` ;
- `ZONE_MOVABLE` - zone which contains movable pages.

which are presented by the `zone_type` enum. We can get information about zones with the:

```
$ cat /proc/zoneinfo
Node 0, zone      DMA
  pages free     3975
        min      3
        low      3
        ...
        ...
Node 0, zone    DMA32
  pages free     694163
        min      875
        low      1093
        ...
        ...
Node 0, zone   Normal
  pages free     2529995
        min      3146
        low      3932
        ...
        ...
```

As I wrote above all nodes are described with the `pglist_data` or `pg_data_t` structure in memory. This structure is defined in the include/linux/mmzone.h. The `build_all_zonelists` function from the mm/page_alloc.c constructs an ordered `zonelist` (of different zones `DMA` , `DMA32` , `NORMAL` , `HIGH_MEMORY` , `MOVABLE`) which specifies the zones/nodes to visit when a selected `zone` or `node` cannot satisfy the allocation request. That's all. More about `NUMA` and multiprocessor systems will be in the special part.

The rest of the stuff before scheduler initialization

Before we will start to dive into linux kernel scheduler initialization process we must do a couple of things. The first thing is the `page_alloc_init` function from the mm/page_alloc.c. This function looks pretty easy:

```
void __init page_alloc_init(void)
{
        hotcpu_notifier(page_alloc_cpu_notify, 0);
}
```

and initializes handler for the `CPU` hotplug. Of course the `hotcpu_notifier` depends on the `CONFIG_HOTPLUG_CPU` configuration option and if this option is set, it just calls `cpu_notifier` macro which expands to the call of the `register_cpu_notifier` which adds hotplug cpu handler (`page_alloc_cpu_notify` in our case).

After this we can see the kernel command line in the initialization output:

```
Linux version 4.1.0-rc2+ (alex@localhost) (gcc version 4.9.2 (Ubuntu 4.9.2-10ubuntu13) ) #493 SMP Thu
Command line: root=/dev/sdb earlyprintk=ttyS0,115200 loglevel=7 debug rdinit=/sbin/init root=/dev/ram
```

And a couple of functions such as `parse_early_param` and `parse_args` which handles linux kernel command line. You may remember that we already saw the call of the `parse_early_param` function in the sixth part of the kernel initialization chapter, so why we call it again? Answer is simple: we call this function in the architecture-specific code (`x86_64` in our case), but not all architecture calls this function. And we need to call the second function `parse_args` to parse and handle non-early command line arguments.

In the next step we can see the call of the `jump_label_init` from the kernel/jump_label.c. and initializes jump label.

After this we can see the call of the `setup_log_buf` function which setups the printk log buffer. We already saw this function in the seventh part of the linux kernel initialization process chapter.

PID hash initialization

The next is `pidhash_init` function. As you know each process has assigned a unique number which called - `process identification number` or `PID` . Each process generated with fork or clone is automatically assigned a new unique `PID` value by the kernel. The management of `PIDs` centered around the two special data structures: `struct pid` and `struct upid` . First structure represents information about a `PID` in the kernel. The second structure represents the information that is visible in a specific namespace. All `PID` instances stored in the special hash table:

```
static struct hlist_head *pid_hash;
```

This hash table is used to find the pid instance that belongs to a numeric `PID` value. So, `pidhash_init` initializes this hash table. In the start of the `pidhash_init` function we can see the call of the `alloc_large_system_hash` :

```
pid_hash = alloc_large_system_hash("PID", sizeof(*pid_hash), 0, 18,
                                   HASH_EARLY | HASH_SMALL,
                                   &pidhash_shift, NULL,
                                   0, 4096);
```

The number of elements of the `pid_hash` depends on the `RAM` configuration, but it can be between `2^4` and `2^12` . The `pidhash_init` computes the size and allocates the required storage (which is `hlist` in our case - the same as doubly linked list, but contains one pointer instead on the struct hlist_head]. The `alloc_large_system_hash` function allocates a large system hash table with `memblock_virt_alloc_nopanic` if we pass `HASH_EARLY` flag (as it in our case) or with `__vmalloc` if we did no pass this flag.

The result we can see in the `dmesg` output:

```
$ dmesg | grep hash
[    0.000000] PID hash table entries: 4096 (order: 3, 32768 bytes)
...
...
...
```

That's all. The rest of the stuff before scheduler initialization is the following functions: `vfs_caches_init_early` does early initialization of the virtual file system (more about it will be in the chapter which will describe virtual file system), `sort_main_extable` sorts the kernel's built-in exception table entries which are between `__start___ex_table` and `__stop___ex_table` , and `trap_init` initializes trap handlers (more about last two function we will know in the separate chapter about interrupts).

The last step before the scheduler initialization is initialization of the memory manager with the `mm_init` function from the init/main.c. As we can see, the `mm_init` function initializes different parts of the linux kernel memory manager:

```
page_ext_init_flatmem();
mem_init();
kmem_cache_init();
percpu_init_late();
pgtable_init();
vmalloc_init();
```

The first is `page_ext_init_flatmem` which depends on the `CONFIG_SPARSEMEM` kernel configuration option and initializes extended data per page handling. The `mem_init` releases all `bootmem` , the `kmem_cache_init` initializes kernel cache, the `percpu_init_late` - replaces `percpu` chunks with those allocated by slub, the `pgtable_init` - initializes the `page->ptl` kernel cache, the `vmalloc_init` - initializes `vmalloc` . Please, **NOTE** that we will not dive into details about all of these functions and concepts, but we will see all of they it in the Linux kernel memory manager chapter.

That's all. Now we can look on the `scheduler` .

Scheduler initialization

And now we come to the main purpose of this part - initialization of the task scheduler. I want to say again as I already did it many times, you will not see the full explanation of the scheduler here, there will be special chapter about this. Ok, next point is the `sched_init` function from the kernel/sched/core.c and as we can understand from the function's name, it initializes scheduler. Let's start to dive into this function and try to understand how the scheduler is initialized. At the start of the `sched_init` function we can see the following code:

```
#ifdef CONFIG_FAIR_GROUP_SCHED
        alloc_size += 2 * nr_cpu_ids * sizeof(void **);
#endif
#ifdef CONFIG_RT_GROUP_SCHED
        alloc_size += 2 * nr_cpu_ids * sizeof(void **);
#endif
```

First of all we can see two configuration options here:

- `CONFIG_FAIR_GROUP_SCHED`
- `CONFIG_RT_GROUP_SCHED`

Both of this options provide two different planning models. As we can read from the documentation, the current scheduler - `CFS` or `Completely Fair Scheduler` use a simple concept. It models process scheduling as if the system has an ideal multitasking processor where each process would receive `1/n` processor time, where `n` is the number of the runnable processes. The scheduler uses the special set of rules. These rules determine when and how to select a new process to run and they are called `scheduling policy` . The Completely Fair Scheduler supports following `normal` or `non-real-time` scheduling policies: `SCHED_NORMAL` , `SCHED_BATCH` and `SCHED_IDLE` . The `SCHED_NORMAL` is used for the most normal applications, the amount of cpu each process consumes is mostly determined by the nice value, the `SCHED_BATCH` used for the 100% non-interactive tasks and the `SCHED_IDLE` runs tasks only when the processor has no task to run besides this task. The `real-time` policies are also supported for the time-critical applications: `SCHED_FIFO` and `SCHED_RR` . If you've read something about the Linux kernel scheduler, you can know that it is modular. It means that it supports different algorithms to schedule different types of processes. Usually this modularity is called `scheduler classes` . These modules encapsulate scheduling policy details and are handled by the scheduler core without knowing too much about them.

Now let's back to the our code and look on the two configuration options `CONFIG_FAIR_GROUP_SCHED` and `CONFIG_RT_GROUP_SCHED` . The scheduler operates on an individual task. These options allows to schedule group tasks (more about it you can read in the CFS group scheduling). We can see that we assign the `alloc_size` variables which represent size based on amount of the processors to allocate for the `sched_entity` and `cfs_rq` to the `2 * nr_cpu_ids * sizeof(void **)` expression with `kzalloc` :

```
ptr = (unsigned long)kzalloc(alloc_size, GFP_NOWAIT);

#ifdef CONFIG_FAIR_GROUP_SCHED
        root_task_group.se = (struct sched_entity **)ptr;
        ptr += nr_cpu_ids * sizeof(void **);
```

```
        root_task_group.cfs_rq = (struct cfs_rq **)ptr;
        ptr += nr_cpu_ids * sizeof(void **);
#endif
```

The `sched_entity` is a structure which is defined in the include/linux/sched.h and used by the scheduler to keep track of process accounting. The `cfs_rq` presents run queue. So, you can see that we allocated space with size `alloc_size` for the run queue and scheduler entity of the `root_task_group`. The `root_task_group` is an instance of the `task_group` structure from the kernel/sched/sched.h which contains task group related information:

```
struct task_group {
    ...
    ...
    struct sched_entity **se;
    struct cfs_rq **cfs_rq;
    ...
    ...
}
```

The root task group is the task group which belongs to every task in system. As we allocated space for the root task group scheduler entity and runqueue, we go over all possible CPUs (`cpu_possible_mask` bitmap) and allocate zeroed memory from a particular memory node with the `kzalloc_node` function for the `load_balance_mask` `percpu` variable:

```
DECLARE_PER_CPU(cpumask_var_t, load_balance_mask);
```

Here `cpumask_var_t` is the `cpumask_t` with one difference: `cpumask_var_t` is allocated only `nr_cpu_ids` bits when the `cpumask_t` always has `NR_CPUS` bits (more about `cpumask` you can read in the CPU masks part). As you can see:

```
#ifdef CONFIG_CPUMASK_OFFSTACK
    for_each_possible_cpu(i) {
        per_cpu(load_balance_mask, i) = (cpumask_var_t)kzalloc_node(
                cpumask_size(), GFP_KERNEL, cpu_to_node(i));
    }
#endif
```

this code depends on the `CONFIG_CPUMASK_OFFSTACK` configuration option. This configuration options says to use dynamic allocation for `cpumask`, instead of putting it on the stack. All groups have to be able to rely on the amount of CPU time. With the call of the two following functions:

```
init_rt_bandwidth(&def_rt_bandwidth,
                  global_rt_period(), global_rt_runtime());
init_dl_bandwidth(&def_dl_bandwidth,
                  global_rt_period(), global_rt_runtime());
```

we initialize bandwidth management for the `SCHED_DEADLINE` real-time tasks. These functions initializes `rt_bandwidth` and `dl_bandwidth` structures which store information about maximum `deadline` bandwidth of the system. For example, let's look on the implementation of the `init_rt_bandwidth` function:

```
void init_rt_bandwidth(struct rt_bandwidth *rt_b, u64 period, u64 runtime)
{
        rt_b->rt_period = ns_to_ktime(period);
        rt_b->rt_runtime = runtime;

        raw_spin_lock_init(&rt_b->rt_runtime_lock);

        hrtimer_init(&rt_b->rt_period_timer,
                     CLOCK_MONOTONIC, HRTIMER_MODE_REL);
        rt_b->rt_period_timer.function = sched_rt_period_timer;
}
```

It takes three parameters:

- address of the `rt_bandwidth` structure which contains information about the allocated and consumed quota within a period;
- `period` - period over which real-time task bandwidth enforcement is measured in `us`;
- `runtime` - part of the period that we allow tasks to run in `us`.

As `period` and `runtime` we pass result of the `global_rt_period` and `global_rt_runtime` functions. Which are `1s` second and and `0.95s` by default. The `rt_bandwidth` structure is defined in the kernel/sched/sched.h and looks:

```
struct rt_bandwidth {
        raw_spinlock_t          rt_runtime_lock;
        ktime_t                 rt_period;
        u64                     rt_runtime;
        struct hrtimer          rt_period_timer;
};
```

As you can see, it contains `runtime` and `period` and also two following fields:

- `rt_runtime_lock` - spinlock for the `rt_time` protection;
- `rt_period_timer` - high-resolution kernel timer for unthrottled of real-time tasks.

So, in the `init_rt_bandwidth` we initialize `rt_bandwidth` period and runtime with the given parameters, initialize the spinlock and high-resolution time. In the next step, depends on enable of SMP, we make initialization of the root domain:

```
#ifdef CONFIG_SMP
    init_defrootdomain();
#endif
```

The real-time scheduler requires global resources to make scheduling decision. But unfortunately scalability bottlenecks appear as the number of CPUs increase. The concept of root domains was introduced for improving scalability. The linux kernel provides a special mechanism for assigning a set of CPUs and memory nodes to a set of tasks and it is called - `cpuset`. If a `cpuset` contains non-overlapping with other `cpuset` CPUs, it is `exclusive cpuset`. Each exclusive cpuset defines an isolated domain or `root domain` of CPUs partitioned from other cpusets or CPUs. A `root domain` is presented by the `struct root_domain` from the kernel/sched/sched.h in the linux kernel and its main purpose is to narrow the scope of the global variables to per-domain variables and all real-time scheduling decisions are made only within the scope of a root domain. That's all about it, but we will see more details about it in the chapter about real-time scheduler.

After `root domain` initialization, we make initialization of the bandwidth for the real-time tasks of the root task group as we did it above:

```
#ifdef CONFIG_RT_GROUP_SCHED
    init_rt_bandwidth(&root_task_group.rt_bandwidth,
            global_rt_period(), global_rt_runtime());
#endif
```

In the next step, depends on the `CONFIG_CGROUP_SCHED` kernel configuration option we initialize the `siblings` and `children` lists of the root task group. As we can read from the documentation, the `CONFIG_CGROUP_SCHED` is:

```
This option allows you to create arbitrary task groups using the "cgroup" pseudo
filesystem and control the cpu bandwidth allocated to each such task group.
```

As we finished with the lists initialization, we can see the call of the `autogroup_init` function:

```
#ifdef CONFIG_CGROUP_SCHED
        list_add(&root_task_group.list, &task_groups);
        INIT_LIST_HEAD(&root_task_group.children);
        INIT_LIST_HEAD(&root_task_group.siblings);
        autogroup_init(&init_task);
#endif
```

which initializes automatic process group scheduling.

After this we are going through the all `possible` cpu (you can remember that `possible` CPUs store in the `cpu_possible_mask` bitmap that can ever be available in the system) and initialize a `runqueue` for each possible cpu:

```
for_each_possible_cpu(i) {
    struct rq *rq;
    ...
    ...
    ...
```

Each processor has its own locking and individual runqueue. All runnable tasks are stored in an active array and indexed according to its priority. When a process consumes its time slice, it is moved to an expired array. All of these arras are stored in the special structure which names is `runqueue`. As there are no global lock and runqueue, we are going through the all possible CPUs and initialize runqueue for the every cpu. The `runqueue` is presented by the `rq` structure in the linux kernel which is defined in the kernel/sched/sched.h.

```
rq = cpu_rq(i);
raw_spin_lock_init(&rq->lock);
rq->nr_running = 0;
rq->calc_load_active = 0;
rq->calc_load_update = jiffies + LOAD_FREQ;
init_cfs_rq(&rq->cfs);
init_rt_rq(&rq->rt);
init_dl_rq(&rq->dl);
rq->rt.rt_runtime = def_rt_bandwidth.rt_runtime;
```

Here we get the runqueue for the every CPU with the `cpu_rq` macro which returns `runqueues` percpu variable and start to initialize it with runqueue lock, number of running tasks, `calc_load` relative fields (`calc_load_active` and `calc_load_update`) which are used in the reckoning of a CPU load and initialization of the completely fair, real-time and deadline related fields in a runqueue. After this we initialize `cpu_load` array with zeros and set the last load update tick to the `jiffies` variable which determines the number of time ticks (cycles), since the system boot:

```
for (j = 0; j < CPU_LOAD_IDX_MAX; j++)
    rq->cpu_load[j] = 0;

rq->last_load_update_tick = jiffies;
```

where `cpu_load` keeps history of runqueue loads in the past, for now `CPU_LOAD_IDX_MAX` is 5. In the next step we fill `runqueue` fields which are related to the SMP, but we will not cover them in this part. And in the end of the loop we initialize high-resolution timer for the give `runqueue` and set the `iowait` (more about it in the separate part about scheduler) number:

```
init_rq_hrtick(rq);
atomic_set(&rq->nr_iowait, 0);
```

Now we come out from the `for_each_possible_cpu` loop and the next we need to set load weight for the `init` task with the `set_load_weight` function. Weight of process is calculated through its dynamic priority which is static priority + scheduling class of the process. After this we increase memory usage counter of the memory descriptor of the `init` process and set scheduler class for the current process:

```
atomic_inc(&init_mm.mm_count);
current->sched_class = &fair_sched_class;
```

And make current process (it will be the first `init` process) `idle` and update the value of the `calc_load_update` with the 5 seconds interval:

```
init_idle(current, smp_processor_id());
calc_load_update = jiffies + LOAD_FREQ;
```

So, the `init` process will be run, when there will be no other candidates (as it is the first process in the system). In the end we just set `scheduler_running` variable:

```
scheduler_running = 1;
```

That's all. Linux kernel scheduler is initialized. Of course, we have skipped many different details and explanations here, because we need to know and understand how different concepts (like process and process groups, runqueue, rcu, etc.) works in the linux kernel , but we took a short look on the scheduler initialization process. We will look all other details in the separate part which will be fully dedicated to the scheduler.

Conclusion

It is the end of the eighth part about the linux kernel initialization process. In this part, we looked on the initialization process of the scheduler and we will continue in the next part to dive in the linux kernel initialization process and will see initialization of the RCU and many other initialization stuff in the next part.

If you have any questions or suggestions write me a comment or ping me at twitter.

Please note that English is not my first language, And I am really sorry for any inconvenience. If you find any mistakes please send me PR to linux-insides.

Links

- CPU masks
- high-resolution kernel timer
- spinlock
- Run queue
- Linux kernem memory manager
- slub
- virtual file system
- Linux kernel hotplug documentation
- IRQ
- Global Descriptor Table
- Per-CPU variables
- SMP
- RCU
- CFS Scheduler documentation
- Real-Time group scheduling
- Previous part

Kernel initialization. Part 9.

RCU initialization

This is ninth part of the Linux Kernel initialization process and in the previous part we stopped at the scheduler initialization. In this part we will continue to dive to the linux kernel initialization process and the main purpose of this part will be to learn about initialization of the RCU. We can see that the next step in the init/main.c after the `sched_init` is the call of the `preempt_disable`. There are two macros:

- `preempt_disable`
- `preempt_enable`

for preemption disabling and enabling. First of all let's try to understand what is `preempt` in the context of an operating system kernel. In simple words, preemption is ability of the operating system kernel to preempt current task to run task with higher priority. Here we need to disable preemption because we will have only one `init` process for the early boot time and we don't need to stop it before we call `cpu_idle` function. The `preempt_disable` macro is defined in the include/linux/preempt.h and depends on the `CONFIG_PREEMPT_COUNT` kernel configuration option. This macro is implemented as:

```
#define preempt_disable() \
do { \
        preempt_count_inc(); \
        barrier(); \
} while (0)
```

and if `CONFIG_PREEMPT_COUNT` is not set just:

```
#define preempt_disable()                       barrier()
```

Let's look on it. First of all we can see one difference between these macro implementations. The `preempt_disable` with `CONFIG_PREEMPT_COUNT` set contains the call of the `preempt_count_inc`. There is special `percpu` variable which stores the number of held locks and `preempt_disable` calls:

```
DECLARE_PER_CPU(int, __preempt_count);
```

In the first implementation of the `preempt_disable` we increment this `__preempt_count`. There is API for returning value of the `__preempt_count`, it is the `preempt_count` function. As we called `preempt_disable`, first of all we increment preemption counter with the `preempt_count_inc` macro which expands to the:

```
#define preempt_count_inc() preempt_count_add(1)
#define preempt_count_add(val)  __preempt_count_add(val)
```

where `preempt_count_add` calls the `raw_cpu_add_4` macro which adds `1` to the given `percpu` variable (`__preempt_count`) in our case (more about `precpu` variables you can read in the part about Per-CPU variables). Ok, we increased `__preempt_count` and the next step we can see the call of the `barrier` macro in the both macros. The `barrier` macro inserts an optimization barrier. In the processors with `x86_64` architecture independent memory access operations can be performed in any order. That's why we need the opportunity to point compiler and processor on compliance of order. This mechanism is memory barrier. Let's consider a simple example:

```
preempt_disable();
foo();
preempt_enable();
```

Compiler can rearrange it as:

```
preempt_disable();
preempt_enable();
foo();
```

In this case non-preemptible function `foo` can be preempted. As we put `barrier` macro in the `preempt_disable` and `preempt_enable` macros, it prevents the compiler from swapping `preempt_count_inc` with other statements. More about barriers you can read here and here.

In the next step we can see following statement:

```
if (WARN(!irqs_disabled(),
     "Interrupts were enabled *very* early, fixing it\n"))
    local_irq_disable();
```

which check IRQs state, and disabling (with `cli` instruction for `x86_64`) if they are enabled.

That's all. Preemption is disabled and we can go ahead.

Initialization of the integer ID management

In the next step we can see the call of the `idr_init_cache` function which defined in the lib/idr.c. The `idr` library is used in a various places in the linux kernel to manage assigning integer `IDs` to objects and looking up objects by id.

Let's look on the implementation of the `idr_init_cache` function:

```
void __init idr_init_cache(void)
{
        idr_layer_cache = kmem_cache_create("idr_layer_cache",
                                sizeof(struct idr_layer), 0, SLAB_PANIC, NULL);
}
```

Here we can see the call of the `kmem_cache_create`. We already called the `kmem_cache_init` in the init/main.c. This function create generalized caches again using the `kmem_cache_alloc` (more about caches we will see in the Linux kernel memory management chapter). In our case, as we are using `kmem_cache_t` which will be used by the slab allocator and `kmem_cache_create` creates it. As you can see we pass five parameters to the `kmem_cache_create`:

- name of the cache;
- size of the object to store in cache;
- offset of the first object in the page;
- flags;
- constructor for the objects.

and it will create `kmem_cache` for the integer IDs. Integer `IDs` is commonly used pattern to map set of integer IDs to the set of pointers. We can see usage of the integer IDs in the i2c drivers subsystem. For example drivers/i2c/i2c-core.c which represents the core of the `i2c` subsystem defines `ID` for the `i2c` adapter with the `DEFINE_IDR` macro:

```
static DEFINE_IDR(i2c_adapter_idr);
```

and then uses it for the declaration of the `i2c` adapter:

```
static int __i2c_add_numbered_adapter(struct i2c_adapter *adap)
{
  int     id;
  ...
  ...
  ...
```

```
    id = idr_alloc(&i2c_adapter_idr, adap, adap->nr, adap->nr + 1, GFP_KERNEL);
    ...
    ...
    ...
}
```

and `id2_adapter_idr` presents dynamically calculated bus number.

More about integer ID management you can read here.

RCU initialization

The next step is RCU initialization with the `rcu_init` function and it's implementation depends on two kernel configuration options:

- `CONFIG_TINY_RCU`
- `CONFIG_TREE_RCU`

In the first case `rcu_init` will be in the kernel/rcu/tiny.c and in the second case it will be defined in the kernel/rcu/tree.c. We will see the implementation of the `tree rcu`, but first of all about the `RCU` in general.

`RCU` or read-copy update is a scalable high-performance synchronization mechanism implemented in the Linux kernel. On the early stage the linux kernel provided support and environment for the concurrently running applications, but all execution was serialized in the kernel using a single global lock. In our days linux kernel has no single global lock, but provides different mechanisms including lock-free data structures, percpu data structures and other. One of these mechanisms is - the `read-copy update`. The `RCU` technique is designed for rarely-modified data structures. The idea of the `RCU` is simple. For example we have a rarely-modified data structure. If somebody wants to change this data structure, we make a copy of this data structure and make all changes in the copy. In the same time all other users of the data structure use old version of it. Next, we need to choose safe moment when original version of the data structure will have no users and update it with the modified copy.

Of course this description of the `RCU` is very simplified. To understand some details about `RCU`, first of all we need to learn some terminology. Data readers in the `RCU` executed in the critical section. Every time when data reader get to the critical section, it calls the `rcu_read_lock`, and `rcu_read_unlock` on exit from the critical section. If the thread is not in the critical section, it will be in state which called - `quiescent state`. The moment when every thread is in the `quiescent state` called - `grace period`. If a thread wants to remove an element from the data structure, this occurs in two steps. First step is `removal` - atomically removes element from the data structure, but does not release the physical memory. After this thread-writer announces and waits until it is finished. From this moment, the removed element is available to the thread-readers. After the `grace period` finished, the second step of the element removal will be started, it just removes the element from the physical memory.

There a couple of implementations of the `RCU`. Old `RCU` called classic, the new implementation called `tree` RCU. As you may already understand, the `CONFIG_TREE_RCU` kernel configuration option enables tree `RCU`. Another is the `tiny` RCU which depends on `CONFIG_TINY_RCU` and `CONFIG_SMP=n`. We will see more details about the `RCU` in general in the separate chapter about synchronization primitives, but now let's look on the `rcu_init` implementation from the kernel/rcu/tree.c:

```
void __init rcu_init(void)
{
        int cpu;

        rcu_bootup_announce();
        rcu_init_geometry();
        rcu_init_one(&rcu_bh_state, &rcu_bh_data);
        rcu_init_one(&rcu_sched_state, &rcu_sched_data);
        __rcu_init_preempt();
        open_softirq(RCU_SOFTIRQ, rcu_process_callbacks);

        /*
         * We don't need protection against CPU-hotplug here because
         * this is called early in boot, before either interrupts
         * or the scheduler are operational.
         */
        cpu_notifier(rcu_cpu_notify, 0);
        pm_notifier(rcu_pm_notify, 0);
```

```
        for_each_online_cpu(cpu)
                rcu_cpu_notify(NULL, CPU_UP_PREPARE, (void *)(long)cpu);

        rcu_early_boot_tests();
}
```

In the beginning of the `rcu_init` function we define `cpu` variable and call `rcu_bootup_announce` . The `rcu_bootup_announce` function is pretty simple:

```
static void __init rcu_bootup_announce(void)
{
        pr_info("Hierarchical RCU implementation.\n");
        rcu_bootup_announce_oddness();
}
```

It just prints information about the `RCU` with the `pr_info` function and `rcu_bootup_announce_oddness` which uses `pr_info` too, for printing different information about the current `RCU` configuration which depends on different kernel configuration options like `CONFIG_RCU_TRACE` , `CONFIG_PROVE_RCU` , `CONFIG_RCU_FANOUT_EXACT` , etc. In the next step, we can see the call of the `rcu_init_geometry` function. This function is defined in the same source code file and computes the node tree geometry depends on the amount of CPUs. Actually `RCU` provides scalability with extremely low internal RCU lock contention. What if a data structure will be read from the different CPUs? `RCU` API provides the `rcu_state` structure which presents RCU global state including node hierarchy. Hierarchy is presented by the:

```
struct rcu_node node[NUM_RCU_NODES];
```

array of structures. As we can read in the comment of above definition:

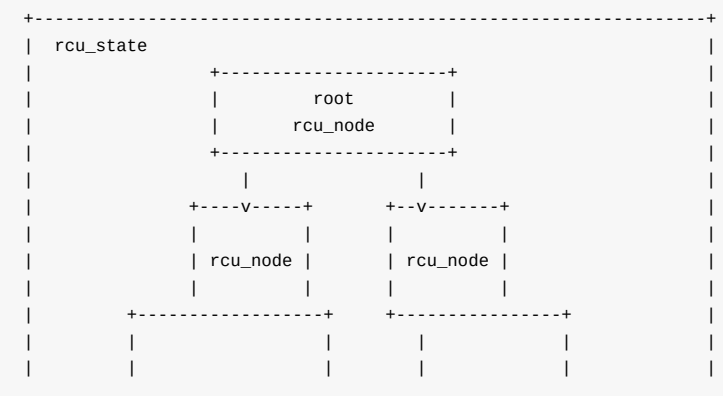
```
The root (first level) of the hierarchy is in ->node[0] (referenced by ->level[0]), the second
level in ->node[1] through ->node[m] (->node[1] referenced by ->level[1]), and the third level
in ->node[m+1] and following (->node[m+1] referenced by ->level[2]).  The number of levels is
determined by the number of CPUs and by CONFIG_RCU_FANOUT.

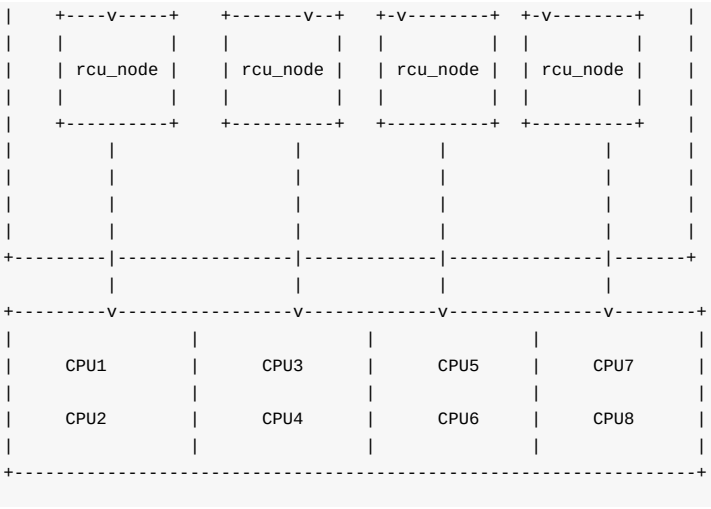
Small systems will have a "hierarchy" consisting of a single rcu_node.
```

The `rcu_node` structure is defined in the kernel/rcu/tree.h and contains information about current grace period, is grace period completed or not, CPUs or groups that need to switch in order for current grace period to proceed, etc. Every `rcu_node` contains a lock for a couple of CPUs. These `rcu_node` structures are embedded into a linear array in the `rcu_state` structure and represented as a tree with the root as the first element and covers all CPUs. As you can see the number of the rcu nodes determined by the `NUM_RCU_NODES` which depends on number of available CPUs:

```
#define NUM_RCU_NODES (RCU_SUM - NR_CPUS)
#define RCU_SUM (NUM_RCU_LVL_0 + NUM_RCU_LVL_1 + NUM_RCU_LVL_2 + NUM_RCU_LVL_3 + NUM_RCU_LVL_4)
```

where levels values depend on the `CONFIG_RCU_FANOUT_LEAF` configuration option. For example for the simplest case, one `rcu_node` will cover two CPU on machine with the eight CPUs:





So, in the `rcu_init_geometry` function we just need to calculate the total number of `rcu_node` structures. We start to do it with the calculation of the `jiffies` till to the first and next `fqs` which is `force-quiescent-state` (read above about it):

```
d = RCU_JIFFIES_TILL_FORCE_QS + nr_cpu_ids / RCU_JIFFIES_FQS_DIV;
if (jiffies_till_first_fqs == ULONG_MAX)
        jiffies_till_first_fqs = d;
if (jiffies_till_next_fqs == ULONG_MAX)
        jiffies_till_next_fqs = d;
```

where:

```
#define RCU_JIFFIES_TILL_FORCE_QS (1 + (HZ > 250) + (HZ > 500))
#define RCU_JIFFIES_FQS_DIV     256
```

As we calculated these jiffies, we check that previous defined `jiffies_till_first_fqs` and `jiffies_till_next_fqs` variables are equal to the ULONG_MAX (their default values) and set they equal to the calculated value. As we did not touch these variables before, they are equal to the `ULONG_MAX` :

```
static ulong jiffies_till_first_fqs = ULONG_MAX;
static ulong jiffies_till_next_fqs = ULONG_MAX;
```

In the next step of the `rcu_init_geometry`, we check that `rcu_fanout_leaf` didn't change (it has the same value as `CONFIG_RCU_FANOUT_LEAF` in compile-time) and equal to the value of the `CONFIG_RCU_FANOUT_LEAF` configuration option, we just return:

```
if (rcu_fanout_leaf == CONFIG_RCU_FANOUT_LEAF &&
    nr_cpu_ids == NR_CPUS)
    return;
```

After this we need to compute the number of nodes that an `rcu_node` tree can handle with the given number of levels:

```
rcu_capacity[0] = 1;
rcu_capacity[1] = rcu_fanout_leaf;
for (i = 2; i <= MAX_RCU_LVLS; i++)
    rcu_capacity[i] = rcu_capacity[i - 1] * CONFIG_RCU_FANOUT;
```

And in the last step we calculate the number of rcu_nodes at each level of the tree in the loop.

As we calculated geometry of the `rcu_node` tree, we need to go back to the `rcu_init` function and next step we need to initialize two `rcu_state` structures with the `rcu_init_one` function:

```
rcu_init_one(&rcu_bh_state, &rcu_bh_data);
```

```
rcu_init_one(&rcu_sched_state, &rcu_sched_data);
```

The `rcu_init_one` function takes two arguments:

- Global `RCU` state;
- Per-CPU data for `RCU` .

Both variables defined in the kernel/rcu/tree.h with its `percpu` data:

```
extern struct rcu_state rcu_bh_state;
DECLARE_PER_CPU(struct rcu_data, rcu_bh_data);
```

About this states you can read here. As I wrote above we need to initialize `rcu_state` structures and `rcu_init_one` function will help us with it. After the `rcu_state` initialization, we can see the call of the `__rcu_init_preempt` which depends on the `CONFIG_PREEMPT_RCU` kernel configuration option. It does the same as previous functions - initialization of the `rcu_preempt_state` structure with the `rcu_init_one` function which has `rcu_state` type. After this, in the `rcu_init` , we can see the call of the:

```
open_softirq(RCU_SOFTIRQ, rcu_process_callbacks);
```

function. This function registers a handler of the `pending interrupt` . Pending interrupt or `softirq` supposes that part of actions can be delayed for later execution when the system is less loaded. Pending interrupts is represented by the following structure:

```
struct softirq_action
{
        void    (*action)(struct softirq_action *);
};
```

which is defined in the include/linux/interrupt.h and contains only one field - handler of an interrupt. You can check about `softirqs` in the your system with the:

```
$ cat /proc/softirqs
                    CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6       CPU7
          HI:          2          0          0          1          0          2          0          0
       TIMER:     137779     108110     139573     107647     107408     114972      99653      98665
      NET_TX:       1127          0          4          0          1          1          0          0
      NET_RX:        334        221     132939       3076        451        361        292        303
       BLOCK:       5253       5596          8        779       2016      37442         28       2855
BLOCK_IOPOLL:          0          0          0          0          0          0          0          0
     TASKLET:         66          0       2916        113          0         24      26708          0
       SCHED:     102350      75950      91705      75356      75323      82627      69279      69914
     HRTIMER:        510        302        368        260        219        255        248        246
         RCU:      81290      68062      82979      69015      68390      69385      63304      63473
```

The `open_softirq` function takes two parameters:

- index of the interrupt;
- interrupt handler.

and adds interrupt handler to the array of the pending interrupts:

```
void open_softirq(int nr, void (*action)(struct softirq_action *))
{
        softirq_vec[nr].action = action;
}
```

In our case the interrupt handler is - `rcu_process_callbacks` which is defined in the kernel/rcu/tree.c and does the `RCU` core processing for the current CPU. After we registered `softirq` interrupt for the `RCU` , we can see the following code:

```
cpu_notifier(rcu_cpu_notify, 0);
pm_notifier(rcu_pm_notify, 0);
```

```
for_each_online_cpu(cpu)
    rcu_cpu_notify(NULL, CPU_UP_PREPARE, (void *)(long)cpu);
```

Here we can see registration of the `cpu` notifier which needs in systems which supports CPU hotplug and we will not dive into details about this theme. The last function in the `rcu_init` is the `rcu_early_boot_tests`:

```
void rcu_early_boot_tests(void)
{
        pr_info("Running RCU self tests\n");

        if (rcu_self_test)
                early_boot_test_call_rcu();
         if (rcu_self_test_bh)
                early_boot_test_call_rcu_bh();
         if (rcu_self_test_sched)
               early_boot_test_call_rcu_sched();
}
```

which runs self tests for the `RCU`.

That's all. We saw initialization process of the `RCU` subsystem. As I wrote above, more about the `RCU` will be in the separate chapter about synchronization primitives.

Rest of the initialization process

Ok, we already passed the main theme of this part which is `RCU` initialization, but it is not the end of the linux kernel initialization process. In the last paragraph of this theme we will see a couple of functions which work in the initialization time, but we will not dive into deep details around this function for different reasons. Some reasons not to dive into details are following:

- They are not very important for the generic kernel initialization process and depend on the different kernel configuration;
- They have the character of debugging and not important for now;
- We will see many of this stuff in the separate parts/chapters.

After we initialized `RCU`, the next step which you can see in the init/main.c is the - `trace_init` function. As you can understand from its name, this function initialize tracing subsystem. You can read more about linux kernel trace system - here.

After the `trace_init`, we can see the call of the `radix_tree_init`. If you are familiar with the different data structures, you can understand from the name of this function that it initializes kernel implementation of the Radix tree. This function is defined in the lib/radix-tree.c and you can read more about it in the part about Radix tree.

In the next step we can see the functions which are related to the `interrupts handling` subsystem, they are:

- `early_irq_init`
- `init_IRQ`
- `softirq_init`

We will see explanation about this functions and their implementation in the special part about interrupts and exceptions handling. After this many different functions (like `init_timers`, `hrtimers_init`, `time_init`, etc.) which are related to different timing and timers stuff. We will see more about these function in the chapter about timers.

The next couple of functions are related with the perf events - `perf_event-init` (there will be separate chapter about perf), initialization of the `profiling` with the `profile_init`. After this we enable `irq` with the call of the:

```
local_irq_enable();
```

which expands to the `sti` instruction and making post initialization of the SLAB with the call of the `kmem_cache_init_late` function (As I wrote above we will know about the `SLAB` in the Linux memory management chapter).

After the post initialization of the `SLAB`, next point is initialization of the console with the `console_init` function from the drivers/tty/tty_io.c.

After the console initialization, we can see the `lockdep_info` function which prints information about the Lock dependency validator. After this, we can see the initialization of the dynamic allocation of the `debug objects` with the `debug_objects_mem_init`, kernel memory leak detector initialization with the `kmemleak_init`, `percpu` pageset setup with the `setup_per_cpu_pageset`, setup of the NUMA policy with the `numa_policy_init`, setting time for the scheduler with the `sched_clock_init`, `pidmap` initialization with the call of the `pidmap_init` function for the initial `PID` namespace, cache creation with the `anon_vma_init` for the private virtual memory areas and early initialization of the ACPI with the `acpi_early_init`.

This is the end of the ninth part of the linux kernel initialization process and here we saw initialization of the RCU. In the last paragraph of this part (`Rest of the initialization process`) we will go through many functions but did not dive into details about their implementations. Do not worry if you do not know anything about these stuff or you know and do not understand anything about this. As I already wrote many times, we will see details of implementations in other parts or other chapters.

Conclusion

It is the end of the ninth part about the linux kernel initialization process. In this part, we looked on the initialization process of the `RCU` subsystem. In the next part we will continue to dive into linux kernel initialization process and I hope that we will finish with the `start_kernel` function and will go to the `rest_init` function from the same init/main.c source code file and will see the start of the first process.

If you have any questions or suggestions write me a comment or ping me at twitter.

Please note that English is not my first language, And I am really sorry for any inconvenience. If you find any mistakes please send me PR to linux-insides.

Links

- lock-free data structures
- kmemleak
- ACPI
- IRQs
- RCU
- RCU documentation
- integer ID management
- Documentation/memory-barriers.txt
- Runtime locking correctness validator
- Per-CPU variables
- Linux kernel memory management
- slab
- i2c
- Previous part

Kernel initialization. Part 10.

End of the linux kernel initialization process

This is tenth part of the chapter about linux kernel initialization process and in the previous part we saw the initialization of the RCU and stopped on the call of the `acpi_early_init` function. This part will be the last part of the Kernel initialization process chapter, so let's finish it.

After the call of the `acpi_early_init` function from the init/main.c, we can see the following code:

```
#ifdef CONFIG_X86_ESPFIX64
    init_espfix_bsp();
#endif
```

Here we can see the call of the `init_espfix_bsp` function which depends on the `CONFIG_X86_ESPFIX64` kernel configuration option. As we can understand from the function name, it does something with the stack. This function is defined in the arch/x86/kernel/espfix_64.c and prevents leaking of `31:16` bits of the `esp` register during returning to 16-bit stack. First of all we install `espfix` page upper directory into the kernel page directory in the `init_espfix_bs` :

```
pgd_p = &init_level4_pgt[pgd_index(ESPFIX_BASE_ADDR)];
pgd_populate(&init_mm, pgd_p, (pud_t *)espfix_pud_page);
```

Where `ESPFIX_BASE_ADDR` is:

```
#define PGDIR_SHIFT     39
#define ESPFIX_PGD_ENTRY _AC(-2, UL)
#define ESPFIX_BASE_ADDR (ESPFIX_PGD_ENTRY << PGDIR_SHIFT)
```

Also we can find it in the Documentation/x86/x86_64/mm:

```
... unused hole ...
ffffff0000000000 - ffffff7fffffffff (=39 bits) %esp fixup stacks
... unused hole ...
```

After we've filled page global directory with the `espfix` pud, the next step is call of the `init_espfix_random` and `init_espfix_ap` functions. The first function returns random locations for the `espfix` page and the second enables the `espfix` for the current CPU. After the `init_espfix_bsp` finished the work, we can see the call of the `thread_info_cache_init` function which defined in the kernel/fork.c and allocates cache for the `thread_info` if `THREAD_SIZE` is less than `PAGE_SIZE` :

```
# if THREAD_SIZE >= PAGE_SIZE
...
...
...
void thread_info_cache_init(void)
{
        thread_info_cache = kmem_cache_create("thread_info", THREAD_SIZE,
                                              THREAD_SIZE, 0, NULL);
        BUG_ON(thread_info_cache == NULL);
}
...
...
...
#endif
```

As we already know the `PAGE_SIZE` is `(_AC(1,UL) << PAGE_SHIFT)` or `4096` bytes and `THREAD_SIZE` is `(PAGE_SIZE << THREAD_SIZE_ORDER)` or `16384` bytes for the `x86_64`. The next function after the `thread_info_cache_init` is the `cred_init` from the kernel/cred.c. This function just allocates cache for the credentials (like `uid`, `gid`, etc.):

```
void __init cred_init(void)
{
         cred_jar = kmem_cache_create("cred_jar", sizeof(struct cred),
                                      0, SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL);
}
```

more about credentials you can read in the Documentation/security/credentials.txt. Next step is the `fork_init` function from the kernel/fork.c. The `fork_init` function allocates cache for the `task_struct`. Let's look on the implementation of the `fork_init`. First of all we can see definitions of the `ARCH_MIN_TASKALIGN` macro and creation of a slab where task_structs will be allocated:

```
#ifndef CONFIG_ARCH_TASK_STRUCT_ALLOCATOR
#ifndef ARCH_MIN_TASKALIGN
#define ARCH_MIN_TASKALIGN      L1_CACHE_BYTES
#endif
        task_struct_cachep =
                kmem_cache_create("task_struct", sizeof(struct task_struct),
                        ARCH_MIN_TASKALIGN, SLAB_PANIC | SLAB_NOTRACK, NULL);
#endif
```

As we can see this code depends on the `CONFIG_ARCH_TASK_STRUCT_ACLLOCATOR` kernel configuration option. This configuration option shows the presence of the `alloc_task_struct` for the given architecture. As `x86_64` has no `alloc_task_struct` function, this code will not work and even will not be compiled on the `x86_64`.

Allocating cache for init task

After this we can see the call of the `arch_task_cache_init` function in the `fork_init`:

```
void arch_task_cache_init(void)
{
        task_xstate_cachep =
                kmem_cache_create("task_xstate", xstate_size,
                                  __alignof__(union thread_xstate),
                                  SLAB_PANIC | SLAB_NOTRACK, NULL);
        setup_xstate_comp();
}
```

The `arch_task_cache_init` does initialization of the architecture-specific caches. In our case it is `x86_64`, so as we can see, the `arch_task_cache_init` allocates cache for the `task_xstate` which represents FPU state and sets up offsets and sizes of all extended states in xsave area with the call of the `setup_xstate_comp` function. After the `arch_task_cache_init` we calculate default maximum number of threads with the:

```
set_max_threads(MAX_THREADS);
```

where default maximum number of threads is:

```
#define FUTEX_TID_MASK  0x3fffffff
#define MAX_THREADS     FUTEX_TID_MASK
```

In the end of the `fork_init` function we initialize signal handler:

```
init_task.signal->rlim[RLIMIT_NPROC].rlim_cur = max_threads/2;
init_task.signal->rlim[RLIMIT_NPROC].rlim_max = max_threads/2;
init_task.signal->rlim[RLIMIT_SIGPENDING] =
        init_task.signal->rlim[RLIMIT_NPROC];
```

As we know the `init_task` is an instance of the `task_struct` structure, so it contains `signal` field which represents signal handler. It has following type `struct signal_struct` . On the first two lines we can see setting of the current and maximum limit of the `resource limits` . Every process has an associated set of resource limits. These limits specify amount of resources which current process can use. Here `rlim` is resource control limit and presented by the:

```
struct rlimit {
        __kernel_ulong_t        rlim_cur;
        __kernel_ulong_t        rlim_max;
};
```

structure from the include/uapi/linux/resource.h. In our case the resource is the `RLIMIT_NPROC` which is the maximum number of processes that user can own and `RLIMIT_SIGPENDING` - the maximum number of pending signals. We can see it in the:

```
cat /proc/self/limits
Limit                     Soft Limit           Hard Limit           Units
...
...
...
Max processes             63815                63815                processes
Max pending signals       63815                63815                signals
...
...
...
```

Initialization of the caches

The next function after the `fork_init` is the `proc_caches_init` from the kernel/fork.c. This function allocates caches for the memory descriptors (or `mm_struct` structure). At the beginning of the `proc_caches_init` we can see allocation of the different SLAB caches with the call of the `kmem_cache_create` :

- `sighand_cachep` - manage information about installed signal handlers;
- `signal_cachep` - manage information about process signal descriptor;
- `files_cachep` - manage information about opened files;
- `fs_cachep` - manage filesystem information.

After this we allocate `SLAB` cache for the `mm_struct` structures:

```
mm_cachep = kmem_cache_create("mm_struct",
                        sizeof(struct mm_struct), ARCH_MIN_MMSTRUCT_ALIGN,
                        SLAB_HWCACHE_ALIGN|SLAB_PANIC|SLAB_NOTRACK, NULL);
```

After this we allocate `SLAB` cache for the important `vm_area_struct` which used by the kernel to manage virtual memory space:

```
vm_area_cachep = KMEM_CACHE(vm_area_struct, SLAB_PANIC);
```

Note, that we use `KMEM_CACHE` macro here instead of the `kmem_cache_create` . This macro is defined in the include/linux/slab.h and just expands to the `kmem_cache_create` call:

```
#define KMEM_CACHE(__struct, __flags) kmem_cache_create(#__struct,\
                sizeof(struct __struct), __alignof__(struct __struct),\
                (__flags), NULL)
```

The `KMEM_CACHE` has one difference from `kmem_cache_create` . Take a look on `__alignof__` operator. The `KMEM_CACHE` macro aligns `SLAB` to the size of the given structure, but `kmem_cache_create` uses given value to align space. After this we can see the call of the `mmap_init` and `nsproxy_cache_init` functions. The first function initializes virtual memory area `SLAB` and the second function initializes `SLAB` for namespaces.

The next function after the `proc_caches_init` is `buffer_init`. This function is defined in the fs/buffer.c source code file and allocate cache for the `buffer_head`. The `buffer_head` is a special structure which defined in the include/linux/buffer_head.h and used for managing buffers. In the start of the `buffer_init` function we allocate cache for the `struct buffer_head` structures with the call of the `kmem_cache_create` function as we did in the previous functions. And calculate the maximum size of the buffers in memory with:

```
nrpages = (nr_free_buffer_pages() * 10) / 100;
max_buffer_heads = nrpages * (PAGE_SIZE / sizeof(struct buffer_head));
```

which will be equal to the `10%` of the `ZONE_NORMAL` (all RAM from the 4GB on the `x86_64`). The next function after the `buffer_init` is - `vfs_caches_init`. This function allocates `SLAB` caches and hashtable for different VFS caches. We already saw the `vfs_caches_init_early` function in the eighth part of the linux kernel initialization process which initialized caches for `dcache` (or directory-cache) and inode cache. The `vfs_caches_init` function makes post-early initialization of the `dcache` and `inode` caches, private data cache, hash tables for the mount points, etc. More details about VFS will be described in the separate part. After this we can see `signals_init` function. This function is defined in the kernel/signal.c and allocates a cache for the `sigqueue` structures which represents queue of the real time signals. The next function is `page_writeback_init`. This function initializes the ratio for the dirty pages. Every low-level page entry contains the `dirty` bit which indicates whether a page has been written to after been loaded into memory.

Creation of the root for the procfs

After all of this preparations we need to create the root for the proc filesystem. We will do it with the call of the `proc_root_init` function from the fs/proc/root.c. At the start of the `proc_root_init` function we allocate the cache for the inodes and register a new filesystem in the system with the:

```
err = register_filesystem(&proc_fs_type);
      if (err)
                return;
```

As I wrote above we will not dive into details about VFS and different filesystems in this chapter, but will see it in the chapter about the `VFS`. After we've registered a new filesystem in our system, we call the `proc_self_init` function from the fs/proc/self.c and this function allocates `inode` number for the `self` (`/proc/self` directory refers to the process accessing the `/proc` filesystem). The next step after the `proc_self_init` is `proc_setup_thread_self` which setups the `/proc/thread-self` directory which contains information about current thread. After this we create `/proc/self/mounts` symlink which will contains mount points with the call of the

```
proc_symlink("mounts", NULL, "self/mounts");
```

and a couple of directories depends on the different configuration options:

```
#ifdef CONFIG_SYSVIPC
        proc_mkdir("sysvipc", NULL);
#endif
        proc_mkdir("fs", NULL);
        proc_mkdir("driver", NULL);
        proc_mkdir("fs/nfsd", NULL);
#if defined(CONFIG_SUN_OPENPROMFS) || defined(CONFIG_SUN_OPENPROMFS_MODULE)
        proc_mkdir("openprom", NULL);
#endif
        proc_mkdir("bus", NULL);
        ...
        ...
        ...
        if (!proc_mkdir("tty", NULL))
                return;
        proc_mkdir("tty/ldisc", NULL);
        ...
        ...
        ...
```

In the end of the `proc_root_init` we call the `proc_sys_init` function which creates `/proc/sys` directory and initializes the Sysctl.

It is the end of `start_kernel` function. I did not describe all functions which are called in the `start_kernel`. I skipped them, because they are not important for the generic kernel initialization stuff and depend on only different kernel configurations. They are `taskstats_init_early` which exports per-task statistic to the user-space, `delayacct_init` - initializes per-task delay accounting, `key_init` and `security_init` initialize different security stuff, `check_bugs` - fix some architecture-dependent bugs, `ftrace_init` function executes initialization of the ftrace, `cgroup_init` makes initialization of the rest of the cgroup subsystem,etc. Many of these parts and subsystems will be described in the other chapters.

That's all. Finally we have passed through the long-long `start_kernel` function. But it is not the end of the linux kernel initialization process. We haven't run the first process yet. In the end of the `start_kernel` we can see the last call of the - `rest_init` function. Let's go ahead.

First steps after the start_kernel

The `rest_init` function is defined in the same source code file as `start_kernel` function, and this file is init/main.c. In the beginning of the `rest_init` we can see call of the two following functions:

```
    rcu_scheduler_starting();
    smpboot_thread_init();
```

The first `rcu_scheduler_starting` makes RCU scheduler active and the second `smpboot_thread_init` registers the `smpboot_thread_notifier` CPU notifier (more about it you can read in the CPU hotplug documentation. After this we can see the following calls:

```
kernel_thread(kernel_init, NULL, CLONE_FS);
pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
```

Here the `kernel_thread` function (defined in the kernel/fork.c) creates new kernel thread.As we can see the `kernel_thread` function takes three arguments:

- Function which will be executed in a new thread;
- Parameter for the `kernel_init` function;
- Flags.

We will not dive into details about `kernel_thread` implementation (we will see it in the chapter which describe scheduler, just need to say that `kernel_thread` invokes clone). Now we only need to know that we create new kernel thread with `kernel_thread` function, parent and child of the thread will use shared information about filesystem and it will start to execute `kernel_init` function. A kernel thread differs from a user thread that it runs in kernel mode. So with these two `kernel_thread` calls we create two new kernel threads with the `PID = 1` for `init` process and `PID = 2` for `kthreadd`. We already know what is `init` process. Let's look on the `kthreadd`. It is a special kernel thread which manages and helps different parts of the kernel to create another kernel thread. We can see it in the output of the `ps` util:

```
$ ps -ef | grep kthread
root         2     0  0 Jan11 ?        00:00:00 [kthreadd]
```

Let's postpone `kernel_init` and `kthreadd` for now and go ahead in the `rest_init`. In the next step after we have created two new kernel threads we can see the following code:

```
    rcu_read_lock();
    kthreadd_task = find_task_by_pid_ns(pid, &init_pid_ns);
    rcu_read_unlock();
```

The first `rcu_read_lock` function marks the beginning of an RCU read-side critical section and the `rcu_read_unlock` marks the end of an RCU read-side critical section. We call these functions because we need to protect the `find_task_by_pid_ns`. The `find_task_by_pid_ns` returns pointer to the `task_struct` by the given pid. So, here we are getting the pointer to the `task_struct` for `PID = 2` (we got it after `kthreadd` creation with the `kernel_thread`). In the next step we call `complete` function

```
complete(&kthreadd_done);
```

and pass address of the `kthreadd_done`. The `kthreadd_done` defined as

```
static __initdata DECLARE_COMPLETION(kthreadd_done);
```

where `DECLARE_COMPLETION` macro defined as:

```
#define DECLARE_COMPLETION(work) \
        struct completion work = COMPLETION_INITIALIZER(work)
```

and expands to the definition of the `completion` structure. This structure is defined in the include/linux/completion.h and presents `completions` concept. Completions is a code synchronization mechanism which provides race-free solution for the threads that must wait for some process to have reached a point or a specific state. Using completions consists of three parts: The first is definition of the `complete` structure and we did it with the `DECLARE_COMPLETION`. The second is call of the `wait_for_completion`. After the call of this function, a thread which called it will not continue to execute and will wait while other thread did not call `complete` function. Note that we call `wait_for_completion` with the `kthreadd_done` in the beginning of the `kernel_init_freeable`:

```
wait_for_completion(&kthreadd_done);
```

And the last step is to call `complete` function as we saw it above. After this the `kernel_init_freeable` function will not be executed while `kthreadd` thread will not be set. After the `kthreadd` was set, we can see three following functions in the `rest_init`:

```
    init_idle_bootup_task(current);
    schedule_preempt_disabled();
    cpu_startup_entry(CPUHP_ONLINE);
```

The first `init_idle_bootup_task` function from the kernel/sched/core.c sets the Scheduling class for the current process (`idle` class in our case):

```
void init_idle_bootup_task(struct task_struct *idle)
{
        idle->sched_class = &idle_sched_class;
}
```

where `idle` class is a low task priority and tasks can be run only when the processor doesn't have anything to run besides this tasks. The second function `schedule_preempt_disabled` disables preempt in `idle` tasks. And the third function `cpu_startup_entry` is defined in the kernel/sched/idle.c and calls `cpu_idle_loop` from the kernel/sched/idle.c. The `cpu_idle_loop` function works as process with `PID = 0` and works in the background. Main purpose of the `cpu_idle_loop` is to consume the idle CPU cycles. When there is no process to run, this process starts to work. We have one process with `idle` scheduling class (we just set the `current` task to the `idle` with the call of the `init_idle_bootup_task` function), so the `idle` thread does not do useful work but just checks if there is an active task to switch to:

```
static void cpu_idle_loop(void)
{
        ...
        ...
        ...
        while (1) {
                while (!need_resched()) {
                ...
```

```
                ...
                ...
                }
        ...
        }
```

More about it will be in the chapter about scheduler. So for this moment the `start_kernel` calls the `rest_init` function which spawns an `init` (`kernel_init` function) process and become `idle` process itself. Now is time to look on the `kernel_init` . Execution of the `kernel_init` function starts from the call of the `kernel_init_freeable` function. The `kernel_init_freeable` function first of all waits for the completion of the `kthreadd` setup. I already wrote about it above:

```
wait_for_completion(&kthreadd_done);
```

After this we set `gfp_allowed_mask` to `__GFP_BITS_MASK` which means that system is already running, set allowed cpus/mems to all CPUs and NUMA nodes with the `set_mems_allowed` function, allow `init` process to run on any CPU with the `set_cpus_allowed_ptr` , set pid for the `cad` or `Ctrl-Alt-Delete` , do preparation for booting of the other CPUs with the call of the `smp_prepare_cpus` , call early initcalls with the `do_pre_smp_initcalls` , initialize `SMP` with the `smp_init` and initialize lockup_detector with the call of the `lockup_detector_init` and initialize scheduler with the `sched_init_smp` .

After this we can see the call of the following functions - `do_basic_setup` . Before we will call the `do_basic_setup` function, our kernel already initialized for this moment. As comment says:

```
Now we can finally start doing some real work..
```

The `do_basic_setup` will reinitialize cpuset to the active CPUs, initialize the `khelper` - which is a kernel thread which used for making calls out to userspace from within the kernel, initialize tmpfs, initialize `drivers` subsystem, enable the user-mode helper `workqueue` and make post-early call of the `initcalls` . We can see opening of the `dev/console` and dup twice file descriptors from `0` to `2` after the `do_basic_setup` :

```
if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) < 0)
    pr_err("Warning: unable to open an initial console.\n");

(void) sys_dup(0);
(void) sys_dup(0);
```

We are using two system calls here `sys_open` and `sys_dup` . In the next chapters we will see explanation and implementation of the different system calls. After we opened initial console, we check that `rdinit=` option was passed to the kernel command line or set default path of the ramdisk:

```
if (!ramdisk_execute_command)
    ramdisk_execute_command = "/init";
```

Check user's permissions for the `ramdisk` and call the `prepare_namespace` function from the init/do_mounts.c which checks and mounts the initrd:

```
if (sys_access((const char __user *) ramdisk_execute_command, 0) != 0) {
    ramdisk_execute_command = NULL;
    prepare_namespace();
}
```

This is the end of the `kernel_init_freeable` function and we need return to the `kernel_init` . The next step after the `kernel_init_freeable` finished its execution is the `async_synchronize_full` . This function waits until all asynchronous function calls have been done and after it we will call the `free_initmem` which will release all memory occupied by the initialization stuff which located between `__init_begin` and `__init_end` . After this we protect `.rodata` with the `mark_rodata_ro` and update state of the system from the `SYSTEM_BOOTING` to the

```
system_state = SYSTEM_RUNNING;
```

And tries to run the `init` process:

```
if (ramdisk_execute_command) {
    ret = run_init_process(ramdisk_execute_command);
    if (!ret)
        return 0;
    pr_err("Failed to execute %s (error %d)\n",
           ramdisk_execute_command, ret);
}
```

First of all it checks the `ramdisk_execute_command` which we set in the `kernel_init_freeable` function and it will be equal to the value of the `rdinit=` kernel command line parameters or `/init` by default. The `run_init_process` function fills the first element of the `argv_init` array:

```
static const char *argv_init[MAX_INIT_ARGS+2] = { "init", NULL, };
```

which represents arguments of the `init` program and call `do_execve` function:

```
argv_init[0] = init_filename;
return do_execve(getname_kernel(init_filename),
    (const char __user *const __user *)argv_init,
    (const char __user *const __user *)envp_init);
```

The `do_execve` function is defined in the include/linux/sched.h and runs program with the given file name and arguments. If we did not pass `rdinit=` option to the kernel command line, kernel starts to check the `execute_command` which is equal to value of the `init=` kernel command line parameter:

```
    if (execute_command) {
        ret = run_init_process(execute_command);
        if (!ret)
            return 0;
        panic("Requested init %s failed (error %d).",
              execute_command, ret);
    }
```

If we did not pass `init=` kernel command line parameter either, kernel tries to run one of the following executable files:

```
if (!try_to_run_init_process("/sbin/init") ||
    !try_to_run_init_process("/etc/init") ||
    !try_to_run_init_process("/bin/init") ||
    !try_to_run_init_process("/bin/sh"))
    return 0;
```

Otherwise we finish with panic:

```
panic("No working init found.  Try passing init= option to kernel. "
      "See Linux Documentation/init.txt for guidance.");
```

That's all! Linux kernel initialization process is finished!

Conclusion

It is the end of the tenth part about the linux kernel initialization process. It is not only the `tenth` part, but also is the last part which describes initialization of the linux kernel. As I wrote in the first part of this chapter, we will go through all steps of the kernel initialization and we did it. We started at the first architecture-independent function - `start_kernel` and finished with the launch of the

first `init` process in the our system. I skipped details about different subsystem of the kernel, for example I almost did not cover scheduler, interrupts, exception handling, etc. From the next part we will start to dive to the different kernel subsystems. Hope it will be interesting.

If you have any questions or suggestions write me a comment or ping me at twitter.

Please note that English is not my first language, And I am really sorry for any inconvenience. If you find any mistakes please send me PR to linux-insides.

Links

- SLAB
- xsave
- FPU
- Documentation/security/credentials.txt
- Documentation/x86/x86_64/mm
- RCU
- VFS
- inode
- proc
- man proc
- Sysctl
- ftrace
- cgroup
- CPU hotplug documentation
- completions - wait for completion handling
- NUMA
- cpus/mems
- initcalls
- Tmpfs
- initrd
- panic
- Previous part

linux

- -
- Linux -
- -
- - fourth part describes first non-early interrupt handlers.
- -
- -
- -
- IRQs -
- Softirq, Tasklets and Workqueues - softirqstasklets workqueues
- -

Part 1.

Introduction

linux `init` interrupts

`interrupts`

- `interrupts`
- `interrupt handlers`

Linux

CPU `event` CPU CPU CPU PIC Advanced Programmable Interrupt Controller `APIC` APIC

- `Local APIC`
- `I/O APIC`

- `Local APIC` CPULocal APIC CPU Local APIC APIC APIC-timer I/O

- `I/O APIC` CPU local I/O APIC

- ()
- ()
-

3

`Interrupt Descriptor Table` `IDT` `vector number` `IDT` `0` `255` Linux

```
BUG_ON((unsigned)n > 0xFF);
```

Linux (`set_intr_gate` , `void set_system_intr_gate` arch/x86/include/asm/desc.h) `0` `31` 32 Linux - `32` `255` I/O

-

-

- `Local APIC` `Local APIC` - () `syscall`

CPU `synchronous` 3

- `Faults`
- `Traps`
- `Aborts`

“”

`maskable` `non-maskable` `x86_64` - `sti` `cli` Linux

```
static inline void native_irq_disable(void)
{
        asm volatile("cli": : :"memory");
}
```

and

```
static inline void native_irq_enable(void)
{
        asm volatile("sti": : :"memory");
}
```

`IF` `sti` `IF` `cli`

```
+---------------------------------------------------------------+
|              |                                                |
|   Priority   | Description                                    |
|              |                                                |
+--------------+------------------------------------------------+
|              | Hardware Reset and Machine Checks              |
|     1        | - RESET                                        |
|              | - Machine Check                                |
+--------------+------------------------------------------------+
|              | Trap on Task Switch                            |
|     2        | - T flag in TSS is set                         |
|              |                                                |
+--------------+------------------------------------------------+
|              | External Hardware Interventions                |
|              | - FLUSH                                        |
|     3        | - STOPCLK                                      |
|              | - SMI                                          |
|              | - INIT                                         |
+--------------+------------------------------------------------+
|              | Traps on the Previous Instruction              |
|     4        | - Breakpoints                                  |
|              | - Debug Trap Exceptions                        |
+--------------+------------------------------------------------+
|     5        | Nonmaskable Interrupts                         |
+--------------+------------------------------------------------+
|     6        | Maskable Hardware Interrupts                   |
+--------------+------------------------------------------------+
|     7        | Code Breakpoint Fault                          |
+--------------+------------------------------------------------+
|     8        | Faults from Fetching Next Instruction          |
|              | Code-Segment Limit Violation                   |
|              | Code Page Fault                                |
+--------------+------------------------------------------------+
|              | Faults from Decoding the Next Instruction      |
|              | Instruction length > 15 bytes                  |
|     9        | Invalid Opcode                                 |
|              | Coprocessor Not Available                      |
|              |                                                |
+--------------+------------------------------------------------+
|     10       | Faults on Executing an Instruction             |
|              | Overflow                                       |
|              | Bound error                                    |
|              | Invalid TSS                                    |
|              | Segment Not Present                            |
|              | Stack fault                                    |
|              | General Protection                             |
|              | Data Page Fault                                |
|              | Alignment Check                                |
|              | x87 FPU Floating-point exception               |
|              | SIMD floating-point exception                  |
|              | Virtualization exception                       |
+--------------+------------------------------------------------+
```

`IDT` `IDT` `IDT` `Global Descriptor Table` `IDT` `gates` `descriptors`

- Interrupt gates
- Task gates
- Trap gates

`x86` long mode `x86_64` `x86` 8 `x86_64` 16 `NULL` `NULL`

```
/*
 * Set up the IDT
 */
static void setup_idt(void)
{
    static const struct gdt_ptr null_idt = {0, 0};
    asm volatile("lidtl %0" : : "m" (null_idt));
}
```

arch/x86/boot/pm.c `x86` 8 `x86_64` 16 `IDT` - IDTR `x86` - `IDTR`

- `LIDT`
- `SIDT`

`LIDT` `IDT` `IDTR` `SIDT` `IDTR` `x86` `IDTR` 48

```
+-----------------------------------+----------------------+
|                                   |                      |
|     Base address of the IDT       |   Limit of the IDT   |
|                                   |                      |
+-----------------------------------+----------------------+
47                                16 15                    0
```

`setup_idt` `null_idt` `lidt` `IDTR` `null_idt` `gdt_ptr`

```
struct gdt_ptr {
        u16 len;
        u32 ptr;
} __attribute__((packed));
```

`IDTR` 2 4 48 `IDT` `x86` 16

```
127                                                                     96
+-----------------------------------------------------------------------+
|                                                                       |
|                               Reserved                                |
|                                                                       |
+-----------------------------------------------------------------------
95                                                                      64
+-----------------------------------------------------------------------+
|                                                                       |
|                             Offset 63..32                             |
|                                                                       |
+-----------------------------------------------------------------------+
63                              48 47      46  44   42    39            34    32
+-----------------------------------------------------------------------+
|                                 |       | D  |   |     |      |   |   |     |
|       Offset 31..16             |   P   |  P  | 0 |Type |0 0 0 | 0 | 0 | IST |
|                                 |       | L  |   |     |      |   |   |     |
 -----------------------------------------------------------------------+
31                                16 15                                 0
+-----------------------------------------------------------------------+
|                                   |                                   |
|          Segment Selector         |               Offset 15..0        |
|                                   |                                   |
```

```
+---------------------------------------------------------------------------+
```

IDT 16 `call` `IDT`

`IDT`

- `0-15` bits -
- `16-31` bits -
- `IST` - `x86_64`
- `DPL` -
- `P` -
- `48-63` bits -
- `64-95` bits -
- `96-127` bits - CPU .

`Type` `IDT`

- Interrupt gate
- Trap gate
- Task gate

`IST` `Interrupt Stack Table` `x86_64` `x86` `IST` `x86` `IDT` `IST`
`IST` Task State Segment `TSS` 7 `IST` `TSS` Linux `IDT`

`gate_desc`

```
extern gate_desc idt_table[];
```

`gate_desc`

```
#ifdef CONFIG_X86_64
...
...
...
typedef struct gate_struct64 gate_desc;
...
...
...
#endif
```

`gate_struct64`

```
struct gate_struct64 {
        u16 offset_low;
        u16 segment;
        unsigned ist : 3, zero0 : 5, type : 5, dpl : 2, p : 1;
        u16 offset_middle;
        u32 offset_high;
        u32 zero1;
} __attribute__((packed));
```

`x86_64` Linux `THREAD_SIZE`

```
#define PAGE_SHIFT      12
#define PAGE_SIZE       (_AC(1,UL) << PAGE_SHIFT)
...
...
...
#define THREAD_SIZE_ORDER       (2 + KASAN_STACK_ORDER)
#define THREAD_SIZE  (PAGE_SIZE << THREAD_SIZE_ORDER)
```

`PAGE_SIZE` `4096` `THREAD_SIZE_ORDER` `KASAN_STACK_ORDER` `KASAN_STACK` `CONFIG_KASAN`

```
#ifdef CONFIG_KASAN
    #define KASAN_STACK_ORDER 1
#else
    #define KASAN_STACK_ORDER 0
#endif
```

`KASan` `CONFIG_KASAN` `THREAD_SIZE` `16384` `THREAD_SIZE` `32768` `thread_info`
Linux CPU CPU CPU CPU per-cpu `interrup`

```
#define IRQ_STACK_ORDER (2 + KASAN_STACK_ORDER)
#define IRQ_STACK_SIZE (PAGE_SIZE << IRQ_STACK_ORDER)
```

`16384` Per-cpu `x86_64` `irq_stack_union` :

```
union irq_stack_union {
    char irq_stack[IRQ_STACK_SIZE];

    struct {
        char gs_base[40];
        unsigned long stack_canary;
    };
};
```

`irq_stack` 16KB `irq_stack_union`

- `gs_base` - `irqstack` `gs` `x86_64` per-cpu `per-cpu` stack canary `gs` per-cpu
  `gs` per-cpu Model specific registers - `fs` `gs` Linux `gs`

```
    movl    $MSR_GS_BASE,%ecx
    movl    initial_gs(%rip),%eax
    movl    initial_gs+4(%rip),%edx
    wrmsr
```

`initial_gs` `irq_stack_union` :

```
GLOBAL(initial_gs)
.quad    INIT_PER_CPU_VAR(irq_stack_union)
```

- `stack_canary` - Stack canary `stack protector` `gs_base` 40 `GCC` stack canary `gs`
  `x86_64` `40` `x86` `20`

`irq_stack_union` `percpu` , `System.map`

```
0000000000000000 D __per_cpu_start
0000000000000000 D irq_stack_union
0000000000004000 d exception_stacks
0000000000009000 D gdt_page
...
...
...
```

:

```
DECLARE_PER_CPU_FIRST(union irq_stack_union, irq_stack_union) __visible;
```

`irq_stack_union` `irq_stack_union` arch/x86/include/asm/processor.h per-cpu

```
DECLARE_PER_CPU(char *, irq_stack_ptr);
DECLARE_PER_CPU(unsigned int, irq_count);
```

`irq_stack_ptr` `irq_count` CPU `irq_stack_ptr` arch/x86/kernel/setup_percpu.c `setup_per_cpu_areas`

```
void __init setup_per_cpu_areas(void)
{
...
...
#ifdef CONFIG_X86_64
for_each_possible_cpu(cpu) {
    ...
    ...
    ...
    per_cpu(irq_stack_ptr, cpu) =
            per_cpu(irq_stack_union.irq_stack, cpu) +
            IRQ_STACK_SIZE - 64;
    ...
    ...
    ...
#endif
...
...
}
```

CPU `irq_stack_ptr` `64` `64` TODO [arch/x86/kernel/cpu/common.c]

```
void load_percpu_segment(int cpu)
{
        ...
        ...
        ...
        __loadsegment_simple(gs, 0);
        wrmsrl(MSR_GS_BASE, cpu_kernelmode_gs_base(cpu));
        ...
        load_stack_canary_segment();
}
```

`gs`

```
    movl    $MSR_GS_BASE,%ecx
    movl    initial_gs(%rip),%eax
    movl    initial_gs+4(%rip),%edx
    wrmsr

SYM_DATA(initial_gs,
.quad INIT_PER_CPU_VAR(fixed_percpu_data))
```

`wrmsr` `edx:eax` `ecx` MSR)MSR `MSR_GS_BASE` `gs` `edx:eax` `initial_gs` `fixed_percpu_data`

`x86_64` `Interrupt Stack Table` `IST` 7 `IST` per-cpu ;

- `DOUBLEFAULT_STACK`
- `NMI_STACK`
- `DEBUG_STACK`
- `MCE_STACK`

```
#define DOUBLEFAULT_STACK 1
#define NMI_STACK 2
```

```
#define DEBUG_STACK 3
#define MCE_STACK 4
```

`IST` `set_intr_gate_ist` :

```
static const __initconst struct idt_data def_idts[] = {
    ...
    INTG(X86_TRAP_NMI,        nmi),
    ...
    INTG(X86_TRAP_DF,        double_fault),
```

`&nmi` `&double_fault`

arch/x86/kernel/entry_64.S

```
idtentry double_fault            do_double_fault            has_error_code=1 paranoid=2 read_cr2=1
...
...
...
SYM_CODE_START(nmi)
...
...
...
SYM_CODE_END(nmi)
SYM_CODE_END(nmi)
```

arch/x86/include/asm/traps.h:

```
asmlinkage void nmi(void);
asmlinkage void double_fault(void);
```

`ss` `NULL` `ss` `rpl` `cpl` `ss rsp` `cs rip` 64 8 :

```
+---------------+
|               |
|      SS       | 40
|      RSP      | 32
|     RFLAGS    | 24
|      CS       | 16
|      RIP      | 8
|   Error code  | 0
|               |
+---------------+
```

`IST` `0` `IST` `rsp` CS `21` 64 `L` `Global Descriptor Table`
`rip` `rip` `iret` `iret` `ss:rsp` `cpl`

Linux -

Twitter

PR **linux-insides(PR** **linux-insides-zh)**

- PIC

- Advanced Programmable Interrupt Controller
- protected mode
- long mode
- kernel stacks
- Task State Segement
- segmented memory model
- Model specific registers
- Stack canary
- Previous chapter

Part 2.

Linux

LinuxLinuxLinux LinuxLinux Linux

Linux `x86_64` arch/x86/boot/pm.c (IDT)IDT `go_to_protected_mode` `setup_idt` IDT :

```
void go_to_protected_mode(void)
{
        ...
        setup_idt();
        ...
}
```

`setup_idt` `NULL` :

```
static void setup_idt(void)
{
        static const struct gdt_ptr null_idt = {0, 0};
        asm volatile("lidtl %0" : : "m" (null_idt));
}
```

`gdt_ptr` 48-bit `GDTR` `Global Descriptor` :

```
struct gdt_ptr {
        u16 len;
        u32 ptr;
} __attribute__((packed));
```

`gdt_prt` `GDTR` `IDTR` Linux `idt_ptr` `gdt_prt` `Interrupt Descriptor Table` `NULL` IDT

Interrupt descriptor table, Global Descriptor Table arch/x86/boot/pmjump.S

`boot_params.hdr.code32_start` arch/x86/boot/pm.c `boot_params` `protected_mode_jump` :

```
protected_mode_jump(boot_params.hdr.code32_start,
                            (u32)&boot_params + (ds() << 4));
```

arch/x86/boot/pmjump.S `protected_mode_jump` 8086 `ax` `dx` :

```
GLOBAL(protected_mode_jump)
        ...
        ...
        ...
        .byte   0x66, 0xea              # ljmpl opcode
2:      .long   in_pm32                 # offset
        .word   __BOOT_CS               # segment
...
...
...
ENDPROC(protected_mode_jump)
```

`in_pm32` 32-bit:

```
GLOBAL(in_pm32)
        ...
        ...
```

```
        jmpl    *%eax // %eax contains address of the `startup_32`
        ...
        ...
ENDPROC(in_pm32)
```

32-bit arch/x86/boot/compressed/head_64.S `_64` `arch/x86/boot/compressed`：

- `arch/x86/boot/compressed/head_32.S`.
- `arch/x86/boot/compressed/head_64.S`；

32-bit `x86_64` arch/x86/boot/compressed/Makefile:

```
vmlinux-objs-y := $(obj)/vmlinux.lds $(obj)/head_$(BITS).o $(obj)/misc.o \
...
...
```

`head_*` `$(BITS)` "" arch/x86/Makefile:

```
ifeq ($(CONFIG_X86_32),y)
...
        BITS := 32
else
        BITS := 64
        ...
endif
```

arch/x86/boot/compressed/head_64.S `startup_32` `startup_32` long mode `long mode` `long mode` `startup_64` arch/x86/boot/compressed/misc.c `decompress_kernel` arch/x86/kernel/head_64.S `startup_64` `identity-mapped pages` NX `Extended Feature Enable Register` () `lgdt` `Global Descriptor Table` `gs`：

```
movl    $MSR_GS_BASE,%ecx
movl    initial_gs(%rip),%eax
movl    initial_gs+4(%rip),%edx
wrmsr
```

`wrmsr` `edx:eax` `ecx` model specific register `ecx` `$MSR_GS_BASE` arch/x86/include/uapi/asm/msr-index.h:

```
#define MSR_GS_BASE             0xc0000101
```

`MSR_GS_BASE` `model specific register` `cs`, `ds`, `es`, `ss` 64-bit `fs` `gs` `model specific register` `back door` `fs` `gs` 64-bit `GS.base` `initial_gs`：

```
GLOBAL(initial_gs)
        .quad   INIT_PER_CPU_VAR(irq_stack_union)
```

`irq_stack_union` `INIT_PER_CPU_VAR` `init_per_cpu__` `init_per_cpu__irq_stack_union` ：

```
#define INIT_PER_CPU(x) init_per_cpu__##x = x + __per_cpu_load
INIT_PER_CPU(irq_stack_union);
```

`init_per_cpu__irq_stack_union` `irq_stack_union + __per_cpu_load` `init_per_cpu__irq_stack_union` `__per_cpu_load` `irq_stack_union` arch/x86/include/asm/processor.h `DECLARE_INIT_PER_CPU` `init_per_cpu_var`：

```
DECLARE_INIT_PER_CPU(irq_stack_union);

#define DECLARE_INIT_PER_CPU(var) \
       extern typeof(per_cpu_var(var)) init_per_cpu_var(var)
```

```
#define init_per_cpu_var(var)  init_per_cpu__##var
```

`init_per_cpu__irq_stack_union` `typeof(per_cpu_var(var))` , `var` `irq_stack_union` `per_cpu_var` arch/x86/include/asm/percpu.h:

```
#define PER_CPU_VAR(var)        %__percpu_seg:var
```

:

```
#ifdef CONFIG_X86_64
    #define __percpu_seg gs
endif
```

`gs:irq_stack_union` `irq_union` `__per_cpu_load` include/asm-generic/sections.h `per-cpu` :

```
extern char __per_cpu_load[], __per_cpu_start[], __per_cpu_end[];
```

`irq_stack_union` `__per_cpu_load` `init_per_cpu__irq_stack_union` `__per_cpu_load` System.map:

```
...
...
...
ffffffff819ed000 D __init_begin
ffffffff819ed000 D __per_cpu_load
ffffffff819ed000 A init_per_cpu__irq_stack_union
...
...
...
```

`initial_gs` :

```
movl    $MSR_GS_BASE,%ecx
movl    initial_gs(%rip),%eax
movl    initial_gs+4(%rip),%edx
wrmsr
```

`MSR_GS_BASE` `initial_gs` 64-bit `edx:eax` `wrmsr` `init_per_cpu__irq_stack_union` `gs` `x86_64_start_kernel` C arch/x86/kernel/head64.c `Interrupt Descriptor Table`

```
for (i = 0; i < NUM_EXCEPTION_VECTORS; i++)
        set_intr_gate(i, early_idt_handlers[i]);

load_idt((const struct desc_ptr *)&idt_descr);
```

Linux `3.18` Linux `4.1.0-rc6+` `Andy Lutomirski` `early_idt_handlers` patch **NOTE** patchLinux:

```
for (i = 0; i < NUM_EXCEPTION_VECTORS; i++)
        set_intr_gate(i, early_idt_handler_array[i]);

load_idt((const struct desc_ptr *)&idt_descr);
```

`early_idt_handler_array` :

```
extern const char early_idt_handler_array[NUM_EXCEPTION_VECTORS][EARLY_IDT_HANDLER_SIZE];
```

`NUM_EXCEPTION_VECTORS` `EARLY_IDT_HANDLER_SIZE` :

```
#define NUM_EXCEPTION_VECTORS 32
#define EARLY_IDT_HANDLER_SIZE 9
```

`early_idt_handler_array` 9 `early_idt_handlers` arch/x86/kernel/head_64.S `early_idt_handler_array` :

```
ENTRY(early_idt_handler_array)
...
...
...
ENDPROC(early_idt_handler_common)
```

`.rept NUM_EXCEPTION_VECTORS` `early_idt_handler_array` `early_make_pgtable` ( ) `x86-64` `setup_arch` `x86_64`

Stack Canary

Linuxarch/x86/kernel/head_64.Sinit/main.c `start_kernel` pid - `1 init` `boot_init_stack_canary` canary `boot_init_stack_canary` arch/x86/include/asm/stackprotector.h `CONFIG_CC_STACKPROTECTOR` :

```
#ifdef CONFIG_CC_STACKPROTECTOR
...
...
...
#else
static inline void boot_init_stack_canary(void)
{
}
#endif
```

`CONFIG_CC_STACKPROTECTOR` `boot_init_stack_canary` `irq_stack_union` per-cpu `stack_canary` 40 `offset` :

```
#ifdef CONFIG_X86_64
        BUILD_BUG_ON(offsetof(union irq_stack_union, stack_canary) != 40);
#endif
```

`irq_stack_union` :

```
union irq_stack_union {
        char irq_stack[IRQ_STACK_SIZE];

    struct {
                char gs_base[40];
                unsigned long stack_canary;
        };
};
```

arch/x86/include/asm/processor.h C `gs_base` 40 bytes `irq_stack` `BUILD_BUG_ON` (`BUILD_BUG_ON` Linux)

`canary` :

```
get_random_bytes(&canary, sizeof(canary));
tsc = __native_read_tsc();
canary += tsc + (tsc << 32UL);
```

`this_cpu_write` `canary` `irq_stack_union` :

```
this_cpu_write(irq_stack_union.stack_canary, canary);
```

`this_cpu_*` Linux kernel documentation

/

init/main.c `canary` `local_irq_disable`

include/linux/irqflags.h CPU `CONFIG_TRACE_IRQFLAGS_SUPPORT` :

```
#ifdef CONFIG_TRACE_IRQFLAGS_SUPPORT
...
#define local_irq_disable() \
        do { raw_local_irq_disable(); trace_hardirqs_off(); } while (0)
...
#else
...
#define local_irq_disable()     do { raw_local_irq_disable(); } while (0)
...
#endif
```

`CONFIG_TRACE_IRQFLAGS_SUPPORT` `local_irq_disable` `trace_hardirqs_off` Linux lockdep `irq-flags` `tracing` `hardirq` `softirq` `lockdep` // `trace_hardirqs_off` kernel/locking/lockdep.c:

```
void trace_hardirqs_off(void)
{
        trace_hardirqs_off_caller(CALLER_ADDR0);
}
EXPORT_SYMBOL(trace_hardirqs_off);
```

`trace_hardirqs_off_caller` `trace_hardirqs_off_caller` , `hardirqs_enabled` `local_irq_disable` `redundant_hardirqs_off` `hardirqs_off_events` `lockdep` kernel/locking/lockdep_insides.h `lockdep_stats` :

```
struct lockdep_stats {
...
...
...
int     softirqs_off_events;
int     redundant_softirqs_off;
...
...
...
}
```

`CONFIG_DEBUG_LOCKDEP` `lockdep_stats_debug_show` `/proc/lockdep` :

```
static void lockdep_stats_debug_show(struct seq_file *m)
{
#ifdef CONFIG_DEBUG_LOCKDEP
    unsigned long long hi1 = debug_atomic_read(hardirqs_on_events),
        hi2 = debug_atomic_read(hardirqs_off_events),
        hr1 = debug_atomic_read(redundant_hardirqs_on),
        ...
    ...
    ...
    seq_printf(m, " hardirq on events:             %11llu\n", hi1);
    seq_printf(m, " hardirq off events:            %11llu\n", hi2);
    seq_printf(m, " redundant hardirq ons:         %11llu\n", hr1);
#endif
}
```

:

```
$ sudo cat /proc/lockdep
 hardirq on events:             12838248974
 hardirq off events:            12838248979
 redundant hardirq ons:               67792
 redundant hardirq offs:         3836339146
 softirq on events:                 38002159
 softirq off events:                38002187
 redundant softirq ons:                    0
 redundant softirq offs:                   0
```

`trace_hardirqs_off` `lockdep` `trancing` `local_disable_irq` `raw_local_irq_disable` arch/x86/include/asm/irqflags.h :

```
static inline void native_irq_disable(void)
{
        asm volatile("cli": : :"memory");
}
```

`cli` IF `local_irq_disable` `local_irq_enable` `local_irq_disable` `sti` :

```
static inline void native_irq_enable(void)
{
        asm volatile("sti": : :"memory");
}
```

`local_irq_disable` `local_irq_enable` `local_irq_disable` Linux init/main.c `start_kernel` "" `cli` `local_irq_{enabled,disable}` `loc`

```
early_boot_irqs_disabled = true;
```

`early_boot_irqs_disabled` include/linux/kernel.h :

```
extern bool early_boot_irqs_disabled;
```

kernel/smp.c `smp_call_function_many` :

```
WARN_ON_ONCE(cpu_online(this_cpu) && irqs_disabled()
                    && !oops_in_progress && !early_boot_irqs_disabled);
```

trap

`local_disable_irq` `boot_cpu_init` `page_address_init` ( ) `setup_arch` arch/x86/kernel/setup.c `setup_arch` `early_trap_init` arch/x86/kernel/traps.c `Interrupt Descriptor Table` :

```
void __init early_trap_init(void)
{
        set_intr_gate_ist(X86_TRAP_DB, &debug, DEBUG_STACK);
        set_system_intr_gate_ist(X86_TRAP_BP, &int3, DEBUG_STACK);
#ifdef CONFIG_X86_32
        set_intr_gate(X86_TRAP_PF, page_fault);
#endif
        load_idt(&idt_descr);
}
```

- `set_intr_gate_ist`
- `set_system_intr_gate_ist`
- `set_intr_gate`

arch/x86/include/asm/desc.h `set_intr_gate_ist` `IDT` :

```
static inline void set_intr_gate_ist(int n, void *addr, unsigned ist)
{
        BUG_ON((unsigned)n > 0xFF);
        _set_gate(n, GATE_INTERRUPT, addr, 0, ist, __KERNEL_CS);
}
```

`n` `0xff` 255 [] (http://0xax.gitbooks.io/linux-insides/content/interrupts/interrupts-1.html) 0 255 `_set_gate` `IDT` :

```
static inline void _set_gate(int gate, unsigned type, void *addr,
                             unsigned dpl, unsigned ist, unsigned seg)
{
        gate_desc s;

        pack_gate(&s, type, (unsigned long)addr, dpl, ist, seg);
        write_idt_entry(idt_table, gate, &s);
        write_trace_idt_entry(gate, &s);
}
```

`pack_gate` `IDT` `gate_desc` , :

- `GATE_INTERRUPT`
- `GATE_TRAP`
- `GATE_CALL`
- `GATE_TASK`

`IDT` `present` :

```
static inline void pack_gate(gate_desc *gate, unsigned type, unsigned long func,
                             unsigned dpl, unsigned ist, unsigned seg)
{
        gate->offset_low        = PTR_LOW(func);
        gate->segment           = __KERNEL_CS;
        gate->ist               = ist;
        gate->p                 = 1;
        gate->dpl               = dpl;
        gate->zero0             = 0;
        gate->zero1             = 0;
        gate->type              = type;
        gate->offset_middle     = PTR_MIDDLE(func);
        gate->offset_high       = PTR_HIGH(func);
}
```

`write_idt_entry` `IDT` `native_write_idt_entry` `idt_table` :

```
#define write_idt_entry(dt, entry, g)           native_write_idt_entry(dt, entry, g)

static inline void native_write_idt_entry(gate_desc *idt, int entry, const gate_desc *gate)
{
        memcpy(&idt[entry], gate, sizeof(*gate));
}
```

`idt_table` `gate_desc` :

```
extern gate_desc idt_table[];
```

`set_intr_gate_ist` `set_system_intr_gate_ist` :

```
static inline void set_system_intr_gate_ist(int n, void *addr, unsigned ist)
{
        BUG_ON((unsigned)n > 0xFF);
        _set_gate(n, GATE_INTERRUPT, addr, 0x3, ist, __KERNEL_CS);
}
```

`_set_gate` `0x3` `set_intr_gate_ist` `0x0` `DPL` `0` `3` `set_system_intr_gate_ist` , `set_intr_gate_ist` , `set_intr_gate` `early_trap_init` :

```
set_intr_gate_ist(X86_TRAP_DB, &debug, DEBUG_STACK);
set_system_intr_gate_ist(X86_TRAP_BP, &int3, DEBUG_STACK);
```

`#DB` `int3` `IDT` :

- vector number of an interrupt;
- address of an interrupt handler;
- interrupt stack table index.

`early_trap_init`

LinuxLinux

twitter

Please note that English is not my first language, And I am really sorry for any inconvenience. If you find any mistakes please send me PR to linux-insides.

- IDT
- Protected mode
- List of x86 calling conventions
- 8086
- Long mode
- NX
- Extended Feature Enable Register
- Model-specific register
- Process identifier
- lockdep
- irqflags tracing
- IF
- Stack canary
- Union type
- this*cpu** operations
- vector number
- Interrupt Stack Table
- Privilege level
- Previous part

. Part 3.

chapter Linux part `setup_arch` arch/x86/kernel/setup.c .

`setup_arch` x86_64 architecture `setup_arch`

- `#DB` -
- `#BP` - `int`

x86_64 kgdb `early_trap_init`

```
void __init early_trap_init(void)
{
        set_intr_gate_ist(X86_TRAP_DB, &debug, DEBUG_STACK);
        set_system_intr_gate_ist(X86_TRAP_BP, &int3, DEBUG_STACK);
        load_idt(&idt_descr);
}
```

arch/x86/kernel/traps.c. `set_intr_gate_ist` `set_system_intr_gate_ist`

Ok `early_trap_init` `#DB` `#BP`

- `DB` `debug` - debug register debug register Intel 80386 `x86` CPU

debug register general protection fault `#DB` `set_intr_gate_ist` `set_system_intr_gate_ist`

`#DB` 1X86_TRAP_DB

```
+-------------------------------------------------+
|Vector|Mnemonic|Description         |Type |Error Code|
+-------------------------------------------------+
|1     | #DB    |Reserved            |F/T  |NO        |
+-------------------------------------------------+
```

int 3 `#BP` `breakpointv` `DB` `BP`

```
// breakpoint.c
#include <stdio.h>

int main() {
    int i;
    while (i < 6){
        printf("i equal to: %d\n", i);
        __asm__("int3");
        ++i;
    }
}
```

```
$ gcc breakpoint.c -o breakpoint
i equal to: 0
Trace/breakpoint trap
```

gdb

```
$ gdb breakpoint
...
...
...
(gdb) run
Starting program: /home/alex/breakpoints
i equal to: 0

Program received signal SIGTRAP, Trace/breakpoint trap.
0x0000000000400585 in main ()
=> 0x0000000000400585 <main+31>:    83 45 fc 01    add    DWORD PTR [rbp-0x4],0x1
(gdb) c
Continuing.
i equal to: 1

Program received signal SIGTRAP, Trace/breakpoint trap.
0x0000000000400585 in main ()
=> 0x0000000000400585 <main+31>:    83 45 fc 01    add    DWORD PTR [rbp-0x4],0x1
(gdb) c
Continuing.
i equal to: 2

Program received signal SIGTRAP, Trace/breakpoint trap.
0x0000000000400585 in main ()
=> 0x0000000000400585 <main+31>:    83 45 fc 01    add    DWORD PTR [rbp-0x4],0x1
...
...
...
```

`set_intr_gate_ist` `set_system_intr_gate_ist`

- `debug` ;
- `int3` .

C `*.c/*.h` arch/x86/include/asm/traps.h

```
asmlinkage void debug(void);
```

and

```
asmlinkage void int3(void);
```

`asmlinkage` gcc C `asmlinkage` `gcc` So, both handlers are defined in the arch/x86/entry/entry_64.S assembly source code file with the `idtentry` macro:

arch/x86/entry/entry_64.S `idtentry`

```
idtentry debug do_debug has_error_code=0 paranoid=1 shift_ist=DEBUG_STACK
```

and

```
idtentry int3 do_int3 has_error_code=0 paranoid=1 shift_ist=DEBUG_STACK
```

general purpose registers `SIGILL` signal arch/x86/kernel/entry_64.S `idtentry`

`sym globl name` `do_sym` * `has_error_code`

`paranoid` - `shift_ist` -“”

`idtentry`

```
.macro idtentry sym do_sym has_error_code:req paranoid=0 shift_ist=-1
ENTRY(\sym)
...
...
...
END(\sym)
.endm
```

Before we will consider internals of the `idtentry` macro, we should to know state of stack when an exception occurs. As we may read in the Intel® 64 and IA-32 Architectures Software Developer’s Manual 3A, the state of stack when an exception occurs is following:

`identry` Intel®64 and IA-32 Architectures Software Developer's Manual 3A

```
    +------------+
+40 | %SS        |
+32 | %RSP       |
+24 | %RFLAGS    |
+16 | %CS        |
 +8 | %RIP       |
  0 | ERROR CODE | <-- %RSP
    +------------+
```

`idtmacro` `#DB BP`

```
idtentry debug do_debug has_error_code=0 paranoid=1 shift_ist=DEBUG_STACK
idtentry int3 do_int3 has_error_code=0 paranoid=1 shift_ist=DEBUG_STACK
```

`debug int3` `do_debug do_int3` `debug int3` `idtentry`

```
.ifeq \has_error_code
    pushq    $-1
.endif
```

“-1”

`idtentry shift_ist paranoid Interrupt Stack Table` - x86_64 double fault shift_istIST

`paranoid` `CS CPL Current Privilege Level 3`

```
testl $3,CS(%rsp)
jnz userspace
...
...
...

//
```

100

> if we are in an NMI/MCE/DEBUG/whatever super-atomic entry context, which might have triggered right after a normal entry wrote CS to the stack but before we executed SWAPGS, then the only safe way to check for GS is the slower method: the RDMSR.

`NMI` swapgs `MSR_GS_BASE` cpu `MSR_GS_BASE`

```
movl $MSR_GS_BASE,%ecx
rdmsr
testl %edx,%edx
js 1f
```

`MSR_GS_BASE` edx:eax gs `0xffff880000000000` `MSR_GS_BASE`
`0xffff880000000000` `0xffffc7fffffffff` `rdmsr` `edx` - `0xffff8800` 4 `-30720` `CPU` `gs`

```
ALLOC_PT_GPREGS_ON_STACK
```

arch / x86 / entry / calling.h 15 * 8

```
.macro ALLOC_PT_GPREGS_ON_STACK addskip=0
    addq    $-(15*8+\addskip), %rsp
.endm
```

`ALLOC_PT_GPREGS_ON_STACK`

```
     +------------+
+160 | %SS        |
+152 | %RSP       |
+144 | %RFLAGS    |
+136 | %CS        |
+128 | %RIP       |
+120 | ERROR CODE |
     |------------|
+112 |            |
+104 |            |
 +96 |            |
 +88 |            |
 +80 |            |
 +72 |            |
 +64 |            |
 +56 |            |
 +48 |            |
 +40 |            |
 +32 |            |
 +24 |            |
 +16 |            |
  +8 |            |
  +0 |            | <- %RSP
     +------------+
```

```
.if \paranoid
    .if \paranoid == 1
        testb    $3, CS(%rsp)
        jnz    1f
    .endif
    call    paranoid_entry
.else
    call    error_entry
.endif
```

`debug int3 paranoid = 1` CS `1f` `paranoid_entry` Let's consider first case when we came from userspace to an exception handler. As described above we should jump at `1` label. The `1` label starts from the call of the

```
call    error_entry
```

:

```
SAVE_C_REGS 8
SAVE_EXTRA_REGS 8
```

arch/x86/entry/calling.h

```
.macro SAVE_EXTRA_REGS offset=0
    movq %r15, 0*8+\offset(%rsp)
    movq %r14, 1*8+\offset(%rsp)
    movq %r13, 2*8+\offset(%rsp)
    movq %r12, 3*8+\offset(%rsp)
    movq %rbp, 4*8+\offset(%rsp)
    movq %rbx, 5*8+\offset(%rsp)
.endm
```

`SAVE_C_REGS` `SAVE_EXTRA_REGS`:

```
     +------------+
+160 | %SS        |
+152 | %RSP       |
+144 | %RFLAGS    |
+136 | %CS        |
+128 | %RIP       |
+120 | ERROR CODE |
     |------------|
+112 | %RDI       |
+104 | %RSI       |
 +96 | %RDX       |
 +88 | %RCX       |
 +80 | %RAX       |
 +72 | %R8        |
 +64 | %R9        |
 +56 | %R10       |
 +48 | %R11       |
 +40 | %RBX       |
 +32 | %RBP       |
 +24 | %R12       |
 +16 | %R13       |
  +8 | %R14       |
  +0 | %R15       | <- %RSP
     +------------+
```

```
testb   $3, CS+8(%rsp)
jz    .Lerror_kernelspace
```

`RIP` SWAPGS `MSR_KERNEL_GS_BASE` `MSR_GS_BASE` `gs` `SWAPGS` `error_entry`

`idtentry` `error_entry`

```
movq    %rsp, %rdi
call    sync_regs
```

`rdi` `sync_regs` (x86_64 ABI) arch / x86 / kernel / traps.c

```
asmlinkage __visible notrace struct pt_regs *sync_regs(struct pt_regs *eregs)
{
    struct pt_regs *regs = task_pt_regs(current);
    *regs = *eregs;
    return regs;
}
```

[arch/x86/include/asm/processor.h] `task_ptr_regs`
(https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/arch/x86/ include / asm / processor.h)
`task_ptr_regs` `thread.sp0`

```
#define task_pt_regs(tsk)       ((struct pt_regs *)(tsk)->thread.sp0 - 1)
```

`sync_regs`

```
movq    %rax, %rsp
```

1. `pt_regs` `rdi`

```
movq    %rsp, %rdi
```

2. `rsi` -1

```
.if \has_error_code
    movq    ORIG_RAX(%rsp), %rsi
    movq    $-1, ORIG_RAX(%rsp)
.else
    xorl    %esi, %esi
.endif
```

`esi`

```
call    \do_sym
```

which:

```
dotraplinkage void do_debug(struct pt_regs *regs, long error_code);
```

`debug`

```
dotraplinkage void notrace do_int3(struct pt_regs *regs, long error_code);
```

`int 3`

> 0

`paranoid = 1` `idtentry` `paranoid` `paranoid_entry`

```
ENTRY(paranoid_entry)
    cld
    SAVE_C_REGS 8
    SAVE_EXTRA_REGS 8
    movl    $1, %ebx
    movl    $MSR_GS_BASE, %ecx
    rdmsr
    testl    %edx, %edx
    js    1f
    SWAPGS
    xorl    %ebx, %ebx
1:    ret
END(paranoid_entry)
```

`SWAPGS` `rdi` rsi

```
movq    %rsp, %rdi

.if \has_error_code
    movq    ORIG_RAX(%rsp), %rsi
    movq    $-1, ORIG_RAX(%rsp)
.else
    xorl    %esi, %esi
.endif
```

`IST`

```
.if \shift_ist != -1
    subq    $EXCEPTION_STKSZ, CPU_TSS_IST(\shift_ist)
.endif
```

`shift_ist` `iddentry` -1 `shift_ist`

```
call    \do_sym
```

`paranoid = 0`

`idtentry` `error_exit`

```
jmp    error_exit
```

arch/x86/entry/entry_64.S `error_exit` `SWPAGS` `iret`

That's all.

Linux `#DB` `#BP` `setup_arch` twitter ping **PR[linux-insides]** **https://github.com/0xAX/linux-insides**

- Debug registers
- Intel 80385
- INT 3
- gcc
- TSS
- GNU assembly .error directive
- dwarf2
- CFI directives
- IRQ
- system call
- swapgs
- SIGTRAP
- Per-CPU variables
- kgdb
- ACPI
- Previous part

Interrupts and Interrupt Handling. Part 4.

Initialization of non-early interrupt gates

This is fourth part about an interrupts and exceptions handling in the Linux kernel and in the previous part we saw first early `#DB` and `#BP` exceptions handlers from the arch/x86/kernel/traps.c. We stopped on the right after the `early_trap_init` function that called in the `setup_arch` function which defined in the arch/x86/kernel/setup.c. In this part we will continue to dive into an interrupts and exceptions handling in the Linux kernel for `x86_64` and continue to do it from the place where we left off in the last part. First thing which is related to the interrupts and exceptions handling is the setup of the `#PF` or page fault handler with the `early_trap_pf_init` function. Let's start from it.

Early page fault handler

The `early_trap_pf_init` function defined in the arch/x86/kernel/traps.c. It uses `set_intr_gate` macro that fills Interrupt Descriptor Table with the given entry:

```
void __init early_trap_pf_init(void)
{
#ifdef CONFIG_X86_64
        set_intr_gate(X86_TRAP_PF, page_fault);
#endif
}
```

This macro defined in the arch/x86/include/asm/desc.h. We already saw macros like this in the previous part - `set_system_intr_gate` and `set_intr_gate_ist`. This macro checks that given vector number is not greater than `255` (maximum vector number) and calls `_set_gate` function as `set_system_intr_gate` and `set_intr_gate_ist` did it:

```
#define set_intr_gate(n, addr)                                  \
do {                                                            \
        BUG_ON((unsigned)n > 0xFF);                             \
        _set_gate(n, GATE_INTERRUPT, (void *)addr, 0, 0,        \
                  __KERNEL_CS);                                 \
        _trace_set_gate(n, GATE_INTERRUPT, (void *)trace_##addr,\
                        0, 0, __KERNEL_CS);                     \
} while (0)
```

The `set_intr_gate` macro takes two parameters:

- vector number of a interrupt;
- address of an interrupt handler;

In our case they are:

- `X86_TRAP_PF` - `14`;
- `page_fault` - the interrupt handler entry point.

The `X86_TRAP_PF` is the element of enum which defined in the arch/x86/include/asm/traprs.h:

```
enum {
    ...
    ...
    ...
    ...
    X86_TRAP_PF,            /* 14, Page Fault */
    ...
    ...
    ...
```

```
}
```

When the `early_trap_pf_init` will be called, the `set_intr_gate` will be expanded to the call of the `_set_gate` which will fill the `IDT` with the handler for the page fault. Now let's look on the implementation of the `page_fault` handler. The `page_fault` handler defined in the arch/x86/kernel/entry_64.S assembly source code file as all exceptions handlers. Let's look on it:

```
trace_idtentry page_fault do_page_fault has_error_code=1
```

We saw in the previous part how `#DB` and `#BP` handlers defined. They were defined with the `idtentry` macro, but here we can see `trace_idtentry`. This macro defined in the same source code file and depends on the `CONFIG_TRACING` kernel configuration option:

```
#ifdef CONFIG_TRACING
.macro trace_idtentry sym do_sym has_error_code:req
idtentry trace(\sym) trace(\do_sym) has_error_code=\has_error_code
idtentry \sym \do_sym has_error_code=\has_error_code
.endm
#else
.macro trace_idtentry sym do_sym has_error_code:req
idtentry \sym \do_sym has_error_code=\has_error_code
.endm
#endif
```

We will not dive into exceptions Tracing now. If `CONFIG_TRACING` is not set, we can see that `trace_idtentry` macro just expands to the normal `idtentry`. We already saw implementation of the `idtentry` macro in the previous part, so let's start from the `page_fault` exception handler.

As we can see in the `idtentry` definition, the handler of the `page_fault` is `do_page_fault` function which defined in the arch/x86/mm/fault.c and as all exceptions handlers it takes two arguments:

- `regs` - `pt_regs` structure that holds state of an interrupted process;
- `error_code` - error code of the page fault exception.

Let's look inside this function. First of all we read content of the cr2 control register:

```
dotraplinkage void notrace
do_page_fault(struct pt_regs *regs, unsigned long error_code)
{
    unsigned long address = read_cr2();
    ...
    ...
    ...
}
```

This register contains a linear address which caused `page fault`. In the next step we make a call of the `exception_enter` function from the include/linux/context_tracking.h. The `exception_enter` and `exception_exit` are functions from context tracking subsystem in the Linux kernel used by the RCU to remove its dependency on the timer tick while a processor runs in userspace. Almost in the every exception handler we will see similar code:

```
enum ctx_state prev_state;
prev_state = exception_enter();
...
... // exception handler here
...
exception_exit(prev_state);
```

The `exception_enter` function checks that `context tracking` is enabled with the `context_tracking_is_enabled` and if it is in enabled state, we get previous context with the `this_cpu_read` (more about `this_cpu_*` operations you can read in the Documentation). After this it calls `context_tracking_user_exit` function which informs the context tracking that the processor is exiting userspace mode and entering the kernel:

```
static inline enum ctx_state exception_enter(void)
{
        enum ctx_state prev_ctx;

        if (!context_tracking_is_enabled())
                return 0;

        prev_ctx = this_cpu_read(context_tracking.state);
        context_tracking_user_exit();

        return prev_ctx;
}
```

The state can be one of the:

```
enum ctx_state {
    IN_KERNEL = 0,
    IN_USER,
} state;
```

And in the end we return previous context. Between the `exception_enter` and `exception_exit` we call actual page fault handler:

```
__do_page_fault(regs, error_code, address);
```

The `__do_page_fault` is defined in the same source code file as `do_page_fault` - arch/x86/mm/fault.c. In the beginning of the `__do_page_fault` we check state of the kmemcheck checker. The `kmemcheck` detects warns about some uses of uninitialized memory. We need to check it because page fault can be caused by kmemcheck:

```
if (kmemcheck_active(regs))
        kmemcheck_hide(regs);
    prefetchw(&mm->mmap_sem);
```

After this we can see the call of the `prefetchw` which executes instruction with the same name which fetches X86_FEATURE_3DNOW to get exclusive cache line. The main purpose of prefetching is to hide the latency of a memory access. In the next step we check that we got page fault not in the kernel space with the following condition:

```
if (unlikely(fault_in_kernel_space(address))) {
...
...
...
}
```

where `fault_in_kernel_space` is:

```
static int fault_in_kernel_space(unsigned long address)
{
        return address >= TASK_SIZE_MAX;
}
```

The `TASK_SIZE_MAX` macro expands to the:

```
#define TASK_SIZE_MAX   ((1UL << 47) - PAGE_SIZE)
```

or `0x00007fffffff000` . Pay attention on `unlikely` macro. There are two macros in the Linux kernel:

```
#define likely(x)      __builtin_expect(!!(x), 1)
#define unlikely(x)    __builtin_expect(!!(x), 0)
```

You can often find these macros in the code of the Linux kernel. Main purpose of these macros is optimization. Sometimes this situation is that we need to check the condition of the code and we know that it will rarely be `true` or `false`. With these macros we can tell to the compiler about this. For example

```
static int proc_root_readdir(struct file *file, struct dir_context *ctx)
{
        if (ctx->pos < FIRST_PROCESS_ENTRY) {
                int error = proc_readdir(file, ctx);
                if (unlikely(error <= 0))
                        return error;
...
...
...
}
```

Here we can see `proc_root_readdir` function which will be called when the Linux VFS needs to read the `root` directory contents. If condition marked with `unlikely`, compiler can put `false` code right after branching. Now let's back to the our address check. Comparison between the given address and the `0x00007fffffff000` will give us to know, was page fault in the kernel mode or user mode. After this check we know it. After this `__do_page_fault` routine will try to understand the problem that provoked page fault exception and then will pass address to the appropriate routine. It can be `kmemcheck` fault, spurious fault, kprobes fault and etc. Will not dive into implementation details of the page fault exception handler in this part, because we need to know many different concepts which are provided by the Linux kernel, but will see it in the chapter about the memory management in the Linux kernel.

Back to start_kernel

There are many different function calls after the `early_trap_pf_init` in the `setup_arch` function from different kernel subsystems, but there are no one interrupts and exceptions handling related. So, we have to go back where we came from - `start_kernel` function from the init/main.c. The first things after the `setup_arch` is the `trap_init` function from the arch/x86/kernel/traps.c. This function makes initialization of the remaining exceptions handlers (remember that we already setup 3 handlers for the `#DB` - debug exception, `#BP` - breakpoint exception and `#PF` - page fault exception). The `trap_init` function starts from the check of the Extended Industry Standard Architecture:

```
#ifdef CONFIG_EISA
        void __iomem *p = early_ioremap(0x0FFFD9, 4);

        if (readl(p) == 'E' + ('I'<<8) + ('S'<<16) + ('A'<<24))
                EISA_bus = 1;
        early_iounmap(p, 4);
#endif
```

Note that it depends on the `CONFIG_EISA` kernel configuration parameter which represents `EISA` support. Here we use `early_ioremap` function to map `I/O` memory on the page tables. We use `readl` function to read first `4` bytes from the mapped region and if they are equal to `EISA` string we set `EISA_bus` to one. In the end we just unmap previously mapped region. More about `early_ioremap` you can read in the part which describes Fix-Mapped Addresses and ioremap.

After this we start to fill the `Interrupt Descriptor Table` with the different interrupt gates. First of all we set `#DE` or `Divide Error` and `#NMI` or `Non-maskable Interrupt`:

```
set_intr_gate(X86_TRAP_DE, divide_error);
set_intr_gate_ist(X86_TRAP_NMI, &nmi, NMI_STACK);
```

We use `set_intr_gate` macro to set the interrupt gate for the `#DE` exception and `set_intr_gate_ist` for the `#NMI`. You can remember that we already used these macros when we have set the interrupts gates for the page fault handler, debug handler and etc, you can find explanation of it in the previous part. After this we setup exception gates for the following exceptions:

```
set_system_intr_gate(X86_TRAP_OF, &overflow);
set_intr_gate(X86_TRAP_BR, bounds);
```

```
set_intr_gate(X86_TRAP_UD, invalid_op);
set_intr_gate(X86_TRAP_NM, device_not_available);
```

Here we can see:

- `#OF` or `Overflow` exception. This exception indicates that an overflow trap occurred when an special INTO instruction was executed;
- `#BR` or `BOUND Range exceeded` exception. This exception indicates that a `BOUND-range-exceed` fault occurred when a BOUND instruction was executed;
- `#UD` or `Invalid Opcode` exception. Occurs when a processor attempted to execute invalid or reserved opcode, processor attempted to execute instruction with invalid operand(s) and etc;
- `#NM` or `Device Not Available` exception. Occurs when the processor tries to execute `x87 FPU` floating point instruction while `EM` flag in the control register `cr0` was set.

In the next step we set the interrupt gate for the `#DF` or `Double fault` exception:

```
set_intr_gate_ist(X86_TRAP_DF, &double_fault, DOUBLEFAULT_STACK);
```

This exception occurs when processor detected a second exception while calling an exception handler for a prior exception. In usual way when the processor detects another exception while trying to call an exception handler, the two exceptions can be handled serially. If the processor cannot handle them serially, it signals the double-fault or `#DF` exception.

The following set of the interrupt gates is:

```
set_intr_gate(X86_TRAP_OLD_MF, &coprocessor_segment_overrun);
set_intr_gate(X86_TRAP_TS, &invalid_TSS);
set_intr_gate(X86_TRAP_NP, &segment_not_present);
set_intr_gate_ist(X86_TRAP_SS, &stack_segment, STACKFAULT_STACK);
set_intr_gate(X86_TRAP_GP, &general_protection);
set_intr_gate(X86_TRAP_SPURIOUS, &spurious_interrupt_bug);
set_intr_gate(X86_TRAP_MF, &coprocessor_error);
set_intr_gate(X86_TRAP_AC, &alignment_check);
```

Here we can see setup for the following exception handlers:

- `#CSO` or `Coprocessor Segment Overrun` - this exception indicates that math coprocessor of an old processor detected a page or segment violation. Modern processors do not generate this exception
- `#TS` or `Invalid TSS` exception - indicates that there was an error related to the Task State Segment.
- `#NP` or `Segment Not Present` exception indicates that the `present flag` of a segment or gate descriptor is clear during attempt to load one of `cs`, `ds`, `es`, `fs`, or `gs` register.
- `#SS` or `Stack Fault` exception indicates one of the stack related conditions was detected, for example a not-present stack segment is detected when attempting to load the `ss` register.
- `#GP` or `General Protection` exception indicates that the processor detected one of a class of protection violations called general-protection violations. There are many different conditions that can cause general-protection exception. For example loading the `ss`, `ds`, `es`, `fs`, or `gs` register with a segment selector for a system segment, writing to a code segment or a read-only data segment, referencing an entry in the `Interrupt Descriptor Table` (following an interrupt or exception) that is not an interrupt, trap, or task gate and many many more.
- `Spurious Interrupt` - a hardware interrupt that is unwanted.
- `#MF` or `x87 FPU Floating-Point Error` exception caused when the x87 FPU has detected a floating point error.
- `#AC` or `Alignment Check` exception Indicates that the processor detected an unaligned memory operand when alignment checking was enabled.

After that we setup this exception gates, we can see setup of the `Machine-Check` exception:

```
#ifdef CONFIG_X86_MCE
    set_intr_gate_ist(X86_TRAP_MC, &machine_check, MCE_STACK);
#endif
```

Note that it depends on the `CONFIG_X86_MCE` kernel configuration option and indicates that the processor detected an internal machine error or a bus error, or that an external agent detected a bus error. The next exception gate is for the SIMD Floating-Point exception:

```
set_intr_gate(X86_TRAP_XF, &simd_coprocessor_error);
```

which indicates the processor has detected an `SSE` or `SSE2` or `SSE3` SIMD floating-point exception. There are six classes of numeric exception conditions that can occur while executing an SIMD floating-point instruction:

- Invalid operation
- Divide-by-zero
- Denormal operand
- Numeric overflow
- Numeric underflow
- Inexact result (Precision)

In the next step we fill the `used_vectors` array which defined in the arch/x86/include/asm/desc.h header file and represents `bitmap`:

```
DECLARE_BITMAP(used_vectors, NR_VECTORS);
```

of the first `32` interrupts (more about bitmaps in the Linux kernel you can read in the part which describes cpumasks and bitmaps)

```
for (i = 0; i < FIRST_EXTERNAL_VECTOR; i++)
    set_bit(i, used_vectors)
```

where `FIRST_EXTERNAL_VECTOR` is:

```
#define FIRST_EXTERNAL_VECTOR          0x20
```

After this we setup the interrupt gate for the `ia32_syscall` and add `0x80` to the `used_vectors` bitmap:

```
#ifdef CONFIG_IA32_EMULATION
        set_system_intr_gate(IA32_SYSCALL_VECTOR, ia32_syscall);
        set_bit(IA32_SYSCALL_VECTOR, used_vectors);
#endif
```

There is `CONFIG_IA32_EMULATION` kernel configuration option on `x86_64` Linux kernels. This option provides ability to execute 32-bit processes in compatibility-mode. In the next parts we will see how it works, in the meantime we need only to know that there is yet another interrupt gate in the `IDT` with the vector number `0x80`. In the next step we maps `IDT` to the fixmap area:

```
__set_fixmap(FIX_RO_IDT, __pa_symbol(idt_table), PAGE_KERNEL_RO);
idt_descr.address = fix_to_virt(FIX_RO_IDT);
```

and write its address to the `idt_descr.address` (more about fix-mapped addresses you can read in the second part of the Linux kernel memory management chapter). After this we can see the call of the `cpu_init` function that defined in the arch/x86/kernel/cpu/common.c. This function makes initialization of the all `per-cpu` state. In the beginning of the `cpu_init` we do the following things: First of all we wait while current cpu is initialized and than we call the `cr4_init_shadow` function which stores shadow copy of the `cr4` control register for the current cpu and load CPU microcode if need with the following function calls:

```
wait_for_master_cpu(cpu);
cr4_init_shadow();
load_ucode_ap();
```

Next we get the `Task State Segment` for the current cpu and `orig_ist` structure which represents origin `Interrupt Stack Table` values with the:

```
t = &per_cpu(cpu_tss, cpu);
oist = &per_cpu(orig_ist, cpu);
```

As we got values of the `Task State Segment` and `Interrupt Stack Table` for the current processor, we clear following bits in the `cr4` control register:

```
cr4_clear_bits(X86_CR4_VME|X86_CR4_PVI|X86_CR4_TSD|X86_CR4_DE);
```

with this we disable `vm86` extension, virtual interrupts, timestamp (RDTSC can only be executed with the highest privilege) and debug extension. After this we reload the `Global Descriptor Table` and `Interrupt Descriptor table` with the:

```
    switch_to_new_gdt(cpu);
    loadsegment(fs, 0);
    load_current_idt();
```

After this we setup array of the Thread-Local Storage Descriptors, configure NX and load CPU microcode. Now is time to setup and load `per-cpu` Task State Segments. We are going in a loop through the all exception stack which is `N_EXCEPTION_STACKS` or `4` and fill it with `Interrupt Stack Tables` :

```
    if (!oist->ist[0]) {
        char *estacks = per_cpu(exception_stacks, cpu);

        for (v = 0; v < N_EXCEPTION_STACKS; v++) {
            estacks += exception_stack_sizes[v];
            oist->ist[v] = t->x86_tss.ist[v] =
                    (unsigned long)estacks;
            if (v == DEBUG_STACK-1)
                per_cpu(debug_stack_addr, cpu) = (unsigned long)estacks;
        }
    }
```

As we have filled `Task State Segments` with the `Interrupt Stack Tables` we can set `TSS` descriptor for the current processor and load it with the:

```
set_tss_desc(cpu, t);
load_TR_desc();
```

where `set_tss_desc` macro from the arch/x86/include/asm/desc.h writes given descriptor to the `Global Descriptor Table` of the given processor:

```
#define set_tss_desc(cpu, addr) __set_tss_desc(cpu, GDT_ENTRY_TSS, addr)
static inline void __set_tss_desc(unsigned cpu, unsigned int entry, void *addr)
{
        struct desc_struct *d = get_cpu_gdt_table(cpu);
        tss_desc tss;
        set_tssldt_descriptor(&tss, (unsigned long)addr, DESC_TSS,
                              IO_BITMAP_OFFSET + IO_BITMAP_BYTES +
                              sizeof(unsigned long) - 1);
        write_gdt_entry(d, entry, &tss, DESC_TSS);
}
```

and `load_TR_desc` macro expands to the `ltr` or `Load Task Register` instruction:

```
#define load_TR_desc()                          native_load_tr_desc()
static inline void native_load_tr_desc(void)
{
        asm volatile("ltr %w0"::"q" (GDT_ENTRY_TSS*8));
}
```

In the end of the `trap_init` function we can see the following code:

```
set_intr_gate_ist(X86_TRAP_DB, &debug, DEBUG_STACK);
set_system_intr_gate_ist(X86_TRAP_BP, &int3, DEBUG_STACK);
...
...
...
#ifdef CONFIG_X86_64
        memcpy(&nmi_idt_table, &idt_table, IDT_ENTRIES * 16);
        set_nmi_gate(X86_TRAP_DB, &debug);
        set_nmi_gate(X86_TRAP_BP, &int3);
#endif
```

Here we copy `idt_table` to the `nmi_dit_table` and setup exception handlers for the `#DB` or `Debug exception` and `#BR` or `Breakpoint exception`. You can remember that we already set these interrupt gates in the previous part, so why do we need to setup it again? We setup it again because when we initialized it before in the `early_trap_init` function, the `Task State Segment` was not ready yet, but now it is ready after the call of the `cpu_init` function.

That's all. Soon we will consider all handlers of these interrupts/exceptions.

Conclusion

It is the end of the fourth part about interrupts and interrupt handling in the Linux kernel. We saw the initialization of the Task State Segment in this part and initialization of the different interrupt handlers as `Divide Error`, `Page Fault` exception and etc. You can note that we saw just initialization stuff, and will dive into details about handlers for these exceptions. In the next part we will start to do it.

If you have any questions or suggestions write me a comment or ping me at twitter.

Please note that English is not my first language, And I am really sorry for any inconvenience. If you find any mistakes please send me PR to linux-insides.

Links

- page fault
- Interrupt Descriptor Table
- Tracing
- cr2
- RCU
- this*cpu** operations
- kmemcheck
- prefetchw
- 3DNow
- CPU caches
- VFS
- Linux kernel memory management
- Fix-Mapped Addresses and ioremap
- Extended Industry Standard Architecture
- INT isntruction
- INTO
- BOUND
- opcode
- control register
- x87 FPU
- MCE exception

- SIMD
- cpumasks and bitmaps
- NX
- Task State Segment
- Previous part

Interrupts and Interrupt Handling. Part 5.

Implementation of exception handlers

This is the fifth part about an interrupts and exceptions handling in the Linux kernel and in the previous part we stopped on the setting of interrupt gates to the Interrupt descriptor Table. We did it in the `trap_init` function from the arch/x86/kernel/traps.c source code file. We saw only setting of these interrupt gates in the previous part and in the current part we will see implementation of the exception handlers for these gates. The preparation before an exception handler will be executed is in the arch/x86/entry/entry_64.S assembly file and occurs in the idtentry macro that defines exceptions entry points:

```
idtentry divide_error                    do_divide_error                    has_error_code=0
idtentry overflow                        do_overflow                        has_error_code=0
idtentry invalid_op                        do_invalid_op                    has_error_code=0
idtentry bounds                            do_bounds                        has_error_code=0
idtentry device_not_available             do_device_not_available           has_error_code=0
idtentry coprocessor_segment_overrun    do_coprocessor_segment_overrun has_error_code=0
idtentry invalid_TSS                    do_invalid_TSS                    has_error_code=1
idtentry segment_not_present            do_segment_not_present            has_error_code=1
idtentry spurious_interrupt_bug            do_spurious_interrupt_bug        has_error_code=0
idtentry coprocessor_error                do_coprocessor_error             has_error_code=0
idtentry alignment_check                do_alignment_check                has_error_code=1
idtentry simd_coprocessor_error            do_simd_coprocessor_error        has_error_code=0
```

The `idtentry` macro does following preparation before an actual exception handler (`do_divide_error` for the `divide_error` , `do_overflow` for the `overflow` and etc.) will get control. In another words the `idtentry` macro allocates place for the registers (pt_regs structure) on the stack, pushes dummy error code for the stack consistency if an interrupt/exception has no error code, checks the segment selector in the `cs` segment register and switches depends on the previous state(userspace or kernelspace). After all of these preparations it makes a call of an actual interrupt/exception handler:

```
.macro idtentry sym do_sym has_error_code:req paranoid=0 shift_ist=-1
ENTRY(\sym)
    ...
    ...
    ...
    call    \do_sym
    ...
    ...
    ...
END(\sym)
.endm
```

After an exception handler will finish its work, the `idtentry` macro restores stack and general purpose registers of an interrupted task and executes iret instruction:

```
ENTRY(paranoid_exit)
    ...
    ...
    ...
    RESTORE_EXTRA_REGS
    RESTORE_C_REGS
    REMOVE_PT_GPREGS_FROM_STACK 8
    INTERRUPT_RETURN
END(paranoid_exit)
```

where `INTERRUPT_RETURN` is:

```
#define INTERRUPT_RETURN    jmp native_iret
...
```

```
ENTRY(native_iret)
.global native_irq_return_iret
native_irq_return_iret:
iretq
```

More about the `idtentry` macro you can read in the third part of the http://0xax.gitbooks.io/linux-insides/content/interrupts/interrupts-3.html chapter. Ok, now we saw the preparation before an exception handler will be executed and now time to look on the handlers. First of all let's look on the following handlers:

- divide_error
- overflow
- invalid_op
- coprocessor_segment_overrun
- invalid_TSS
- segment_not_present
- stack_segment
- alignment_check

All these handlers defined in the arch/x86/kernel/traps.c source code file with the `DO_ERROR` macro:

```
DO_ERROR(X86_TRAP_DE,     SIGFPE,  "divide error",                divide_error)
DO_ERROR(X86_TRAP_OF,     SIGSEGV, "overflow",                    overflow)
DO_ERROR(X86_TRAP_UD,     SIGILL,  "invalid opcode",              invalid_op)
DO_ERROR(X86_TRAP_OLD_MF, SIGFPE,  "coprocessor segment overrun", coprocessor_segment_overrun)
DO_ERROR(X86_TRAP_TS,     SIGSEGV, "invalid TSS",                 invalid_TSS)
DO_ERROR(X86_TRAP_NP,     SIGBUS,  "segment not present",         segment_not_present)
DO_ERROR(X86_TRAP_SS,     SIGBUS,  "stack segment",               stack_segment)
DO_ERROR(X86_TRAP_AC,     SIGBUS,  "alignment check",             alignment_check)
```

As we can see the `DO_ERROR` macro takes 4 parameters:

- Vector number of an interrupt;
- Signal number which will be sent to the interrupted process;
- String which describes an exception;
- Exception handler entry point.

This macro defined in the same source code file and expands to the function with the `do_handler` name:

```
#define DO_ERROR(trapnr, signr, str, name)                              \
dotraplinkage void do_##name(struct pt_regs *regs, long error_code)     \
{                                                                       \
        do_error_trap(regs, error_code, str, trapnr, signr);            \
}
```

Note on the `##` tokens. This is special feature - GCC macro Concatenation which concatenates two given strings. For example, first `DO_ERROR` in our example will expands to the:

```
dotraplinkage void do_divide_error(struct pt_regs *regs, long error_code)     \
{
    ...
}
```

We can see that all functions which are generated by the `DO_ERROR` macro just make a call of the `do_error_trap` function from the arch/x86/kernel/traps.c. Let's look on implementation of the `do_error_trap` function.

Trap handlers

The `do_error_trap` function starts and ends from the two following functions:

```
enum ctx_state prev_state = exception_enter();
...
...
...
exception_exit(prev_state);
```

from the include/linux/context_tracking.h. The context tracking in the Linux kernel subsystem which provide kernel boundaries probes to keep track of the transitions between level contexts with two basic initial contexts: `user` or `kernel`. The `exception_enter` function checks that context tracking is enabled. After this if it is enabled, the `exception_enter` reads previous context and compares it with the `CONTEXT_KERNEL`. If the previous context is `user`, we call `context_tracking_exit` function from the kernel/context_tracking.c which inform the context tracking subsystem that a processor is exiting user mode and entering the kernel mode:

```
if (!context_tracking_is_enabled())
    return 0;

prev_ctx = this_cpu_read(context_tracking.state);
if (prev_ctx != CONTEXT_KERNEL)
    context_tracking_exit(prev_ctx);

return prev_ctx;
```

If previous context is non `user`, we just return it. The `pre_ctx` has `enum ctx_state` type which defined in the include/linux/context_tracking_state.h and looks as:

```
enum ctx_state {
    CONTEXT_KERNEL = 0,
    CONTEXT_USER,
    CONTEXT_GUEST,
} state;
```

The second function is `exception_exit` defined in the same include/linux/context_tracking.h file and checks that context tracking is enabled and call the `contert_tracking_enter` function if the previous context was `user`:

```
static inline void exception_exit(enum ctx_state prev_ctx)
{
    if (context_tracking_is_enabled()) {
        if (prev_ctx != CONTEXT_KERNEL)
            context_tracking_enter(prev_ctx);
    }
}
```

The `context_tracking_enter` function informs the context tracking subsystem that a processor is going to enter to the user mode from the kernel mode. We can see the following code between the `exception_enter` and `exception_exit`:

```
if (notify_die(DIE_TRAP, str, regs, error_code, trapnr, signr) !=
        NOTIFY_STOP) {
    conditional_sti(regs);
    do_trap(trapnr, signr, str, regs, error_code,
        fill_trap_info(regs, signr, trapnr, &info));
}
```

First of all it calls the `notify_die` function which defined in the kernel/notifier.c. To get notified for kernel panic, kernel oops, Non-Maskable Interrupt or other events the caller needs to insert itself in the `notify_die` chain and the `notify_die` function does it. The Linux kernel has special mechanism that allows kernel to ask when something happens and this mechanism called `notifiers` or `notifier chains`. This mechanism used for example for the `USB` hotplug events (look on the drivers/usb/core/notify.c), for the memory hotplug (look on the include/linux/memory.h, the `hotplug_memory_notifier` macro and etc...), system reboots and etc. A notifier chain is thus a simple, singly-linked list. When a Linux kernel subsystem wants to be notified of specific events, it fills out a

special `notifier_block` structure and passes it to the `notifier_chain_register` function. An event can be sent with the call of the `notifier_call_chain` function. First of all the `notify_die` function fills `die_args` structure with the trap number, trap string, registers and other values:

```
struct die_args args = {
       .regs   = regs,
       .str    = str,
       .err    = err,
       .trapnr = trap,
       .signr  = sig,
}
```

and returns the result of the `atomic_notifier_call_chain` function with the `die_chain`:

```
static ATOMIC_NOTIFIER_HEAD(die_chain);
return atomic_notifier_call_chain(&die_chain, val, &args);
```

which just expands to the `atomic_notifier_head` structure that contains lock and `notifier_block`:

```
struct atomic_notifier_head {
        spinlock_t lock;
        struct notifier_block __rcu *head;
};
```

The `atomic_notifier_call_chain` function calls each function in a notifier chain in turn and returns the value of the last notifier function called. If the `notify_die` in the `do_error_trap` does not return `NOTIFY_STOP` we execute `conditional_sti` function from the arch/x86/kernel/traps.c that checks the value of the interrupt flag and enables interrupt depends on it:

```
static inline void conditional_sti(struct pt_regs *regs)
{
        if (regs->flags & X86_EFLAGS_IF)
                local_irq_enable();
}
```

more about `local_irq_enable` macro you can read in the second part of this chapter. The next and last call in the `do_error_trap` is the `do_trap` function. First of all the `do_trap` function defined the `tsk` variable which has `task_struct` type and represents the current interrupted process. After the definition of the `tsk`, we can see the call of the `do_trap_no_signal` function:

```
struct task_struct *tsk = current;

if (!do_trap_no_signal(tsk, trapnr, str, regs, error_code))
    return;
```

The `do_trap_no_signal` function makes two checks:

- Did we come from the Virtual 8086 mode;
- Did we come from the kernelspace.

```
if (v8086_mode(regs)) {
    ...
}

if (!user_mode(regs)) {
    ...
}

return -1;
```

We will not consider first case because the long mode does not support the Virtual 8086 mode. In the second case we invoke `fixup_exception` function which will try to recover a fault and `die` if we can't:

```
if (!fixup_exception(regs)) {
    tsk->thread.error_code = error_code;
    tsk->thread.trap_nr = trapnr;
    die(str, regs, error_code);
}
```

The `die` function defined in the arch/x86/kernel/dumpstack.c source code file, prints useful information about stack, registers, kernel modules and caused kernel oops. If we came from the userspace the `do_trap_no_signal` function will return `-1` and the execution of the `do_trap` function will continue. If we passed through the `do_trap_no_signal` function and did not exit from the `do_trap` after this, it means that previous context was - `user`. Most exceptions caused by the processor are interpreted by Linux as error conditions, for example division by zero, invalid opcode and etc. When an exception occurs the Linux kernel sends a signal to the interrupted process that caused the exception to notify it of an incorrect condition. So, in the `do_trap` function we need to send a signal with the given number (`SIGFPE` for the divide error, `SIGILL` for the overflow exception and etc...). First of all we save error code and vector number in the current interrupts process with the filling `thread.error_code` and `thread_trap_nr`:

```
tsk->thread.error_code = error_code;
tsk->thread.trap_nr = trapnr;
```

After this we make a check do we need to print information about unhandled signals for the interrupted process. We check that `show_unhandled_signals` variable is set, that `unhandled_signal` function from the kernel/signal.c will return unhandled signal(s) and printk rate limit:

```
#ifdef CONFIG_X86_64
    if (show_unhandled_signals && unhandled_signal(tsk, signr) &&
        printk_ratelimit()) {
        pr_info("%s[%d] trap %s ip:%lx sp:%lx error:%lx",
            tsk->comm, tsk->pid, str,
            regs->ip, regs->sp, error_code);
        print_vma_addr(" in ", regs->ip);
        pr_cont("\n");
    }
#endif
```

And send a given signal to interrupted process:

```
force_sig_info(signr, info ?: SEND_SIG_PRIV, tsk);
```

This is the end of the `do_trap`. We just saw generic implementation for eight different exceptions which are defined with the `DO_ERROR` macro. Now let's look on another exception handlers.

Double fault

The next exception is `#DF` or `Double fault`. This exception occurs when the processor detected a second exception while calling an exception handler for a prior exception. We set the trap gate for this exception in the previous part:

```
set_intr_gate_ist(X86_TRAP_DF, &double_fault, DOUBLEFAULT_STACK);
```

Note that this exception runs on the `DOUBLEFAULT_STACK` Interrupt Stack Table which has index - `1`:

```
#define DOUBLEFAULT_STACK 1
```

The `double_fault` is handler for this exception and defined in the arch/x86/kernel/traps.c. The `double_fault` handler starts from the definition of two variables: string that describes exception and interrupted process, as other exception handlers:

```
static const char str[] = "double fault";
```

```
struct task_struct *tsk = current;
```

The handler of the double fault exception split on two parts. The first part is the check which checks that a fault is a `non-IST` fault on the `espfix64` stack. Actually the `iret` instruction restores only the bottom `16` bits when returning to a `16` bit segment. The `espfix` feature solves this problem. So if the `non-IST` fault on the espfix64 stack we modify the stack to make it look like `General Protection Fault` :

```
struct pt_regs *normal_regs = task_pt_regs(current);

memmove(&normal_regs->ip, (void *)regs->sp, 5*8);
ormal_regs->orig_ax = 0;
regs->ip = (unsigned long)general_protection;
regs->sp = (unsigned long)&normal_regs->orig_ax;
return;
```

In the second case we do almost the same that we did in the previous exception handlers. The first is the call of the `ist_enter` function that discards previous context, `user` in our case:

```
ist_enter(regs);
```

And after this we fill the interrupted process with the vector number of the `Double fault` exception and error code as we did it in the previous handlers:

```
tsk->thread.error_code = error_code;
tsk->thread.trap_nr = X86_TRAP_DF;
```

Next we print useful information about the double fault (PID number, registers content):

```
#ifdef CONFIG_DOUBLEFAULT
    df_debug(regs, error_code);
#endif
```

And die:

```
    for (;;)
        die(str, regs, error_code);
```

That's all.

Device not available exception handler

The next exception is the `#NM` or `Device not available` . The `Device not available` exception can occur depending on these things:

- The processor executed an x87 FPU floating-point instruction while the EM flag in control register `cr0` was set;
- The processor executed a `wait` or `fwait` instruction while the `MP` and `TS` flags of register `cr0` were set;
- The processor executed an x87 FPU, MMX or SSE instruction while the `TS` flag in control register `cr0` was set and the `EM` flag is clear.

The handler of the `Device not available` exception is the `do_device_not_available` function and it defined in the arch/x86/kernel/traps.c source code file too. It starts and ends from the getting of the previous context, as other traps which we saw in the beginning of this part:

```
enum ctx_state prev_state;
prev_state = exception_enter();
...
```

```
...
...
exception_exit(prev_state);
```

In the next step we check that `FPU` is not eager:

```
BUG_ON(use_eager_fpu());
```

When we switch into a task or interrupt we may avoid loading the `FPU` state. If a task will use it, we catch `Device not Available exception` exception. If we loading the `FPU` state during task switching, the `FPU` is eager. In the next step we check `cr0` control register on the `EM` flag which can show us is `x87` floating point unit present (flag clear) or not (flag set):

```
#ifdef CONFIG_MATH_EMULATION
    if (read_cr0() & X86_CR0_EM) {
        struct math_emu_info info = { };

        conditional_sti(regs);

        info.regs = regs;
        math_emulate(&info);
        exception_exit(prev_state);
        return;
    }
#endif
```

If the `x87` floating point unit not presented, we enable interrupts with the `conditional_sti`, fill the `math_emu_info` (defined in the arch/x86/include/asm/math_emu.h) structure with the registers of an interrupt task and call `math_emulate` function from the arch/x86/math-emu/fpu_entry.c. As you can understand from function's name, it emulates `X87 FPU` unit (more about the `x87` we will know in the special chapter). In other way, if `X86_CR0_EM` flag is clear which means that `x87 FPU` unit is presented, we call the `fpu__restore` function from the arch/x86/kernel/fpu/core.c which copies the `FPU` registers from the `fpustate` to the live hardware registers. After this `FPU` instructions can be used:

```
fpu__restore(&current->thread.fpu);
```

General protection fault exception handler

The next exception is the `#GP` or `General protection fault`. This exception occurs when the processor detected one of a class of protection violations called `general-protection violations`. It can be:

- Exceeding the segment limit when accessing the `cs`, `ds`, `es`, `fs` or `gs` segments;
- Loading the `ss`, `ds`, `es`, `fs` or `gs` register with a segment selector for a system segment.;
- Violating any of the privilege rules;
- and other...

The exception handler for this exception is the `do_general_protection` from the arch/x86/kernel/traps.c. The `do_general_protection` function starts and ends as other exception handlers from the getting of the previous context:

```
prev_state = exception_enter();
...
exception_exit(prev_state);
```

After this we enable interrupts if they were disabled and check that we came from the Virtual 8086 mode:

```
conditional_sti(regs);

if (v8086_mode(regs)) {
    local_irq_enable();
    handle_vm86_fault((struct kernel_vm86_regs *) regs, error_code);
```

```
    goto exit;
}
```

As long mode does not support this mode, we will not consider exception handling for this case. In the next step check that previous mode was kernel mode and try to fix the trap. If we can't fix the current general protection fault exception we fill the interrupted process with the vector number and error code of the exception and add it to the `notify_die` chain:

```
if (!user_mode(regs)) {
    if (fixup_exception(regs))
        goto exit;

    tsk->thread.error_code = error_code;
    tsk->thread.trap_nr = X86_TRAP_GP;
    if (notify_die(DIE_GPF, "general protection fault", regs, error_code,
                X86_TRAP_GP, SIGSEGV) != NOTIFY_STOP)
        die("general protection fault", regs, error_code);
    goto exit;
}
```

If we can fix exception we go to the `exit` label which exits from exception state:

```
exit:
    exception_exit(prev_state);
```

If we came from user mode we send `SIGSEGV` signal to the interrupted process from user mode as we did it in the `do_trap` function:

```
if (show_unhandled_signals && unhandled_signal(tsk, SIGSEGV) &&
        printk_ratelimit()) {
    pr_info("%s[%d] general protection ip:%lx sp:%lx error:%lx",
        tsk->comm, task_pid_nr(tsk),
        regs->ip, regs->sp, error_code);
    print_vma_addr(" in ", regs->ip);
    pr_cont("\n");
}

force_sig_info(SIGSEGV, SEND_SIG_PRIV, tsk);
```

That's all.

Conclusion

It is the end of the fifth part of the Interrupts and Interrupt Handling chapter and we saw implementation of some interrupt handlers in this part. In the next part we will continue to dive into interrupt and exception handlers and will see handler for the Non-Maskable Interrupts, handling of the math coprocessor and SIMD coprocessor exceptions and many many more.

If you have any questions or suggestions write me a comment or ping me at twitter.

Please note that English is not my first language, And I am really sorry for any inconvenience. If you find any mistakes please send me PR to linux-insides.

Links

- Interrupt descriptor Table
- iret instruction
- GCC macro Concatenation
- kernel panic
- kernel oops
- Non-Maskable Interrupt

- hotplug
- interrupt flag
- long mode
- signal
- printk
- coprocessor
- SIMD
- Interrupt Stack Table
- PID
- x87 FPU
- control register
- MMX
- Previous part

Interrupts and Interrupt Handling. Part 6.

Non-maskable interrupt handler

It is sixth part of the Interrupts and Interrupt Handling in the Linux kernel chapter and in the previous part we saw implementation of some exception handlers for the General Protection Fault exception, divide exception, invalid opcode exceptions and etc. As I wrote in the previous part we will see implementations of the rest exceptions in this part. We will see implementation of the following handlers:

- Non-Maskable interrupt;
- BOUND Range Exceeded Exception;
- Coprocessor exception;
- SIMD coprocessor exception.

in this part. So, let's start.

Non-Maskable interrupt handling

A Non-Maskable interrupt is a hardware interrupt that cannot be ignored by standard masking techniques. In a general way, a non-maskable interrupt can be generated in either of two ways:

- External hardware asserts the non-maskable interrupt pin on the CPU.
- The processor receives a message on the system bus or the APIC serial bus with a delivery mode `NMI` .

When the processor receives a `NMI` from one of these sources, the processor handles it immediately by calling the `NMI` handler pointed to by interrupt vector which has number `2` (see table in the first part). We already filled the Interrupt Descriptor Table with the vector number, address of the `nmi` interrupt handler and `NMI_STACK` Interrupt Stack Table entry:

```
set_intr_gate_ist(X86_TRAP_NMI, &nmi, NMI_STACK);
```

in the `trap_init` function which defined in the arch/x86/kernel/traps.c source code file. In the previous parts we saw that entry points of the all interrupt handlers are defined with the:

```
.macro idtentry sym do_sym has_error_code:req paranoid=0 shift_ist=-1
ENTRY(\sym)
...
...
...
END(\sym)
.endm
```

macro from the arch/x86/entry/entry_64.S assembly source code file. But the handler of the `Non-Maskable` interrupts is not defined with this macro. It has own entry point:

```
ENTRY(nmi)
...
...
...
END(nmi)
```

in the same arch/x86/entry/entry_64.S assembly file. Lets dive into it and will try to understand how `Non-Maskable` interrupt handler works. The `nmi` handlers starts from the call of the:

```
PARAVIRT_ADJUST_EXCEPTION_FRAME
```

macro but we will not dive into details about it in this part, because this macro related to the Paravirtualization stuff which we will see in another chapter. After this save the content of the `rdx` register on the stack:

```
pushq   %rdx
```

And allocated check that `cs` was not the kernel segment when an non-maskable interrupt occurs:

```
cmpl    $__KERNEL_CS, 16(%rsp)
jne    first_nmi
```

The `__KERNEL_CS` macro defined in the arch/x86/include/asm/segment.h and represented second descriptor in the Global Descriptor Table:

```
#define GDT_ENTRY_KERNEL_CS    2
#define __KERNEL_CS    (GDT_ENTRY_KERNEL_CS*8)
```

more about `GDT` you can read in the second part of the Linux kernel booting process chapter. If `cs` is not kernel segment, it means that it is not nested `NMI` and we jump on the `first_nmi` label. Let's consider this case. First of all we put address of the current stack pointer to the `rdx` and pushes `1` to the stack in the `first_nmi` label:

```
first_nmi:
    movq    (%rsp), %rdx
    pushq    $1
```

Why do we push `1` on the stack? As the comment says: `We allow breakpoints in NMIs`. On the x86_64, like other architectures, the CPU will not execute another `NMI` until the first `NMI` is completed. A `NMI` interrupt finished with the iret instruction like other interrupts and exceptions do it. If the `NMI` handler triggers either a page fault or breakpoint or another exception which are use `iret` instruction too. If this happens while in `NMI` context, the CPU will leave `NMI` context and a new `NMI` may come in. The `iret` used to return from those exceptions will re-enable `NMIs` and we will get nested non-maskable interrupts. The problem the `NMI` handler will not return to the state that it was, when the exception triggered, but instead it will return to a state that will allow new `NMIs` to preempt the running `NMI` handler. If another `NMI` comes in before the first NMI handler is complete, the new NMI will write all over the preempted `NMIs` stack. We can have nested `NMIs` where the next `NMI` is using the top of the stack of the previous `NMI`. It means that we cannot execute it because a nested non-maskable interrupt will corrupt stack of a previous non-maskable interrupt. That's why we have allocated space on the stack for temporary variable. We will check this variable that it was set when a previous `NMI` is executing and clear if it is not nested `NMI`. We push `1` here to the previously allocated space on the stack to denote that a `non-maskable` interrupt executed currently. Remember that when and `NMI` or another exception occurs we have the following stack frame:

```
+------------------------+
|         SS             |
|         RSP            |
|        RFLAGS          |
|         CS             |
|         RIP            |
+------------------------+
```

and also an error code if an exception has it. So, after all of these manipulations our stack frame will look like this:

```
+------------------------+
|         SS             |
|         RSP            |
|        RFLAGS          |
|         CS             |
|         RIP            |
|         RDX            |
|          1             |
+------------------------+
```

In the next step we allocate yet another `40` bytes on the stack:

```
subq    $(5*8), %rsp
```

and pushes the copy of the original stack frame after the allocated space:

```
.rept 5
pushq    11*8(%rsp)
.endr
```

with the .rept assembly directive. We need in the copy of the original stack frame. Generally we need in two copies of the interrupt stack. First is `copied` interrupts stack: `saved` stack frame and `copied` stack frame. Now we pushes original stack frame to the `saved` stack frame which locates after the just allocated `40` bytes (`copied` stack frame). This stack frame is used to fixup the `copied` stack frame that a nested NMI may change. The second - `copied` stack frame modified by any nested `NMIs` to let the first `NMI` know that we triggered a second `NMI` and we should repeat the first `NMI` handler. Ok, we have made first copy of the original stack frame, now time to make second copy:

```
addq    $(10*8), %rsp

.rept 5
pushq    -6*8(%rsp)
.endr
subq    $(5*8), %rsp
```

After all of these manipulations our stack frame will be like this:

```
+-------------------------+
| original SS             |
| original Return RSP     |
| original RFLAGS         |
| original CS             |
| original RIP            |
+-------------------------+
| temp storage for rdx    |
+-------------------------+
| NMI executing variable  |
+-------------------------+
| copied SS               |
| copied Return RSP       |
| copied RFLAGS           |
| copied CS               |
| copied RIP              |
+-------------------------+
| Saved SS                |
| Saved Return RSP        |
| Saved RFLAGS            |
| Saved CS                |
| Saved RIP               |
+-------------------------+
```

After this we push dummy error code on the stack as we did it already in the previous exception handlers and allocate space for the general purpose registers on the stack:

```
pushq    $-1
ALLOC_PT_GPREGS_ON_STACK
```

We already saw implementation of the `ALLOC_PT_GREGS_ON_STACK` macro in the third part of the interrupts chapter. This macro defined in the arch/x86/entry/calling.h and yet another allocates `120` bytes on stack for the general purpose registers, from the `rdi` to the `r15`:

```
.macro ALLOC_PT_GPREGS_ON_STACK addskip=0
```

```
addq    $-(15*8+\addskip), %rsp
.endm
```

After space allocation for the general registers we can see call of the `paranoid_entry`:

```
call    paranoid_entry
```

We can remember from the previous parts this label. It pushes general purpose registers on the stack, reads `MSR_GS_BASE` Model Specific register and checks its value. If the value of the `MSR_GS_BASE` is negative, we came from the kernel mode and just return from the `paranoid_entry`, in other way it means that we came from the usermode and need to execute `swapgs` instruction which will change user `gs` with the kernel `gs`:

```
ENTRY(paranoid_entry)
    cld
    SAVE_C_REGS 8
    SAVE_EXTRA_REGS 8
    movl    $1, %ebx
    movl    $MSR_GS_BASE, %ecx
    rdmsr
    testl    %edx, %edx
    js    1f
    SWAPGS
    xorl    %ebx, %ebx
1:    ret
END(paranoid_entry)
```

Note that after the `swapgs` instruction we zeroed the `ebx` register. Next time we will check content of this register and if we executed `swapgs` than `ebx` must contain `0` and `1` in other way. In the next step we store value of the `cr2` control register to the `r12` register, because the `NMI` handler can cause `page fault` and corrupt the value of this control register:

```
movq    %cr2, %r12
```

Now time to call actual `NMI` handler. We push the address of the `pt_regs` to the `rdi`, error code to the `rsi` and call the `do_nmi` handler:

```
movq    %rsp, %rdi
movq    $-1, %rsi
call    do_nmi
```

We will back to the `do_nmi` little later in this part, but now let's look what occurs after the `do_nmi` will finish its execution. After the `do_nmi` handler will be finished we check the `cr2` register, because we can got page fault during `do_nmi` performed and if we got it we restore original `cr2`, in other way we jump on the label `1`. After this we test content of the `ebx` register (remember it must contain `0` if we have used `swapgs` instruction and `1` if we didn't use it) and execute `SWAPGS_UNSAFE_STACK` if it contains `1` or jump to the `nmi_restore` label. The `SWAPGS_UNSAFE_STACK` macro just expands to the `swapgs` instruction. In the `nmi_restore` label we restore general purpose registers, clear allocated space on the stack for this registers, clear our temporary variable and exit from the interrupt handler with the `INTERRUPT_RETURN` macro:

```
    movq    %cr2, %rcx
    cmpq    %rcx, %r12
    je    1f
    movq    %r12, %cr2
1:
    testl    %ebx, %ebx
    jnz    nmi_restore
nmi_swapgs:
    SWAPGS_UNSAFE_STACK
nmi_restore:
    RESTORE_EXTRA_REGS
    RESTORE_C_REGS
    /* Pop the extra iret frame at once */
```

```
    REMOVE_PT_GPREGS_FROM_STACK 6*8
    /* Clear the NMI executing stack variable */
    movq    $0, 5*8(%rsp)
    INTERRUPT_RETURN
```

where `INTERRUPT_RETURN` is defined in the arch/x86/include/irqflags.h and just expands to the `iret` instruction. That's all.

Now let's consider case when another `NMI` interrupt occurred when previous `NMI` interrupt didn't finish its execution. You can remember from the beginning of this part that we've made a check that we came from userspace and jump on the `first_nmi` in this case:

```
cmpl    $__KERNEL_CS, 16(%rsp)
jne    first_nmi
```

Note that in this case it is first `NMI` every time, because if the first `NMI` catched page fault, breakpoint or another exception it will be executed in the kernel mode. If we didn't come from userspace, first of all we test our temporary variable:

```
cmpl    $1, -8(%rsp)
je    nested_nmi
```

and if it is set to `1` we jump to the `nested_nmi` label. If it is not `1`, we test the `IST` stack. In the case of nested `NMIs` we check that we are above the `repeat_nmi`. In this case we ignore it, in other way we check that we above than `end_repeat_nmi` and jump on the `nested_nmi_out` label.

Now let's look on the `do_nmi` exception handler. This function defined in the arch/x86/kernel/nmi.c source code file and takes two parameters:

- address of the `pt_regs`;
- error code.

as all exception handlers. The `do_nmi` starts from the call of the `nmi_nesting_preprocess` function and ends with the call of the `nmi_nesting_postprocess`. The `nmi_nesting_preprocess` function checks that we likely do not work with the debug stack and if we on the debug stack set the `update_debug_stack` per-cpu variable to `1` and call the `debug_stack_set_zero` function from the arch/x86/kernel/cpu/common.c. This function increases the `debug_stack_use_ctr` per-cpu variable and loads new `Interrupt Descriptor Table`:

```
static inline void nmi_nesting_preprocess(struct pt_regs *regs)
{
        if (unlikely(is_debug_stack(regs->sp))) {
                debug_stack_set_zero();
                this_cpu_write(update_debug_stack, 1);
        }
}
```

The `nmi_nesting_postprocess` function checks the `update_debug_stack` per-cpu variable which we set in the `nmi_nesting_preprocess` and resets debug stack or in another words it loads origin `Interrupt Descriptor Table`. After the call of the `nmi_nesting_preprocess` function, we can see the call of the `nmi_enter` in the `do_nmi`. The `nmi_enter` increases `lockdep_recursion` field of the interrupted process, update preempt counter and informs the RCU subsystem about `NMI`. There is also `nmi_exit` function that does the same stuff as `nmi_enter`, but vice-versa. After the `nmi_enter` we increase `__nmi_count` in the `irq_stat` structure and call the `default_do_nmi` function. First of all in the `default_do_nmi` we check the address of the previous nmi and update address of the last nmi to the actual:

```
if (regs->ip == __this_cpu_read(last_nmi_rip))
    b2b = true;
else
    __this_cpu_write(swallow_nmi, false);

__this_cpu_write(last_nmi_rip, regs->ip);
```

After this first of all we need to handle CPU-specific `NMIs`:

```
handled = nmi_handle(NMI_LOCAL, regs, b2b);
__this_cpu_add(nmi_stats.normal, handled);
```

And then non-specific `NMIs` depends on its reason:

```
reason = x86_platform.get_nmi_reason();
if (reason & NMI_REASON_MASK) {
    if (reason & NMI_REASON_SERR)
        pci_serr_error(reason, regs);
    else if (reason & NMI_REASON_IOCHK)
        io_check_error(reason, regs);

    __this_cpu_add(nmi_stats.external, 1);
    return;
}
```

That's all.

Range Exceeded Exception

The next exception is the `BOUND` range exceeded exception. The `BOUND` instruction determines if the first operand (array index) is within the bounds of an array specified the second operand (bounds operand). If the index is not within bounds, a `BOUND` range exceeded exception or `#BR` is occurred. The handler of the `#BR` exception is the `do_bounds` function that defined in the arch/x86/kernel/traps.c. The `do_bounds` handler starts with the call of the `exception_enter` function and ends with the call of the `exception_exit`:

```
prev_state = exception_enter();

if (notify_die(DIE_TRAP, "bounds", regs, error_code,
               X86_TRAP_BR, SIGSEGV) == NOTIFY_STOP)
    goto exit;
...
...
...
exception_exit(prev_state);
return;
```

After we have got the state of the previous context, we add the exception to the `notify_die` chain and if it will return `NOTIFY_STOP` we return from the exception. More about notify chains and the `context tracking` functions you can read in the previous part. In the next step we enable interrupts if they were disabled with the `contidional_sti` function that checks `IF` flag and call the `local_irq_enable` depends on its value:

```
conditional_sti(regs);

if (!user_mode(regs))
    die("bounds", regs, error_code);
```

and check that if we didn't came from user mode we send `SIGSEGV` signal with the `die` function. After this we check is MPX enabled or not, and if this feature is disabled we jump on the `exit_trap` label:

```
if (!cpu_feature_enabled(X86_FEATURE_MPX)) {
    goto exit_trap;
}

where we execute `do_trap` function (more about it you can find in the previous part):

```C
exit_trap:
```
```

```
    do_trap(X86_TRAP_BR, SIGSEGV, "bounds", regs, error_code, NULL);
    exception_exit(prev_state);
```

If `MPX` feature is enabled we check the `BNDSTATUS` with the `get_xsave_field_ptr` function and if it is zero, it means that the `MPX` was not responsible for this exception:

```
bndcsr = get_xsave_field_ptr(XSTATE_BNDCSR);
if (!bndcsr)
        goto exit_trap;
```

After all of this, there is still only one way when `MPX` is responsible for this exception. We will not dive into the details about Intel Memory Protection Extensions in this part, but will see it in another chapter.

Coprocessor exception and SIMD exception

The next two exceptions are x87 FPU Floating-Point Error exception or `#MF` and SIMD Floating-Point Exception or `#XF`. The first exception occurs when the `x87 FPU` has detected floating point error. For example divide by zero, numeric overflow and etc. The second exception occurs when the processor has detected SSE/SSE2/SSE3 `SIMD` floating-point exception. It can be the same as for the `x87 FPU`. The handlers for these exceptions are `do_coprocessor_error` and `do_simd_coprocessor_error` are defined in the arch/x86/kernel/traps.c and very similar on each other. They both make a call of the `math_error` function from the same source code file but pass different vector number. The `do_coprocessor_error` passes `X86_TRAP_MF` vector number to the `math_error`:

```
dotraplinkage void do_coprocessor_error(struct pt_regs *regs, long error_code)
{
    enum ctx_state prev_state;

    prev_state = exception_enter();
    math_error(regs, error_code, X86_TRAP_MF);
    exception_exit(prev_state);
}
```

and `do_simd_coprocessor_error` passes `X86_TRAP_XF` to the `math_error` function:

```
dotraplinkage void
do_simd_coprocessor_error(struct pt_regs *regs, long error_code)
{
    enum ctx_state prev_state;

    prev_state = exception_enter();
    math_error(regs, error_code, X86_TRAP_XF);
    exception_exit(prev_state);
}
```

First of all the `math_error` function defines current interrupted task, address of its fpu, string which describes an exception, add it to the `notify_die` chain and return from the exception handler if it will return `NOTIFY_STOP`:

```
    struct task_struct *task = current;
    struct fpu *fpu = &task->thread.fpu;
    siginfo_t info;
    char *str = (trapnr == X86_TRAP_MF) ? "fpu exception" :
                        "simd exception";

    if (notify_die(DIE_TRAP, str, regs, error_code, trapnr, SIGFPE) == NOTIFY_STOP)
        return;
```

After this we check that we are from the kernel mode and if yes we will try to fix an exception with the `fixup_exception` function. If we cannot we fill the task with the exception's error code and vector number and die:

```
if (!user_mode(regs)) {
```

```
    if (!fixup_exception(regs)) {
        task->thread.error_code = error_code;
        task->thread.trap_nr = trapnr;
        die(str, regs, error_code);
    }
    return;
}
```

If we came from the user mode, we save the `fpu` state, fill the task structure with the vector number of an exception and `siginfo_t` with the number of signal, `errno`, the address where exception occurred and signal code:

```
fpu__save(fpu);

task->thread.trap_nr    = trapnr;
task->thread.error_code = error_code;
info.si_signo        = SIGFPE;
info.si_errno        = 0;
info.si_addr        = (void __user *)uprobe_get_trap_addr(regs);
info.si_code = fpu__exception_code(fpu, trapnr);
```

After this we check the signal code and if it is non-zero we return:

```
if (!info.si_code)
    return;
```

Or send the `SIGFPE` signal in the end:

```
force_sig_info(SIGFPE, &info, task);
```

That's all.

Conclusion

It is the end of the sixth part of the Interrupts and Interrupt Handling chapter and we saw implementation of some exception handlers in this part, like `non-maskable` interrupt, SIMD and x87 FPU floating point exception. Finally we have finsihed with the `trap_init` function in this part and will go ahead in the next part. The next our point is the external interrupts and the `early_irq_init` function from the init/main.c.

If you have any questions or suggestions write me a comment or ping me at twitter.

Please note that English is not my first language, And I am really sorry for any inconvenience. If you find any mistakes please send me PR to linux-insides.

Links

- General Protection Fault
- opcode
- Non-Maskable
- BOUND instruction
- CPU socket
- Interrupt Descriptor Table
- Interrupt Stack Table
- Paravirtualization
- .rept
- SIMD
- Coprocessor

- x86_64
- iret
- page fault
- breakpoint
- Global Descriptor Table
- stack frame
- Model Specific regiser
- percpu
- RCU
- MPX
- x87 FPU
- Previous part

Interrupts and Interrupt Handling. Part 7.

Introduction to external interrupts

This is the seventh part of the Interrupts and Interrupt Handling in the Linux kernel chapter and in the previous part we have finished with the exceptions which are generated by the processor. In this part we will continue to dive to the interrupt handling and will start with the external hardware interrupt handling. As you can remember, in the previous part we have finished with the `trap_init` function from the arch/x86/kernel/trap.c and the next step is the call of the `early_irq_init` function from the init/main.c.

Interrupts are signal that are sent across IRQ or `Interrupt Request Line` by a hardware or software. External hardware interrupts allow devices like keyboard, mouse and etc, to indicate that it needs attention of the processor. Once the processor receives the `Interrupt Request`, it will temporary stop execution of the running program and invoke special routine which depends on an interrupt. We already know that this routine is called interrupt handler (or how we will call it `ISR` or `Interrupt Service Routine` from this part). The `ISR` or `Interrupt Handler Routine` can be found in Interrupt Vector table that is located at fixed address in the memory. After the interrupt is handled processor resumes the interrupted process. At the boot/initialization time, the Linux kernel identifies all devices in the machine, and appropriate interrupt handlers are loaded into the interrupt table. As we saw in the previous parts, most exceptions are handled simply by the sending a Unix signal to the interrupted process. That's why kernel is can handle an exception quickly. Unfortunately we can not use this approach for the external hardware interrupts, because often they arrive after (and sometimes long after) the process to which they are related has been suspended. So it would make no sense to send a Unix signal to the current process. External interrupt handling depends on the type of an interrupt:

- `I/O` interrupts;
- Timer interrupts;
- Interprocessor interrupts.

I will try to describe all types of interrupts in this book.

Generally, a handler of an `I/O` interrupt must be flexible enough to service several devices at the same time. For example in the PCI bus architecture several devices may share the same `IRQ` line. In the simplest way the Linux kernel must do following thing when an `I/O` interrupt occurred:

- Save the value of an `IRQ` and the register's contents on the kernel stack;
- Send an acknowledgment to the hardware controller which is servicing the `IRQ` line;
- Execute the interrupt service routine (next we will call it `ISR`) which is associated with the device;
- Restore registers and return from an interrupt;

Ok, we know a little theory and now let's start with the `early_irq_init` function. The implementation of the `early_irq_init` function is in the kernel/irq/irqdesc.c. This function make early initialization of the `irq_desc` structure. The `irq_desc` structure is the foundation of interrupt management code in the Linux kernel. An array of this structure, which has the same name - `irq_desc`, keeps track of every interrupt request source in the Linux kernel. This structure defined in the include/linux/irqdesc.h and as you can note it depends on the `CONFIG_SPARSE_IRQ` kernel configuration option. This kernel configuration option enables support for sparse irqs. The `irq_desc` structure contains many different files:

- `irq_common_data` - per irq and chip data passed down to chip functions;
- `status_use_accessors` - contains status of the interrupt source which is combination of the values from the `enum` from the include/linux/irq.h and different macros which are defined in the same source code file;
- `kstat_irqs` - irq stats per-cpu;
- `handle_irq` - highlevel irq-events handler;
- `action` - identifies the interrupt service routines to be invoked when the IRQ occurs;
- `irq_count` - counter of interrupt occurrences on the IRQ line;
- `depth` - `0` if the IRQ line is enabled and a positive value if it has been disabled at least once;
- `last_unhandled` - aging timer for unhandled count;
- `irqs_unhandled` - count of the unhandled interrupts;
- `lock` - a spin lock used to serialize the accesses to the `IRQ` descriptor;

- `pending_mask` - pending rebalanced interrupts;
- `owner` - an owner of interrupt descriptor. Interrupt descriptors can be allocated from modules. This field is need to proved refcount on the module which provides the interrupts;
- and etc.

Of course it is not all fields of the `irq_desc` structure, because it is too long to describe each field of this structure, but we will see it all soon. Now let's start to dive into the implementation of the `early_irq_init` function.

Early external interrupts initialization

Now, let's look on the implementation of the `early_irq_init` function. Note that implementation of the `early_irq_init` function depends on the `CONFIG_SPARSE_IRQ` kernel configuration option. Now we consider implementation of the `early_irq_init` function when the `CONFIG_SPARSE_IRQ` kernel configuration option is not set. This function starts from the declaration of the following variables: `irq` descriptors counter, loop counter, memory node and the `irq_desc` descriptor:

```
int __init early_irq_init(void)
{
        int count, i, node = first_online_node;
        struct irq_desc *desc;
        ...
        ...
        ...
}
```

The `node` is an online NUMA node which depends on the `MAX_NUMNODES` value which depends on the `CONFIG_NODES_SHIFT` kernel configuration parameter:

```
#define MAX_NUMNODES    (1 << NODES_SHIFT)
...
...
...
#ifdef CONFIG_NODES_SHIFT
    #define NODES_SHIFT     CONFIG_NODES_SHIFT
#else
    #define NODES_SHIFT     0
#endif
```

As I already wrote, implementation of the `first_online_node` macro depends on the `MAX_NUMNODES` value:

```
#if MAX_NUMNODES > 1
  #define first_online_node       first_node(node_states[N_ONLINE])
#else
  #define first_online_node       0
```

The `node_states` is the enum which defined in the include/linux/nodemask.h and represent the set of the states of a node. In our case we are searching an online node and it will be `0` if `MAX_NUMNODES` is one or zero. If the `MAX_NUMNODES` is greater than one, the `node_states[N_ONLINE]` will return `1` and the `first_node` macro will be expands to the call of the `__first_node` function which will return `minimal` or the first online node:

```
#define first_node(src) __first_node(&(src))

static inline int __first_node(const nodemask_t *srcp)
{
        return min_t(int, MAX_NUMNODES, find_first_bit(srcp->bits, MAX_NUMNODES));
}
```

More about this will be in the another chapter about the `NUMA`. The next step after the declaration of these local variables is the call of the:

```
init_irq_default_affinity();
```

function. The `init_irq_default_affinity` function defined in the same source code file and depends on the `CONFIG_SMP` kernel configuration option allocates a given cpumask structure (in our case it is the `irq_default_affinity`):

```
#if defined(CONFIG_SMP)
cpumask_var_t irq_default_affinity;

static void __init init_irq_default_affinity(void)
{
        alloc_cpumask_var(&irq_default_affinity, GFP_NOWAIT);
        cpumask_setall(irq_default_affinity);
}
#else
static void __init init_irq_default_affinity(void)
{
}
#endif
```

We know that when a hardware, such as disk controller or keyboard, needs attention from the processor, it throws an interrupt. The interrupt tells to the processor that something has happened and that the processor should interrupt current process and handle an incoming event. In order to prevent multiple devices from sending the same interrupts, the IRQ system was established where each device in a computer system is assigned its own special IRQ so that its interrupts are unique. Linux kernel can assign certain `IRQs` to specific processors. This is known as `SMP IRQ affinity` , and it allows you control how your system will respond to various hardware events (that's why it has certain implementation only if the `CONFIG_SMP` kernel configuration option is set). After we allocated `irq_default_affinity` cpumask, we can see `printk` output:

```
printk(KERN_INFO "NR_IRQS:%d\n", NR_IRQS);
```

which prints `NR_IRQS` :

```
~$ dmesg | grep NR_IRQS
[    0.000000] NR_IRQS:4352
```

The `NR_IRQS` is the maximum number of the `irq` descriptors or in another words maximum number of interrupts. Its value depends on the state of the `CONFIG_X86_IO_APIC` kernel configuration option. If the `CONFIG_X86_IO_APIC` is not set and the Linux kernel uses an old PIC chip, the `NR_IRQS` is:

```
#define NR_IRQS_LEGACY                  16

#ifdef CONFIG_X86_IO_APIC
...
...
...
#else
# define NR_IRQS                        NR_IRQS_LEGACY
#endif
```

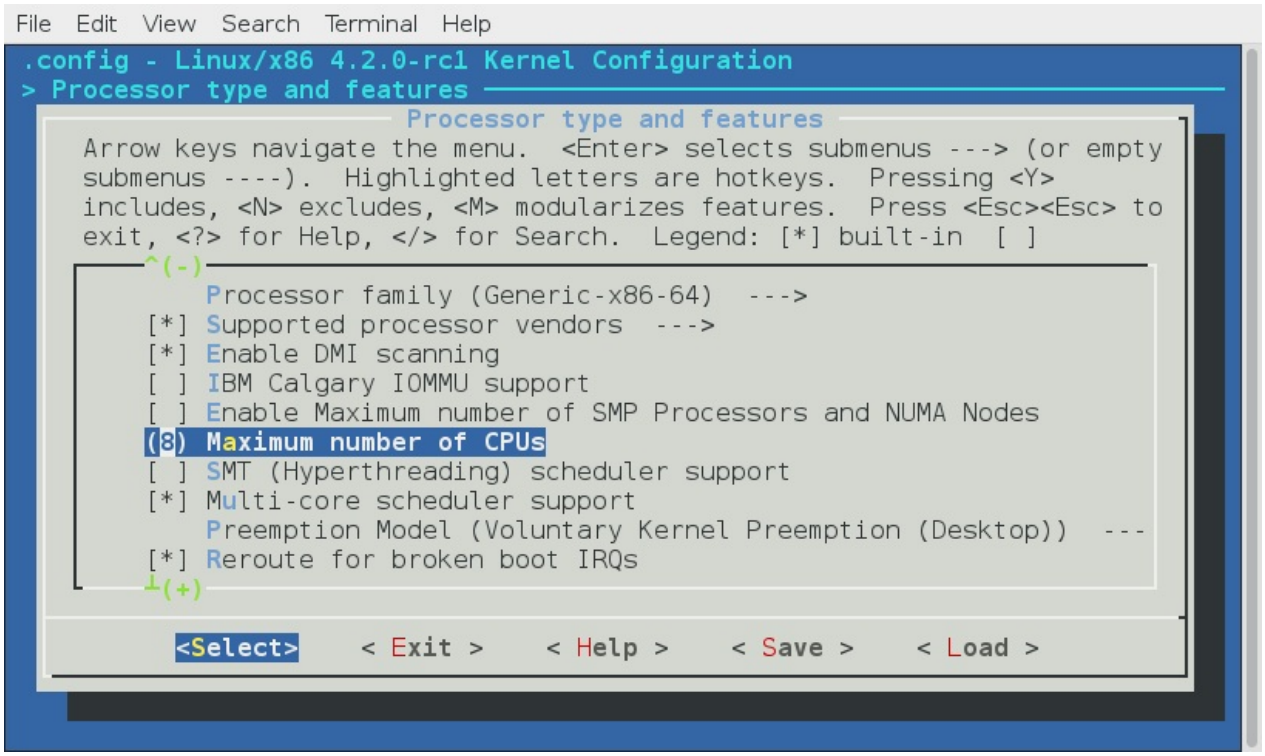
In other way, when the `CONFIG_X86_IO_APIC` kernel configuration option is set, the `NR_IRQS` depends on the amount of the processors and amount of the interrupt vectors:

```
#define CPU_VECTOR_LIMIT              (64 * NR_CPUS)
#define NR_VECTORS                    256
#define IO_APIC_VECTOR_LIMIT          ( 32 * MAX_IO_APICS )
#define MAX_IO_APICS                  128

# define NR_IRQS                                        \
        (CPU_VECTOR_LIMIT > IO_APIC_VECTOR_LIMIT ?      \
                (NR_VECTORS + CPU_VECTOR_LIMIT)  :      \
                (NR_VECTORS + IO_APIC_VECTOR_LIMIT))
...
```

```
...
...
```

We remember from the previous parts, that the amount of processors we can set during Linux kernel configuration process with the `CONFIG_NR_CPUS` configuration option:



In the first case (`CPU_VECTOR_LIMIT > IO_APIC_VECTOR_LIMIT`), the `NR_IRQS` will be `4352` , in the second case (`CPU_VECTOR_LIMIT < IO_APIC_VECTOR_LIMIT`), the `NR_IRQS` will be `768` . In my case the `NR_CPUS` is `8` as you can see in the my configuration, the `CPU_VECTOR_LIMIT` is `512` and the `IO_APIC_VECTOR_LIMIT` is `4096` . So `NR_IRQS` for my configuration is `4352` :

```
~$ dmesg | grep NR_IRQS
[    0.000000] NR_IRQS:4352
```

In the next step we assign array of the IRQ descriptors to the `irq_desc` variable which we defined in the start of the `early_irq_init` function and calculate count of the `irq_desc` array with the `ARRAY_SIZE` macro:

```
desc = irq_desc;
count = ARRAY_SIZE(irq_desc);
```

The `irq_desc` array defined in the same source code file and looks like:

```
struct irq_desc irq_desc[NR_IRQS] __cacheline_aligned_in_smp = {
        [0 ... NR_IRQS-1] = {
                .handle_irq     = handle_bad_irq,
                .depth          = 1,
                .lock           = __RAW_SPIN_LOCK_UNLOCKED(irq_desc->lock),
        }
};
```

The `irq_desc` is array of the `irq` descriptors. It has three already initialized fields:

- `handle_irq` - as I already wrote above, this field is the highlevel irq-event handler. In our case it initialized with the `handle_bad_irq` function that defined in the kernel/irq/handle.c source code file and handles spurious and unhandled irqs;
- `depth` - `0` if the IRQ line is enabled and a positive value if it has been disabled at least once;
- `lock` - A spin lock used to serialize the accesses to the `IRQ` descriptor.

As we calculated count of the interrupts and initialized our `irq_desc` array, we start to fill descriptors in the loop:

```
for (i = 0; i < count; i++) {
    desc[i].kstat_irqs = alloc_percpu(unsigned int);
    alloc_masks(&desc[i], GFP_KERNEL, node);
    raw_spin_lock_init(&desc[i].lock);
    lockdep_set_class(&desc[i].lock, &irq_desc_lock_class);
    desc_set_defaults(i, &desc[i], node, NULL);
}
```

We are going through the all interrupt descriptors and do the following things:

First of all we allocate percpu variable for the `irq` kernel statistic with the `alloc_percpu` macro. This macro allocates one instance of an object of the given type for every processor on the system. You can access kernel statistic from the userspace via `/proc/stat`:

```
~$ cat /proc/stat
cpu  207907 68 53904 5427850 14394 0 394 0 0 0
cpu0 25881 11 6684 679131 1351 0 18 0 0 0
cpu1 24791 16 5894 679994 2285 0 24 0 0 0
cpu2 26321 4 7154 678924 664 0 71 0 0 0
cpu3 26648 8 6931 678891 414 0 244 0 0 0
...
...
...
```

Where the sixth column is the servicing interrupts. After this we allocate cpumask for the given irq descriptor affinity and initialize the spinlock for the given interrupt descriptor. After this before the critical section, the lock will be acquired with a call of the `raw_spin_lock` and unlocked with the call of the `raw_spin_unlock`. In the next step we call the `lockdep_set_class` macro which set the Lock validator `irq_desc_lock_class` class for the lock of the given interrupt descriptor. More about `lockdep`, `spinlock` and other synchronization primitives will be described in the separate chapter.

In the end of the loop we call the `desc_set_defaults` function from the kernel/irq/irqdesc.c. This function takes four parameters:

- number of a irq;
- interrupt descriptor;
- online `NUMA` node;
- owner of interrupt descriptor. Interrupt descriptors can be allocated from modules. This field is need to proved refcount on the module which provides the interrupts;

and fills the rest of the `irq_desc` fields. The `desc_set_defaults` function fills interrupt number, `irq` chip, platform-specific per-chip private data for the chip methods, per-IRQ data for the `irq_chip` methods and MSI descriptor for the per `irq` and `irq` chip data:

```
desc->irq_data.irq = irq;
desc->irq_data.chip = &no_irq_chip;
desc->irq_data.chip_data = NULL;
desc->irq_data.handler_data = NULL;
desc->irq_data.msi_desc = NULL;
...
...
...
```

The `irq_data.chip` structure provides general `API` like the `irq_set_chip`, `irq_set_irq_type` and etc, for the irq controller drivers. You can find it in the kernel/irq/chip.c source code file.

After this we set the status of the accessor for the given descriptor and set disabled state of the interrupts:

```
...
...
...
irq_settings_clr_and_set(desc, ~0, _IRQ_DEFAULT_INIT_FLAGS);
irqd_set(&desc->irq_data, IRQD_IRQ_DISABLED);
```

```
...
...
...
```

In the next step we set the high level interrupt handlers to the `handle_bad_irq` which handles spurious and unhandled irqs (as the hardware stuff is not initialized yet, we set this handler), set `irq_desc.desc` to `1` which means that an `IRQ` is disabled, reset count of the unhandled interrupts and interrupts in general:

```
...
...
...
desc->handle_irq = handle_bad_irq;
desc->depth = 1;
desc->irq_count = 0;
desc->irqs_unhandled = 0;
desc->name = NULL;
desc->owner = owner;
...
...
...
```

After this we go through the all possible processor with the for_each_possible_cpu helper and set the `kstat_irqs` to zero for the given interrupt descriptor:

```
    for_each_possible_cpu(cpu)
        *per_cpu_ptr(desc->kstat_irqs, cpu) = 0;
```

and call the `desc_smp_init` function from the kernel/irq/irqdesc.c that initializes `NUMA` node of the given interrupt descriptor, sets default `SMP` affinity and clears the `pending_mask` of the given interrupt descriptor depends on the value of the `CONFIG_GENERIC_PENDING_IRQ` kernel configuration option:

```
static void desc_smp_init(struct irq_desc *desc, int node)
{
        desc->irq_data.node = node;
        cpumask_copy(desc->irq_data.affinity, irq_default_affinity);
#ifdef CONFIG_GENERIC_PENDING_IRQ
        cpumask_clear(desc->pending_mask);
#endif
}
```

In the end of the `early_irq_init` function we return the return value of the `arch_early_irq_init` function:

```
return arch_early_irq_init();
```

This function defined in the kernel/apic/vector.c and contains only one call of the `arch_early_ioapic_init` function from the kernel/apic/io_apic.c. As we can understand from the `arch_early_ioapic_init` function's name, this function makes early initialization of the I/O APIC. First of all it make a check of the number of the legacy interrupts with the call of the `nr_legacy_irqs` function. If we have no legacy interrupts with the Intel 8259 programmable interrupt controller we set `io_apic_irqs` to the `0xffffffffffffffff`:

```
if (!nr_legacy_irqs())
    io_apic_irqs = ~0UL;
```

After this we are going through the all `I/O APICs` and allocate space for the registers with the call of the `alloc_ioapic_saved_registers`:

```
for_each_ioapic(i)
    alloc_ioapic_saved_registers(i);
```

And in the end of the `arch_early_ioapic_init` function we are going through the all legacy irqs (from `IRQ0` to `IRQ15`) in the loop and allocate space for the `irq_cfg` which represents configuration of an irq on the given `NUMA` node:

```
for (i = 0; i < nr_legacy_irqs(); i++) {
    cfg = alloc_irq_and_cfg_at(i, node);
    cfg->vector = IRQ0_VECTOR + i;
    cpumask_setall(cfg->domain);
}
```

That's all.

Sparse IRQs

We already saw in the beginning of this part that implementation of the `early_irq_init` function depends on the `CONFIG_SPARSE_IRQ` kernel configuration option. Previously we saw implementation of the `early_irq_init` function when the `CONFIG_SPARSE_IRQ` configuration option is not set, now let's look on the its implementation when this option is set. Implementation of this function very similar, but little differ. We can see the same definition of variables and call of the `init_irq_default_affinity` in the beginning of the `early_irq_init` function:

```
#ifdef CONFIG_SPARSE_IRQ
int __init early_irq_init(void)
{
    int i, initcnt, node = first_online_node;
    struct irq_desc *desc;

    init_irq_default_affinity();
    ...
    ...
    ...
}
#else
...
...
...
```

But after this we can see the following call:

```
initcnt = arch_probe_nr_irqs();
```

The `arch_probe_nr_irqs` function defined in the arch/x86/kernel/apic/vector.c and calculates count of the pre-allocated irqs and update `nr_irqs` with its number. But stop. Why there are pre-allocated irqs? There is alternative form of interrupts called - Message Signaled Interrupts available in the PCI. Instead of assigning a fixed number of the interrupt request, the device is allowed to record a message at a particular address of RAM, in fact, the display on the Local APIC. `MSI` permits a device to allocate `1`, `2`, `4`, `8`, `16` or `32` interrupts and `MSI-X` permits a device to allocate up to `2048` interrupts. Now we know that irqs can be pre-allocated. More about `MSI` will be in a next part, but now let's look on the `arch_probe_nr_irqs` function. We can see the check which assign amount of the interrupt vectors for the each processor in the system to the `nr_irqs` if it is greater and calculate the `nr` which represents number of `MSI` interrupts:

```
int nr_irqs = NR_IRQS;

if (nr_irqs > (NR_VECTORS * nr_cpu_ids))
    nr_irqs = NR_VECTORS * nr_cpu_ids;

nr = (gsi_top + nr_legacy_irqs()) + 8 * nr_cpu_ids;
```

Take a look on the `gsi_top` variable. Each `APIC` is identified with its own `ID` and with the offset where its `IRQ` starts. It is called `GSI` base or `Global System Interrupt` base. So the `gsi_top` represents it. We get the `Global System Interrupt` base from the MultiProcessor Configuration Table table (you can remember that we have parsed this table in the sixth part of the Linux Kernel

initialization process chapter).

After this we update the `nr` depends on the value of the `gsi_top`:

```
#if defined(CONFIG_PCI_MSI) || defined(CONFIG_HT_IRQ)
        if (gsi_top <= NR_IRQS_LEGACY)
                nr +=  8 * nr_cpu_ids;
        else
                nr += gsi_top * 16;
#endif
```

Update the `nr_irqs` if it less than `nr` and return the number of the legacy irqs:

```
if (nr < nr_irqs)
    nr_irqs = nr;

return nr_legacy_irqs();
}
```

The next after the `arch_probe_nr_irqs` is printing information about number of `IRQs`:

```
printk(KERN_INFO "NR_IRQS:%d nr_irqs:%d %d\n", NR_IRQS, nr_irqs, initcnt);
```

We can find it in the dmesg output:

```
$ dmesg | grep NR_IRQS
[    0.000000] NR_IRQS:4352 nr_irqs:488 16
```

After this we do some checks that `nr_irqs` and `initcnt` values is not greater than maximum allowable number of `irqs`:

```
if (WARN_ON(nr_irqs > IRQ_BITMAP_BITS))
    nr_irqs = IRQ_BITMAP_BITS;

if (WARN_ON(initcnt > IRQ_BITMAP_BITS))
    initcnt = IRQ_BITMAP_BITS;
```

where `IRQ_BITMAP_BITS` is equal to the `NR_IRQS` if the `CONFIG_SPARSE_IRQ` is not set and `NR_IRQS + 8196` in other way. In the next step we are going over all interrupt descriptors which need to be allocated in the loop and allocate space for the descriptor and insert to the `irq_desc_tree` radix tree:

```
for (i = 0; i < initcnt; i++) {
    desc = alloc_desc(i, node, NULL);
    set_bit(i, allocated_irqs);
    irq_insert_desc(i, desc);
}
```

In the end of the `early_irq_init` function we return the value of the call of the `arch_early_irq_init` function as we did it already in the previous variant when the `CONFIG_SPARSE_IRQ` option was not set:

```
return arch_early_irq_init();
```

That's all.

Conclusion

It is the end of the seventh part of the Interrupts and Interrupt Handling chapter and we started to dive into external hardware interrupts in this part. We saw early initialization of the `irq_desc` structure which represents description of an external interrupt and contains information about it like list of irq actions, information about interrupt handler, interrupt's owner, count of the unhandled interrupt and etc. In the next part we will continue to research external interrupts.

If you have any questions or suggestions write me a comment or ping me at twitter.

Please note that English is not my first language, And I am really sorry for any inconvenience. If you find any mistakes please send me PR to linux-insides.

Links

- IRQ
- numa
- Enum type
- cpumask
- percpu
- spinlock
- critical section
- Lock validator
- MSI
- I/O APIC
- Local APIC
- Intel 8259
- PIC
- MultiProcessor Configuration Table
- radix tree
- dmesg

Interrupts and Interrupt Handling. Part 8.

Non-early initialization of the IRQs

This is the eighth part of the Interrupts and Interrupt Handling in the Linux kernel chapter and in the previous part we started to dive into the external hardware interrupts. We looked on the implementation of the `early_irq_init` function from the kernel/irq/irqdesc.c source code file and saw the initialization of the `irq_desc` structure in this function. Remind that `irq_desc` structure (defined in the include/linux/irqdesc.h is the foundation of interrupt management code in the Linux kernel and represents an interrupt descriptor. In this part we will continue to dive into the initialization stuff which is related to the external hardware interrupts.

Right after the call of the `early_irq_init` function in the init/main.c we can see the call of the `init_IRQ` function. This function is architecture-specific and defined in the arch/x86/kernel/irqinit.c. The `init_IRQ` function makes initialization of the `vector_irq` percpu variable that defined in the same arch/x86/kernel/irqinit.c source code file:

```
...
DEFINE_PER_CPU(vector_irq_t, vector_irq) = {
        [0 ... NR_VECTORS - 1] = -1,
};
...
```

and represents `percpu` array of the interrupt vector numbers. The `vector_irq_t` defined in the arch/x86/include/asm/hw_irq.h and expands to the:

```
typedef int vector_irq_t[NR_VECTORS];
```

where `NR_VECTORS` is count of the vector number and as you can remember from the first part of this chapter it is `256` for the x86_64:

```
#define NR_VECTORS                      256
```

So, in the start of the `init_IRQ` function we fill the `vector_irq` percpu array with the vector number of the `legacy` interrupts:

```
void __init init_IRQ(void)
{
    int i;

    for (i = 0; i < nr_legacy_irqs(); i++)
        per_cpu(vector_irq, 0)[IRQ0_VECTOR + i] = i;
...
...
...
}
```

This `vector_irq` will be used during the first steps of an external hardware interrupt handling in the `do_IRQ` function from the arch/x86/kernel/irq.c:

```
__visible unsigned int __irq_entry do_IRQ(struct pt_regs *regs)
{
    ...
    ...
    ...
    irq = __this_cpu_read(vector_irq[vector]);

    if (!handle_irq(irq, regs)) {
        ...
        ...
        ...
    }
```

```
    exiting_irq();
    ...
    ...
    return 1;
}
```

Why is `legacy` here? Actually all interrupts are handled by the modern IO-APIC controller. But these interrupts (from `0x30` to `0x3f`) by legacy interrupt-controllers like Programmable Interrupt Controller. If these interrupts are handled by the `I/O APIC` then this vector space will be freed and re-used. Let's look on this code closer. First of all the `nr_legacy_irqs` defined in the arch/x86/include/asm/i8259.h and just returns the `nr_legacy_irqs` field from the `legacy_pic` structure:

```
static inline int nr_legacy_irqs(void)
{
        return legacy_pic->nr_legacy_irqs;
}
```

This structure defined in the same header file and represents non-modern programmable interrupts controller:

```
struct legacy_pic {
        int nr_legacy_irqs;
        struct irq_chip *chip;
        void (*mask)(unsigned int irq);
        void (*unmask)(unsigned int irq);
        void (*mask_all)(void);
        void (*restore_mask)(void);
        void (*init)(int auto_eoi);
        int (*irq_pending)(unsigned int irq);
        void (*make_irq)(unsigned int irq);
};
```

Actual default maximum number of the legacy interrupts represented by the `NR_IRQ_LEGACY` macro from the arch/x86/include/asm/irq_vectors.h:

```
#define NR_IRQS_LEGACY                  16
```

In the loop we are accessing the `vecto_irq` per-cpu array with the `per_cpu` macro by the `IRQ0_VECTOR + i` index and write the legacy vector number there. The `IRQ0_VECTOR` macro defined in the arch/x86/include/asm/irq_vectors.h header file and expands to the `0x30`:

```
#define FIRST_EXTERNAL_VECTOR           0x20

#define IRQ0_VECTOR                     ((FIRST_EXTERNAL_VECTOR + 16) & ~15)
```

Why is `0x30` here? You can remember from the first part of this chapter that first 32 vector numbers from `0` to `31` are reserved by the processor and used for the processing of architecture-defined exceptions and interrupts. Vector numbers from `0x30` to `0x3f` are reserved for the ISA. So, it means that we fill the `vector_irq` from the `IRQ0_VECTOR` which is equal to the `32` to the `IRQ0_VECTOR + 16` (before the `0x30`).

In the end of the `init_IRQ` function we can see the call of the following function:

```
x86_init.irqs.intr_init();
```

from the arch/x86/kernel/x86_init.c source code file. If you have read chapter about the Linux kernel initialization process, you can remember the `x86_init` structure. This structure contains a couple of files which are points to the function related to the platform setup (`x86_64` in our case), for example `resources` - related with the memory resources, `mpparse` - related with the parsing of the MultiProcessor Configuration Table table and etc.). As we can see the `x86_init` also contains the `irqs` field which contains three following fields:

```
struct x86_init_ops x86_init __initdata
{
    ...
    ...
    ...
    .irqs = {
                .pre_vector_init        = init_ISA_irqs,
                .intr_init              = native_init_IRQ,
                .trap_init              = x86_init_noop,
    },
    ...
    ...
    ...
}
```

Now, we are interesting in the `native_init_IRQ` . As we can note, the name of the `native_init_IRQ` function contains the `native_` prefix which means that this function is architecture-specific. It defined in the arch/x86/kernel/irqinit.c and executes general initialization of the Local APIC and initialization of the ISA irqs. Let's look on the implementation of the `native_init_IRQ` function and will try to understand what occurs there. The `native_init_IRQ` function starts from the execution of the following function:

```
x86_init.irqs.pre_vector_init();
```

As we can see above, the `pre_vector_init` points to the `init_ISA_irqs` function that defined in the same source code file and as we can understand from the function's name, it makes initialization of the `ISA` related interrupts. The `init_ISA_irqs` function starts from the definition of the `chip` variable which has a `irq_chip` type:

```
void __init init_ISA_irqs(void)
{
    struct irq_chip *chip = legacy_pic->chip;
    ...
    ...
    ...
```

The `irq_chip` structure defined in the include/linux/irq.h header file and represents hardware interrupt chip descriptor. It contains:

- `name` - name of a device. Used in the `/proc/interrupts` :

```
$ cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6       CPU7
  0:         16          0          0          0          0          0          0          0   IO-APIC   2-edge
timer
  1:          2          0          0          0          0          0          0          0   IO-APIC   1-edge
i8042
  8:          1          0          0          0          0          0          0          0   IO-APIC   8-edge
rtc0
```

look on the last column;

- `(*irq_mask)(struct irq_data *data)` - mask an interrupt source;
- `(*irq_ack)(struct irq_data *data)` - start of a new interrupt;
- `(*irq_startup)(struct irq_data *data)` - start up the interrupt;
- `(*irq_shutdown)(struct irq_data *data)` - shutdown the interrupt
- and etc.

fields. Note that the `irq_data` structure represents set of the per irq chip data passed down to chip functions. It contains `mask` - precomputed bitmask for accessing the chip registers, `irq` - interrupt number, `hwirq` - hardware interrupt number, local to the interrupt domain chip low level interrupt hardware access and etc.

After this depends on the `CONFIG_X86_64` and `CONFIG_X86_LOCAL_APIC` kernel configuration option call the `init_bsp_APIC` function from the arch/x86/kernel/apic/apic.c:

```
#if defined(CONFIG_X86_64) || defined(CONFIG_X86_LOCAL_APIC)
    init_bsp_APIC();
#endif
```

This function makes initialization of the APIC of `bootstrap processor` (or processor which starts first). It starts from the check that we found SMP config (read more about it in the sixth part of the Linux kernel initialization process chapter) and the processor has `APIC` :

```
if (smp_found_config || !cpu_has_apic)
    return;
```

In other way we return from this function. In the next step we call the `clear_local_APIC` function from the same source code file that shutdowns the local `APIC` (more about it will be in the chapter about the `Advanced Programmable Interrupt Controller`) and enable `APIC` of the first processor by the setting `unsigned int value` to the `APIC_SPIV_APIC_ENABLED` :

```
value = apic_read(APIC_SPIV);
value &= ~APIC_VECTOR_MASK;
value |= APIC_SPIV_APIC_ENABLED;
```

and writing it with the help of the `apic_write` function:

```
apic_write(APIC_SPIV, value);
```

After we have enabled `APIC` for the bootstrap processor, we return to the `init_ISA_irqs` function and in the next step we initialize legacy `Programmable Interrupt Controller` and set the legacy chip and handler for the each legacy irq:

```
legacy_pic->init(0);

for (i = 0; i < nr_legacy_irqs(); i++)
    irq_set_chip_and_handler(i, chip, handle_level_irq);
```

Where can we find `init` function? The `legacy_pic` defined in the arch/x86/kernel/i8259.c and it is:

```
struct legacy_pic *legacy_pic = &default_legacy_pic;
```

Where the `default_legacy_pic` is:

```
struct legacy_pic default_legacy_pic = {
    ...
    ...
    ...
    .init = init_8259A,
    ...
    ...
    ...
}
```

The `init_8259A` function defined in the same source code file and executes initialization of the Intel 8259 `Programmable Interrupt Controller` (more about it will be in the separate chapter about `Programmable Interrupt Controllers` and `APIC`).

Now we can return to the `native_init_IRQ` function, after the `init_ISA_irqs` function finished its work. The next step is the call of the `apic_intr_init` function that allocates special interrupt gates which are used by the SMP architecture for the Inter-processor interrupt. The `alloc_intr_gate` macro from the arch/x86/include/asm/desc.h used for the interrupt descriptor allocation:

```
#define alloc_intr_gate(n, addr)                    \
do {                                                \
        alloc_system_vector(n);                     \
        set_intr_gate(n, addr);                     \
```

```
} while (0)
```

As we can see, first of all it expands to the call of the `alloc_system_vector` function that checks the given vector number in the `used_vectors` bitmap (read previous part about it) and if it is not set in the `used_vectors` bitmap we set it. After this we test that the `first_system_vector` is greater than given interrupt vector number and if it is greater we assign it:

```
if (!test_bit(vector, used_vectors)) {
    set_bit(vector, used_vectors);
    if (first_system_vector > vector)
        first_system_vector = vector;
} else {
    BUG();
}
```

We already saw the `set_bit` macro, now let's look on the `test_bit` and the `first_system_vector`. The first `test_bit` macro defined in the arch/x86/include/asm/bitops.h and looks like this:

```
#define test_bit(nr, addr)                      \
        (__builtin_constant_p((nr))             \
         ? constant_test_bit((nr), (addr))      \
         : variable_test_bit((nr), (addr)))
```

We can see the ternary operator here make a test with the gcc built-in function `__builtin_constant_p` tests that given vector number (`nr`) is known at compile time. If you're feeling misunderstanding of the `__builtin_constant_p`, we can make simple test:

```
#include <stdio.h>

#define PREDEFINED_VAL 1

int main() {
    int i = 5;
    printf("__builtin_constant_p(i) is %d\n", __builtin_constant_p(i));
    printf("__builtin_constant_p(PREDEFINED_VAL) is %d\n", __builtin_constant_p(PREDEFINED_VAL));
    printf("__builtin_constant_p(100) is %d\n", __builtin_constant_p(100));

    return 0;
}
```

and look on the result:

```
$ gcc test.c -o test
$ ./test
__builtin_constant_p(i) is 0
__builtin_constant_p(PREDEFINED_VAL) is 1
__builtin_constant_p(100) is 1
```

Now I think it must be clear for you. Let's get back to the `test_bit` macro. If the `__builtin_constant_p` will return non-zero, we call `constant_test_bit` function:

```
static inline int constant_test_bit(int nr, const void *addr)
{
    const u32 *p = (const u32 *)addr;

    return ((1UL << (nr & 31)) & (p[nr >> 5])) != 0;
}
```

and the `variable_test_bit` in other way:

```
static inline int variable_test_bit(int nr, const void *addr)
{
        u8 v;
```

```
        const u32 *p = (const u32 *)addr;

        asm("btl %2,%1; setc %0" : "=qm" (v) : "m" (*p), "Ir" (nr));
        return v;
}
```

What's the difference between two these functions and why do we need in two different functions for the same purpose? As you already can guess main purpose is optimization. If we will write simple example with these functions:

```
#define CONST 25

int main() {
    int nr = 24;
    variable_test_bit(nr, (int*)0x10000000);
    constant_test_bit(CONST, (int*)0x10000000)
    return 0;
}
```

and will look on the assembly output of our example we will see following assembly code:

```
pushq    %rbp
movq    %rsp, %rbp

movl    $268435456, %esi
movl    $25, %edi
call    constant_test_bit
```

for the `constant_test_bit`, and:

```
pushq    %rbp
movq    %rsp, %rbp

subq    $16, %rsp
movl    $24, -4(%rbp)
movl    -4(%rbp), %eax
movl    $268435456, %esi
movl    %eax, %edi
call    variable_test_bit
```

for the `variable_test_bit`. These two code listings starts with the same part, first of all we save base of the current stack frame in the `%rbp` register. But after this code for both examples is different. In the first example we put `$268435456` (here the `$268435456` is our second parameter - `0x10000000`) to the `esi` and `$25` (our first parameter) to the `edi` register and call `constant_test_bit`. We put function parameters to the `esi` and `edi` registers because as we are learning Linux kernel for the `x86_64` architecture we use `System V AMD64 ABI` calling convention. All is pretty simple. When we are using predefined constant, the compiler can just substitute its value. Now let's look on the second part. As you can see here, the compiler can not substitute value from the `nr` variable. In this case compiler must calculate its offset on the program's stack frame. We subtract `16` from the `rsp` register to allocate stack for the local variables data and put the `$24` (value of the `nr` variable) to the `rbp` with offset `-4`. Our stack frame will be like this:

```
         <- stack grows

              %[rbp]
                 |
+----------+ +---------+ +---------+ +--------+
|          | |         | | return  | |        |
|    nr    |-|         |-|         |-|  argc  |
|          | |         | | address | |        |
+----------+ +---------+ +---------+ +--------+
                 |
              %[rsp]
```

After this we put this value to the `eax`, so `eax` register now contains value of the `nr`. In the end we do the same that in the first example, we put the `$268435456` (the first parameter of the `variable_test_bit` function) and the value of the `eax` (value of `nr`) to the `edi` register (the second parameter of the `variable_test_bit function`).

The next step after the `apic_intr_init` function will finish its work is the setting interrupt gates from the `FIRST_EXTERNAL_VECTOR` or `0x20` to the `0x256`:

```
i = FIRST_EXTERNAL_VECTOR;

#ifndef CONFIG_X86_LOCAL_APIC
#define first_system_vector NR_VECTORS
#endif

for_each_clear_bit_from(i, used_vectors, first_system_vector) {
    set_intr_gate(i, irq_entries_start + 8 * (i - FIRST_EXTERNAL_VECTOR));
}
```

But as we are using the `for_each_clear_bit_from` helper, we set only non-initialized interrupt gates. After this we use the same `for_each_clear_bit_from` helper to fill the non-filled interrupt gates in the interrupt table with the `spurious_interrupt`:

```
#ifdef CONFIG_X86_LOCAL_APIC
for_each_clear_bit_from(i, used_vectors, NR_VECTORS)
    set_intr_gate(i, spurious_interrupt);
#endif
```

Where the `spurious_interrupt` function represent interrupt handler for the `spurious` interrupt. Here the `used_vectors` is the `unsigned long` that contains already initialized interrupt gates. We already filled first `32` interrupt vectors in the `trap_init` function from the arch/x86/kernel/setup.c source code file:

```
for (i = 0; i < FIRST_EXTERNAL_VECTOR; i++)
    set_bit(i, used_vectors);
```

You can remember how we did it in the sixth part of this chapter.

In the end of the `native_init_IRQ` function we can see the following check:

```
if (!acpi_ioapic && !of_ioapic && nr_legacy_irqs())
    setup_irq(2, &irq2);
```

First of all let's deal with the condition. The `acpi_ioapic` variable represents existence of I/O APIC. It defined in the arch/x86/kernel/acpi/boot.c. This variable set in the `acpi_set_irq_model_ioapic` function that called during the processing `Multiple APIC Description Table`. This occurs during initialization of the architecture-specific stuff in the arch/x86/kernel/setup.c (more about it we will know in the other chapter about APIC). Note that the value of the `acpi_ioapic` variable depends on the `CONFIG_ACPI` and `CONFIG_X86_LOCAL_APIC` Linux kernel configuration options. If these options did not set, this variable will be just zero:

```
#define acpi_ioapic 0
```

The second condition - `!of_ioapic && nr_legacy_irqs()` checks that we do not use Open Firmware `I/O APIC` and legacy interrupt controller. We already know about the `nr_legacy_irqs`. The second is `of_ioapic` variable defined in the arch/x86/kernel/devicetree.c and initialized in the `dtb_ioapic_setup` function that build information about `APICs` in the devicetree. Note that `of_ioapic` variable depends on the `CONFIG_OF` Linux kernel configuration option. If this option is not set, the value of the `of_ioapic` will be zero too:

```
#ifdef CONFIG_OF
extern int of_ioapic;
...
...
...
#else
```

```
#define of_ioapic 0
...
...
...
#endif
```

If the condition will return non-zero value we call the:

```
setup_irq(2, &irq2);
```

function. First of all about the `irq2`. The `irq2` is the `irqaction` structure that defined in the arch/x86/kernel/irqinit.c source code file and represents `IRQ 2` line that is used to query devices connected cascade:

```
static struct irqaction irq2 = {
    .handler = no_action,
    .name = "cascade",
    .flags = IRQF_NO_THREAD,
};
```

Some time ago interrupt controller consisted of two chips and one was connected to second. The second chip that was connected to the first chip via this `IRQ 2` line. This chip serviced lines from `8` to `15` and after this lines of the first chip. So, for example Intel 8259A has following lines:

- `IRQ 0` - system time;
- `IRQ 1` - keyboard;
- `IRQ 2` - used for devices which are cascade connected;
- `IRQ 8` - RTC;
- `IRQ 9` - reserved;
- `IRQ 10` - reserved;
- `IRQ 11` - reserved;
- `IRQ 12` - `ps/2` mouse;
- `IRQ 13` - coprocessor;
- `IRQ 14` - hard drive controller;
- `IRQ 1` - reserved;
- `IRQ 3` - `COM2` and `COM4`;
- `IRQ 4` - `COM1` and `COM3`;
- `IRQ 5` - `LPT2`;
- `IRQ 6` - drive controller;
- `IRQ 7` - `LPT1`.

The `setup_irq` function defined in the kernel/irq/manage.c and takes two parameters:

- vector number of an interrupt;
- `irqaction` structure related with an interrupt.

This function initializes interrupt descriptor from the given vector number at the beginning:

```
struct irq_desc *desc = irq_to_desc(irq);
```

And call the `__setup_irq` function that setups given interrupt:

```
chip_bus_lock(desc);
retval = __setup_irq(irq, desc, act);
chip_bus_sync_unlock(desc);
return retval;
```

Note that the interrupt descriptor is locked during `__setup_irq` function will work. The `__setup_irq` function makes many different things: It creates a handler thread when a thread function is supplied and the interrupt does not nest into another interrupt thread, sets the flags of the chip, fills the `irqaction` structure and many many more.

All of the above it creates `/prov/vector_number` directory and fills it, but if you are using modern computer all values will be zero there:

```
$ cat /proc/irq/2/node
0

$cat /proc/irq/2/affinity_hint
00

cat /proc/irq/2/spurious
count 0
unhandled 0
last_unhandled 0 ms
```

because probably `APIC` handles interrupts on the our machine.

That's all.

Conclusion

It is the end of the eighth part of the Interrupts and Interrupt Handling chapter and we continued to dive into external hardware interrupts in this part. In the previous part we started to do it and saw early initialization of the `IRQs`. In this part we already saw non-early interrupts initialization in the `init_IRQ` function. We saw initialization of the `vector_irq` per-cpu array which is store vector numbers of the interrupts and will be used during interrupt handling and initialization of other stuff which is related to the external hardware interrupts.

In the next part we will continue to learn interrupts handling related stuff and will see initialization of the `softirqs`.

If you have any questions or suggestions write me a comment or ping me at twitter.

Please note that English is not my first language, And I am really sorry for any inconvenience. If you find any mistakes please send me PR to linux-insides.

Links

- IRQ
- percpu
- x86_64
- Intel 8259
- Programmable Interrupt Controller
- ISA
- MultiProcessor Configuration Table
- Local APIC
- I/O APIC
- SMP
- Inter-processor interrupt
- ternary operator
- gcc
- calling convention
- PDF. System V Application Binary Interface AMD64
- Call stack
- Open Firmware

- devicetree
- RTC
- Previous part

()

(Tasklets)

Linux arch/x86/kernel/irqinit.c `init_IRQ`

-
-

-
-

 Linux

-
- `tasklets`
-

`ksoftirqd` () `ksoftirqd/n` n `systemd-cgls`

```
$ systemd-cgls -k | grep ksoft
├─   3 [ksoftirqd/0]
├─  13 [ksoftirqd/1]
├─  18 [ksoftirqd/2]
├─  23 [ksoftirqd/3]
├─  28 [ksoftirqd/4]
├─  33 [ksoftirqd/5]
├─  38 [ksoftirqd/6]
├─  43 [ksoftirqd/7]
```

`spawn_ksoftirqd` initcall

```
early_initcall(spawn_ksoftirqd);
```

Linux `open_softirq` `softirq` kernel/softirq.c

```
void open_softirq(int nr, void (*action)(struct softirq_action *))
{
    softirq_vec[nr].action = action;
}
```

- `softirq_vec`
-

`softirq_vec`

```
static struct softirq_action softirq_vec[NR_SOFTIRQS] __cacheline_aligned_in_smp;
```

`softirq_vec` `NR_SOFTIRQS` (10) `softirq` `softirq_action` Linux tasklet RCU

```
enum
{
        HI_SOFTIRQ=0,
        TIMER_SOFTIRQ,
        NET_TX_SOFTIRQ,
        NET_RX_SOFTIRQ,
        BLOCK_SOFTIRQ,
        BLOCK_IOPOLL_SOFTIRQ,
        TASKLET_SOFTIRQ,
        SCHED_SOFTIRQ,
        HRTIMER_SOFTIRQ,
        RCU_SOFTIRQ,
        NR_SOFTIRQS
};
```

```
const char * const softirq_to_name[NR_SOFTIRQS] = {
        "HI", "TIMER", "NET_TX", "NET_RX", "BLOCK", "BLOCK_IOPOLL",
        "TASKLET", "SCHED", "HRTIMER", "RCU"
};
```

`/proc/softirqs`

```
~$ cat /proc/softirqs
                    CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6       CPU7
          HI:           5          0          0          0          0          0          0          0
       TIMER:      332519     310498     289555     272913     282535     279467     282895     270979
      NET_TX:        2320          0          0          2          1          1          0          0
      NET_RX:      270221        225        338        281        311        262        430        265
       BLOCK:      134282         32         40         10         12          7          8          8
BLOCK_IOPOLL:           0          0          0          0          0          0          0          0
     TASKLET:      196835          2          3          0          0          0          0          0
       SCHED:      161852     146745     129539     126064     127998     128014     120243     117391
     HRTIMER:           0          0          0          0          0          0          0          0
         RCU:      337707     289397     251874     239796     254377     254898     267497     256624
```

`softirq_vec` `softirq_action`

```
struct softirq_action
{
         void    (*action)(struct softirq_action *);
};
```

`open_softirq` `softirq_action` `softirq_vec` `open_softirq` `raise_softirq` -- `nr`

```
void raise_softirq(unsigned int nr)
{
        unsigned long flags;

        local_irq_save(flags);
        raise_softirq_irqoff(nr);
        local_irq_restore(flags);
}
```

`local_irq_save` `local_irq_restore` `raise_softirq_irqoff` `local_irq_save` include/linux/irqflags.h eflags IF `local_irq_restore` `softirq`

`raise_softirq_irqoff` nr(`__softirq_pending`)

```
__raise_softirq_irqoff(nr);
```

`in_interrupt` `irq_count` cpu `raise_softirq_irqoff` `IF` `wakeup_softirqd`

```
if (!in_interrupt())
    wakeup_softirqd();
```

`wakeup_softirqd` `ksoftirqd`

```
static void wakeup_softirqd(void)
{
    struct task_struct *tsk = __this_cpu_read(ksoftirqd);

    if (tsk && tsk->state != TASK_RUNNING)
        wake_up_process(tsk);
}
```

`ksoftirqd` `run_ksoftirqd` `__do_softirq` `__do_softirq` `__softirq_pending` `__do_softirq`

```
unsigned long end = jiffies + MAX_SOFTIRQ_TIME;
...
...
...
restart:
while ((softirq_bit = ffs(pending))) {
    ...
    h->action(h);
    ...
}
...
...
...
pending = local_softirq_pending();
if (pending) {
    if (time_before(jiffies, end) && !need_resched() &&
        --max_restart)
            goto restart;
}
...
```

arch/x86/kernel/irq.c `do_IRQ` Linux arch/x86/include/asm/apic.h `exiting_irq` `exiting_irq` `irq_exit` `irq_exit` `invoke_softirq`

```
if (!in_interrupt() && local_softirq_pending())
    invoke_softirq();
```

`__do_softirq` `softirq` `open_softirq` `raise_softirq` Linux `tasklets`

Tasklets

Linux `tasklets` `tasklets` `softirq`

- `TASKLET_SOFTIRQ` ;
- `HI_SOFTIRQ` .

`tasklets` `tasklets` `softirq_init` kernel/softirq.c

```
void __init softirq_init(void)
{
        int cpu;

        for_each_possible_cpu(cpu) {
                per_cpu(tasklet_vec, cpu).tail =
                        &per_cpu(tasklet_vec, cpu).head;
                per_cpu(tasklet_hi_vec, cpu).tail =
                        &per_cpu(tasklet_hi_vec, cpu).head;
        }

        open_softirq(TASKLET_SOFTIRQ, tasklet_action);
        open_softirq(HI_SOFTIRQ, tasklet_hi_action);
}
```

cpu integer `for_each_possible_cpu` `possible_cpu` CPU masks `possible_cpu` `possible processor` `cpu_possible_bits` kernel/cpu.c

```
static DECLARE_BITMAP(cpu_possible_bits, CONFIG_NR_CPUS) __read_mostly;
...
...
...
const struct cpumask *const cpu_possible_mask = to_cpumask(cpu_possible_bits);
```

integer `cpu` `for_each_possible_cpu` `per-cpu`

- `tasklet_vec` ;
- `tasklet_hi_vec` ;

`per-cpu` `softirq_init` `tasklet_head`

```
static DEFINE_PER_CPU(struct tasklet_head, tasklet_vec);
static DEFINE_PER_CPU(struct tasklet_head, tasklet_hi_vec);
```

`tasklet_head` `Tasklets` head tail

```
struct tasklet_head {
        struct tasklet_struct *head;
        struct tasklet_struct **tail;
};
```

`tasklet_struct` include/linux/interrupt.h `Tasklet` `Tasklet` `Tasklet` `tasklet_struct`

```
struct tasklet_struct
{
        struct tasklet_struct *next;
        unsigned long state;
        atomic_t count;
        void (*func)(unsigned long);
        unsigned long data;
};
```

5

- `Tasklet`
- `Tasklet`
- `Tasklet`
- `Tasklet`

- `softirq_init` tasklets `tasklet_vec` `tasklet_hi_vec` Tasklets Tasklets kernel/softirq.c `softirq_init` `open_softirq`

```
open_softirq(TASKLET_SOFTIRQ, tasklet_action);
open_softirq(HI_SOFTIRQ, tasklet_hi_action);
```

`open_softirq` Tasklets `tasklet_action` `tasklet_hi_action` `tasklet_hi_action` `HI_SOFTIRQ` `tasklet_action` `TASKLET_SOFTIRQ`

Linux API Tasklets `tasklet_init` `task_struct` `task_struct`

```
void tasklet_init(struct tasklet_struct *t,
                  void (*func)(unsigned long), unsigned long data)
{
    t->next = NULL;
    t->state = 0;
    atomic_set(&t->count, 0);
    t->func = func;
    t->data = data;
}
```

tasklet

```
DECLARE_TASKLET(name, func, data);
DECLARE_TASKLET_DISABLED(name, func, data);
```

Linux tasklet

```
void tasklet_schedule(struct tasklet_struct *t);
void tasklet_hi_schedule(struct tasklet_struct *t);
void tasklet_hi_schedule_first(struct tasklet_struct *t);
```

tasklet `tasklet_schedule`

```
static inline void tasklet_schedule(struct tasklet_struct *t)
{
    if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state))
        __tasklet_schedule(t);
}

void __tasklet_schedule(struct tasklet_struct *t)
{
        unsigned long flags;

        local_irq_save(flags);
        t->next = NULL;
        *__this_cpu_read(tasklet_vec.tail) = t;
        __this_cpu_write(tasklet_vec.tail, &(t->next));
        raise_softirq_irqoff(TASKLET_SOFTIRQ);
        local_irq_restore(flags);
}
```

tasklet `TASKLET_STATE_SCHED` tasklet `__tasklet_schedule` `__tasklet_schedule` `raise_softirq` tasklet `tasklet_vec` `raise_softirq_irqoff` Linux `tasklet_action` `TASKLET_SOFTIRQ` `tasklet_hi_action` `HI_SOFTIRQ` --- `tasklet_action` `tasklet_hi_vec`

`tasklet_action`

```
static void tasklet_action(struct softirq_action *a)
```

```
{
    local_irq_disable();
    list = __this_cpu_read(tasklet_vec.head);
    __this_cpu_write(tasklet_vec.head, NULL);
    __this_cpu_write(tasklet_vec.tail, this_cpu_ptr(&tasklet_vec.head));
    local_irq_enable();

    while (list) {
        if (tasklet_trylock(t)) {
            t->func(t->data);
            tasklet_unlock(t);
        }
        ...
        ...
        ...
    }
}
```

`tasklet_action` `local_irq_disable` ( ) tasklet `NULL` tasklet tasklet tasklet `tasklet_trylock` `TASKLET_STATE_RUN`

```
static inline int tasklet_trylock(struct tasklet_struct *t)
{
    return !test_and_set_bit(TASKLET_STATE_RUN, &(t)->state);
}
```

tasklet (`tasklet_init`) `tasklet_unlock` `TASKLET_STATE_RUN`

`tasklet` `tasklets`

`tasklets` Linux --

`tasklets` `tasklets` `tasklets` Tasklets Linux kernel/workqueue.c

```
struct worker_pool {
    spinlock_t              lock;
    int                     cpu;
    int                     node;
    int                     id;
    unsigned int            flags;

    struct list_head        worklist;
    int                     nr_workers;
...
...
...
```

`worker thread` include/linux/workqueue.h `work_struct`

```
struct work_struct {
    atomic_long_t data;
    struct list_head entry;
    work_func_t func;
#ifdef CONFIG_LOCKDEP
    struct lockdep_map lockdep_map;
#endif
};
```

`func` -- `data` --Linux `kworker` cpu

```
systemd-cgls -k | grep kworker
├─    5 [kworker/0:0H]
├─   15 [kworker/1:0H]
├─   20 [kworker/2:0H]
├─   25 [kworker/3:0H]
├─   30 [kworker/4:0H]
...
...
...
```

(`ksoftirqd`) Linux

```
#define DECLARE_WORK(n, f) \
    struct work_struct n = __WORK_INITIALIZER(n, f)
```

```
#define INIT_WORK(_work, _func)       \
    __INIT_WORK((_work), (_func), 0)

#define __INIT_WORK(_work, _func, _onstack)                    \
    do {                                                       \
            __init_work((_work), _onstack);                    \
            (_work)->data = (atomic_long_t) WORK_DATA_INIT();  \
            INIT_LIST_HEAD(&(_work)->entry);                   \
             (_work)->func = (_func);                          \
    } while (0)
```

`work_struct` `work` `queue_work` `queue_delayed_work`

```
static inline bool queue_work(struct workqueue_struct *wq,
                              struct work_struct *work)
{
    return queue_work_on(WORK_CPU_UNBOUND, wq, work);
}
```

`queue_work` `queue_work_on` `queue_work_on` `WORK_CPU_UNBOUND`
include/linux/workqueue.h `queue_work_on` `WORK_STRUCT_PENDING_BIT` `__queue_work`

```
bool queue_work_on(int cpu, struct workqueue_struct *wq,
           struct work_struct *work)
{
    bool ret = false;
    ...
    if (!test_and_set_bit(WORK_STRUCT_PENDING_BIT, work_data_bits(work))) {
        __queue_work(cpu, wq, work);
        ret = true;
    }
    ...
    return ret;
}
```

`__queue_work` `work poll` `work poll` `workqueue` `works` `workqueue` Linux `worker_pool`
`work poll` `workqueue_struct` `pwqs` `worker_pool` `workqueue` `worker_pool` `worker_pool`
`pool_workqueue` `workqueue` `worker_pool` `__queue_work` `raw_smp_processor_id` cpu)
`work_struct` `pool_workqueue` `work` `workqueue`

```
static void __queue_work(int cpu, struct workqueue_struct *wq,
                         struct work_struct *work)
{
...
...
...
```

```
if (req_cpu == WORK_CPU_UNBOUND)
    cpu = raw_smp_processor_id();

if (!(wq->flags & WQ_UNBOUND))
    pwq = per_cpu_ptr(wq->cpu_pwqs, cpu);
else
    pwq = unbound_pwq_by_node(wq, cpu_to_node(cpu));
...
...
...
insert_work(pwq, work, worklist, work_flags);
```

`works` `workqueue` `works` `workqueue` `works` `worker_thread` `work_struct` `works` `workqueue`

`IRQs` `irq_desc` `tasklet`

Twitter

PR **linux-insides(PR** **linux-insides-zh)**

- initcall
- IF
- eflags
- CPU masks
- per-cpu
- Workqueue
- Previous part

()

Linux `softirq` `tasklet` `workqueue`

StringARM** SA-100/21285 IRQ drivers/tty/serial/21285.c

Linux

- `module_init`
- `module_exit`

:

```
module_init(serial21285_init);
module_exit(serial21285_exit);
```

Linux `module_init` `module_exit` include/linux/init.h :

```
#define module_init(initfn)                                     \
        static inline initcall_t __inittest(void)               \
        { return initfn; }                                      \
        int init_module(void) __attribute__((alias(#initfn)));

#define module_exit(exitfn)                                     \
        static inline exitcall_t __exittest(void)               \
        { return exitfn; }                                      \
        void cleanup_module(void) __attribute__((alias(#exitfn)));
```

initcall

- `early_initcall`
- `pure_initcall`
- `core_initcall`
- `postcore_initcall`
- `arch_initcall`
- `subsys_initcall`
- `fs_initcall`
- `rootfs_initcall`
- `device_initcall`
- `late_initcall`

init/main.c `do_initcalls` Linux

```
#define module_init(x)  __initcall(x);
#define module_exit(x)  __exitcall(x);
```

kernel/module.c `do_init_module` Linux `module_init` - `serial21285_init`

```
static int __init serial21285_init(void)
{
    int ret;
```

```
    printk(KERN_INFO "Serial: 21285 driver\n");

    serial21285_setup_ports();

    ret = uart_register_driver(&serial21285_reg);
    if (ret == 0)
        uart_add_one_port(&serial21285_reg, &serial21285_port);

    return ret;
}
```

`serial21285_setup_ports` `serial21285_port` uart

```
unsigned int mem_fclk_21285 = 50000000;

static void serial21285_setup_ports(void)
{
    serial21285_port.uartclk = mem_fclk_21285 / 4;
}
```

`serial21285` `uart`

```
static struct uart_driver serial21285_reg = {
    .owner            = THIS_MODULE,
    .driver_name    = "ttyFB",
    .dev_name        = "ttyFB",
    .major            = SERIAL_21285_MAJOR,
    .minor            = SERIAL_21285_MINOR,
    .nr                = 1,
    .cons            = SERIAL_21285_CONSOLE,
};
```

drivers/tty/serial/serial_core.c `uart_add_one_port` `serial21285_port` `serial21285_init`

```
if (ret == 0)
    uart_add_one_port(&serial21285_reg, &serial21285_port);

return ret;
```

`uart` drivers/tty/serial/serial_core.c `uart_open` `uart_startup` `startup` `uart_ops` `uart`

```
static struct uart_ops serial21285_ops = {
    ...
    .startup    = serial21285_startup,
    ...
}
```

`.startup` `serial21285_startup`

`serial21285_startup`

```
static int serial21285_startup(struct uart_port *port)
{
    int ret;

    tx_enabled(port) = 1;
    rx_enabled(port) = 1;
```

```
    ret = request_irq(IRQ_CONRX, serial21285_rx_chars, 0,
              serial21285_name, port);
    if (ret == 0) {
        ret = request_irq(IRQ_CONTX, serial21285_tx_chars, 0,
                  serial21285_name, port);
        if (ret)
            free_irq(IRQ_CONRX, port);
    }

    return ret;
}
```

`TX` `RX` - `RX` - `TX` `tx_enabled` `rx_enalbed` `request_irq`
include/linux/interrupt.h

```
static inline int __must_check
request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
            const char *name, void *dev)
{
        return request_threaded_irq(irq, handler, NULL, flags, name, dev);
}
```

`request_irq`

- `irq` -
- `handler` -
- `flags` -
- `name` -
- `dev` -

`request_irq` `IRQ_CONRX` `CONRX` arch/arm/mach-footbridge/include/mach/irqs.h `21285` `request_irq` `IRQ_CONTX` `RX` `TX`

```
#define IRQ_CONRX               _DC21285_IRQ(0)
#define IRQ_CONTX               _DC21285_IRQ(1)
...
...
...
#define _DC21285_IRQ(x)         (16 + (x))
```

ISA `0` `15` `16` `17` `request_irq` `serial21285_rx_chars` `serial21285_tx_chars` `RX` `TX` `flags` `request_irq` `flags` include/linux/interrupt.h `IRQF_*`

- `IRQF_SHARED` -
- `IRQF_PERCPU` - cpu(per cpu)
- `IRQF_NO_THREAD` -
- `IRQF_NOBALANCING` - irq
- `IRQF_IRQPOLL` -
-

`0` `IRQF_TRIGGER_NONE` (`name`) `serial21285_name`

```
static const char serial21285_name[] = "Footbridge UART";
```

`/proc/interrupts` `uart_port` `request_irq` `request_irq` kernel/irq/manage.c `request_threaded_irq` `irqaction` `irq_desc`

```
int request_threaded_irq(unsigned int irq, irq_handler_t handler,
                         irq_handler_t thread_fn, unsigned long irqflags,
                         const char *devname, void *dev_id)
{
```

```
        struct irqaction *action;
        struct irq_desc *desc;
        int retval;
        ...
        ...
        ...
}
```

`irqaction` `irq_desc` `irqaction` `request_threaded_irq` `request_irq` `irq_handler_t thread_fn` `NULL` `irq` `irq`

```
if (((irqflags & IRQF_SHARED) && !dev_id) ||
            (!(irqflags & IRQF_SHARED) && (irqflags & IRQF_COND_SUSPEND)) ||
            ((irqflags & IRQF_NO_SUSPEND) && (irqflags & IRQF_COND_SUSPEND)))
               return -EINVAL;
```

`dev_id` () `IRQF_COND_SUSPEND` `-EINVAL` kernel/irq/irqdesc.c `irq_to_desc` `irq` `irq` `-EINVAL`

```
desc = irq_to_desc(irq);
if (!desc)
    return -EINVAL;
```

`irq_to_desc` `irq` `irq` `irq_desc`

```
struct irq_desc *irq_to_desc(unsigned int irq)
{
        return (irq < NR_IRQS) ? irq_desc + irq : NULL;
}
```

`irq` `irq`

```
if (!irq_settings_can_request(desc) || WARN_ON(irq_settings_is_per_cpu_devid(desc)))
    return -EINVAL;
```

`-EINVAL` (`handler`) `request_irq` `thread_fn` `NULL` `-EINVAL` `request_irq` `thread_fn` `handler` `irq_default_primary_handler`

```
if (!handler) {
    if (!thread_fn)
        return -EINVAL;
    handler = irq_default_primary_handler;
}
```

`kzalloc` `irqaction`

```
action = kzalloc(sizeof(struct irqaction), GFP_KERNEL);
if (!action)
    return -ENOMEM;
```

`kzalloc` Linux `irqaction`

```
action->handler = handler;
action->thread_fn = thread_fn;
action->flags = irqflags;
action->name = devname;
action->dev_id = dev_id;
```

`request_threaded_irq` kernel/irq/manage.c `__setup_irq` `irqaction` `irqaction`

```
chip_bus_lock(desc);
retval = __setup_irq(irq, desc, action);
chip_bus_sync_unlock(desc);

if (retval)
    kfree(action);

return retval;
```

`__setup_irq` `chip_bus_lock` `chip_bus_sync_unlock` (i2c) `__setup_irq` `__setup_irq` `NULL` `irqchip` `NULL` `NULL` `irq_nested_primary_handler` `irq_default_priamry_handler` `thread_fn` `kthread_create`

```
if (new->thread_fn && !nested) {
    struct task_struct *t;
    t = kthread_create(irq_thread, new, "irq/%d-%s", irq, new->name);
    ...
}
```

`16` `17` `serial21285_rx_chars` `serial21285_tx_chars`

`irqaction` `native_init_IRQ` APIC

```
for_each_clear_bit_from(i, used_vectors, first_system_vector) {
    set_intr_gate(i, irq_entries_start +
        8 * (i - FIRST_EXTERNAL_VECTOR));
}
```

`first_system_vector` `used_vectors`

```
int first_system_vector = FIRST_SYSTEM_VECTOR; // 0xef
```

`i` `irq_entries_start + 8 * (i - FIRST_EXTERNAL_VECTOR)` - `irq_entries_start` arch/x86/entry/entry_64.S `irq`

```
    .align 8
ENTRY(irq_entries_start)
    vector=FIRST_EXTERNAL_VECTOR
    .rept (FIRST_SYSTEM_VECTOR - FIRST_EXTERNAL_VECTOR)
    pushq    $(~vector+0x80)
    vector=vector+1
    jmp    common_interrupt
    .align    8
    .endr
END(irq_entries_start)
```

GNU `.rept` `.endr` `FIRST_SYSTEM_VECTOR - FIRST_EXTERNAL_VECTOR` `FIRST_SYSTEM_VECTOR` `0xef` `FIRST_EXTERNAL_VECTOR` `0x20`

```
>>> 0xef - 0x20
207
```

`.rept` ( ) `vector` 1 `common_interrupt` `common_interrupt` `interrupt` `do_IRQ`

```
common_interrupt:
    addq    $-0x80, (%rsp)
    interrupt do_IRQ
```

`interrupt` `SWAPGS` `gs` per-cpu `irq_count` `do_IRQ` arch/x86/kernel/irq.c
`do_IRQ` - `pt_regs`

```
__visible unsigned int __irq_entry do_IRQ(struct pt_regs *regs)
{
    struct pt_regs *old_regs = set_irq_regs(regs);
    unsigned vector = ~regs->orig_ax;
    unsigned irq;

    irq_enter();
    exit_idle();
    ...
    ...
    ...
}
```

`set_irq_regs` `per-cpu` `irq_enter` `exit_idle` `irq_enter` `__preempt_count`
`exit_idle` pid `0` `idle` `IDLE_END` `idle_notifier`

cpu `irq` `handle_irq`

```
irq = __this_cpu_read(vector_irq[vector]);

if (!handle_irq(irq, regs)) {
    ...
    ...
    ...
}
...
...
...
```

`handle_irq` arch/x86/kernel/irq_64.c `generic_handle_irq_desc`

```
desc = irq_to_desc(irq);
    if (unlikely(!desc))
        return false;
generic_handle_irq_desc(irq, desc);
```

```
static inline void generic_handle_irq_desc(unsigned int irq, struct irq_desc *desc)
{
       desc->handle_irq(irq, desc);
}
```

..... `handle_irq` `irqaction` `irq_desc->handle_irq` API APIC `irq->actions(s)`
`serial21285_tx_chars` `serial21285_rx_chars`

`do_IRQ` `irq_exit` `set_irq_regs`

```
irq_exit();
set_irq_regs(old_regs);
return 1;
```

`IRQ`

`do_IRQ` arch/x86/entry/entry_64.S `ret_from_intr` `DISABLE_INTERRUPTS` `cli` per-cpu `irq_count` 1 `1`

```
DISABLE_INTERRUPTS(CLBR_NONE)
TRACE_IRQS_OFF
decl    PER_CPU_VAR(irq_count)
```

()

```
INTERRUPT_RETURN
```

`INTERRUPT_RETURN`

```
#define INTERRUPT_RETURN    jmp native_iret
```

```
ENTRY(native_iret)

.global native_irq_return_iret
native_irq_return_iret:
    iretq
```

Linux

twitter

linux-insides PR(PR linux-insides-zh)

-
- StrongARM** SA-110/21285
- IRQ
-
- initcall
- uart
- ISA
-
- i2c
- APIC
- GNU
-
- per-cpu
- pid
-
-
-

Linux

- - Linux
- Linux - Linux
- vsyscall and vDSO - `vsyscall` `vDSO`
- Linux -
- open - open
- Linux - `getrlimit/setrlimit`

Linux

linux , Linux System Call Linux VDSO vsyscall

Linux

?

socket [C] (https://en.wikipedia.org/wiki/C_%28programming_language%29)

Linux CPU x86_64 322 x86 358 `Hello world` :

```
.data

msg:
    .ascii "Hello, world!\n"
    len = . - msg

.text
    .global _start

_start:
    movq  $1, %rax
    movq  $1, %rdi
    movq  $msg, %rsi
    movq  $len, %rdx
    syscall

    movq  $60, %rax
    xorq  %rdi, %rdi
    syscall
```

:

```
$ gcc -c test.S
$ ld -o test test.o
```

:

```
./test
Hello, world!
```

Linux `x86_64` `Hello world` :

- `.data`
- `.text`

- `.data` (`Hello world`) - `.text` : `syscall` `syscall` `syscall` 64-ia-32-architectures-software-developer-vol-2b-manual:

#

```
SYSCALL 0IA32_LSTAR MSRRIP(RCX SYSCALL )
(WRMSR IA32_LSTAR MSR)
...
...
```

```
...
SYSCALL  IA32_STAR MSR  4732  CS  SS CS  SS ( GDT  LDT )

 SYSCALL
```

arch/x86/entry/entry_64.S `entry_SYSCALL_64` `syscalls` `SYSCALL` arch/x86/kernel/cpu/common.c `IA32_STAR` Model specific register:

```
wrmsrl(MSR_LSTAR, entry_SYSCALL_64);
```

`syscall` , - `write` write . write - `1` `rax` : `%rdi` , `%rsi` `%rdx` `write`

- (`1` stdout)
-
-

, `c` write [fs/read_write.c] (https://github.com/torvalds/linux/blob/master/fs/read_write.c) :

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
        size_t, count)
{
    ...
    ...
    ...
}
```

:

```
ssize_t write(int fd, const void *buf, size_t nbytes);
```

`SYSCALL_DEFINE3` ,

, exit:

- Return value

strace :

```
$ strace test
execve("./test", ["./test"], [/* 62 vars */]) = 0
write(1, "Hello, world!\n", 14Hello, world!
)         = 14
_exit(0)                                = ?

+++ exited with 0 +++
```

`strace` , execve : `write` `exit` - [x86-64 calling conventions] (https://en.wikipedia.org/wiki/X86_calling_conventions#x86-64_calling_conventions) `x86_64` - System V Application Binary Interface. PDF:

- `rdi`
- `rsi`
- `rdx`
- `rcx`
- `r8`
- `r9`

:

```
#include <stdio.h>

int main(int argc, char **argv)
{
   FILE *fp;
   char buff[255];

   fp = fopen("test.txt", "r");
   fgets(buff, 255, fp);
   printf("%s\n", buff);
   fclose(fp);

   return 0;
}
```

Linux `fopen` , `fgets` , `printf` `fclose` `open` , `read` `write` `close` `fopen` , `fgets` , `printf` `fclose` `C` standard library :

```
$ gcc test.c -o test
```

ltrace:

```
$ ltrace ./test
__libc_start_main([ "./test" ] <unfinished ...>
fopen("test.txt", "r")                                  = 0x602010
fgets("Hello World!\n", 255, 0x602010)                  = 0x7ffd2745e700
puts("Hello World!\n"Hello World!

)                                                       = 14
fclose(0x602010)                                        = 0
+++ exited (status 0) +++
```

`ltrace` `fopen` , `fgets` `buf` `puts` `stdout` `fclose` `puts` `write` `ltrace` `-S` :

```
write@SYS(1, "Hello World!\n\n", 14) = 14
```

// proc : `/proc/${pid}/syscall` , 1 systemd:

```
$ sudo cat /proc/1/comm
systemd

$ sudo cat /proc/1/syscall
232 0x4 0x7ffdf82e11b0 0x1f 0xffffffff 0x100 0x7ffdf82e11bf 0x7ffdf82e11a0 0x7f9114681193
```

`232` epoll_wait epoll I/O. `emacs` :

```
$ ps ax | grep emacs
2093 ?        Sl     2:40 emacs

$ sudo cat /proc/2093/comm
emacs

$ sudo cat /proc/2093/syscall
270 0xf 0x7fff068a5a90 0x7fff068a5b10 0x0 0x7fff068a59c0 0x7fff068a59d0 0x7fff068a59b0 0x7f777dd8813c
```

`270` sys_pselect6 `emacs`

`write`

write

Linux fs/read_write.c `write`

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
        size_t, count)
{
    struct fd f = fdget_pos(fd);
    ssize_t ret = -EBADF;

    if (f.file) {
        loff_t pos = file_pos_read(f.file);
        ret = vfs_write(f.file, buf, count, &pos);
        if (ret >= 0)
            file_pos_write(f.file, pos);
        fdput_pos(f);
    }

    return ret;
}
```

`SYSCALL_DEFINE3` include/linux/syscalls.h `sys_name(...)` :

```
#define SYSCALL_DEFINE3(name, ...) SYSCALL_DEFINEx(3, _##name, __VA_ARGS__)

#define SYSCALL_DEFINEx(x, sname, ...)                \
        SYSCALL_METADATA(sname, x, __VA_ARGS__)       \
        __SYSCALL_DEFINEx(x, sname, __VA_ARGS__)
```

`SYSCALL_DEFINE3` `name` `SYSCALL_DEFINEx` `_##name` (`##` documentation of gcc) `SYSCALL_DEFINEx` :

- `SYSCALL_METADATA` ;
- `__SYSCALL_DEFINEx` .

`SYSCALL_METADATA` `CONFIG_FTRACE_SYSCALLS` tracer `SYSCALL_METADATA` include/trace/syscall.h `syscall_metadata` , :

```
#define SYSCALL_METADATA(sname, nb, ...)                            \
    ...                                                             \
    ...                                                             \
    ...                                                             \
    struct syscall_metadata __used                                  \
          __syscall_meta_##sname = {                                \
                .name           = "sys"#sname,                      \
                .syscall_nr     = -1,                               \
                .nb_args        = nb,                               \
                .types          = nb ? types_##sname : NULL,        \
                .args           = nb ? args_##sname : NULL,         \
                .enter_event    = &event_enter_##sname,             \
                .exit_event     = &event_exit_##sname,              \
                .enter_fields   = LIST_HEAD_INIT(__syscall_meta_##sname.enter_fields), \
            };                                                      \

    static struct syscall_metadata __used                           \
          __attribute__((section("__syscalls_metadata")))           \
          *__p_syscall_meta_##sname = &__syscall_meta_##sname;
```

`CONFIG_FTRACE_SYSCALLS` `SYSCALL_METADATA` :

```
#define SYSCALL_METADATA(sname, nb, ...)
```

`__SYSCALL_DEFINEx` :

```
#define __SYSCALL_DEFINEx(x, name, ...)                                 \
        asmlinkage long sys##name(__MAP(x,__SC_DECL,__VA_ARGS__))       \
                __attribute__((alias(__stringify(SyS##name))));         \
                                                                        \
        static inline long SYSC##name(__MAP(x,__SC_DECL,__VA_ARGS__));  \
                                                                        \
        asmlinkage long SyS##name(__MAP(x,__SC_LONG,__VA_ARGS__));      \
                                                                        \
        asmlinkage long SyS##name(__MAP(x,__SC_LONG,__VA_ARGS__))       \
        {                                                               \
                long ret = SYSC##name(__MAP(x,__SC_CAST,__VA_ARGS__));  \
                __MAP(x,__SC_TEST,__VA_ARGS__);                         \
                __PROTECT(x, ret,__MAP(x,__SC_ARGS,__VA_ARGS__));       \
                return ret;                                             \
        }                                                               \
                                                                        \
        static inline long SYSC##name(__MAP(x,__SC_DECL,__VA_ARGS__))
```

`sys##name` `sys_system_call_name` `__SC_DECL` `__VA_ARGS__` `__MAP` `__SC_DECL` `__VA_ARGS__` `__SYSCALL_DEFINEx` CVE-2009-0029 write:

```
asmlinkage long sys_write(unsigned int fd, const char __user * buf, size_t count);
```

`write` :

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
        size_t, count)
{
    struct fd f = fdget_pos(fd);
    ssize_t ret = -EBADF;

    if (f.file) {
        loff_t pos = file_pos_read(f.file);
        ret = vfs_write(f.file, buf, count, &pos);
        if (ret >= 0)
            file_pos_write(f.file, pos);
        fdput_pos(f);
    }

    return ret;
}
```

:

- `fd` -
- `buf` -
- `count` -

`buf` , `__user` sparse Linux sparse [include/linux/compiler.h] (https://github.com/torvalds/linux/blob/master/include/linux/compiler.h) Linux `__CHECKER__` `sys_write` `f` `f` `fd` `fd` Linux `fdget_pos` `fdget_pos` `__to_fd` :

```
static inline struct fd fdget_pos(int fd)
{
        return __to_fd(__fdget_pos(fd));
}
```

`fdget_pos` `fd` `fdget_pos` , `current->files` , `fd` , `file_pos_read` `f_pos` :

```
static inline loff_t file_pos_read(struct file *file)
{
        return file->f_pos;
```

```
}
```

`vfs_write` `vfs_write` fs/read_write.c - `vfs_write` `vfs_write`, `file_pos_write`:

```
if (ret >= 0)
    file_pos_write(f.file, pos);
```

`f_pos`:

```
static inline void file_pos_write(struct file *file, loff_t pos)
{
        file->f_pos = pos;
}
```

`write`,:

```
fdput_pos(f);
```

`f_pos_lock`

Linux `write`, .

LinuxLinux

, twitter @ 0xAX, email issue.

PR **linux-insides.**

- system call
- vdso
- vsyscall
- general purpose registers
- socket
- C programming language
- x86
- x86_64
- x86-64 calling conventions
- System V Application Binary Interface. PDF
- GCC
- Intel manual. PDF
- system call table
- GCC macro documentation
- file descriptor
- stdout
- strace
- standard library
- wrapper functions
- ltrace
- sparse

- proc file system
- Virtual file system
- systemd
- epoll
- Previous chapter

Linux

Linux

Linux system call write Linux

`Hello World`

```
int main(int argc, char **argv)
{
    ...
    ...
    ...
    sys_write(fd1, buf, strlen(buf));
    ...
    ...
}
```

C standard library :

```
#include <unistd.h>

int main(int argc, char **argv)
{
    ...
    ...
    ...
    write(fd1, buf, strlen(buf));
    ...
    ...
}
```

`write` `syscall` Linux `syscall`

`syscall` (C) Linux Linux `system call table` Linux arch/x86/entry/syscall_64.c `sys_call_table` :

```
asmlinkage const sys_call_ptr_t sys_call_table[__NR_syscall_max+1] = {
    [0 ... __NR_syscall_max] = &sys_ni_syscall,
    #include <asm/syscalls_64.h>
};
```

`sys_call_table` `__NR_syscall_max + 1` `__NR_syscall_max` x86_64 , `__NR_syscall_max` `547` (Linux `5.0.0-rc7`) Kbuild - include/generated/asm-offsets.h`:

```
#define __NR_syscall_max 547
```

`x86_64` arch/x86/entry/syscalls/syscall_64.tbl ; `sys_call_table` `sys_call_p`

```
typedef void (*sys_call_ptr_t)(void);
```

`sys_call_table` `sys_ni_syscall` `sys_ni_syscall` “not-implemented” `sys_call_table` “not-implemented” `sys`

```
asmlinkage long sys_ni_syscall(void)
{
    return -ENOSYS;
}
```

The `-ENOSYS` error tells us that:

```
ENOSYS          Function not implemented (POSIX.1)
```

`sys_call_table` `...` GCC - Designated Initializers `asm/syscalls_64.h` arch/x86/entry/syscalls/syscalltbl.sh syscall table `asm/syscalls_64.h` :

```
__SYSCALL_COMMON(0, sys_read, sys_read)
__SYSCALL_COMMON(1, sys_write, sys_write)
__SYSCALL_COMMON(2, sys_open, sys_open)
__SYSCALL_COMMON(3, sys_close, sys_close)
__SYSCALL_COMMON(5, sys_newfstat, sys_newfstat)
...
...
...
```

`__SYSCALL_COMMON` `__SYSCALL_64` :

```
#define __SYSCALL_COMMON(nr, sym, compat) __SYSCALL_64(nr, sym, compat)
#define __SYSCALL_64(nr, sym, compat) [nr] = sym,
```

`sys_call_table` :

```
asmlinkage const sys_call_ptr_t sys_call_table[__NR_syscall_max+1] = {
    [0 ... __NR_syscall_max] = &sys_ni_syscall,
    [0] = sys_read,
    [1] = sys_write,
    [2] = sys_open,
    ...
    ...
    ...
};
```

“non-implemented” `sys_ni_syscall` `-ENOSYS` `sys_syscall_name`

Linux Linux `sys_syscall_name` Linux Linux Linux Linux

? Intel - 64-ia-32-architectures-software-developer-vol-2b-manual:

```
SYSCALL  0 IA32_LSTAR MSR  RIP
```

`IA32_LSTAR` model specific register Linux Linux Linux `trap_init` arch/x86/kernel/setup.c `non-early` `non-early` arch/x86/kernel/cpu/common.c `cpu_init` `per-cpu`

```
wrmsrl(MSR_STAR,  ((u64)__USER32_CS)<<48  | ((u64)__KERNEL_CS)<<32);
wrmsrl(MSR_LSTAR, entry_SYSCALL_64);
```

- `MSR_STAR` `63:48` `CS` `SS` `sysret` `MSR_STAR` `47:32` `CS` and `SS` `entry_SYSCALL_64` `MSR_LSTAR` `entry_SYSCALL_64` arch/x86/entry/entry_64.S () `entry_SYSCALL_64`

- `MSR_CSTAR` - target `rip` for the compability mode callers;
- `MSR_IA32_SYSENTER_CS` - target `cs` for the `sysenter` instruction;
- `MSR_IA32_SYSENTER_ESP` - target `esp` for the `sysenter` instruction;
- `MSR_IA32_SYSENTER_EIP` - target `eip` for the `sysenter` instruction.

`CONFIG_IA32_EMULATION` 64 32 `CONFIG_IA32_EMULATION`

```
wrmsrl(MSR_CSTAR, entry_SYSCALL_compat);
```

`entry_SYSENTER_compat` :

```
wrmsrl_safe(MSR_IA32_SYSENTER_CS, (u64)__KERNEL_CS);
wrmsrl_safe(MSR_IA32_SYSENTER_ESP, 0ULL);
wrmsrl_safe(MSR_IA32_SYSENTER_EIP, (u64)entry_SYSENTER_compat);
```

`CONFIG_IA32_EMULATION` `ignore_sysret` `MSR_CSTAR` :

```
wrmsrl(MSR_CSTAR, ignore_sysret);
```

arch/x86/entry/entry_64.S `-ENOSYS` :

```
ENTRY(ignore_sysret)
    mov    $-ENOSYS, %eax
    sysret
END(ignore_sysret)
```

`CONFIG_IA32_EMULATION` `MSR_IA32_SYSENTER_CS` `MSR_IA32_SYSENTER_ESP` `MSR_IA32_SYSENTER_EIP` `CONFIG_IA32_EMULATION` `MSR_IA32_SYSENTER_ESP` `MSR_IA32_SYSENTER_EIP` Global Descriptor Table `MSR_IA32_SYSENTER_CS` :

```
wrmsrl_safe(MSR_IA32_SYSENTER_CS, (u64)GDT_ENTRY_INVALID_SEG);
wrmsrl_safe(MSR_IA32_SYSENTER_ESP, 0ULL);
wrmsrl_safe(MSR_IA32_SYSENTER_EIP, 0ULL);
```

Linux `Global Descriptor Table`

`syscall_init` `MSR_SYSCALL_MASK`

```
wrmsrl(MSR_SYSCALL_MASK,
       X86_EFLAGS_TF|X86_EFLAGS_DF|X86_EFLAGS_IF|
       X86_EFLAGS_IOPL|X86_EFLAGS_AC|X86_EFLAGS_NT);
```

syscall `syscall_init` `syscall`

Linux `idtentry` `interrupt` `entry_SYSCALL_64`

`entry_SYSCALL_64` arch/x86/entry/entry_64.S :

```
SWAPGS_UNSAFE_STACK
```

arch/x86/include/asm/irqflags.h `swapgs` :

```
#define SWAPGS_UNSAFE_STACK    swapgs
```

GS `MSR_KERNEL_GS_BASE` `rsp_scratch` per-cpu

```
movq    %rsp, PER_CPU_VAR(rsp_scratch)
movq    PER_CPU_VAR(cpu_current_top_of_stack), %rsp
```

```
pushq   $__USER_DS
pushq   PER_CPU_VAR(rsp_scratch)
```

(`bp` `bx` `r12` `r15`)“non-implemented” `-ENOSYS` :

```
ENABLE_INTERRUPTS(CLBR_NONE)

pushq   %r11
pushq   $__USER_CS
pushq   %rcx
pushq   %rax
pushq   %rdi
pushq   %rsi
pushq   %rdx
pushq   %rcx
pushq   $-ENOSYS
pushq   %r8
pushq   %r9
pushq   %r10
pushq   %r11
sub    $(6*8), %rsp
```

:

- `rax` -
- `rcx` - contains return address to the user space;
- `r11` -
- `rdi` - system call handler
- `rsi` - system call handler
- `rdx` - system call handler
- `r10` - system call handler
- `r8` - system call handler
- `r9` - system call handler

(`rbp` `rbx` `r12` `r15`) C ABI) “non-implemented” dump

`thread_info` `_TIF_WORK_SYSCALL_ENTRY`

```
testl   $_TIF_WORK_SYSCALL_ENTRY, ASM_THREAD_INFO(TI_flags, %rsp, SIZEOF_PTREGS)
jnz    tracesys
```

`_TIF_WORK_SYSCALL_ENTRY` arch/x86/include/asm/thread_info.h

```
#define _TIF_WORK_SYSCALL_ENTRY \
```

```
    (_TIF_SYSCALL_TRACE | _TIF_SYSCALL_EMU | _TIF_SYSCALL_AUDIT |   \
    _TIF_SECCOMP | _TIF_SINGLESTEP | _TIF_SYSCALL_TRACEPOINT |     \
    _TIF_NOHZ)
```

/ Linux `tracesys` `entry_SYSCALL_64_fastpath` `entry_SYSCALL_64_fastpath` arch/x86/include/asm/unistd.h `__SYSCALL_MASK`

```
# ifdef CONFIG_X86_X32_ABI
#  define __SYSCALL_MASK (~(__X32_SYSCALL_BIT))
# else
#  define __SYSCALL_MASK (~0)
# endif
```

`__X32_SYSCALL_BIT`

```
#define __X32_SYSCALL_BIT    0x40000000
```

`__SYSCALL_MASK` `CONFIG_X86_X32_ABI` 64 32 ABI

`__SYSCALL_MASK` `CONFIG_X86_X32_ABI` `rax` (`__NR_syscall_max`) `CNOFIG_X86_X32_ABI` `eax` `X32_SYSCALL_BIT`

```
#if __SYSCALL_MASK == ~0
    cmpq    $__NR_syscall_max, %rax
#else
    andl    $__SYSCALL_MASK, %eax
    cmpl    $__NR_syscall_max, %eax
#endif
```

`ja` `CF` `ZF` 0 :

```
ja    1f
```

`r10` `rcx` x86_64 C ABI `call`

```
movq    %r10, %rcx
call    *sys_call_table(, %rax, 8)
```

`sys_call_table` `rax` `sys_call_table` 8 `*sys_call_table(, %rax, 8)` `sys_call_table`

Linux `SYSCALL_DEFINE[N]` `sys_read` `sys_write`

arch/x86/entry/entry_64.S:

```
call    *sys_call_table(, %rax, 8)
```

`rax` `RAX`

```
movq    %rax, RAX(%rsp)
```

arch/x86/include/asm/irqflags.h `LOCKDEP_SYS_EXIT` :

```
LOCKDEP_SYS_EXIT
```

`CONFIG_DEBUG_LOCK_ALLOC` `entry_SYSCALL_64` `rxc` `r11` `rcx` `r11`
flags register `rcx` `r11` `rsp` :

```
RESTORE_C_REGS_EXCEPT_RCX_R11

movq    RIP(%rsp), %rcx
movq    EFLAGS(%rsp), %r11
movq    RSP(%rsp), %rsp

USERGS_SYSRET64
```

`USERGS_SYSRET64` `swapgs` `GS` `GS` `sysretq`

```
#define USERGS_SYSRET64                 \
    swapgs;                             \
    sysretq;
```

- ;
- - `entry_SYSCALL_64` ;
- `entry_SYSCALL_64` ;
- `entry_SYSCALL_64` `rax` `sys_call_table` ;
- ;
- `sysretq` `entry_SYSCALL_64`

Linux Linux

twitter @ 0xAX email issue

PR **linux-insides**

Links

- system call
- write
- C standard library
- list of cpu architectures
- x86_64
- kbuild
- typedef
- errno
- gcc
- model specific register
- intel 2b manual
- coprocessor
- instruction pointer
- flags register
- Global Descriptor Table
- per-cpu
- general purpose registers
- ABI
- x86_64 C ABI

- previous chapter

Linux

vsyscalls vDSO

Linux `vsyscall` `vdso`

 Linux Linux

vsyscalls

`vsyscall` `virtual system call` `vsyscall` Linux X86_64 Linux [] (https://github.com/torvalds/linux/blob/master/Documentation/x86/x86_64/mm.txt)

```
ffffffffff600000 - ffffffffffdfffff (=8 MB) vsyscalls
```

:

```
~$ sudo cat /proc/1/maps | grep vsyscall
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

, `vsyscall` arch/x86/entry/vsyscall/vsyscall_64.c `map_vsyscall` Linux arch/x86/kernel/setup.c `setup_arch` (Linux)

 `map_vsyscall` `CONFIG_X86_VSYSCALL_EMULATION` :

```
#ifdef CONFIG_X86_VSYSCALL_EMULATION
extern void map_vsyscall(void);
#else
static inline void map_vsyscall(void) {}
#endif
```

, `CONFIG_X86_VSYSCALL_EMULATION` : `vsyscall` . `vsyscall` ?, `vsyscall` ABI , `vsyscall` `map_vsyscall` :

```
void __init map_vsyscall(void)
{
    extern char __vsyscall_page;
    unsigned long physaddr_vsyscall = __pa_symbol(&__vsyscall_page);
    ...
    ...
    ...
}
```

 `map_vsyscall` `__pa_symbol` `vsyscall` (of the Linux kernel initialization process) `__vsyscall_page` arch/x86/entry/vsyscall/vsyscall_emu_64.S :

```
ffffffff81881000 D __vsyscall_page
```

 `.data..page_aligned, aw` :

- `gettimeofday` ;
- `time` ;
- `getcpu` .

:

```
__vsyscall_page:
    mov $__NR_gettimeofday, %rax
    syscall
    ret

    .balign 1024, 0xcc
    mov $__NR_time, %rax
    syscall
    ret

    .balign 1024, 0xcc
    mov $__NR_getcpu, %rax
    syscall
    ret
```

`map_vsyscall` `__vsyscall_page` `__vsyscall_page` `__set_fixmap` `vsyscall` fix-mapped `vsyscall_mode`:

```
if (vsyscall_mode != NONE)
    __set_fixmap(VSYSCALL_PAGE, physaddr_vsyscall,
                 vsyscall_mode == NATIVE
                             ? PAGE_KERNEL_VSYSCALL
                             : PAGE_KERNEL_VVAR);
```

The `__set_fixmap` takes three arguments: The first is index of the `fixed_addresses` enum. In our case `VSYSCALL_PAGE` is the first element of the `fixed_addresses` enum for the `x86_64` architecture:

```
enum fixed_addresses {
...
...
...
#ifdef CONFIG_X86_VSYSCALL_EMULATION
    VSYSCALL_PAGE = (FIXADDR_TOP - VSYSCALL_ADDR) >> PAGE_SHIFT,
#endif
...
...
...
```

`511` `VSYSCALL_PAGE` `vsyscall_mode` `vsyscall_mode` `NATIVE` `PAGE_KERNEL_VSYSCALL` `PAGE_KERNEL_VVAR` (`PAGE_KERNEL_VSYSCALL` `PAGE_KERNEL_VVAR`):

```
#define __PAGE_KERNEL_VSYSCALL          (__PAGE_KERNEL_RX | _PAGE_USER)
#define __PAGE_KERNEL_VVAR              (__PAGE_KERNEL_RO | _PAGE_USER)
```

`vsyscall` `_PAGE_USER` `vsyscall_mode` (`__PAGE_KERNEL_VSYSCALL`) `vsyscall_mode` `NATIVE` `syscall` `vsyscall_mode` `emulate` vsyscall `PAGE_KERNEL_VVAR` `vsyscall_setup`

```
static int __init vsyscall_setup(char *str)
{
    if (str) {
        if (!strcmp("emulate", str))
            vsyscall_mode = EMULATE;
        else if (!strcmp("native", str))
            vsyscall_mode = NATIVE;
        else if (!strcmp("none", str))
            vsyscall_mode = NONE;
        else
            return -EINVAL;

        return 0;
```

```
    }

    return -EINVAL;
}
```

```
early_param("vsyscall", vsyscall_setup);
```

`early_param` Linux

`vsyscall_map` BUILD_BUG_ON `vsyscall` `VSYSCALL_ADDR` :

```
BUILD_BUG_ON((unsigned long)__fix_to_virt(VSYSCALL_PAGE) !=
             (unsigned long)VSYSCALL_ADDR);
```

`vsyscall` `vsyscall=native` arch/x86/entry/vsyscall/vsyscall_emu_64.S glibc `1024` (`0x400`)

```
__vsyscall_page:
    mov $__NR_gettimeofday, %rax
    syscall
    ret

    .balign 1024, 0xcc
    mov $__NR_time, %rax
    syscall
    ret

    .balign 1024, 0xcc
    mov $__NR_getcpu, %rax
    syscall
    ret
```

`vsyscall` `ffffffffff600000` , glibc `glibc`

```
#define VSYSCALL_ADDR_vgettimeofday   0xffffffffff600000
#define VSYSCALL_ADDR_vtime           0xffffffffff600400
#define VSYSCALL_ADDR_vgetcpu          0xffffffffff600800
```

`__vsyscall_page` + `VSYSCALL_ADDR_vsyscall_name` , x86_64

, `vsyscall=emulate` , page fault , `vsyscall` `__PAGE_KERNEL_VVAR` `do_page_fault` `#PF` page fault page fault `vsyscall` `emulate` `vsyscall` arch/x86/entry/vsyscall/vsyscall_64.c `emulate_vsyscall`

The `emulate_vsyscall` function gets the number of a virtual system call, checks it, prints error and sends segementation fault single:

```
...
...
...
vsyscall_nr = addr_to_vsyscall_nr(address);
if (vsyscall_nr < 0) {
    warn_bad_vsyscall(KERN_WARNING, regs, "misaligned vsyscall...);
    goto sigsegv;
}
...
...
...
sigsegv:
    force_sig(SIGSEGV, current);
    reutrn true;
```

As it checked number of a virtual system call, it does some yet another checks like `access_ok` violations and execute system call function depends on the number of a virtual system call:

```
switch (vsyscall_nr) {
    case 0:
        ret = sys_gettimeofday(
            (struct timeval __user *)regs->di,
            (struct timezone __user *)regs->si);
        break;
    ...
    ...
    ...
}
```

In the end we put the result of the `sys_gettimeofday` or another virtual system call handler to the `ax` general purpose register, as we did it with the normal system calls and restore the instruction pointer register and add `8` bytes to the stack pointer register. This operation emulates `ret` instruction.

```
    regs->ax = ret;

do_ret:
    regs->ip = caller;
    regs->sp += 8;
    return true;
```

That's all. Now let's look on the modern concept - `vDSO`.

Introduction to vDSO

As I already wrote above, `vsyscall` is an obsolete concept and replaced by the `vDSO` or `virtual dynamic shared object`. The main difference between the `vsyscall` and `vDSO` mechanisms is that `vDSO` maps memory pages into each process in a shared object form, but `vsyscall` is static in memory and has the same address every time. For the `x86_64` architecture it is called - `linux-vdso.so.1`. All userspace applications linked with this shared library via the `glibc`. For example:

```
~$ ldd /bin/uname
    linux-vdso.so.1 (0x00007ffe014b7000)
    libc.so.6 => /lib64/libc.so.6 (0x00007fbfee2fe000)
    /lib64/ld-linux-x86-64.so.2 (0x00005559aab7c000)
```

Or:

```
~$ sudo cat /proc/1/maps | grep vdso
7fff39f73000-7fff39f75000 r-xp 00000000 00:00 0       [vdso]
```

Here we can see that uname util was linked with the three libraries:

- `linux-vdso.so.1`;
- `libc.so.6`;
- `ld-linux-x86-64.so.2`.

The first provides `vDSO` functionality, the second is `C` standard library and the third is the program interpreter (more about this you can read in the part that describes linkers). So, the `vDSO` solves limitations of the `vsyscall`. Implementation of the `vDSO` is similar to `vsyscall`.

Initialization of the `vDSO` occurs in the `init_vdso` function that defined in the arch/x86/entry/vdso/vma.c source code file. This function starts from the initialization of the `vDSO` images for 32-bits and 64-bits depends on the `CONFIG_X86_X32_ABI` kernel configuration option:

```
static int __init init_vdso(void)
{
    init_vdso_image(&vdso_image_64);

#ifdef CONFIG_X86_X32_ABI
    init_vdso_image(&vdso_image_x32);
#endif
```

Both function initialize the `vdso_image` structure. This structure is defined in the two generated source code files: the arch/x86/entry/vdso/vdso-image-64.c and the arch/x86/entry/vdso/vdso-image-64.c. These source code files generated by the vdso2c program from the different source code files, represent different approaches to call a system call like `int 0x80`, `sysenter` and etc. The full set of the images depends on the kernel configuration.

For example for the `x86_64` Linux kernel it will contain `vdso_image_64`:

```
#ifdef CONFIG_X86_64
extern const struct vdso_image vdso_image_64;
#endif
```

But for the `x86` - `vdso_image_32`:

```
#ifdef CONFIG_X86_X32
extern const struct vdso_image vdso_image_x32;
#endif
```

If our kernel is configured for the `x86` architecture or for the `x86_64` and compability mode, we will have ability to call a system call with the `int 0x80` interrupt, if compability mode is enabled, we will be able to call a system call with the native `syscall instruction` or `sysenter` instruction in other way:

```
#if defined CONFIG_X86_32 || defined CONFIG_COMPAT
  extern const struct vdso_image vdso_image_32_int80;
#ifdef CONFIG_COMPAT
  extern const struct vdso_image vdso_image_32_syscall;
#endif
 extern const struct vdso_image vdso_image_32_sysenter;
#endif
```

As we can understand from the name of the `vdso_image` structure, it represents image of the `vDSO` for the certain mode of the system call entry. This structure contains information about size in bytes of the `vDSO` area that always a multiple of `PAGE_SIZE` (`4096` bytes), pointer to the text mapping, start and end address of the `alternatives` (set of instructions with better alternatives for the certain type of the processor) and etc. For example `vdso_image_64` looks like this:

```
const struct vdso_image vdso_image_64 = {
    .data = raw_data,
    .size = 8192,
    .text_mapping = {
        .name = "[vdso]",
        .pages = pages,
    },
    .alt = 3145,
    .alt_len = 26,
    .sym_vvar_start = -8192,
    .sym_vvar_page = -8192,
    .sym_hpet_page = -4096,
};
```

Where the `raw_data` contains raw binary code of the 64-bit `vDSO` system calls which are `2` page size:

```
static struct page *pages[2];
```

or 8 Kilobytes.

The `init_vdso_image` function is defined in the same source code file and just initializes the `vdso_image.text_mapping.pages`. First of all this function calculates the number of pages and initializes each `vdso_image.text_mapping.pages[number_of_page]` with the `virt_to_page` macro that converts given address to the `page` structure:

```
void __init init_vdso_image(const struct vdso_image *image)
{
    int i;
    int npages = (image->size) / PAGE_SIZE;

    for (i = 0; i < npages; i++)
        image->text_mapping.pages[i] =
            virt_to_page(image->data + i*PAGE_SIZE);
    ...
    ...
    ...
}
```

The `init_vdso` function passed to the `subsys_initcall` macro adds the given function to the `initcalls` list. All functions from this list will be called in the `do_initcalls` function from the init/main.c source code file:

```
subsys_initcall(init_vdso);
```

Ok, we just saw initialization of the `vDSO` and initialization of `page` structures that are related to the memory pages that contain `vDSO` system calls. But to where do their pages map? Actually they are mapped by the kernel, when it loads binary to the memory. The Linux kernel calls the `arch_setup_additional_pages` function from the arch/x86/entry/vdso/vma.c source code file that checks that `vDSO` enabled for the `x86_64` and calls the `map_vdso` function:

```
int arch_setup_additional_pages(struct linux_binprm *bprm, int uses_interp)
{
    if (!vdso64_enabled)
        return 0;

    return map_vdso(&vdso_image_64, true);
}
```

The `map_vdso` function is defined in the same source code file and maps pages for the `vDSO` and for the shared `vDSO` variables. That's all. The main differences between the `vsyscall` and the `vDSO` concepts is that `vsyscal` has a static address of `ffffffffff600000` and implements `3` system calls, whereas the `vDSO` loads dynamically and implements four system calls:

- `__vdso_clock_gettime`;
- `__vdso_getcpu`;
- `__vdso_gettimeofday`;
- `__vdso_time`.

That's all.

Conclusion

This is the end of the third part about the system calls concept in the Linux kernel. In the previous part we discussed the implementation of the preparation from the Linux kernel side, before a system call will be handled and implementation of the `exit` process from a system call handler. In this part we continued to dive into the stuff which is related to the system call concept and learned two new concepts that are very similar to the system call - the `vsyscall` and the `vDSO`.

After all of these three parts, we know almost all things that are related to system calls, we know what system call is and why user applications need them. We also know what occurs when a user application calls a system call and how the kernel handles system calls.

The next part will be the last part in this chapter and we will see what occurs when a user runs the program.

If you have questions or suggestions, feel free to ping me in twitter 0xAX, drop me email or just create issue.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-insides.

Links

- x86_64 memory map
- x86_64
- context switching
- ABI
- virtual address
- Segmentation
- enum
- fix-mapped addresses
- glibc
- BUILD_BUG_ON
- Processor register
- Page fault
- segementation fault
- instruction pointer
- stack pointer
- uname
- Linkers
- Previous part

System calls in the Linux kernel. Part 4.

How does the Linux kernel run a program

This is the fourth part of the chapter that describes system calls in the Linux kernel and as I wrote in the conclusion of the previous - this part will be last in this chapter. In the previous part we stopped at the two new concepts:

- `vsyscall`;
- `vDSO`;

that are related and very similar on system call concept.

This part will be last part in this chapter and as you can understand from the part's title - we will see what does occur in the Linux kernel when we run our programs. So, let's start.

how do we launch our programs?

There are many different ways to launch an application from an user perspective. For example we can run a program from the shell or double-click on the application icon. It does not matter. The Linux kernel handles application launch regardless how we do launch this application.

In this part we will consider the way when we just launch an application from the shell. As you know, the standard way to launch an application from shell is the following: We just launch a terminal emulator application and just write the name of the program and pass or not arguments to our program, for example:

```
~$ ls --version
ls (GNU coreutils) 8.23
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Written by Richard M. Stallman and David MacKenzie.
```

Let's consider what does occur when we launch an application from the shell, what does shell do when we write program name, what does Linux kernel do etc. But before we will start to consider these interesting things, I want to warn that this book is about the Linux kernel. That's why we will see Linux kernel insides related stuff mostly in this part. We will not consider in details what does shell do, we will not consider complex cases, for example subshells etc.

My default shell is - bash, so I will consider how do bash shell launches a program. So let's start. The `bash` shell as well as any program that written with C programming language starts from the main function. If you will look on the source code of the `bash` shell, you will find the `main` function in the shell.c source code file. This function makes many different things before the main thread loop of the `bash` started to work. For example this function:

- checks and tries to open `/dev/tty`;
- check that shell running in debug mode;
- parses command line arguments;
- reads shell environment;
- loads `.bashrc`, `.profile` and other configuration files;
- and many many more.

After all of these operations we can see the call of the `reader_loop` function. This function defined in the eval.c source code file and represents main thread loop or in other words it reads and executes commands. As the `reader_loop` function made all checks and read the given program name and arguments, it calls the `execute_command` function from the execute_cmd.c source code file. The `execute_command` function through the chain of the functions calls:

```
execute_command
--> execute_command_internal
----> execute_simple_command
------> execute_disk_command
--------> shell_execve
```

makes different checks like do we need to start `subshell`, was it builtin `bash` function or not etc. As I already wrote above, we will not consider all details about things that are not related to the Linux kernel. In the end of this process, the `shell_execve` function calls the `execve` system call:

```
execve (command, args, env);
```

The `execve` system call has the following signature:

```
int execve(const char *filename, char *const argv [], char *const envp[]);
```

and executes a program by the given filename, with the given arguments and environment variables. This system call is the first in our case and only, for example:

```
$ strace ls
execve("/bin/ls", ["ls"], [/* 62 vars */]) = 0

$ strace echo
execve("/bin/echo", ["echo"], [/* 62 vars */]) = 0

$ strace uname
execve("/bin/uname", ["uname"], [/* 62 vars */]) = 0
```

So, an user application (`bash` in our case) calls the system call and as we already know the next step is Linux kernel.

execve system call

We saw preparation before a system call called by an user application and after a system call handler finished its work in the second part of this chapter. We stopped at the call of the `execve` system call in the previous paragraph. This system call defined in the fs/exec.c source code file and as we already know it takes three arguments:

```
SYSCALL_DEFINE3(execve,
        const char __user *, filename,
        const char __user *const __user *, argv,
        const char __user *const __user *, envp)
{
    return do_execve(getname(filename), argv, envp);
}
```

Implementation of the `execve` is pretty simple here, as we can see it just returns the result of the `do_execve` function. The `do_execve` function defined in the same source code file and do the following things:

- Initialize two pointers on a userspace data with the given arguments and environment variables;
- return the result of the `do_execveat_common`.

We can see its implementation:

```
struct user_arg_ptr argv = { .ptr.native = __argv };
struct user_arg_ptr envp = { .ptr.native = __envp };
return do_execveat_common(AT_FDCWD, filename, argv, envp, 0);
```

The `do_execveat_common` function does main work - it executes a new program. This function takes similar set of arguments, but as you can see it takes five arguments instead of three. The first argument is the file descriptor that represent directory with our application, in our case the `AT_FDCWD` means that the given pathname is interpreted relative to the current working directory of the calling process. The fifth argument is flags. In our case we passed `0` to the `do_execveat_common`. We will check in a next step, so will see it latter.

First of all the `do_execveat_common` function checks the `filename` pointer and returns if it is `NULL`. After this we check flags of the current process that limit of running processes is not exceed:

```
if (IS_ERR(filename))
    return PTR_ERR(filename);

if ((current->flags & PF_NPROC_EXCEEDED) &&
    atomic_read(&current_user()->processes) > rlimit(RLIMIT_NPROC)) {
    retval = -EAGAIN;
    goto out_ret;
}

current->flags &= ~PF_NPROC_EXCEEDED;
```

If these two checks were successful we unset `PF_NPROC_EXCEEDED` flag in the flags of the current process to prevent fail of the `execve`. You can see that in the next step we call the `unshare_files` function that defined in the kernel/fork.c and unshares the files of the current task and check the result of this function:

```
retval = unshare_files(&displaced);
if (retval)
    goto out_ret;
```

We need to call this function to eliminate potential leak of the execve'd binary's file descriptor. In the next step we start preparation of the `bprm` that represented by the `struct linux_binprm` structure (defined in the include/linux/binfmts.h header file). The `linux_binprm` structure is used to hold the arguments that are used when loading binaries. For example it contains `vma` field which has `vm_area_struct` type and represents single memory area over a contiguous interval in a given address space where our application will be loaded, `mm` field which is memory descriptor of the binary, pointer to the top of memory and many other different fields.

First of all we allocate memory for this structure with the `kzalloc` function and check the result of the allocation:

```
bprm = kzalloc(sizeof(*bprm), GFP_KERNEL);
if (!bprm)
    goto out_files;
```

After this we start to prepare the `binprm` credentials with the call of the `prepare_bprm_creds` function:

```
retval = prepare_bprm_creds(bprm);
    if (retval)
        goto out_free;

check_unsafe_exec(bprm);
current->in_execve = 1;
```

Initialization of the `binprm` credentials in other words is initialization of the `cred` structure that stored inside of the `linux_binprm` structure. The `cred` structure contains the security context of a task for example real uid of the task, real guid of the task, `uid` and `guid` for the virtual file system operations etc. In the next step as we executed preparation of the `bprm` credentials we check that now we can safely execute a program with the call of the `check_unsafe_exec` function and set the current process to the `in_execve` state.

After all of these operations we call the `do_open_execat` function that checks the flags that we passed to the `do_execveat_common` function (remember that we have `0` in the `flags`) and searches and opens executable file on disk, checks that our we will load a binary file from `noexec` mount points (we need to avoid execute a binary from filesystems that do not contain executable binaries like proc or sysfs), intializes `file` structure and returns pointer on this structure. Next we can see the call the `sched_exec` after this:

```
file = do_open_execat(fd, filename, flags);
```

```
retval = PTR_ERR(file);
if (IS_ERR(file))
    goto out_unmark;

sched_exec();
```

The `sched_exec` function is used to determine the least loaded processor that can execute the new program and to migrate the current process to it.

After this we need to check file descriptor of the give executable binary. We try to check does the name of the our binary file starts from the `/` symbol or does the path of the given executable binary is interpreted relative to the current working directory of the calling process or in other words file descriptor is `AT_FDCWD` (read above about this).

If one of these checks is successfull we set the binary parameter filename:

```
bprm->file = file;

if (fd == AT_FDCWD || filename->name[0] == '/') {
    bprm->filename = filename->name;
}
```

Otherwise if the filename is empty we set the binary parameter filename to the `/dev/fd/%d` or `/dev/fd/%d/%s` depends on the filename of the given executable binary which means that we will execute the file to which the file descriptor refers:

```
} else {
    if (filename->name[0] == '\0')
        pathbuf = kasprintf(GFP_TEMPORARY, "/dev/fd/%d", fd);
    else
        pathbuf = kasprintf(GFP_TEMPORARY, "/dev/fd/%d/%s",
                            fd, filename->name);
    if (!pathbuf) {
        retval = -ENOMEM;
        goto out_unmark;
    }

    bprm->filename = pathbuf;
}

bprm->interp = bprm->filename;
```

Note that we set not only the `bprm->filename` but also `bprm->interp` that will contain name of the program interpreter. For now we just write the same name there, but later it will be updated with the real name of the program interpreter depends on binary format of a program. You can read above that we already prepared `cred` for the `linux_binprm`. The next step is initalization of other fields of the `linux_binprm`. First of all we call the `bprm_mm_init` function and pass the `bprm` to it:

```
retval = bprm_mm_init(bprm);
if (retval)
    goto out_unmark;
```

The `bprm_mm_init` defined in the same source code file and as we can understand from the function's name, it makes initialization of the memory descriptor or in other words the `bprm_mm_init` function initializes `mm_struct` structure. This structure defined in the include/linux/mm_types.h header file and represents address space of a process. We will not consider implementation of the `bprm_mm_init` function because we do not know many important stuff related to the Linux kernel memory manager, but we just need to know that this function initializes `mm_struct` and populate it with a temporary stack `vm_area_struct`.

After this we calculate the count of the command line arguments which are were passed to the our executable binary, the count of the environment variables and set it to the `bprm->argc` and `bprm->envc` respectively:

```
bprm->argc = count(argv, MAX_ARG_STRINGS);
if ((retval = bprm->argc) < 0)
    goto out;
```

```
bprm->envc = count(envp, MAX_ARG_STRINGS);
if ((retval = bprm->envc) < 0)
    goto out;
```

As you can see we do this operations with the help of the `count` function that defined in the same source code file and calculates the count of strings in the `argv` array. The `MAX_ARG_STRINGS` macro defined in the include/uapi/linux/binfmts.h header file and as we can understand from the macro's name, it represents maximum number of strings that were passed to the `execve` system call. The value of the `MAX_ARG_STRINGS` :

```
#define MAX_ARG_STRINGS 0x7FFFFFFF
```

After we calculated the number of the command line arguments and environment variables, we call the `prepare_binprm` function. We already call the function with the similar name before this moment. This function is called `prepare_binprm_cred` and we remember that this function initializes `cred` structure in the `linux_bprm` . Now the `prepare_binprm` function:

```
retval = prepare_binprm(bprm);
if (retval < 0)
    goto out;
```

fills the `linux_binprm` structure with the `uid` from inode and read `128` bytes from the binary executable file. We read only first `128` from the executable file because we need to check a type of our executable. We will read the rest of the executable file in the later step. After the preparation of the `linux_bprm` structure we copy the filename of the executable binary file, command line arguments and enviroment variables to the `linux_bprm` with the call of the `copy_strings_kernel` function:

```
retval = copy_strings_kernel(1, &bprm->filename, bprm);
if (retval < 0)
    goto out;

retval = copy_strings(bprm->envc, envp, bprm);
if (retval < 0)
    goto out;

retval = copy_strings(bprm->argc, argv, bprm);
if (retval < 0)
    goto out;
```

And set the pointer to the top of new program's stack that we set in the `bprm_mm_init` function:

```
bprm->exec = bprm->p;
```

The top of the stack will contain the program filename and we store this fileneme tothe `exec` field of the `linux_bprm` structure.

Now we have filled `linux_bprm` structure, we call the `exec_binprm` function:

```
retval = exec_binprm(bprm);
if (retval < 0)
    goto out;
```

First of all we store the pid and `pid` that seen from the namespace of the current task in the `exec_binprm` :

```
old_pid = current->pid;
rcu_read_lock();
old_vpid = task_pid_nr_ns(current, task_active_pid_ns(current->parent));
rcu_read_unlock();
```

and call the:

```
search_binary_handler(bprm);
```

function. This function goes through the list of handlers that contains different binary formats. Currently the Linux kernel supports following binary formats:

- `binfmt_script` - support for interpreted scripts that are starts from the #! line;
- `binfmt_misc` - support differnt binary formats, according to runtime configuration of the Linux kernel;
- `binfmt_elf` - support elf format;
- `binfmt_aout` - support a.out format;
- `binfmt_flat` - support for flat format;
- `binfmt_elf_fdpic` - Support for elf FDPIC binaries;
- `binfmt_em86` - support for Intel elf binaries running on Alpha machines.

So, the search-binary_handler tries to call the `load_binary` function and pass `linux_binprm` to it. If the binary handler supports the given executable file format, it starts to prepare the executable binary for execution:

```
int search_binary_handler(struct linux_binprm *bprm)
{
    ...
    ...
    ...
    list_for_each_entry(fmt, &formats, lh) {
        retval = fmt->load_binary(bprm);
        if (retval < 0 && !bprm->mm) {
            force_sigsegv(SIGSEGV, current);
            return retval;
        }
    }

    return retval;
```

Where the `load_binary` for example for the elf checks the magic number (each `elf` binary file contains magic number in the header) in the `linux_bprm` buffer (remember that we read first `128` bytes from the executable binary file): and exit if it is not `elf` binary:

```
static int load_elf_binary(struct linux_binprm *bprm)
{
    ...
    ...
    ...
    loc->elf_ex = *((struct elfhdr *)bprm->buf);

    if (memcmp(elf_ex.e_ident, ELFMAG, SELFMAG) != 0)
        goto out;
```

If the given executable file is in `elf` format, the `load_elf_binary` continues to execute. The `load_elf_binary` does many different things to prepare on execution executable file. For example it checks the architecture and type of the executable file:

```
if (loc->elf_ex.e_type != ET_EXEC && loc->elf_ex.e_type != ET_DYN)
    goto out;
if (!elf_check_arch(&loc->elf_ex))
    goto out;
```

and exit if there is wrong architecture and executable file non executable non shared. Tries to load the `program header table` :

```
elf_phdata = load_elf_phdrs(&loc->elf_ex, bprm->file);
if (!elf_phdata)
    goto out;
```

that describes segments. Read the `program interpreter` and libraries that linked with the our executable binary file from disk and load it to memory. The `program interpreter` specified in the `.interp` section of the executable file and as you can read in the part that describes Linkers it is - `/lib64/ld-linux-x86-64.so.2` for the `x86_64`. It setups the stack and map `elf` binary into the correct location in memory. It maps the bss and the brk sections and does many many other different things to prepare executable file to execute.

In the end of the execution of the `load_elf_binary` we call the `start_thread` function and pass three arguments to it:

```
    start_thread(regs, elf_entry, bprm->p);
    retval = 0;
out:
    kfree(loc);
out_ret:
    return retval;
```

These arguments are:

- Set of registers for the new task;
- Address of the entry point of the new task;
- Address of the top of the stack for the new task.

As we can understand from the function's name, it starts new thread, but it is not so. The `start_thread` function just prepares new task's registers to be ready to run. Let's look on the implementation of this function:

```
void
start_thread(struct pt_regs *regs, unsigned long new_ip, unsigned long new_sp)
{
        start_thread_common(regs, new_ip, new_sp,
                            __USER_CS, __USER_DS, 0);
}
```

As we can see the `start_thread` function just makes a call of the `start_thread_common` function that will do all for us:

```
static void
start_thread_common(struct pt_regs *regs, unsigned long new_ip,
                    unsigned long new_sp,
                    unsigned int _cs, unsigned int _ss, unsigned int _ds)
{
        loadsegment(fs, 0);
        loadsegment(es, _ds);
        loadsegment(ds, _ds);
        load_gs_index(0);
        regs->ip                = new_ip;
        regs->sp                = new_sp;
        regs->cs                = _cs;
        regs->ss                = _ss;
        regs->flags             = X86_EFLAGS_IF;
        force_iret();
}
```

The `start_thread_common` function fills `fs` segment register with zero and `es` and `ds` with the value of the data segment register. After this we set new values to the instruction pointer, `cs` segments etc. In the end of the `start_thread_common` function we can see the `force_iret` macro that force a system call return via `iret` instruction. Ok, we prepared new thread to run in userspace and now we can return from the `exec_binprm` and now we are in the `do_execveat_common` again. After the `exec_binprm` will finish its execution we release memory for structures that was allocated before and return.

After we returned from the `execve` system call handler, execution of our program will be started. We can do it, because all context related information already configured for this purpose. As we saw the `execve` system call does not return control to a process, but code, data and other segments of the caller process are just overwritten of the program segments. The exit from our application will be implemented through the `exit` system call.

That's all. From this point our programm will be executed.

Conclusion

This is the end of the fourth and last part of the about the system calls concept in the Linux kernel. We saw almost all related stuff to the `system call` concept in these four parts. We started from the understanding of the `system call` concept, we have learned what is it and why do users applications need in this concept. Next we saw how does the Linux handle a system call from an user application. We met two similar concepts to the `system call` concept, they are `vsyscall` and `vDSO` and finally we saw how does Linux kernel run an user program.

If you have questions or suggestions, feel free to ping me in twitter 0xAX, drop me email or just create issue.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-insides.

Links

- System call
- shell
- bash
- entry point
- C
- environment variables
- file descriptor
- real uid
- virtual file system
- procfs
- sysfs
- inode
- pid
- namespace
- #!
- elf
- a.out
- flat
- Alpha
- FDPIC
- segments
- Linkers
- Processor register
- instruction pointer
- Previous part

open

Linux Linux Linux Linux Linux Linux sector,tracks

`read` , `write` , `open` , `close` , `dup`

open C `open`

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

int main(int argc, char *argv) {
        int fd = open("test", O_RDONLY);

        if fd < 0 {
                perror("Opening of the file is failed\n");
        }
        else {
                printf("file sucessfully opened\n");
        }

        close(fd);
        return 0;
}
```

`open` `open` `open` proc

```
$ sudo ls /proc/1/fd/

0  10  12  14  16  2   21  23  25  27  29  30  32  34  36  38  4   41  43  45  47  49  50  53  55  58  6   61  63  67
  8
1  11  13  15  19  20  22  24  26  28  3   31  33  35  37  39  40  42  44  46  48  5   51  54  57  59  60  62  65  7
  9
```

`open` `open` man

open

```
SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)
{
    if (force_o_largefile())
        flags |= O_LARGEFILE;

    return do_sys_open(AT_FDCWD, filename, flags, mode);
}
```

`SYSCALL_DEFINE` `open`

`open` fs/open.c

```
SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)
```

```
{
    if (force_o_largefile())
        flags |= O_LARGEFILE;

    return do_sys_open(AT_FDCWD, filename, flags, mode);
}
```

`do_sys_open` `open` `if`

```
if (force_o_largefile())
    flags |= O_LARGEFILE;
```

`force_o_largefile()` true `open` flags `O_LARGEFILE` `O_LARGEFILE` `open(2)` man

> O_LARGEFILE
>
> (LFS) Allow files whose sizes cannot be represented in an off_t (but can be represented in an off64_t) to be opened.

GNU C

> off_t
>
> This is a signed integer type used to represent file sizes. In the GNU C Library, this type is no narrower than int. If the source is compiled with _FILE_OFFSET_BITS == 64 this type is transparently replaced by off64_t.

> off64_t
>
> This type is used similar to off_t. The difference is that even on 32 bit machines, where the off_t type would have 32 bits, off64_t has 64 bits and so is able to address files up to 2^63 bytes in length. When compiling with _FILE_OFFSET_BITS == 64 this type is available under the name off_t.

`off_t` , `off64_t` `O_LARGEFILE` Linux 32 `O_LARGEFILE` 64 `open` include/linux/fcntl.h linux `force_o_largefile`

```
#ifndef force_o_largefile
#define force_o_largefile() (BITS_PER_LONG != 32)
#endif
```

CPU x86_64 `force_o_largefile` include/linux/fcntl.h

`force_o_largefile` x86_64 "true" 64 `force_o_largefile` true `O_LARGEFILE` `open` flags

`O_LARGEFILE` `force_o_largefile` `do_sys_open`

```
long do_sys_open(int dfd, const char __user *filename, int flags, umode_t mode)
{
    struct open_flags op;
    int fd = build_open_flags(flags, mode, &op);
    struct filename *tmp;

    if (fd)
        return fd;

    tmp = getname(filename);
    if (IS_ERR(tmp))
        return PTR_ERR(tmp);

    fd = get_unused_fd_flags(flags);
    if (fd >= 0) {
        struct file *f = do_filp_open(dfd, tmp, &op);
        if (IS_ERR(f)) {
            put_unused_fd(fd);
```

```
            fd = PTR_ERR(f);
        } else {
            fsnotify_open(f);
            fd_install(fd, f);
        }
    }
    putname(tmp);
    return fd;
}
```

`do_sys_open`

open(2) flags

`open` flags `mode` `do_sys_open` `build_open_flags` flags flags mode

`build_open_flags`

- flags -
- mode -

- `op` `open_flags`

```
struct open_flags {
        int open_flag;
        umode_t mode;
        int acc_mode;
        int intent;
        int lookup_flags;
};
```

fs/internal.h flags `build_open_flags` `open_flags`

`build_open_flags`

```
int acc_mode = ACC_MODE(flags);
```

`ACC_MODE` include/linux/fs.h,

```
#define ACC_MODE(x) ("\004\002\006\006"[(x)&O_ACCMODE])
#define O_ACCMODE   00000003
```

`"\004\002\006\006"`

```
"\004\002\006\006" == {'\004', '\002', '\006', '\006'}
```

`ACC_MODE` `[(x) & O_ACCMODE]` `O_ACCMODE` == 00000003. `x & O_ACCMODE` `read` , `write` `read/weite`

```
#define O_RDONLY        00000000
#define O_WRONLY        00000001
#define O_RDWR          00000002
```

`ACC_MODE` `MAY_WRITE` , `MAY_READ`

```
if (flags & (O_CREAT | __O_TMPFILE))
    op->mode = (mode & S_IALLUGO) | S_IFREG;
else
```

```
    op->mode = 0;
```

`open_flags`

> if neither O_CREAT nor O_TMPFILE is specified, then mode is ignored.

`O_CREAT` `O_TMPFILE` `opendir` (http://man7.org/linux/man-pages/man3/opendir.3.html)

fanotify `O_CLOSEXEC`

```
flags &= ~FMODE_NONOTIFY & ~O_CLOEXEC;
```

`execve` `open` `O_CLOSEXEC` `O_CLOSEXEC` fork) + execve)

flags `O_SYNC` `O_DSYNC`

```
if (flags & __O_SYNC)
    flags |= O_DSYNC;
```

`O_SYNC` `O_DSYNC` `O_SYNC` (`O_DSYNC`) `atime` , `mtime` Linux `O_DSYNC` + `__O_SYNC` , `__O_SYNC|O_DSYNC`

flags `O_TMPFILE_MASK` `O_CREAT` | `O_TMPFILE` `O_CREAT` & `O_TMPFILE`

```
if (flags & __O_TMPFILE) {
    if ((flags & O_TMPFILE_MASK) != O_TMPFILE)
        return -EINVAL;
    if (!(acc_mode & MAY_WRITE))
        return -EINVAL;
} else if (flags & O_PATH) {
          flags &= O_DIRECTORY | O_NOFOLLOW | O_PATH;
        acc_mode = 0;
}
```

man

> O_TMPFILE must be specified with one of O_RDWR or O_WRONLY

`O_TMPFILE` `O_PATH` `O_PATH`

- ()
-

`dup` , `fcntl` `read` , `write` `O_DIRECTORY | O_NOFOLLOW | O_PATH` `build_open_flags` `open_flags->open_flag`

```
op->open_flag = flags;
```

`open_flag` flags `umask` `mode` `open_flags` `op->acc_mode` `build_open_flags` `acc_mode` flag

```
if (flags & O_TRUNC)
        acc_mode |= MAY_WRITE;
if (flags & O_APPEND)
    acc_mode |= MAY_APPEND;
op->acc_mode = acc_mode;
```

`O_TRUNC` 0 `O_APPEND` append mode ()

`open_flags` - `intent` flags `O_PATH` `open_flags` 0

```
op->intent = flags & O_PATH ? 0 : LOOKUP_OPEN;
```

`open_flags` `LOOKUP_OPEN` `LOOKUP_CREATE` `O_EXEC`

```
if (flags & O_CREAT) {
    op->intent |= LOOKUP_CREATE;
    if (flags & O_EXCL)
        op->intent |= LOOKUP_EXCL;
}
```

`open_flags` `lookup_flags` :

```
if (flags & O_DIRECTORY)
    lookup_flags |= LOOKUP_DIRECTORY;
if (!(flags & O_NOFOLLOW))
    lookup_flags |= LOOKUP_FOLLOW;
op->lookup_flags = lookup_flags;

return 0;
```

`LOOKUP_DIRECTORY` `LOOKUP_FOLLOW` `build_open_flags` `open_flags` modes flags `do_sys_open`

`build_open_flags` flags modes `getname` `filename` `open`

```
tmp = getname(filename);
if (IS_ERR(tmp))
    return PTR_ERR(tmp);
```

getname fs/namei.c

```
struct filename *
getname(const char __user * filename)
{
        return getname_flags(filename, 0, NULL);
}
```

`getname_flags` `getname_flags` `filename` include/linux/fs.h

- name -
- uptr -
- aname - audit
- refcnt -
- iname - `PATH_MAX`

`getname_flags` `strncpy_from_user` `open`

```
fd = get_unused_fd_flags(flags);
```

`get_unused_fd_flags` minimum (`0`) maximum (`RLIMIT_NOFILE`) `open`
`get_unused_fd_flags` `O_CLOEXEC` flags

`do_sys_open` `do_filp_open function` :

```
struct file *f = do_filp_open(dfd, tmp, &op);
```

```
if (IS_ERR(f)) {
    put_unused_fd(fd);
    fd = PTR_ERR(f);
} else {
    fsnotify_open(f);
    fd_install(fd, f);
}
```

`do_filp_open()` `file` `file` `do_filp_open` `put_unused_fd` `do_filp_open()` `file` `file`

`do_filp_open()` fs/namei.c Linux `nameidata` inode `do_filp_open()` `open` `inode` `nameidata` `path_openat`

```
filp = path_openat(&nd, op, flags | LOOKUP_RCU);

if (unlikely(filp == ERR_PTR(-ECHILD)))
    filp = path_openat(&nd, op, flags);
if (unlikely(filp == ERR_PTR(-ESTALE)))
    filp = path_openat(&nd, op, flags | LOOKUP_REVAL);
```

`path_openat` Linux RCU nfs `path_openat` `path lookup` `dentry` (Linux)

`path_openat` `get_empty_flip()` `get_empty_flip()` `file` `file` `open` `O_TMPFILE` | `O_CREATE` `O_PATH` `do_tmpfile` `do_o_path`

`path_init` `inode` `dentry inode` `AT_CWD` `do_sys_open`

`path_init` loop loop `link_path_walk` `do_last` `link_path_walk` `walk_component` `dcache` `link_path_walk` `do_last` `link_path_walk` `file` `do_last` `vfs_open`

`vfs_open` fs/open.c Linux

`open` `open` `file_operations.open` filesystem

Linux , twitter @0xAX/email, issue. , Linux read

linux-insides PR

- system call
- open
- file descriptor
- proc
- GNU C Library Reference Manual
- IA-64
- x86_64
- opendir
- fanotify
- fork)
- execve)
- symlink
- audit

- inode
- RCU
- read
- previous part

Linux

- - Linux
- - this part describes `clocksource` framework in the Linux kernel.
- The tick broadcast framework and dyntick - tick broadcast framework and dyntick
- - Linux
- Clockevents - : `clockevents`.
- x86 - `x86_64`
- Linux -

Timers and time management in the Linux kernel. Part 1.

Introduction

This is yet another post that opens new chapter in the linux-insides book. The previous part was a list part of the chapter that describes system call concept and now time is to start new chapter. As you can understand from the post's title, this chapter will be devoted to the `timers` and `time management` in the Linux kernel. The choice of topic for the current chapter is not accidental. Timers and generally time management are very important and widely used in the Linux kernel. The Linux kernel uses timers for various tasks, different timeouts for example in TCP implementation, the kernel must know current time, scheduling asynchronous functions, next event interrupt scheduling and many many more.

So, we will start to learn implementation of the different time management related stuff in this part. We will see different types of timers and how do different Linux kernel subsystems use them. As always we will start from the earliest part of the Linux kernel and will go through initialization process of the Linux kernel. We already did it in the special chapter which describes initialization process of the Linux kernel, but as you may remember we missed some things there. And one of them is the initialization of timers.

Let's start.

Initialization of non-standard PC hardware clock

After the Linux kernel was decompressed (more about this you can read in the Kernel decompression part) the architecture non-specific code starts to work in the init/main.c source code file. After initialization of the lock validator, initialization of cgroups and setting canary value we can see the call of the `setup_arch` function.

As you may remember this function defined in the arch/x86/kernel/setup.c source code file and prepares/initializes architecture-specific stuff (for example it reserves place for bss section, reserves place for initrd, parses kernel command line and many many other things). Besides this, we can find some time management related functions there.

The first is:

```
x86_init.timers.wallclock_init();
```

We already saw `x86_init` structure in the chapter that describes initialization of the Linux kernel. This structure contains pointers to the default setup functions for the different platforms like Intel MID, Intel CE4100 and etc. The `x86_init` structure defined in the arch/x86/kernel/x86_init.c and as you can see it determines standard PC hardware by default.

As we can see, the `x86_init` structure has `x86_init_ops` type that provides a set of functions for platform specific setup like reserving standard resources, platform specific memory setup, initialization of interrupt handlers and etc. This structure looks like:

```
struct x86_init_ops {
    struct x86_init_resources       resources;
    struct x86_init_mpparse         mpparse;
    struct x86_init_irqs            irqs;
    struct x86_init_oem             oem;
    struct x86_init_paging          paging;
    struct x86_init_timers          timers;
    struct x86_init_iommu           iommu;
    struct x86_init_pci             pci;
};
```

We can note `timers` field that has `x86_init_timers` type and as we can understand by its name - this field is related to time management and timers. The `x86_init_timers` contains four fields which are all functions that returns pointer on void:

- `setup_percpu_clockev` - set up the per cpu clock event device for the boot cpu;
- `tsc_pre_init` - platform function called before TSC init;

- `timer_init` - initialize the platform timer;
- `wallclock_init` - initialize the wallclock device.

So, as we already know, in our case the `wallclock_init` executes initialization of the wallclock device. If we will look on the `x86_init` structure, we will see that `wallclock_init` points to the `x86_init_noop` :

```
struct x86_init_ops x86_init __initdata = {
    ...
    ...
    ...
    .timers = {
        .wallclock_init            = x86_init_noop,
    },
    ...
    ...
    ...
}
```

Where the `x86_init_noop` is just a function that does nothing:

```
void __cpuinit x86_init_noop(void) { }
```

for the standard PC hardware. Actually, the `wallclock_init` function is used in the Intel MID platform. Initialization of the `x86_init.timers.wallclock_init` located in the arch/x86/platform/intel-mid/intel-mid.c source code file in the `x86_intel_mid_early_setup` function:

```
void __init x86_intel_mid_early_setup(void)
{
    ...
    ...
    ...
    x86_init.timers.wallclock_init = intel_mid_rtc_init;
    ...
    ...
    ...
}
```

Implementation of the `intel_mid_rtc_init` function is in the arch/x86/platform/intel-mid/intel_mid_vrtc.c source code file and looks pretty easy. First of all, this function parses Simple Firmware Interface M-Real-Time-Clock table for the getting such devices to the `sfi_mrtc_array` array and initialization of the `set_time` and `get_time` functions:

```
void __init intel_mid_rtc_init(void)
{
    unsigned long vrtc_paddr;

    sfi_table_parse(SFI_SIG_MRTC, NULL, NULL, sfi_parse_mrtc);

    vrtc_paddr = sfi_mrtc_array[0].phys_addr;
    if (!sfi_mrtc_num || !vrtc_paddr)
        return;

    vrtc_virt_base = (void __iomem *)set_fixmap_offset_nocache(FIX_LNW_VRTC,
                                vrtc_paddr);

    x86_platform.get_wallclock = vrtc_get_time;
    x86_platform.set_wallclock = vrtc_set_mmss;
}
```

That's all, after this a device based on `Intel MID` will be able to get time from hardware clock. As I already wrote, the standard PC x86_64 architecture does not support `x86_init_noop` and just do nothing during call of this function. We just saw initialization of the real time clock for the Intel MID architecture and now times to return to the general `x86_64` architecture and will look on the time management related stuff there.

Acquainted with jiffies

If we will return to the `setup_arch` function which is located as you remember in the arch/x86/kernel/setup.c source code file, we will see the next call of the time management related function:

```
register_refined_jiffies(CLOCK_TICK_RATE);
```

Before we will look on the implementation of this function, we must know about jiffy. As we can read on wikipedia:

```
Jiffy is an informal term for any unspecified short period of time
```

This definition is very similar to the `jiffy` in the Linux kernel. There is global variable with the `jiffies` which holds the number of ticks that have occurred since the system booted. The Linux kernel sets this variable to zero:

```
extern unsigned long volatile __jiffy_data jiffies;
```

during initialization process. This global variable will be increased each time during timer interrupt. Besides this, near the `jiffies` variable we can see definition of the similar variable

```
extern u64 jiffies_64;
```

Actually only one of these variables is in use in the Linux kernel. And it depends on the processor type. For the x86_64 it will be `u64` use and for the x86 is `unsigned long` . We will see this if we will look on the arch/x86/kernel/vmlinux.lds.S linker script:

```
#ifdef CONFIG_X86_32
...
jiffies = jiffies_64;
...
#else
...
jiffies_64 = jiffies;
...
#endif
```

In the case of `x86_32` the `jiffies` will be lower `32` bits of the `jiffies_64` variable. Schematically, we can imagine it as follows

```
                    jiffies_64
+-----------------------------------------------------+
|                       |                             |
|                       |                             |
|                       |       jiffies on `x86_32`   |
|                       |                             |
|                       |                             |
+-----------------------------------------------------+
63                     31                             0
```

Now we know a little theory about `jiffies` and we can return to the our function. There is no architecture-specific implementation for our function - the `register_refined_jiffies` . This function located in the generic kernel code - kernel/time/jiffies.c source code file. Main point of the `register_refined_jiffies` is registration of the jiffy `clocksource` . Before we will look on the implementation of the `register_refined_jiffies` function, we must know what is it `clocksource` . As we can read in the comments:

```
The `clocksource` is hardware abstraction for a free-running counter.
```

I'm not sure about you, but that description didn't give a good understanding about the `clocksource` concept. Let's try to understand what is it, but we will not go deeper because this topic will be described in a separate part in much more detail. The main point of the `clocksource` is timekeeping abstraction or in very simple words - it provides a time value to the kernel. We already know about

`jiffies` interface that represents number of ticks that have occurred since the system booted. It represented by the global variable in the Linux kernel and increased each timer interrupt. The Linux kernel can use `jiffies` for time measurement. So why do we need in separate context like the `clocksource` ? Actually different hardware devices provide different clock sources that are widely in their capabilities. The availability of more precise techniques for time intervals measurement is hardware-dependent.

For example `x86` has on-chip a 64-bit counter that is called Time Stamp Counter and its frequency can be equal to processor frequency. Or for example High Precision Event Timer that consists of a `64-bit` counter of at least `10 MHz` frequency. Two different timers and they are both for `x86` . If we will add timers from other architectures, this only makes this problem more complex. The Linux kernel provides `clocksource` concept to solve the problem.

The clocksource concept represented by the `clocksource` structure in the Linux kernel. This structure defined in the include/linux/clocksource.h header file and contains a couple of fields that describe a time counter. For example it contains - `name` field which is the name of a counter, `flags` field that describes different properties of a counter, pointers to the `suspend` and `resume` functions, and many more.

Let's look on the `clocksource` structure for jiffies that defined in the kernel/time/jiffies.c source code file:

```
static struct clocksource clocksource_jiffies = {
    .name        = "jiffies",
    .rating        = 1,
    .read        = jiffies_read,
    .mask        = 0xffffffff,
    .mult        = NSEC_PER_JIFFY << JIFFIES_SHIFT,
    .shift        = JIFFIES_SHIFT,
    .max_cycles    = 10,
};
```

We can see definition of the default name here - `jiffies` , the next is `rating` field allows the best registered clock source to be chosen by the clock source management code available for the specified hardware. The `rating` may have following value:

- `1-99` - Only available for bootup and testing purposes;
- `100-199` - Functional for real use, but not desired.
- `200-299` - A correct and usable clocksource.
- `300-399` - A reasonably fast and accurate clocksource.
- `400-499` - The ideal clocksource. A must-use where available;

For example rating of the time stamp counter is `300` , but rating of the high precision event timer is `250` . The next field is `read` - is pointer to the function that allows to read clocksource's cycle value or in other words it just returns `jiffies` variable with `cycle_t` type:

```
static cycle_t jiffies_read(struct clocksource *cs)
{
        return (cycle_t) jiffies;
}
```

that is just 64-bit unsigned type:

```
typedef u64 cycle_t;
```

The next field is the `mask` value ensures that subtraction between counters values from non `64 bit` counters do not need special overflow logic. In our case the mask is `0xffffffff` and it is `32` bits. This means that `jiffy` wraps around to zero after `42` seconds:

```
>>> 0xffffffff
4294967295
# 42 nanoseconds
>>> 42 * pow(10, -9)
4.2000000000000006e-08
# 43 nanoseconds
>>> 43 * pow(10, -9)
```

```
4.3e-08
```

The next two fields `mult` and `shift` are used to convert the clocksource's period to nanoseconds per cycle. When the kernel calls the `clocksource.read` function, this function returns value in `machine` time units represented with `cycle_t` data type that we saw just now. To convert this return value to the nanoseconds we need in these two fields: `mult` and `shift`. The `clocksource` provides `clocksource_cyc2ns` function that will do it for us with the following expression:

```
((u64) cycles * mult) >> shift;
```

As we can see the `mult` field is equal:

```
NSEC_PER_JIFFY << JIFFIES_SHIFT

#define NSEC_PER_JIFFY  ((NSEC_PER_SEC+HZ/2)/HZ)
#define NSEC_PER_SEC    1000000000L
```

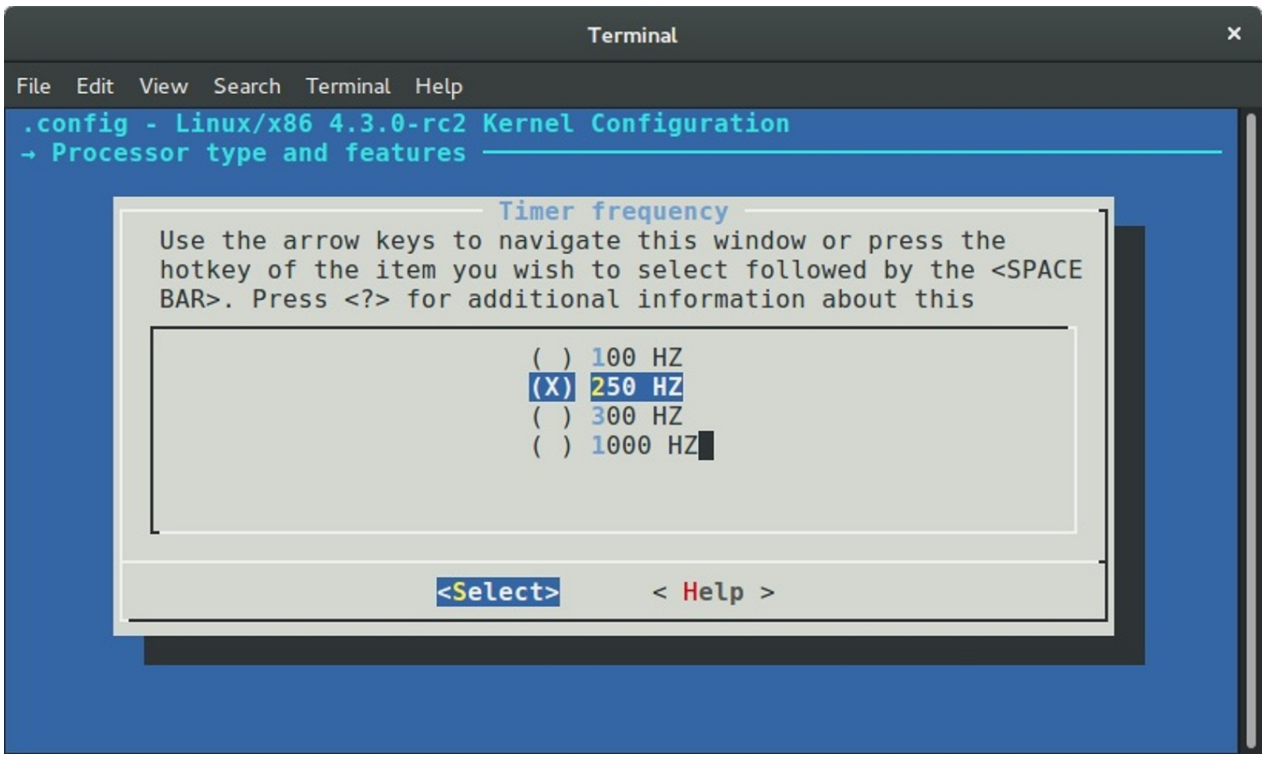
by default, and the `shift` is

```
#if HZ < 34
  #define JIFFIES_SHIFT   6
#elif HZ < 67
  #define JIFFIES_SHIFT   7
#else
  #define JIFFIES_SHIFT   8
#endif
```

The `jiffies` clock source uses the `NSEC_PER_JIFFY` multiplier conversion to specify the nanosecond over cycle ratio. Note that values of the `JIFFIES_SHIFT` and `NSEC_PER_JIFFY` depend on `HZ` value. The `HZ` represents the frequency of the system timer. This macro defined in the include/asm-generic/param.h and depends on the `CONFIG_HZ` kernel configuration option. The value of `HZ` differs for each supported architecture, but for `x86` it's defined like:

```
#define HZ        CONFIG_HZ
```

Where `CONFIG_HZ` can be one of the following values:



This means that in our case the timer interrupt frequency is `250 HZ` or occurs `250` times per second or one timer interrupt each `4ms`.

The last field that we can see in the definition of the `clocksource_jiffies` structure is the - `max_cycles` that holds the maximum cycle value that can safely be multiplied without potentially causing an overflow.

```
Ok, we just saw definition of the `clocksource_jiffies` structure, also we know a little about `jiffies` and `clockso
urce`, now is time to get back to the implementation of the our function. In the beginning of this part we have stopp
ed on the call of the:
```

```
register_refined_jiffies(CLOCK_TICK_RATE);
```

function from the arch/x86/kernel/setup.c source code file.

As I already wrote, the main purpose of the `register_refined_jiffies` function is to register `refined_jiffies` clocksource. We already saw the `clocksource_jiffies` structure represents standard `jiffies` clock source. Now, if you look in the kernel/time/jiffies.c source code file, you will find yet another clock source definition:

```
struct clocksource refined_jiffies;
```

There is one different between `refined_jiffies` and `clocksource_jiffies`: The standard `jiffies` based clock source is the lowest common denominator clock source which should function on all systems. As we already know, the `jiffies` global variable will be increased during each timer interrupt. This means that standard `jiffies` based clock source has the same resolution as the timer interrupt frequency. From this we can understand that standard `jiffies` based clock source may suffer from inaccuracies. The `refined_jiffies` uses `CLOCK_TICK_RATE` as the base of `jiffies` shift.

Let's look on the implementation of this function. First of all we can see that the `refined_jiffies` clock source based on the `clocksource_jiffies` structure:

```
int register_refined_jiffies(long cycles_per_second)
{
    u64 nsec_per_tick, shift_hz;
    long cycles_per_tick;

    refined_jiffies = clocksource_jiffies;
    refined_jiffies.name = "refined-jiffies";
    refined_jiffies.rating++;
    ...
    ...
    ...
```

Here we can see that we update the name of the `refined_jiffies` to `refined-jiffies` and increase the rating of this structure. As you remember, the `clocksource_jiffies` has rating - `1`, so our `refined_jiffies` clocksource will have rating - `2`. This means that the `refined_jiffies` will be best selection for clock source management code.

In the next step we need to calculate number of cycles per one tick:

```
cycles_per_tick = (cycles_per_second + HZ/2)/HZ;
```

Note that we have used `NSEC_PER_SEC` macro as the base of the standard `jiffies` multiplier. Here we are using the `cycles_per_second` which is the first parameter of the `register_refined_jiffies` function. We've passed the `CLOCK_TICK_RATE` macro to the `register_refined_jiffies` function. This macro definied in the arch/x86/include/asm/timex.h header file and expands to the:

```
#define CLOCK_TICK_RATE         PIT_TICK_RATE
```

where the `PIT_TICK_RATE` macro expands to the frequency of the Intel 8253:

```
#define PIT_TICK_RATE 1193182ul
```

After this we calculate `shift_hz` for the `register_refined_jiffies` that will store `hz << 8` or in other words frequency of the system timer. We shift left the `cycles_per_second` or frequency of the programmable interval timer on `8` in order to get extra accuracy:

```
shift_hz = (u64)cycles_per_second << 8;
shift_hz += cycles_per_tick/2;
do_div(shift_hz, cycles_per_tick);
```

In the next step we calculate the number of seconds per one tick by shifting left the `NSEC_PER_SEC` on `8` too as we did it with the `shift_hz` and do the same calculation as before:

```
nsec_per_tick = (u64)NSEC_PER_SEC << 8;
nsec_per_tick += (u32)shift_hz/2;
do_div(nsec_per_tick, (u32)shift_hz);
```

```
refined_jiffies.mult = ((u32)nsec_per_tick) << JIFFIES_SHIFT;
```

In the end of the `register_refined_jiffies` function we register new clock source with the `__clocksource_register` function that defined in the include/linux/clocksource.h header file and return:

```
__clocksource_register(&refined_jiffies);
return 0;
```

The clock source management code provides the API for clock source registration and selection. As we can see, clock sources are registered by calling the `__clocksource_register` function during kernel initialization or from a kernel module. During registration, the clock source management code will choose the best clock source available in the system using the `clocksource.rating` field which we already saw when we initialized `clocksource` structure for `jiffies`.

Using the jiffies

We just saw initialization of two `jiffies` based clock sources in the previous paragraph:

- standard `jiffies` based clock source;
- refined `jiffies` based clock source;

Don't worry if you don't understand the calculations here. They look frightening at first. Soon, step by step we will learn these things. So, we just saw initialization of `jffies` based clock sources and also we know that the Linux kernel has the global variable `jiffies` that holds the number of ticks that have occurred since the kernel started to work. Now, let's look how to use it. To use `jiffies` we just can use `jiffies` global variable by its name or with the call of the `get_jiffies_64` function. This function defined in the kernel/time/jiffies.c source code file and just returns full `64-bit` value of the `jiffies`:

```
u64 get_jiffies_64(void)
{
    unsigned long seq;
    u64 ret;

    do {
        seq = read_seqbegin(&jiffies_lock);
        ret = jiffies_64;
    } while (read_seqretry(&jiffies_lock, seq));
    return ret;
}
EXPORT_SYMBOL(get_jiffies_64);
```

Note that the `get_jiffies_64` function does not implemented as `jiffies_read` for example:

```
static cycle_t jiffies_read(struct clocksource *cs)
{
    return (cycle_t) jiffies;
}
```

We can see that implementation of the `get_jiffies_64` is more complex. The reading of the `jiffies_64` variable is implemented using seqlocks. Actually this is done for machines that cannot atomically read the full 64-bit values.

If we can access the `jiffies` or the `jiffies_64` variable we can convert it to `human` time units. To get one second we can use following expression:

```
jiffies / HZ
```

So, if we know this, we can get any time units. For example:

```
/* Thirty seconds from now */
jiffies + 30*HZ

/* Two minutes from now */
jiffies + 120*HZ

/* One millisecond from now */
jiffies + HZ / 1000
```

That's all.

Conclusion

This concludes the first part covering time and time management related concepts in the Linux kernel. We met first two concepts and its initialization in this part: `jiffies` and `clocksource`. In the next part we will continue to dive into this interesting theme and as I already wrote in this part we will acquainted and try to understand insides of these and other time management concepts in the Linux kernel.

If you have questions or suggestions, feel free to ping me in twitter 0xAX, drop me email or just create issue.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-insides.

Links

- system call
- TCP
- lock validator
- cgroups
- bss
- initrd
- Intel MID
- TSC
- void
- Simple Firmware Interface
- x86_64
- real time clock
- Jiffy

- high precision event timer
- nanoseconds
- Intel 8253
- seqlocks
- cloksource documentation
- Previous chapter

Timers and time management in the Linux kernel. Part 2.

Introduction to the `clocksource` framework

The previous part was the first part in the current chapter that describes timers and time management related stuff in the Linux kernel. We got acquainted with two concepts in the previous part:

- `jiffies`
- `clocksource`

The first is the global variable that is defined in the include/linux/jiffies.h header file and represents the counter that is increased during each timer interrupt. So if we can access this global variable and we know the timer interrupt rate we can convert `jiffies` to the human time units. As we already know the timer interrupt rate represented by the compile-time constant that is called `HZ` in the Linux kernel. The value of `HZ` is equal to the value of the `CONFIG_HZ` kernel configuration option and if we will look into the arch/x86/configs/x86_64_defconfig kernel configuration file, we will see that:

```
CONFIG_HZ_1000=y
```

kernel configuration option is set. This means that value of `CONFIG_HZ` will be `1000` by default for the x86_64 architecture. So, if we divide the value of `jiffies` by the value of `HZ`:

```
jiffies / HZ
```

we will get the amount of seconds that elapsed since the beginning of the moment the Linux kernel started to work or in other words we will get the system uptime. Since `HZ` represents the amount of timer interrupts in a second, we can set a value for some time in the future. For example:

```
/* one minute from now */
unsigned long later = jiffies + 60*HZ;

/* five minutes from now */
unsigned long later = jiffies + 5*60*HZ;
```

This is a very common practice in the Linux kernel. For example, if you will look into the arch/x86/kernel/smpboot.c source code file, you will find the `do_boot_cpu` function. This function boots all processors besides bootstrap processor. You can find a snippet that waits ten seconds for a response from the application processor:

```
if (!boot_error) {
    timeout = jiffies + 10*HZ;
    while (time_before(jiffies, timeout)) {
        ...
        ...
        ...
        udelay(100);
    }
    ...
    ...
    ...
}
```

We assign `jiffies + 10*HZ` value to the `timeout` variable here. As I think you already understood, this means a ten seconds timeout. After this we are entering a loop where we use the `time_before` macro to compare the current `jiffies` value and our timeout.

Or for example if we look into the sound/isa/sscape.c source code file which represents the driver for the Ensoniq Soundscape Elite sound card, we will see the `obp_startup_ack` function that waits upto a given timeout for the On-Board Processor to return its start-up acknowledgement sequence:

```
static int obp_startup_ack(struct soundscape *s, unsigned timeout)
{
    unsigned long end_time = jiffies + msecs_to_jiffies(timeout);

    do {
        ...
        ...
        ...
        x = host_read_unsafe(s->io_base);
        ...
        ...
        ...
        if (x == 0xfe || x == 0xff)
            return 1;
        msleep(10);
    } while (time_before(jiffies, end_time));

    return 0;
}
```

As you can see, the `jiffies` variable is very widely used in the Linux kernel code. As I already wrote, we met yet another new time management related concept in the previous part - `clocksource` . We have only seen a short description of this concept and the API for a clock source registration. Let's take a closer look in this part.

Introduction to `clocksource`

The `clocksource` concept represents the generic API for clock sources management in the Linux kernel. Why do we need a separate framework for this? Let's go back to the beginning. The `time` concept is the fundamental concept in the Linux kernel and other operating system kernels. And the timekeeping is one of the necessities to use this concept. For example Linux kernel must know and update the time elapsed since system startup, it must determine how long the current process has been running for every processor and many many more. Where the Linux kernel can get information about time? First of all it is Real Time Clock or RTC that represents by the a nonvolatile device. You can find a set of architecture-independent real time clock drivers in the Linux kernel in the drivers/rtc directory. Besides this, each architecture can provide a driver for the architecture-dependent real time clock, for example - `CMOS/RTC` - arch/x86/kernel/rtc.c for the x86 architecture. The second is system timer - timer that excites interrupts with a periodic rate. For example, for IBM PC compatibles it was - programmable interval timer.

We already know that for timekeeping purposes we can use `jiffies` in the Linux kernel. The `jiffies` can be considered as read only global variable which is updated with `HZ` frequency. We know that the `HZ` is a compile-time kernel parameter whose reasonable range is from `100` to `1000` Hz. So, it is guaranteed to have an interface for time measurement with `1` - `10` milliseconds resolution. Besides standard `jiffies` , we saw the `refined_jiffies` clock source in the previous part that is based on the `i8253/i8254` programmable interval timer tick rate which is almost `1193182` hertz. So we can get something about `1` microsecond resolution with the `refined_jiffies` . In this time, nanoseconds are the favorite choice for the time value units of the given clock source.

The availability of more precise techniques for time intervals measurement is hardware-dependent. We just knew a little about `x86` dependent timers hardware. But each architecture provides own timers hardware. Earlier each architecture had own implementation for this purpose. Solution of this problem is an abstraction layer and associated API in a common code framework for managing various clock sources and independent of the timer interrupt. This common code framework became - `clocksource` framework.

Generic timeofday and clock source management framework moved a lot of timekeeping code into the architecture independent portion of the code, with the architecture-dependent portion reduced to defining and managing low-level hardware pieces of clocksources. It takes a large amount of funds to measure the time interval on different architectures with different hardware, and it is very complex. Implementation of the each clock related service is strongly associated with an individual hardware device and as you can understand, it results in similar implementations for different architectures.

Within this framework, each clock source is required to maintain a representation of time as a monotonically increasing value. As we can see in the Linux kernel code, nanoseconds are the favorite choice for the time value units of a clock source in this time. One of the main point of the clock source framework is to allow an user to select clock source among a range of available hardware devices supporting clock functions when configuring the system and selecting, accessing and scaling different clock sources.

The clocksource structure

The fundamental of the `clocksource` framework is the `clocksource` structure that defined in the include/linux/clocksource.h header file. We already saw some fields that are provided by the `clocksource` structure in the previous part. Let's look on the full definition of this structure and try to describe all of its fields:

```
struct clocksource {
    cycle_t (*read)(struct clocksource *cs);
    cycle_t mask;
    u32 mult;
    u32 shift;
    u64 max_idle_ns;
    u32 maxadj;
#ifdef CONFIG_ARCH_CLOCKSOURCE_DATA
    struct arch_clocksource_data archdata;
#endif
    u64 max_cycles;
    const char *name;
    struct list_head list;
    int rating;
    int (*enable)(struct clocksource *cs);
    void (*disable)(struct clocksource *cs);
    unsigned long flags;
    void (*suspend)(struct clocksource *cs);
    void (*resume)(struct clocksource *cs);
#ifdef CONFIG_CLOCKSOURCE_WATCHDOG
    struct list_head wd_list;
    cycle_t cs_last;
    cycle_t wd_last;
#endif
    struct module *owner;
} ____cacheline_aligned;
```

We already saw the first field of the `clocksource` structure in the previous part - it is pointer to the `read` function that returns best counter selected by the clocksource framework. For example we use `jiffies_read` function to read `jiffies` value:

```
static struct clocksource clocksource_jiffies = {
    ...
    .read        = jiffies_read,
    ...
}
```

where `jiffies_read` just returns:

```
static cycle_t jiffies_read(struct clocksource *cs)
{
    return (cycle_t) jiffies;
}
```

Or the `read_tsc` function:

```
static struct clocksource clocksource_tsc = {
    ...
    .read                   = read_tsc,
    ...
};
```

for the time stamp counter reading.

The next field is `mask` that allows to ensure that subtraction between counters values from non `64 bit` counters do not need special overflow logic. After the `mask` field, we can see two fields: `mult` and `shift`. These are the fields that are base of mathematical functions that are provide ability to convert time values specific to each clock source. In other words these two fields help us to convert an abstract machine time units of a counter to nanoseconds.

After these two fields we can see the `64` bits `max_idle_ns` field represents max idle time permitted by the clocksource in nanoseconds. We need in this field for the Linux kernel with enabled `CONFIG_NO_HZ` kernel configuration option. This kernel configuration option enables the Linux kernel to run without a regular timer tick (we will see full explanation of this in other part). The problem that dynamic tick allows the kernel to sleep for periods longer than a single tick, moreover sleep time could be unlimited. The `max_idle_ns` field represents this sleeping limit.

The next field after the `max_idle_ns` is the `maxadj` field which is the maximum adjustment value to `mult`. The main formula by which we convert cycles to the nanoseconds:

```
((u64) cycles * mult) >> shift;
```

is not `100%` accurate. Instead the number is taken as close as possible to a nanosecond and `maxadj` helps to correct this and allows clocksource API to avoid `mult` values that might overflow when adjusted. The next four fields are pointers to the function:

- `enable` - optional function to enable clocksource;
- `disable` - optional function to disable clocksource;
- `suspend` - suspend function for the clocksource;
- `resume` - resume function for the clocksource;

The next field is the `max_cycles` and as we can understand from its name, this field represents maximum cycle value before potential overflow. And the last field is `owner` represents reference to a kernel module that is owner of a clocksource. This is all. We just went through all the standard fields of the `clocksource` structure. But you can noted that we missed some fields of the `clocksource` structure. We can divide all of missed field on two types: Fields of the first type are already known for us. For example, they are `name` field that represents name of a `clocksource`, the `rating` field that helps to the Linux kernel to select the best clocksource and etc. The second type, fields which are dependent from the different Linux kernel configuration options. Let's look on these fields.

The first field is the `archdata`. This field has `arch_clocksource_data` type and depends on the `CONFIG_ARCH_CLOCKSOURCE_DATA` kernel configuration option. This field is actual only for the x86 and IA64 architectures for this moment. And again, as we can understand from the field's name, it represents architecture-specific data for a clock source. For example, it represents `vDSO` clock mode:

```
struct arch_clocksource_data {
    int vclock_mode;
};
```

for the `x86` architectures. Where the `vDSO` clock mode can be one of the:

```
#define VCLOCK_NONE 0
#define VCLOCK_TSC  1
#define VCLOCK_HPET 2
#define VCLOCK_PVCLOCK 3
```

The last three fields are `wd_list`, `cs_last` and the `wd_last` depends on the `CONFIG_CLOCKSOURCE_WATCHDOG` kernel configuration option. First of all let's try to understand what is it `watchdog`. In a simple words, watchdog is a timer that is used for detection of the computer malfunctions and recovering from it. All of these three fields contain watchdog related data that is used by the `clocksource` framework. If we will grep the Linux kernel source code, we will see that only arch/x86/KConfig kernel configuration file contains the `CONFIG_CLOCKSOURCE_WATCHDOG` kernel configuration option. So, why do `x86` and `x86_64` need in watchdog? You already may know that all `x86` processors has special 64-bit register - time stamp counter. This register contains number of cycles since the reset. Sometimes the time stamp counter needs to be verified against another clock source. We will not see initialization of the `watchdog` timer in this part, before this we must learn more about timers.

That's all. From this moment we know all fields of the `clocksource` structure. This knowledge will help us to learn insides of the `clocksource` framework.

New clock source registration

We saw only one function from the `clocksource` framework in the previous part. This function was - `__clocksource_register` . This function defined in the include/linux/clocksource.h header file and as we can understand from the function's name, main point of this function is to register new clocksource. If we will look on the implementation of the `__clocksource_register` function, we will see that it just makes call of the `__clocksource_register_scale` function and returns its result:

```
static inline int __clocksource_register(struct clocksource *cs)
{
    return __clocksource_register_scale(cs, 1, 0);
}
```

Before we will see implementation of the `__clocksource_register_scale` function, we can see that `clocksource` provides additional API for a new clock source registration:

```
static inline int clocksource_register_hz(struct clocksource *cs, u32 hz)
{
        return __clocksource_register_scale(cs, 1, hz);
}

static inline int clocksource_register_khz(struct clocksource *cs, u32 khz)
{
        return __clocksource_register_scale(cs, 1000, khz);
}
```

And all of these functions do the same. They return value of the `__clocksource_register_scale` function but with different set of parameters. The `__clocksource_register_scale` function defined in the kernel/time/clocksource.c source code file. To understand difference between these functions, let's look on the parameters of the `clocksource_register_khz` function. As we can see, this function takes three parameters:

- `cs` - clocksource to be installed;
- `scale` - scale factor of a clock source. In other words, if we will multiply value of this parameter on frequency, we will get `hz` of a clocksource;
- `freq` - clock source frequency divided by scale.

Now let's look on the implementation of the `__clocksource_register_scale` function:

```
int __clocksource_register_scale(struct clocksource *cs, u32 scale, u32 freq)
{
        __clocksource_update_freq_scale(cs, scale, freq);
        mutex_lock(&clocksource_mutex);
        clocksource_enqueue(cs);
        clocksource_enqueue_watchdog(cs);
        clocksource_select();
        mutex_unlock(&clocksource_mutex);
        return 0;
}
```

First of all we can see that the `__clocksource_register_scale` function starts from the call of the `__clocksource_update_freq_scale` function that defined in the same source code file and updates given clock source with the new frequency. Let's look on the implementation of this function. In the first step we need to check given frequency and if it was not passed as `zero` , we need to calculate `mult` and `shift` parameters for the given clock source. Why do we need to check value of the `frequency` ? Actually it can be zero. if you attentively looked on the implementation of the `__clocksource_register` function, you

may have noticed that we passed `frequency` as `0`. We will do it only for some clock sources that have self defined `mult` and `shift` parameters. Look in the previous part and you will see that we saw calculation of the `mult` and `shift` for `jiffies`. The `__clocksource_update_freq_scale` function will do it for us for other clock sources.

So in the start of the `__clocksource_update_freq_scale` function we check the value of the `frequency` parameter and if is not zero we need to calculate `mult` and `shift` for the given clock source. Let's look on the `mult` and `shift` calculation:

```
void __clocksource_update_freq_scale(struct clocksource *cs, u32 scale, u32 freq)
{
        u64 sec;

        if (freq) {
             sec = cs->mask;
             do_div(sec, freq);
             do_div(sec, scale);

             if (!sec)
                   sec = 1;
             else if (sec > 600 && cs->mask > UINT_MAX)
                   sec = 600;

             clocks_calc_mult_shift(&cs->mult, &cs->shift, freq,
                                    NSEC_PER_SEC / scale, sec * scale);
        }
        ...
        ...
        ...
}
```

Here we can see calculation of the maximum number of seconds which we can run before a clock source counter will overflow. First of all we fill the `sec` variable with the value of a clock source mask. Remember that a clock source's mask represents maximum amount of bits that are valid for the given clock source. After this, we can see two division operations. At first we divide our `sec` variable on a clock source frequency and then on scale factor. The `freq` parameter shows us how many timer interrupts will be occurred in one second. So, we divide `mask` value that represents maximum number of a counter (for example `jiffy`) on the frequency of a timer and will get the maximum number of seconds for the certain clock source. The second division operation will give us maximum number of seconds for the certain clock source depends on its scale factor which can be `1` hertz or `1` kilohertz (10^ Hz).

After we have got maximum number of seconds, we check this value and set it to `1` or `600` depends on the result at the next step. These values is maximum sleeping time for a clocksource in seconds. In the next step we can see call of the `clocks_calc_mult_shift`. Main point of this function is calculation of the `mult` and `shift` values for a given clock source. In the end of the `__clocksource_update_freq_scale` function we check that just calculated `mult` value of a given clock source will not cause overflow after adjustment, update the `max_idle_ns` and `max_cycles` values of a given clock source with the maximum nanoseconds that can be converted to a clock source counter and print result to the kernel buffer:

```
pr_info("%s: mask: 0x%llx max_cycles: 0x%llx, max_idle_ns: %lld ns\n",
    cs->name, cs->mask, cs->max_cycles, cs->max_idle_ns);
```

that we can see in the dmesg output:

```
$ dmesg | grep "clocksource:"
[    0.000000] clocksource: refined-jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns: 1910969940391419 n
s
[    0.000000] clocksource: hpet: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns: 133484882848 ns
[    0.094084] clocksource: jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns: 1911260446275000 ns
[    0.205302] clocksource: acpi_pm: mask: 0xffffff max_cycles: 0xffffff, max_idle_ns: 2085701024 ns
[    1.452979] clocksource: tsc: mask: 0xffffffffffffffff max_cycles: 0x7350b459580, max_idle_ns: 881591204237 ns
```

After the `__clocksource_update_freq_scale` function will finish its work, we can return back to the `__clocksource_register_scale` function that will register new clock source. We can see the call of the following three functions:

```
mutex_lock(&clocksource_mutex);
```

```
clocksource_enqueue(cs);
clocksource_enqueue_watchdog(cs);
clocksource_select();
mutex_unlock(&clocksource_mutex);
```

Note that before the first will be called, we lock the `clocksource_mutex` mutex. The point of the `clocksource_mutex` mutex is to protect `curr_clocksource` variable which represents currently selected `clocksource` and `clocksource_list` variable which represents list that contains registered `clocksources`. Now, let's look on these three functions.

The first `clocksource_enqueue` function and other two defined in the same source code file. We go through all already registered `clocksources` or in other words we go through all elements of the `clocksource_list` and tries to find best place for a given `clocksource`:

```
static void clocksource_enqueue(struct clocksource *cs)
{
    struct list_head *entry = &clocksource_list;
    struct clocksource *tmp;

    list_for_each_entry(tmp, &clocksource_list, list)
        if (tmp->rating >= cs->rating)
            entry = &tmp->list;
    list_add(&cs->list, entry);
}
```

In the end we just insert new clocksource to the `clocksource_list`. The second function - `clocksource_enqueue_watchdog` does almost the same that previous function, but it inserts new clock source to the `wd_list` depends on flags of a clock source and starts new watchdog timer. As I already wrote, we will not consider `watchdog` related stuff in this part but will do it in next parts.

The last function is the `clocksource_select`. As we can understand from the function's name, main point of this function - select the best `clocksource` from registered clocksources. This function consists only from the call of the function helper:

```
static void clocksource_select(void)
{
    return __clocksource_select(false);
}
```

Note that the `__clocksource_select` function takes one parameter (`false` in our case). This bool parameter shows how to traverse the `clocksource_list`. In our case we pass `false` that is meant that we will go through all entries of the `clocksource_list`. We already know that `clocksource` with the best rating will the first in the `clocksource_list` after the call of the `clocksource_enqueue` function, so we can easily get it from this list. After we found a clock source with the best rating, we switch to it:

```
if (curr_clocksource != best && !timekeeping_notify(best)) {
    pr_info("Switched to clocksource %s\n", best->name);
    curr_clocksource = best;
}
```

The result of this operation we can see in the `dmesg` output:

```
$ dmesg | grep Switched
[    0.199688] clocksource: Switched to clocksource hpet
[    2.452966] clocksource: Switched to clocksource tsc
```

Note that we can see two clock sources in the `dmesg` output (`hpet` and `tsc` in our case). Yes, actually there can be many different clock sources on a particular hardware. So the Linux kernel knows about all registered clock sources and switches to a clock source with a better rating each time after registration of a new clock source.

If we will look on the bottom of the kernel/time/clocksource.c source code file, we will see that it has sysfs interface. Main initialization occurs in the `init_clocksource_sysfs` function which will be called during device `initcalls`. Let's look on the implementation of the `init_clocksource_sysfs` function:

```
static struct bus_type clocksource_subsys = {
    .name = "clocksource",
    .dev_name = "clocksource",
};

static int __init init_clocksource_sysfs(void)
{
    int error = subsys_system_register(&clocksource_subsys, NULL);

    if (!error)
        error = device_register(&device_clocksource);
    if (!error)
        error = device_create_file(
                &device_clocksource,
                &dev_attr_current_clocksource);
    if (!error)
        error = device_create_file(&device_clocksource,
                       &dev_attr_unbind_clocksource);
    if (!error)
        error = device_create_file(
                &device_clocksource,
                &dev_attr_available_clocksource);
    return error;
}
device_initcall(init_clocksource_sysfs);
```

First of all we can see that it registers a `clocksource` subsystem with the call of the `subsys_system_register` function. In other words, after the call of this function, we will have following directory:

```
$ pwd
/sys/devices/system/clocksource
```

After this step, we can see registration of the `device_clocksource` device which is represented by the following structure:

```
static struct device device_clocksource = {
    .id    = 0,
    .bus    = &clocksource_subsys,
};
```

and creation of three files:

- `dev_attr_current_clocksource` ;
- `dev_attr_unbind_clocksource` ;
- `dev_attr_available_clocksource` .

These files will provide information about current clock source in the system, available clock sources in the system and interface which allows to unbind the clock source.

After the `init_clocksource_sysfs` function will be executed, we will be able find some information about available clock sources in the:

```
$ cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
```

Or for example information about current clock source in the system:

```
$ cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
```

In the previous part, we saw API for the registration of the `jiffies` clock source, but didn't dive into details about the `clocksource` framework. In this part we did it and saw implementation of the new clock source registration and selection of a clock source with the best rating value in the system. Of course, this is not all API that `clocksource` framework provides. There a couple additional functions like `clocksource_unregister` for removing given clock source from the `clocksource_list` and etc. But I will not describe this functions in this part, because they are not important for us right now. Anyway if you are interesting in it, you can find it in the kernel/time/clocksource.c.

That's all.

Conclusion

This is the end of the second part of the chapter that describes timers and timer management related stuff in the Linux kernel. In the previous part got acquainted with the following two concepts: `jiffies` and `clocksource`. In this part we saw some examples of the `jiffies` usage and knew more details about the `clocksource` concept.

If you have questions or suggestions, feel free to ping me in twitter 0xAX, drop me email or just create issue.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-insides.

Links

- x86
- x86_64
- uptime
- Ensoniq Soundscape Elite
- RTC
- interrupts
- IBM PC
- programmable interval timer
- Hz
- nanoseconds
- dmesg
- time stamp counter
- loadable kernel module
- IA64
- watchdog
- clock rate
- mutex
- sysfs
- previous part

Timers and time management in the Linux kernel. Part 3.

The tick broadcast framework and dyntick

This is third part of the chapter which describes timers and time management related stuff in the Linux kernel and we stopped on the `clocksource` framework in the previous part. We have started to consider this framework because it is closely related to the special counters which are provided by the Linux kernel. One of these counters which we already saw in the first part of this chapter is - `jiffies`. As I already wrote in the first part of this chapter, we will consider time management related stuff step by step during the Linux kernel initialization. Previous step was call of the:

```
register_refined_jiffies(CLOCK_TICK_RATE);
```

function which defined in the kernel/time/jiffies.c source code file and executes initialization of the `refined_jiffies` clock source for us. Recall that this function is called from the `setup_arch` function that defined in the https://github.com/torvalds/linux/blob/master/arch/x86/kernel/setup.c source code and executes architecture-specific (x86_64 in our case) initialization. Look on the implementation of the `setup_arch` and you will note that the call of the `register_refined_jiffies` is the last step before the `setup_arch` function will finish its work.

There are many different `x86_64` specific things already configured after the end of the `setup_arch` execution. For example some early interrupt handlers already able to handle interrupts, memory space reserved for the initrd, DMI scanned, the Linux kernel log buffer is already set and this means that the printk function is able to work, e820 parsed and the Linux kernel already knows about available memory and and many many other architecture specific things (if you are interesting, you can read more about the `setup_arch` function and Linux kernel initialization process in the second chapter of this book).

Now, the `setup_arch` finished its work and we can back to the generic Linux kernel code. Recall that the `setup_arch` function was called from the `start_kernel` function which is defined in the init/main.c source code file. So, we shall return to this function. You can see that there are many different function are called right after `setup_arch` function inside of the `start_kernel` function, but since our chapter is devoted to timers and time management related stuff, we will skip all code which is not related to this topic. The first function which is related to the time management in the Linux kernel is:

```
tick_init();
```

in the `start_kernel`. The `tick_init` function defined in the kernel/time/tick-common.c source code file and does two things:

- Initialization of `tick broadcast` framework related data structures;
- Initialization of `full` tickless mode related data structures.

We didn't see anything related to the `tick broadcast` framework in this book and didn't know anything about tickless mode in the Linux kernel. So, the main point of this part is to look on these concepts and to know what are they.

The idle process

First of all, let's look on the implementation of the `tick_init` function. As I already wrote, this function defined in the kernel/time/tick-common.c source code file and consists from the two calls of following functions:

```
void __init tick_init(void)
{
    tick_broadcast_init();
    tick_nohz_init();
}
```

As you can understand from the paragraph's title, we are interesting only in the `tick_broadcast_init` function for now. This function defined in the kernel/time/tick-broadcast.c source code file and executes initialization of the `tick broadcast` framework related data structures. Before we will look on the implementation of the `tick_broadcast_init` function and will try to understand what does this function do, we need to know about `tick broadcast` framework.

Main point of a central processor is to execute programs. But sometimes a processor may be in a special state when it is not being used by any program. This special state is called - idle. When the processor has no anything to execute, the Linux kernel launches `idle` task. We already saw a little about this in the last part of the Linux kernel initialization process. When the Linux kernel will finish all initialization processes in the `start_kernel` function from the init/main.c source code file, it will call the `rest_init` function from the same source code file. Main point of this function is to launch kernel `init` thread and the `kthreadd` thread, to call the `schedule` function to start task scheduling and to go to sleep by calling the `cpu_idle_loop` function that defined in the kernel/sched/idle.c source code file.

The `cpu_idle_loop` function represents infinite loop which checks the need for rescheduling on each iteration. After the scheduler finds something to execute, the `idle` process will finish its work and the control will be moved to a new runnable task with the call of the `schedule_preempt_disabled` function:

```
static void cpu_idle_loop(void)
{
    while (1) {
        while (!need_resched()) {
        ...
        ...
        ...
        /* the main idle function */
        cpuidle_idle_call();
    }
    ...
    ...
    ...
    schedule_preempt_disabled();
}
```

Of course, we will not consider full implementation of the `cpu_idle_loop` function and details of the `idle` state in this part, because it is not related to our topic. But there is one interesting moment for us. We know that the processor can execute only one task in one time. How does the Linux kernel decide to reschedule and stop `idle` process if the processor executes infinite loop in the `cpu_idle_loop` ? The answer is system timer interrupts. When an interrupt occurs, the processor stops the `idle` thread and transfers control to an interrupt handler. After the system timer interrupt handler will be handled, the `need_resched` will return true and the Linux kernel will stop `idle` process and will transfer control to the current runnable task. But handling of the system timer interrupts is not effective for power management, because if a processor is in `idle` state, there is little point in sending it a system timer interrupt.

By default, there is the `CONFIG_HZ_PERIODIC` kernel configuration option which is enabled in the Linux kernel and tells to handle each interrupt of the system timer. To solve this problem, the Linux kernel provides two additional ways of managing scheduling-clock interrupts:

The first is to omit scheduling-clock ticks on idle processors. To enable this behaviour in the Linux kernel, we need to enable the `CONFIG_NO_HZ_IDLE` kernel configuration option. This option allows Linux kernel to avoid sending timer interrupts to idle processors. In this case periodic timer interrupts will be replaced with on-demand interrupts. This mode is called - `dyntick-idle` mode. But if the kernel does not handle interrupts of a system timer, how can the kernel decide if the system has nothing to do?

Whenever the idle task is selected to run, the periodic tick is disabled with the call of the `tick_nohz_idle_enter` function that defined in the kernel/time/tick-sched.c source code file and enabled with the call of the `tick_nohz_idle_exit` function. There is special concept in the Linux kernel which is called - `clock event devices` that are used to schedule the next interrupt. This concept provides API for devices which can deliver interrupts at a specific time in the future and represented by the `clock_event_device` structure in the Linux kernel. We will not dive into implementation of the `clock_event_device` structure now. We will see it in the next prat of this chapter. But there is one interesting moment for us right now.

The second way is to omit scheduling-clock ticks on processors that are either in `idle` state or that have only one runnable task or in other words busy processor. We can enable this feature with the `CONFIG_NO_HZ_FULL` kernel configuration option and it allows to reduce the number of timer interrupts significantly.

Besides the `cpu_idle_loop`, idle processor can be in a sleeping state. The Linux kernel provides special `cpuidle` framework. Main point of this framework is to put an idle processor to sleeping states. The name of the set of these states is - `C-states`. But how does a processor will be woken if local timer is disabled? The linux kernel provides `tick broadcast` framework for this. The main point of this framework is assign a timer which is not affected by the `C-states`. This timer will wake a sleeping processor.

Now, after some theory we can return to the implementation of our function. Let's recall that the `tick_init` function just calls two following functions:

```
void __init tick_init(void)
{
    tick_broadcast_init();
    tick_nohz_init();
}
```

Let's consider the first function. The first `tick_broadcast_init` function defined in the kernel/time/tick-broadcast.c source code file and executes initialization of the `tick broadcast` framework related data structures. Let's look on the implementation of the `tick_broadcast_init` function:

```
void __init tick_broadcast_init(void)
{
        zalloc_cpumask_var(&tick_broadcast_mask, GFP_NOWAIT);
        zalloc_cpumask_var(&tick_broadcast_on, GFP_NOWAIT);
        zalloc_cpumask_var(&tmpmask, GFP_NOWAIT);
#ifdef CONFIG_TICK_ONESHOT
         zalloc_cpumask_var(&tick_broadcast_oneshot_mask, GFP_NOWAIT);
         zalloc_cpumask_var(&tick_broadcast_pending_mask, GFP_NOWAIT);
         zalloc_cpumask_var(&tick_broadcast_force_mask, GFP_NOWAIT);
#endif
}
```

As we can see, the `tick_broadcast_init` function allocates different cpumasks with the help of the `zalloc_cpumask_var` function. The `zalloc_cpumask_var` function defined in the lib/cpumask.c source code file and expands to the call of the following function:

```
bool zalloc_cpumask_var(cpumask_var_t *mask, gfp_t flags)
{
        return alloc_cpumask_var(mask, flags | __GFP_ZERO);
}
```

Ultimately, the memory space will be allocated for the given `cpumask` with the certain flags with the help of the `kmalloc_node` function:

```
*mask = kmalloc_node(cpumask_size(), flags, node);
```

Now let's look on the `cpumasks` that will be initialized in the `tick_broadcast_init` function. As we can see, the `tick_broadcast_init` function will initialize six `cpumasks`, and moreover, initialization of the last three `cpumasks` will be depended on the `CONFIG_TICK_ONESHOT` kernel configuration option.

The first three `cpumasks` are:

- `tick_broadcast_mask` - the bitmap which represents list of processors that are in a sleeping mode;
- `tick_broadcast_on` - the bitmap that stores numbers of processors which are in a periodic broadcast state;
- `tmpmask` - this bitmap for temporary usage.

As we already know, the next three `cpumasks` depends on the `CONFIG_TICK_ONESHOT` kernel configuration option. Actually each clock event devices can be in one of two modes:

- `periodic` - clock events devices that support periodic events;
- `oneshot` - clock events devices that capable of issuing events that happen only once.

The linux kernel defines two mask for such clock events devices in the include/linux/clockchips.h header file:

```
#define CLOCK_EVT_FEAT_PERIODIC		0x000001
#define CLOCK_EVT_FEAT_ONESHOT		0x000002
```

So, the last three `cpumasks` are:

- `tick_broadcast_oneshot_mask` - stores numbers of processors that must be notified;
- `tick_broadcast_pending_mask` - stores numbers of processors that pending broadcast;
- `tick_broadcast_force_mask` - stores numbers of processors with enforced broadcast.

We have initialized six `cpumasks` in the `tick broadcast` framework, and now we can proceed to implementation of this framework.

The `tick broadcast` framework

Hardware may provide some clock source devices. When a processor sleeps and its local timer stopped, there must be additional clock source device that will handle awakening of a processor. The Linux kernel uses these `special` clock source devices which can raise an interrupt at a specified time. We already know that such timers called `clock events` devices in the Linux kernel. Besides `clock events` devices. Actually, each processor in the system has its own local timer which is programmed to issue interrupt at the time of the next deferred task. Also these timers can be programmed to do a periodical job, like updating `jiffies` and etc. These timers represented by the `tick_device` structure in the Linux kernel. This structure defined in the kernel/time/tick-sched.h header file and looks:

```
struct tick_device {
	struct clock_event_device *evtdev;
	enum tick_device_mode mode;
};
```

Note, that the `tick_device` structure contains two fields. The first field - `evtdev` represents pointer to the `clock_event_device` structure that defined in the include/linux/clockchips.h header file and represents descriptor of a clock event device. A `clock event` device allows to register an event that will happen in the future. As I already wrote, we will not consider `clock_event_device` structure and related API in this part, but will see it in the next part.

The second field of the `tick_device` structure represents mode of the `tick_device` . As we already know, the mode can be one of the:

```
num tick_device_mode {
	TICKDEV_MODE_PERIODIC,
	TICKDEV_MODE_ONESHOT,
};
```

Each `clock events` device in the system registers itself by the call of the `clockevents_register_device` function or `clockevents_config_and_register` function during initialization process of the Linux kernel. During the registration of a new `clock events` device, the Linux kernel calls the `tick_check_new_device` function that defined in the kernel/time/tick-common.c source code file and checks the given `clock events` device should be used by the Linux kernel. After all checks, the `tick_check_new_device` function executes a call of the:

```
tick_install_broadcast_device(newdev);
```

function that checks that the given `clock event` device can be broadcast device and install it, if the given device can be broadcast device. Let's look on the implementation of the `tick_install_broadcast_device` function:

```
void tick_install_broadcast_device(struct clock_event_device *dev)
```

```
{
    struct clock_event_device *cur = tick_broadcast_device.evtdev;

    if (!tick_check_broadcast_device(cur, dev))
        return;

    if (!try_module_get(dev->owner))
        return;

    clockevents_exchange_device(cur, dev);

    if (cur)
        cur->event_handler = clockevents_handle_noop;

    tick_broadcast_device.evtdev = dev;

    if (!cpumask_empty(tick_broadcast_mask))
        tick_broadcast_start_periodic(dev);

    if (dev->features & CLOCK_EVT_FEAT_ONESHOT)
        tick_clock_notify();
}
```

First of all we get the current `clock event` device from the `tick_broadcast_device`. The `tick_broadcast_device` defined in the kernel/time/tick-common.c source code file:

```
static struct tick_device tick_broadcast_device;
```

and represents external clock device that keeps track of events for a processor. The first step after we got the current clock device is the call of the `tick_check_broadcast_device` function which checks that a given clock events device can be utilized as broadcast device. The main point of the `tick_check_broadcast_device` function is to check value of the `features` field of the given `clock events` device. As we can understand from the name of this field, the `features` field contains a clock event device features. Available values defined in the include/linux/clockchips.h header file and can be one of the `CLOCK_EVT_FEAT_PERIODIC` - which represents a clock events device which supports periodic events and etc. So, the `tick_check_broadcast_device` function check `features` flags for `CLOCK_EVT_FEAT_ONESHOT`, `CLOCK_EVT_FEAT_DUMMY` and other flags and returns `false` if the given clock events device has one of these features. In other way the `tick_check_broadcast_device` function compares `ratings` of the given clock event device and current clock event device and returns the best.

After the `tick_check_broadcast_device` function, we can see the call of the `try_module_get` function that checks module owner of the clock events. We need to do it to be sure that the given `clock events` device was correctly initialized. The next step is the call of the `clockevents_exchange_device` function that defined in the kernel/time/clockevents.c source code file and will release old clock events device and replace the previous functional handler with a dummy handler.

In the last step of the `tick_install_broadcast_device` function we check that the `tick_broadcast_mask` is not empty and start the given `clock events` device in periodic mode with the call of the `tick_broadcast_start_periodic` function:

```
if (!cpumask_empty(tick_broadcast_mask))
    tick_broadcast_start_periodic(dev);

if (dev->features & CLOCK_EVT_FEAT_ONESHOT)
    tick_clock_notify();
```

The `tick_broadcast_mask` filled in the `tick_device_uses_broadcast` function that checks a `clock events` device during registration of this `clock events` device:

```
int cpu = smp_processor_id();

int tick_device_uses_broadcast(struct clock_event_device *dev, int cpu)
{
    ...
    ...
    ...
```

```
    if (!tick_device_is_functional(dev)) {
        ...
        cpumask_set_cpu(cpu, tick_broadcast_mask);
        ...
    }
    ...
    ...
    ...
}
```

More about the `smp_processor_id` macro you can read in the fourth part of the Linux kernel initialization process chapter.

The `tick_broadcast_start_periodic` function check the given `clock event` device and call the `tick_setup_periodic` function:

```
static void tick_broadcast_start_periodic(struct clock_event_device *bc)
{
    if (bc)
        tick_setup_periodic(bc, 1);
}
```

that defined in the kernel/time/tick-common.c source code file and sets broadcast handler for the given `clock event` device by the call of the following function:

```
tick_set_periodic_handler(dev, broadcast);
```

This function checks the second parameter which represents broadcast state (`on` or `off`) and sets the broadcast handler depends on its value:

```
void tick_set_periodic_handler(struct clock_event_device *dev, int broadcast)
{
    if (!broadcast)
        dev->event_handler = tick_handle_periodic;
    else
        dev->event_handler = tick_handle_periodic_broadcast;
}
```

When an `clock event` device will issue an interrupt, the `dev->event_handler` will be called. For example, let's look on the interrupt handler of the high precision event timer which is located in the arch/x86/kernel/hpet.c source code file:

```
static irqreturn_t hpet_interrupt_handler(int irq, void *data)
{
    struct hpet_dev *dev = (struct hpet_dev *)data;
    struct clock_event_device *hevt = &dev->evt;

    if (!hevt->event_handler) {
        printk(KERN_INFO "Spurious HPET timer interrupt on HPET timer %d\n",
                dev->num);
        return IRQ_HANDLED;
    }

    hevt->event_handler(hevt);
    return IRQ_HANDLED;
}
```

The `hpet_interrupt_handler` gets the irq specific data and check the event handler of the `clock event` device. Recall that we just set in the `tick_set_periodic_handler` function. So the `tick_handler_periodic_broadcast` function will be called in the end of the high precision event timer interrupt handler.

The `tick_handler_periodic_broadcast` function calls the

```
bc_local = tick_do_periodic_broadcast();
```

function which stores numbers of processors which have asked to be woken up in the temporary `cpumask` and call the `tick_do_broadcast` function:

```
cpumask_and(tmpmask, cpu_online_mask, tick_broadcast_mask);
return tick_do_broadcast(tmpmask);
```

The `tick_do_broadcast` calls the `broadcast` function of the given clock events which sends IPI interrupt to the set of the processors. In the end we can call the event handler of the given `tick_device` :

```
if (bc_local)
    td->evtdev->event_handler(td->evtdev);
```

which actually represents interrupt handler of the local timer of a processor. After this a processor will wake up. That is all about `tick broadcast` framework in the Linux kernel. We have missed some aspects of this framework, for example reprogramming of a `clock event` device and broadcast with the oneshot timer and etc. But the Linux kernel is very big, it is not real to cover all aspects of it. I think it will be interesting to dive into with yourself.

If you remember, we have started this part with the call of the `tick_init` function. We just consider the `tick_broadcast_init` function and releated theory, but the `tick_init` function contains another call of a function and this function is - `tick_nohz_init` . Let's look on the implementation of this function.

Initialization of dyntick related data structures

We already saw some information about `dyntick` concept in this part and we know that this concept allows kernel to disable system timer interrupts in the `idle` state. The `tick_nohz_init` function makes initialization of the different data structures which are related to this concept. This function defined in the kernel/time/tick-sched.c source code file and starts from the check of the value of the `tick_nohz_full_running` variable which represents state of the tick-less mode for the `idle` state and the state when system timer interrups are disabled during a processor has only one runnable task:

```
if (!tick_nohz_full_running) {
    if (tick_nohz_init_all() < 0)
    return;
}
```

If this mode is not running we call the `tick_nohz_init_all` function that defined in the same source code file and check its result. The `tick_nohz_init_all` function tries to allocate the `tick_nohz_full_mask` with the call of the `alloc_cpumask_var` that will allocate space for a `tick_nohz_full_mask` . The `tck_nohz_full_mask` will store numbers of processors that have enabled full `NO_HZ` . After successful allocation of the `tick_nohz_full_mask` we set all bits in the `tick_nogz_full_mask` , set the `tick_nohz_full_running` and return result to the `tick_nohz_init` function:

```
static int tick_nohz_init_all(void)
{
        int err = -1;
#ifdef CONFIG_NO_HZ_FULL_ALL
        if (!alloc_cpumask_var(&tick_nohz_full_mask, GFP_KERNEL)) {
                WARN(1, "NO_HZ: Can't allocate full dynticks cpumask\n");
                return err;
        }
        err = 0;
        cpumask_setall(tick_nohz_full_mask);
        tick_nohz_full_running = true;
#endif
        return err;
}
```

In the next step we try to allocate a memory space for the `housekeeping_mask` :

```
if (!alloc_cpumask_var(&housekeeping_mask, GFP_KERNEL)) {
    WARN(1, "NO_HZ: Can't allocate not-full dynticks cpumask\n");
    cpumask_clear(tick_nohz_full_mask);
    tick_nohz_full_running = false;
    return;
}
```

This `cpumask` will store number of processor for `housekeeping` or in other words we need at least in one processor that will not be in `NO_HZ` mode, because it will do timekeeping and etc. After this we check the result of the architecture-specific `arch_irq_work_has_interrupt` function. This function checks ability to send inter-processor interrupt for the certain architecture. We need to check this, because system timer of a processor will be disabled during `NO_HZ` mode, so there must be at least one online processor which can send inter-processor interrupt to awake offline processor. This function defined in the arch/x86/include/asm/irq_work.h header file for the x86_64 and just checks that a processor has APIC from the CPUID:

```
static inline bool arch_irq_work_has_interrupt(void)
{
    return cpu_has_apic;
}
```

If a processor has not `APIC`, the Linux kernel prints warning message, clears the `tick_nohz_full_mask` cpumask, copies numbers of all possible processors in the system to the `housekeeping_mask` and resets the value of the `tick_nohz_full_running` variable:

```
if (!arch_irq_work_has_interrupt()) {
    pr_warning("NO_HZ: Can't run full dynticks because arch doesn't "
           "support irq work self-IPIs\n");
    cpumask_clear(tick_nohz_full_mask);
    cpumask_copy(housekeeping_mask, cpu_possible_mask);
    tick_nohz_full_running = false;
    return;
}
```

After this step, we get the number of the current processor by the call of the `smp_processor_id` and check this processor in the `tick_nohz_full_mask`. If the `tick_nohz_full_mask` contains a given processor we clear appropriate bit in the `tick_nohz_full_mask`:

```
cpu = smp_processor_id();

if (cpumask_test_cpu(cpu, tick_nohz_full_mask)) {
    pr_warning("NO_HZ: Clearing %d from nohz_full range for timekeeping\n", cpu);
    cpumask_clear_cpu(cpu, tick_nohz_full_mask);
}
```

Because this processor will be used for timekeeping. After this step we put all numbers of processors that are in the `cpu_possible_mask` and not in the `tick_nohz_full_mask`:

```
cpumask_andnot(housekeeping_mask,
           cpu_possible_mask, tick_nohz_full_mask);
```

After this operation, the `housekeeping_mask` will contain all processors of the system except a processor for timekeeping. In the last step of the `tick_nohz_init_all` function, we are going through all processors that are defined in the `tick_nohz_full_mask` and call the following function for an each processor:

```
for_each_cpu(cpu, tick_nohz_full_mask)
    context_tracking_cpu_set(cpu);
```

The `context_tracking_cpu_set` function defined in the kernel/context_tracking.c source code file and main point of this function is to set the `context_tracking.active` percpu variable to `true`. When the `active` field will be set to `true` for the certain processor, all context switches will be ignored by the Linux kernel context tracking subsystem for this processor.

That's all. This is the end of the `tick_nohz_init` function. After this `NO_HZ` related data structures will be initialzed. We didn't see API of the `NO_HZ` mode, but will see it soon.

Conclusion

This is the end of the third part of the chapter that describes timers and timer management related stuff in the Linux kernel. In the previous part got acquainted with the `clocksource` concept in the Linux kernel which represents framework for managing different clock source in a interrupt and hardware characteristics independent way. We continued to look on the Linux kernel initialization process in a time management context in this part and got acquainted with two new concepts for us: the `tick broadcast` framework and `tick-less` mode. The first concept helps the Linux kernel to deal with processors which are in deep sleep and the second concept represents the mode in which kernel may work to improve power management of `idle` processors.

In the next part we will continue to dive into timer management related things in the Linux kernel and will see new concept for us - `timers`.

If you have questions or suggestions, feel free to ping me in twitter 0xAX, drop me email or just create issue.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-insides.

Links

- x86_64
- initrd
- interrupt
- DMI
- printk
- CPU idle
- power management
- NO_HZ documentation
- cpumasks
- high precision event timer
- irq
- IPI
- CPUID
- APIC
- percpu
- context switches
- Previous part

Timers and time management in the Linux kernel. Part 4.

Timers

This is fourth part of the chapter which describes timers and time management related stuff in the Linux kernel and in the previous part we knew about the `tick broadcast` framework and `NO_HZ` mode in the Linux kernel. We will continue to dive into the time management related stuff in the Linux kernel in this part and will be acquainted with yet another concept in the Linux kernel - `timers`. Before we will look at timers in the Linux kernel, we have to learn some theory about this concept. Note that we will consider software timers in this part.

The Linux kernel provides a `software timer` concept to allow to kernel functions could be invoked at future moment. Timers are widely used in the Linux kernel. For example, look in the net/netfilter/ipset/ip_set_list_set.c source code file. This source code file provides implementation of the framework for the managing of groups of IP addresses.

We can find the `list_set` structure that contains `gc` filed in this source code file:

```
struct list_set {
    ...
    struct timer_list gc;
    ...
};
```

Not that the `gc` filed has `timer_list` type. This structure defined in the include/linux/timer.h header file and main point of this structure is to store `dynamic` timers in the Linux kernel. Actually, the Linux kernel provides two types of timers called dynamic timers and interval timers. First type of timers is used by the kernel, and the second can be used by user mode. The `timer_list` structure contains actual `dynamic` timers. The `list_set` contains `gc` timer in our example represents timer for garbage collection. This timer will be initialized in the `list_set_gc_init` function:

```
static void
list_set_gc_init(struct ip_set *set, void (*gc)(unsigned long ul_set))
{
    struct list_set *map = set->data;
    ...
    ...
    ...
    map->gc.function = gc;
    map->gc.expires = jiffies + IPSET_GC_PERIOD(set->timeout) * HZ;
    ...
    ...
    ...
}
```

A function that is pointed by the `gc` pointer, will be called after timeout which is equal to the `map->gc.expires`.

Ok, we will not dive into this example with the netfilter, because this chapter is not about network related stuff. But we saw that timers are widely used in the Linux kernel and learned that they represent concept which allows to functions to be called in future.

Now let's continue to research source code of Linux kernel which is related to the timers and time management stuff as we did it in all previous chapters.

Introduction to dynamic timers in the Linux kernel

As I already wrote, we knew about the `tick broadcast` framework and `NO_HZ` mode in the previous part. They will be initialized in the init/main.c source code file by the call of the `tick_init` function. If we will look at this source code file, we will see that the next time management related function is:

```
init_timers();
```

This function defined in the kernel/time/timer.c source code file and contains calls of four functions:

```
void __init init_timers(void)
{
    init_timer_cpus();
    init_timer_stats();
    timer_register_cpu_notifier();
    open_softirq(TIMER_SOFTIRQ, run_timer_softirq);
}
```

Let's look on implementation of each function. The first function is `init_timer_cpus` defined in the same source code file and just calls the `init_timer_cpu` function for each possible processor in the system:

```
static void __init init_timer_cpus(void)
{
    int cpu;

    for_each_possible_cpu(cpu)
        init_timer_cpu(cpu);
}
```

If you do not know or do not remember what is it a `possible` cpu, you can read the special part of this book which describes `cpumask` concept in the Linux kernel. In short words, a `possible` processor is a processor which can be plugged in anytime during the life of the system.

The `init_timer_cpu` function does main work for us, namely it executes initialization of the `tvec_base` structure for each processor. This structure defined in the kernel/time/timer.c source code file and stores data related to a `dynamic` timer for a certain processor. Let's look on the definition of this structure:

```
struct tvec_base {
    spinlock_t lock;
    struct timer_list *running_timer;
    unsigned long timer_jiffies;
    unsigned long next_timer;
    unsigned long active_timers;
    unsigned long all_timers;
    int cpu;
    bool migration_enabled;
    bool nohz_active;
    struct tvec_root tv1;
    struct tvec tv2;
    struct tvec tv3;
    struct tvec tv4;
    struct tvec tv5;
} ____cacheline_aligned;
```

The `thec_base` structure contains following fields: The `lock` for `tvec_base` protection, the next `running_timer` field points to the currently running timer for the certain processor, the `timer_jiffies` fields represents the earliest expiration time (it will be used by the Linux kernel to find already expired timers). The next field - `next_timer` contains the next pending timer for a next timer interrupt in a case when a processor goes to sleep and the `NO_HZ` mode is enabled in the Linux kernel. The `active_timers` field provides accounting of non-deferrable timers or in other words all timers that will not be stopped during a processor will go to sleep. The `all_timers` field tracks total number of timers or `active_timers` + deferrable timers. The `cpu` field represents number of a processor which owns timers. The `migration_enabled` and `nohz_active` fields are represent opportunity of timers migration to another processor and status of the `NO_HZ` mode respectively.

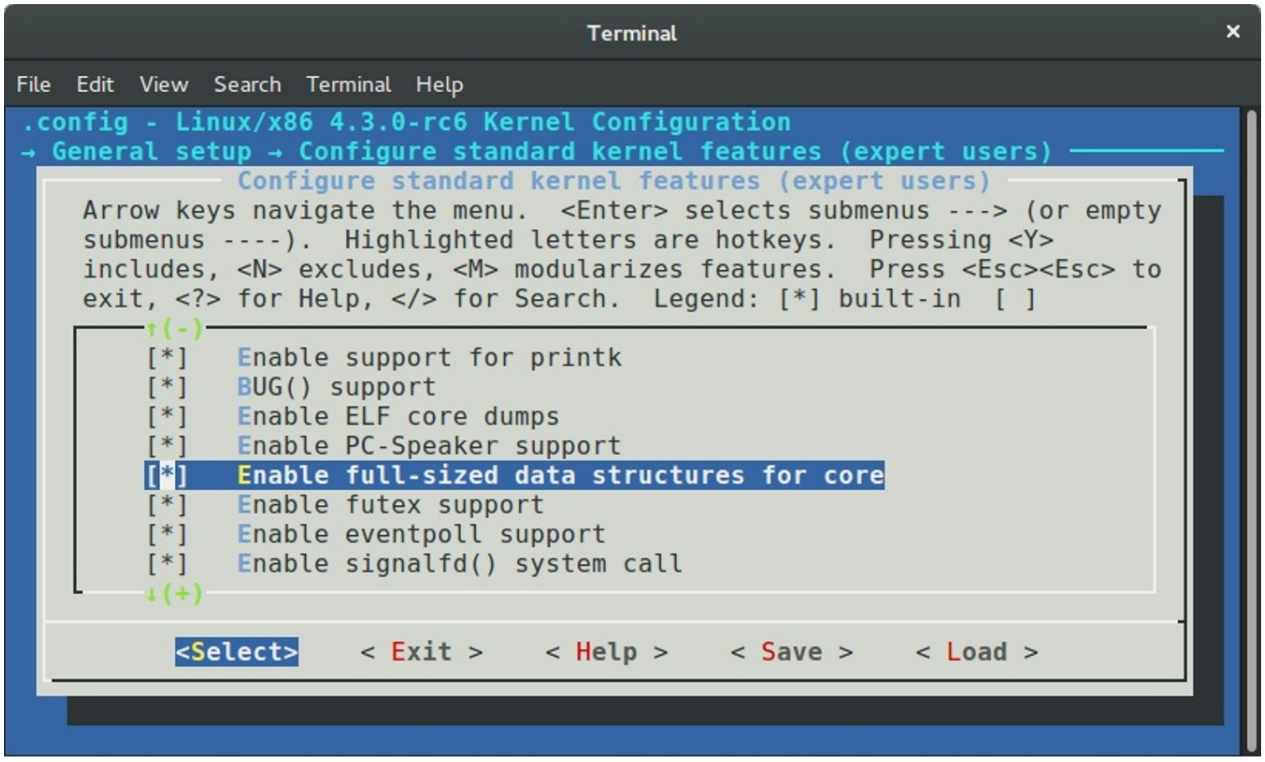
The last five fields of the `tvec_base` structure represent lists of dynamic timers. The first `tv1` field has:

```
#define TVR_SIZE (1 << TVR_BITS)
#define TVR_BITS (CONFIG_BASE_SMALL ? 6 : 8)
```

```
...
...
...

struct tvec_root {
    struct hlist_head vec[TVR_SIZE];
};
```

type. Note that the value of the `TVR_SIZE` depends on the `CONFIG_BASE_SMALL` kernel configuration option:



that reduces size of the kernel data structures if disabled. The `v1` is array that may contain `64` or `256` elements where an each element represents a dynamic timer that will decay within the next `255` system timer interrupts. Next three fields: `tv2`, `tv3` and `tv4` are lists with dynamic timers too, but they store dynamic timers which will decay the next `2^14 - 1`, `2^20 - 1` and `2^26` respectively. The last `tv5` field represents list which stores dynamic timers with a large expiring period.

So, now we saw the `tvec_base` structure and description of its fields and we can look on the implementation of the `init_timer_cpu` function. As I already wrote, this function defined in the kernel/time/timer.c source code file and executes initialization of the `tvec_bases`:

```
static void __init init_timer_cpu(int cpu)
{
    struct tvec_base *base = per_cpu_ptr(&tvec_bases, cpu);

    base->cpu = cpu;
    spin_lock_init(&base->lock);

    base->timer_jiffies = jiffies;
    base->next_timer = base->timer_jiffies;
}
```

The `tvec_bases` represents per-cpu variable which represents main data structure for a dynamic timer for a given processor. This `per-cpu` variable defined in the same source code file:

```
static DEFINE_PER_CPU(struct tvec_base, tvec_bases);
```

First of all we're getting the address of the `tvec_bases` for the given processor to `base` variable and as we got it, we are starting to initialize some of the `tvec_base` fields in the `init_timer_cpu` function. After initialization of the `per-cpu` dynamic timers with the jiffies and the number of a possible processor, we need to initialize a `tstats_lookup_lock` spinlock in the `init_timer_stats` function:

```
void __init init_timer_stats(void)
{
    int cpu;

    for_each_possible_cpu(cpu)
        raw_spin_lock_init(&per_cpu(tstats_lookup_lock, cpu));
}
```

The `tstats_lookcup_lock` variable represents `per-cpu` raw spinlock:

```
static DEFINE_PER_CPU(raw_spinlock_t, tstats_lookup_lock);
```

which will be used for protection of operation with statistics of timers that can be accessed through the procfs:

```
static int __init init_tstats_procfs(void)
{
    struct proc_dir_entry *pe;

    pe = proc_create("timer_stats", 0644, NULL, &tstats_fops);
    if (!pe)
        return -ENOMEM;
    return 0;
}
```

For example:

```
$ cat /proc/timer_stats
Timerstats sample period: 3.888770 s
  12,     0 swapper          hrtimer_stop_sched_tick (hrtimer_sched_tick)
  15,     1 swapper          hcd_submit_urb (rh_timer_func)
   4,   959 kedac            schedule_timeout (process_timeout)
   1,     0 swapper          page_writeback_init (wb_timer_fn)
  28,     0 swapper          hrtimer_stop_sched_tick (hrtimer_sched_tick)
  22,  2948 IRQ 4            tty_flip_buffer_push (delayed_work_timer_fn)
  ...
  ...
  ...
```

The next step after initialization of the `tstats_lookup_lock` spinlock is the call of the `timer_register_cpu_notifier` function. This function depends on the `CONFIG_HOTPLUG_CPU` kernel configuration option which enables support for hotplug processors in the Linux kernel.

When a processor will be logically offlined, a notification will be sent to the Linux kernel with the `CPU_DEAD` or the `CPU_DEAD_FROZEN` event by the call of the `cpu_notifier` macro:

```
#ifdef CONFIG_HOTPLUG_CPU
...
...
static inline void timer_register_cpu_notifier(void)
{
    cpu_notifier(timer_cpu_notify, 0);
}
...
...
#else
...
...
static inline void timer_register_cpu_notifier(void) { }
```

```
...
...
#endif /* CONFIG_HOTPLUG_CPU */
```

In this case the `timer_cpu_notify` will be called which checks an event type and will call the `migrate_timers` function:

```
static int timer_cpu_notify(struct notifier_block *self,
                            unsigned long action, void *hcpu)
{
    switch (action) {
    case CPU_DEAD:
    case CPU_DEAD_FROZEN:
        migrate_timers((long)hcpu);
        break;
    default:
        break;
    }

    return NOTIFY_OK;
}
```

This chapter will not describe `hotplug` related events in the Linux kernel source code, but if you are interesting in such things, you can find implementation of the `migrate_timers` function in the kernel/time/timer.c source code file.

The last step in the `init_timers` function is the call of the:

```
open_softirq(TIMER_SOFTIRQ, run_timer_softirq);
```

function. The `open_softirq` function may be already familar to you if you have read the ninth part about the interrupts and interrupt handling in the Linux kernel. In short words, the `open_softirq` function defined in the kernel/softirq.c source code file and executes initialization of the deferred interrupt handler.

In our case the deferred function is the `run_timer_softirq` function that is will be called after a hardware interrupt in the `do_IRQ` function which defined in the arch/x86/kernel/irq.c source code file. The main point of this function is to handle a software dynamic timer. The Linux kernel does not do this thing during the hardware timer interrupt handling because this is time consuming operation.

Let's look on the implementation of the `run_timer_softirq` function:

```
static void run_timer_softirq(struct softirq_action *h)
{
    struct tvec_base *base = this_cpu_ptr(&tvec_bases);

    if (time_after_eq(jiffies, base->timer_jiffies))
        __run_timers(base);
}
```

At the beginning of the `run_timer_softirq` function we get a `dynamic` timer for a current processor and compares the current value of the jiffies with the value of the `timer_jiffies` for the current structure by the call of the `time_after_eq` macro which is defined in the include/linux/jiffies.h header file:

```
#define time_after_eq(a,b)          \
    (typecheck(unsigned long, a) && \
     typecheck(unsigned long, b) && \
    ((long)((a) - (b)) >= 0))
```

Reclaim that the `timer_jiffies` field of the `tvec_base` structure represents the relative time when functions delayed by the given timer will be executed. So we compare these two values and if the current time represented by the `jiffies` is greater than `base->timer_jiffies`, we call the `__run_timers` function that defined in the same source code file. Let's look on the implementation of this function.

As I just wrote, the `__run_timers` function runs all expired timers for a given processor. This function starts from the acquiring of the `tvec_base's` lock to protect the `tvec_base` structure

```
static inline void __run_timers(struct tvec_base *base)
{
    struct timer_list *timer;

    spin_lock_irq(&base->lock);
    ...
    ...
    ...
    spin_unlock_irq(&base->lock);
}
```

After this it starts the loop while the `timer_jiffies` will not be greater than the `jiffies`:

```
while (time_after_eq(jiffies, base->timer_jiffies)) {
    ...
    ...
    ...
}
```

We can find many different manipulations in the our loop, but the main point is to find expired timers and call delayed functions. First of all we need to calculate the `index` of the `base->tv1` list that stores the next timer to be handled with the following expression:

```
index = base->timer_jiffies & TVR_MASK;
```

where the `TVR_MASK` is a mask for the getting of the `tvec_root->vec` elements. As we got the index with the next timer which must be handled we check its value. If the index is zero, we go through all lists in our cascade table `tv2`, `tv3` and etc., and rehashing it with the call of the `cascade` function:

```
if (!index &&
    (!cascade(base, &base->tv2, INDEX(0))) &&
        (!cascade(base, &base->tv3, INDEX(1))) &&
                !cascade(base, &base->tv4, INDEX(2)))
        cascade(base, &base->tv5, INDEX(3));
```

After this we increase the value of the `base->timer_jiffies`:

```
++base->timer_jiffies;
```

In the last step we are executing a corresponding function for each timer from the list in a following loop:

```
hlist_move_list(base->tv1.vec + index, head);

while (!hlist_empty(head)) {
    ...
    ...
    ...
    timer = hlist_entry(head->first, struct timer_list, entry);
    fn = timer->function;
    data = timer->data;

    spin_unlock(&base->lock);
    call_timer_fn(timer, fn, data);
    spin_lock(&base->lock);

    ...
    ...
    ...
}
```

where the `call_timer_fn` just call the given function:

```
static void call_timer_fn(struct timer_list *timer, void (*fn)(unsigned long),
                          unsigned long data)
{
    ...
    ...
    ...
    fn(data);
    ...
    ...
    ...
}
```

That's all. The Linux kernel has infrastructure for `dynamic timers` from this moment. We will not dive into this interesting theme. As I already wrote the `timers` is a widely used concept in the Linux kernel and nor one part, nor two parts will not cover understanding of such things how it implemented and how it works. But now we know about this concept, why does the Linux kernel needs in it and some data structures around it.

Now let's look usage of `dynamic timers` in the Linux kernel.

Usage of dynamic timers

As you already can noted, if the Linux kernel provides a concept, it also provides API for managing of this concept and the `dynamic timers` concept is not exception here. To use a timer in the Linux kernel code, we must define a variable with a `timer_list` type. We can initialize our `timer_list` structure in two ways. The first is to use the `init_timer` macro that defined in the include/linux/timer.h header file:

```
#define init_timer(timer)    \
    __init_timer((timer), 0)

#define __init_timer(_timer, _flags)   \
         init_timer_key((_timer), (_flags), NULL, NULL)
```

where the `init_timer_key` function just calls the:

```
do_init_timer(timer, flags, name, key);
```

function which fields the given `timer` with default values. The second way is to use the:

```
#define TIMER_INITIALIZER(_function, _expires, _data)        \
    __TIMER_INITIALIZER((_function), (_expires), (_data), 0)
```

macro which will initilize the given `timer_list` structure too.

After a `dynamic timer` is initialzed we can start this `timer` with the call of the:

```
void add_timer(struct timer_list * timer);
```

function and stop it with the:

```
int del_timer(struct timer_list * timer);
```

function.

That's all.

Conclusion

This is the end of the fourth part of the chapter that describes timers and timer management related stuff in the Linux kernel. In the previous part we got acquainted with the two new concepts: the `tick broadcast` framework and the `NO_HZ` mode. In this part we continued to dive into time managemented related stuff and got acquainted with the new concept - `dynamic timer` or software timer. We didn't saw implementation of a `dynamic timers` management code in details in this part but saw data structures and API around this concept.

In the next part we will continue to dive into timer management related things in the Linux kernel and will see new concept for us - `timers`.

If you have questions or suggestions, feel free to ping me in twitter 0xAX, drop me email or just create issue.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-insides.

Links

- IP
- netfilter
- network
- cpumask
- interrupt
- jiffies
- per-cpu
- spinlock
- procfs
- previous part

Timers and time management in the Linux kernel. Part 5.

Introduction to the `clockevents` framework

This is fifth part of the chapter which describes timers and time management related stuff in the Linux kernel. As you might noted from the title of this part, the `clockevents` framework will be discussed. We already saw one framework in the second part of this chapter. It was `clocksource` framework. Both of these frameworks represent timekeeping abstractions in the Linux kernel.

At first let's refresh your memory and try to remember what is it `clocksource` framework and and what its purpose. The main goal of the `clocksource` framework is to provide `timeline`. As described in the documentation:

> For example issuing the command 'date' on a Linux system will eventually read the clock source to determine exactly what time it is.

The Linux kernel supports many different clock sources. You can find some of them in the drivers/clocksource. For example old good Intel 8253 - programmable interval timer with `1193182` Hz frequency, yet another one - ACPI PM timer with `3579545` Hz frequency. Besides the drivers/clocksource directory, each architecture may provide own architecture-specific clock sources. For example x86 architecture provides High Precision Event Timer, or for example powerpc provides access to the processor timer through `timebase` register.

Each clock source provides monotonic atomic counter. As I already wrote, the Linux kernel supports a huge set of different clock source and each clock source has own parameters like frequency. The main goal of the `clocksource` framework is to provide API to select best available clock source in the system i.e. a clock source with the highest frequency. Additional goal of the `clocksource` framework is to represent an atomic counter provided by a clock source in human units. In this time, nanoseconds are the favorite choice for the time value units of the given clock source in the Linux kernel.

The `clocksource` framework represented by the `clocksource` structure which is defined in the include/linux/clocksource.h header code file which contains `name` of a clock source, rating of certain clock source in the system (a clock source with the higher frequency has the biggest rating in the system), `list` of all registered clock source in the system, `enable` and `disable` fields to enable and disable a clock source, pointer to the `read` function which must return an atomic counter of a clock source and etc.

Additionally the `clocksource` structure provides two fields: `mult` and `shift` which are needed for translation of an atomic counter which is provided by a certain clock source to the human units, i.e. nanoseconds. Translation occurs via following formula:

```
ns ~= (clocksource * mult) >> shift
```

As we already know, besides the `clocksource` structure, the `clocksource` framework provides an API for registration of clock source with different frequency scale factor:

```
static inline int clocksource_register_hz(struct clocksource *cs, u32 hz)
static inline int clocksource_register_khz(struct clocksource *cs, u32 khz)
```

A clock source unregistration:

```
int clocksource_unregister(struct clocksource *cs)
```

and etc.

Additionally to the `clocksource` framework, the Linux kernel provides `clockevents` framework. As described in the documentation:

> Clock events are the conceptual reverse of clock sources

Main goal of the is to manage clock event devices or in other words - to manage devices that allow to register an event or in other words interrupt that is going to happen at a defined point of time in the future.

Now we know a little about the `clockevents` framework in the Linux kernel, and now time is to see on it API.

API of `clockevents` framework

The main structure which described a clock event device is `clock_event_device` structure. This structure is defined in the include/linux/clockchips.h header file and contains a huge set of fields. as well as the `clocksource` structure it has `name` fields which contains human readable name of a clock event device, for example local APIC timer:

```
static struct clock_event_device lapic_clockevent = {
    .name                   = "lapic",
    ...
    ...
    ...
}
```

Addresses of the `event_handler`, `set_next_event`, `next_event` functions for a certain clock event device which are an interrupt handler, setter of next event and local storage for next event respectively. Yet another field of the `clock_event_device` structure is - `features` field. Its value maybe on of the following generic features:

```
#define CLOCK_EVT_FEAT_PERIODIC    0x000001
#define CLOCK_EVT_FEAT_ONESHOT        0x000002
```

Where the `CLOCK_EVT_FEAT_PERIODIC` represents device which may be programmed to generate events periodically. The `CLOCK_EVT_FEAT_ONESHOT` represents device which may generate an event only once. Besides these two features, there are also architecture-specific features. For example x86_64 supports two additional features:

```
#define CLOCK_EVT_FEAT_C3STOP        0x000008
```

The first `CLOCK_EVT_FEAT_C3STOP` means that a clock event device will be stopped in the C3 state. Additionally the `clock_event_device` structure has `mult` and `shift` fields as well as `clocksource` structure. The `clocksource` structure also contains other fields, but we will consider it later.

After we considered part of the `clock_event_device` structure, time is to look at the `API` of the `clockevents` framework. To work with a clock event device, first of all we need to initialize `clock_event_device` structure and register a clock events device. The `clockevents` framework provides following `API` for registration of clock event devices:

```
void clockevents_register_device(struct clock_event_device *dev)
{
   ...
   ...
   ...
}
```

This function defined in the kernel/time/clockevents.c source code file and as we may see, the `clockevents_register_device` function takes only one parameter:

- address of a `clock_event_device` structure which represents a clock event device.

So, to register a clock event device, at first we need to initialize `clock_event_device` structure with parameters of a certain clock event device. Let's take a look at one random clock event device in the Linux kernel source code. We can find one in the drivers/closksource directory or try to take a look at an architecture-specific clock event device. Let's take for example - Periodic Interval Timer (PIT) for at91sam926x. You can find its implementation in the drivers/closksource.

First of all let's look at initialization of the `clock_event_device` structure. This occurs in the `at91sam926x_pit_common_init` function:

```
struct pit_data {
    ...
    ...
```

```
    struct clock_event_device       clkevt;
    ...
    ...
};

static void __init at91sam926x_pit_common_init(struct pit_data *data)
{
    ...
    ...
    ...
    data->clkevt.name = "pit";
    data->clkevt.features = CLOCK_EVT_FEAT_PERIODIC;
    data->clkevt.shift = 32;
    data->clkevt.mult = div_sc(pit_rate, NSEC_PER_SEC, data->clkevt.shift);
    data->clkevt.rating = 100;
    data->clkevt.cpumask = cpumask_of(0);

    data->clkevt.set_state_shutdown = pit_clkevt_shutdown;
    data->clkevt.set_state_periodic = pit_clkevt_set_periodic;
    data->clkevt.resume = at91sam926x_pit_resume;
    data->clkevt.suspend = at91sam926x_pit_suspend;
    ...
}
```

Here we can see that `at91sam926x_pit_common_init` takes one parameter - pointer to the `pit_data` structure which contains `clock_event_device` structure which will contain clock event related information of the `at91sam926x` periodic Interval Timer. At the start we fill `name` of the timer device and its `features`. In our case we deal with periodic timer which as we already know may be programmed to generate events periodically.

The next two fields `shift` and `mult` are familiar to us. They will be used to translate counter of our timer to nanoseconds. After this we set rating of the timer to `100`. This means if there will not be timers with higher rating in the system, this timer will be used for timekeeping. The next field - `cpumask` indicates for which processors in the system the device will work. In our case, the device will work for the first processor. The `cpumask_of` macro defined in the include/linux/cpumask.h header file and just expands to the call of the:

```
#define cpumask_of(cpu) (get_cpu_mask(cpu))
```

Where the `get_cpu_mask` returns the cpumask containing just a given `cpu` number. More about `cpumasks` concept you may read in the CPU masks in the Linux kernel part. In the last four lines of code we set callbacks for the clock event device suspend/resume, device shutdown and update of the clock event device state.

After we finished with the initialization of the `at91sam926x` periodic timer, we can register it by the call of the following functions:

```
clockevents_register_device(&data->clkevt);
```

Now we can consider implementation of the `clockevent_register_device` function. As I already wrote above, this function is defined in the kernel/time/clockevents.c source code file and starts from the initialization of the initial event device state:

```
clockevent_set_state(dev, CLOCK_EVT_STATE_DETACHED);
```

Actually, an event device may be in one of this states:

```
enum clock_event_state {
    CLOCK_EVT_STATE_DETACHED,
    CLOCK_EVT_STATE_SHUTDOWN,
    CLOCK_EVT_STATE_PERIODIC,
    CLOCK_EVT_STATE_ONESHOT,
    CLOCK_EVT_STATE_ONESHOT_STOPPED,
};
```

Where:

- `CLOCK_EVT_STATE_DETACHED` - a clock event device is not not used by `clockevents` framework. Actually it is initial state of all clock event devices;
- `CLOCK_EVT_STATE_SHUTDOWN` - a clock event device is powered-off;
- `CLOCK_EVT_STATE_PERIODIC` - a clock event device may be programmed to generate event periodically;
- `CLOCK_EVT_STATE_ONESHOT` - a clock event device may be programmed to generate event only once;
- `CLOCK_EVT_STATE_ONESHOT_STOPPED` - a clock event device was programmed to generate event only once and now it is temporary stopped.

The implementation of the `clock_event_set_state` function is pretty easy:

```
static inline void clockevent_set_state(struct clock_event_device *dev,
                    enum clock_event_state state)
{
    dev->state_use_accessors = state;
}
```

As we can see, it just fills the `state_use_accessors` field of the given `clock_event_device` structure with the given value which is in our case is `CLOCK_EVT_STATE_DETACHED`. Actually all clock event devices has this initial state during registration. The `state_use_accessors` field of the `clock_event_device` structure provides `current` state of the clock event device.

After we have set initial state of the given `clock_event_device` structure we check that the `cpumask` of the given clock event device is not zero:

```
if (!dev->cpumask) {
    WARN_ON(num_possible_cpus() > 1);
    dev->cpumask = cpumask_of(smp_processor_id());
}
```

Remember that we have set the `cpumask` of the `at91sam926x` periodic timer to first processor. If the `cpumask` field is zero, we check the number of possible processors in the system and print warning message if it is less than on. Additionally we set the `cpumask` of the given clock event device to the current processor. If you are interested in how the `smp_processor_id` macro is implemented, you can read more about it in the fourth part of the Linux kernel initialization process chapter.

After this check we lock the actual code of the clock event device registration by the call following macros:

```
raw_spin_lock_irqsave(&clockevents_lock, flags);
...
...
...
raw_spin_unlock_irqrestore(&clockevents_lock, flags);
```

Additionally the `raw_spin_lock_irqsave` and the `raw_spin_unlock_irqrestore` macros disable local interrupts, however interrupts on other processors still may occur. We need to do it to prevent potential deadlock if we adding new clock event device to the list of clock event devices and an interrupt occurs from other clock event device.

We can see following code of clock event device registration between the `raw_spin_lock_irqsave` and `raw_spin_unlock_irqrestore` macros:

```
list_add(&dev->list, &clockevent_devices);
tick_check_new_device(dev);
clockevents_notify_released();
```

First of all we add the given clock event device to the list of clock event devices which is represented by the `clockevent_devices`:

```
static LIST_HEAD(clockevent_devices);
```

At the next step we call the `tick_check_new_device` function which is defined in the kernel/time/tick-common.c source code file and checks do the new registered clock event device should be used or not. The `tick_check_new_device` function checks the given `clock_event_device` gets the current registered tick device which is represented by the `tick_device` structure and compares their ratings and features. Actually `CLOCK_EVT_STATE_ONESHOT` is preferred:

```
static bool tick_check_preferred(struct clock_event_device *curdev,
                 struct clock_event_device *newdev)
{
    if (!(newdev->features & CLOCK_EVT_FEAT_ONESHOT)) {
        if (curdev && (curdev->features & CLOCK_EVT_FEAT_ONESHOT))
            return false;
        if (tick_oneshot_mode_active())
            return false;
    }

    return !curdev ||
        newdev->rating > curdev->rating ||
           !cpumask_equal(curdev->cpumask, newdev->cpumask);
}
```

If the new registered clock event device is more preferred than old tick device, we exchange old and new registered devices and install new device:

```
clockevents_exchange_device(curdev, newdev);
tick_setup_device(td, newdev, cpu, cpumask_of(cpu));
```

The `clockevents_exchange_device` function releases or in other words deleted the old clock event device from the `clockevent_devices` list. The next function - `tick_setup_device` as we may understand from its name, setups new tick device. This function check the mode of the new registered clock event device and call the `tick_setup_periodic` function or the `tick_setup_oneshot` depends on the tick device mode:

```
if (td->mode == TICKDEV_MODE_PERIODIC)
    tick_setup_periodic(newdev, 0);
else
    tick_setup_oneshot(newdev, handler, next_event);
```

Both of this functions calls the `clockevents_switch_state` to change state of the clock event device and the `clockevents_program_event` function to set next event of clock event device based on delta between the maximum and minimum difference current time and time for the next event. The `tick_setup_periodic` :

```
clockevents_switch_state(dev, CLOCK_EVT_STATE_PERIODIC);
clockevents_program_event(dev, next, false))
```

and the `tick_setup_oneshot_periodic` :

```
clockevents_switch_state(newdev, CLOCK_EVT_STATE_ONESHOT);
clockevents_program_event(newdev, next_event, true);
```

The `clockevents_switch_state` function checks that the clock event device is not in the given state and calls the `__clockevents_switch_state` function from the same source code file:

```
if (clockevent_get_state(dev) != state) {
    if (__clockevents_switch_state(dev, state))
        return;
```

The `__clockevents_switch_state` function just makes a call of the certain callback depends on the given state:

```
static int __clockevents_switch_state(struct clock_event_device *dev,
                      enum clock_event_state state)
```

```
{
    if (dev->features & CLOCK_EVT_FEAT_DUMMY)
        return 0;

    switch (state) {
    case CLOCK_EVT_STATE_DETACHED:
    case CLOCK_EVT_STATE_SHUTDOWN:
        if (dev->set_state_shutdown)
            return dev->set_state_shutdown(dev);
        return 0;

    case CLOCK_EVT_STATE_PERIODIC:
        if (!(dev->features & CLOCK_EVT_FEAT_PERIODIC))
            return -ENOSYS;
        if (dev->set_state_periodic)
            return dev->set_state_periodic(dev);
        return 0;
    ...
    ...
    ...
```

In our case for `at91sam926x` periodic timer, the state is the `CLOCK_EVT_FEAT_PERIODIC`:

```
data->clkevt.features = CLOCK_EVT_FEAT_PERIODIC;
data->clkevt.set_state_periodic = pit_clkevt_set_periodic;
```

So, for the `pit_clkevt_set_periodic` callback will be called. If we will read the documentation of the Periodic Interval Timer (PIT) for at91sam926x, we will see that there is `Periodic Interval Timer Mode Register` which allows us to control of periodic interval timer.

It looks like:

```
31                                                  25        24
+-------------------------------------------------------------+
|                                        |  PITIEN  |  PITEN  |
+-------------------------------------------------------------+
23                            19                              16
+-------------------------------------------------------------+
|                              |              PIV               |
+-------------------------------------------------------------+
15                                                            8
+-------------------------------------------------------------+
|                            PIV                              |
+-------------------------------------------------------------+
7                                                             0
+-------------------------------------------------------------+
|                            PIV                              |
+-------------------------------------------------------------+
```

Where `PIV` or `Periodic Interval Value` - defines the value compared with the primary `20-bit` counter of the Periodic Interval Timer. The `PITEN` or `Period Interval Timer Enabled` if the bit is `1` and the `PITIEN` or `Periodic Interval Timer Interrupt Enable` if the bit is `1`. So, to set periodic mode, we need to set `24`, `25` bits in the `Periodic Interval Timer Mode Register`. And we are doing it in the `pit_clkevt_set_periodic` function:

```
static int pit_clkevt_set_periodic(struct clock_event_device *dev)
{
        struct pit_data *data = clkevt_to_pit_data(dev);
        ...
        ...
        ...
        pit_write(data->base, AT91_PIT_MR,
                  (data->cycle - 1) | AT91_PIT_PITEN | AT91_PIT_PITIEN);

        return 0;
}
```

Where the `AT91_PT_MR` , `AT91_PT_PITEN` and the `AT91_PIT_PITIEN` are declared as:

```
#define AT91_PIT_MR             0x00
#define AT91_PIT_PITIEN       BIT(25)
#define AT91_PIT_PITEN        BIT(24)
```

After the setup of the new clock event device is finished, we can return to the `clockevents_register_device` function. The last function in the `clockevents_register_device` function is:

```
clockevents_notify_released();
```

This function checks the `clockevents_released` list which contains released clock event devices (remember that they may occur after the call of the `clockevents_exchange_device` function). If this list is not empty, we go through clock event devices from the `clock_events_released` list and delete it from the `clockevent_devices` :

```
static void clockevents_notify_released(void)
{
    struct clock_event_device *dev;

    while (!list_empty(&clockevents_released)) {
        dev = list_entry(clockevents_released.next,
                 struct clock_event_device, list);
        list_del(&dev->list);
        list_add(&dev->list, &clockevent_devices);
        tick_check_new_device(dev);
    }
}
```

That's all. From this moment we have registered new clock event device. So the usage of the `clockevents` framework is simple and clear. Architectures registered their clock event devices, in the clock events core. Users of the clockevents core can get clock event devices for their use. The `clockevents` framework provides notification mechanisms for various clock related management events like a clock event device registered or unregistered, a processor is offlined in system which supports CPU hotplug and etc.

We saw implementation only of the `clockevents_register_device` function. But generally, the clock event layer API is small. Besides the `API` for clock event device registration, the `clockevents` framework provides functions to schedule the next event interrupt, clock event device notification service and support for suspend and resume for clock event devices.

If you want to know more about `clockevents` API you can start to research following source code and header files: kernel/time/tick-common.c, kernel/time/clockevents.c and include/linux/clockchips.h.

That's all.

Conclusion

This is the end of the fifth part of the chapter that describes timers and timer management related stuff in the Linux kernel. In the previous part got acquainted with the `timers` concept. In this part we continued to learn time management related stuff in the Linux kernel and saw a little about yet another framework - `clockevents` .

If you have questions or suggestions, feel free to ping me in twitter 0xAX, drop me email or just create issue.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-insides.

Links

- timekeeping documentation
- Intel 8253

- programmable interval timer
- ACPI pdf
- x86
- High Precision Event Timer
- powerpc
- frequency
- API
- nanoseconds
- interrupt
- interrupt handler
- local APIC
- C3 state
- Periodic Interval Timer (PIT) for at91sam926x
- CPU masks in the Linux kernel
- deadlock
- CPU hotplug
- previous part

Linux 6

x86_64

LinuxclockeventsLinuxx86

x86 sysfs `/sys/devices/system/clocksource/clocksource0/available_clocksource` `/sys/devices/system/clocksource/clocksourceN`

- `available_clocksource` -
- `current_clocksource` -

```
$ cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
```

- `tsc` - Time Stamp Counter;
- `hpet` - High Precision Event Timer;
- `acpi_pm` - ACPI Power Management Timer.

:

```
$ cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
```

Time Stamp Counter

ACPI3.579545MHz High Precision Event Timer() 10MHz Time Stamp Counter() `TSC` `/proc/cpuinfo`

```
$ cat /proc/cpuinfo
...
model name    : Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz
...
```

`TSC` TSC `ACPI PM HPET`

`/sys/devices/system/clocksource/clocksource0/available_clocksource` `jiffy refined_jiffies` CLOCK_SOURCE_VALID_FOR_HRES

- `hpet`
- `acpi_pm`
- `tsc`

dmesg

```
$ dmesg | grep clocksource
[    0.000000] clocksource: refined-jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns: 1910969940391419 n
s
[    0.000000] clocksource: hpet: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns: 133484882848 ns
[    0.094369] clocksource: jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns: 1911260446275000 ns
[    0.186498] clocksource: Switched to clocksource hpet
[    0.196827] clocksource: acpi_pm: mask: 0xffffff max_cycles: 0xffffff, max_idle_ns: 2085701024 ns
```

```
[    1.413685] tsc: Refined TSC clocksource calibration: 3999.981 MHz
[    1.413688] clocksource: tsc: mask: 0xffffffffffffffff max_cycles: 0x73509721780, max_idle_ns: 881591102108 ns
[    2.413748] clocksource: Switched to clocksource tsc
```

High Precision Event Timer

High Precision Event Timer

x86HPETarch/x86/kernel/hpet.c `hpet_enable` Linux init/main.c `start_kernel` "'early console'

```
if (late_time_init)
    late_time_init();
```

jiffy `x86 late_time_init` arch/x86/kernel/time.c

```
static __init void x86_late_time_init(void)
{
    x86_init.timers.timer_init();
    tsc_init();
}
```

`x86 TSC` `x86_init.timers.timer_init timer_init hpet_time_init` `x86_init` arch/x86/kernel/x86_init.c:

```
struct x86_init_ops x86_init __initdata = {
   ...
   ...
   ...
   .timers = {
        .setup_percpu_clockev    = setup_boot_APIC_clock,
        .timer_init        = hpet_time_init,
        .wallclock_init        = x86_init_noop,
   },
   ...
   ...
   ...
```

`HPET` `hpet_time_init` programmable interval timerIRQ:

```
void __init hpet_time_init(void)
{
    if (!hpet_enable())
        setup_pit_timer();
    setup_default_timer_irq();
}
```

`hpet_enable is_hpet_capable'` HPET`

```
int __init hpet_enable(void)
{
    if (!is_hpet_capable())
        return 0;

    hpet_set_mapping();
}
```

`is_hpet_capable hpet=disable hpet_address` ACPI HPET `hpet_set_mapping`

```
hpet_virt_address = ioremap_nocache(hpet_address, HPET_MMAP_SIZE);
```

IA-PC HPET (High Precision Event Timers) Specification

> 1024

`HPET_MMAP_SIZE` `1024`

```
#define HPET_MMAP_SIZE        1024
```

`HPET` `HPET_ID`：

```
id = hpet_readl(HPET_ID);

last = (id & HPET_ID_NUMBER) >> HPET_ID_NUMBER_SHIFT;
```

`HPET`

```
cfg = hpet_readl(HPET_CFG);

hpet_boot_cfg = kmalloc((last + 2) * sizeof(*hpet_boot_cfg), GFP_KERNEL);
```

`HPET` `HPET_CFG_ENABLE` `HPET_CFG_ENABLE` `hpet_clocksource_register`

```
if (hpet_clocksource_register())
    goto out_nohpet;
```

```
clocksource_register_hz(&clocksource_hpet, (u32)hpet_freq);
```

`clocksource_hpet` `clocksource` `rating 250` `refined_jiffies` `rating 2` `hpet` `read_hpet` `HPET`

```
static struct clocksource clocksource_hpet = {
    .name        = "hpet",
    .rating        = 250,
    .read        = read_hpet,
    .mask        = HPET_MASK,
    .flags        = CLOCK_SOURCE_IS_CONTINUOUS,
    .resume        = hpet_resume_counter,
    .archdata    = { .vclock_mode = VCLOCK_HPET },
};
```

`clocksource_hpet` arch/x86/kernel/time.c `hpet_time_init()`

```
setup_default_timer_irq();
```

`setup_default_timer_irq` `legacy` IRQ i8259 IRQ0

High Precision Event TimerLinux `read_hpet`

```
static cycle_t read_hpet(struct clocksource *cs)
{
    return (cycle_t)hpet_readl(HPET_COUNTER);
}
```

`Main Counter Register`

ACPI PM timer

ACPI Power Management Timerdrivers/clocksource/acpi_pm.c `fs` initcall `init_acpi_pm_clocksource`

`init_acpi_pm_clocksource` `pmtmr_ioport`

```
static int __init init_acpi_pm_clocksource(void)
{
    ...
    ...
    ...
    if (!pmtmr_ioport)
        return -ENODEV;
    ...
    ...
    ...
```

`pmtmr_ioport` Power Management Timer Control Register Block arch/x86/kernel/acpi/boot.c `acpi_parse_fadt` `FADT` `Fixed ACPI Description Table` ACPI `X_PM_TMR_BLK` `Power Management Timer Control Register Blcok` , `Generic Address Structure`

```
static int __init acpi_parse_fadt(struct acpi_table_header *table)
{
#ifdef CONFIG_X86_PM_TIMER
        ...
        ...
        ...
        pmtmr_ioport = acpi_gbl_FADT.xpm_timer_block.address;
        ...
        ...
        ...
#endif
    return 0;
}
```

`CONFIG_X86_PM_TIMER` `acpi_parse_fadt` `Power Management Timer` `init_acpi_pm_clocksource` `pmtmr_ioport` 0

```
clocksource_register_hz(&clocksource_acpi_pm, PMTMR_TICKS_PER_SEC);
```

`clocksource_register_hs` `acpi_pm` `clocksource` :

```
static struct clocksource clocksource_acpi_pm = {
    .name        = "acpi_pm",
    .rating        = 200,
    .read        = acpi_pm_read,
    .mask        = (cycle_t)ACPI_PM_MASK,
    .flags        = CLOCK_SOURCE_IS_CONTINUOUS,
};
```

`rating` `200` `acpi_pm_read` `apci_pm` `acpi_pm_read` `read_pmtmr` :

```
static cycle_t acpi_pm_read(struct clocksource *cs)
{
    return (cycle_t)read_pmtmr();
}
```

`Power Management Timer`

```
+-------------------------------+-----------------------------------+
|                               |                                   |
|  upper eight bits of a        |      running count of the         |
| 32-bit power management timer |     power management timer        |
|                               |                                   |
+-------------------------------+-----------------------------------+
```

```
31          E_TMR_VAL          24              TMR_VAL          0
```

`Fixed ACPI Description Table` ACPI `pmtmr_ioport` `read_pmtmr`

```
static inline u32 read_pmtmr(void)
{
    return inl(pmtmr_ioport) & ACPI_PM_MASK;
}
```

`Power Management Timer` `24`

`Time Stamp Counter`

Time Stamp Counter

Time Stamp Counterarch/x86/kernel/tsc.c `x86_late_time_init` Time Stamp Counter `tsc_init()`

`tsc_init` `Time Stamp Counter`:

```
void __init tsc_init(void)
{
    u64 lpj;
    int cpu;

    if (!cpu_has_tsc) {
        setup_clear_cpu_cap(X86_FEATURE_TSC_DEADLINE_TIMER);
        return;
    }
    ...
    ...
    ...
```

`cpu_has_tsc` `cpu_has` macro:

```
#define cpu_has_tsc        boot_cpu_has(X86_FEATURE_TSC)
#define boot_cpu_has(bit)    cpu_has(&boot_cpu_data, bit)
#define cpu_has(c, bit)                          \
    (__builtin_constant_p(bit) && REQUIRED_MASK_BIT_SET(bit) ? 1 :    \
     test_cpu_cap(c, bit))
```

`boot_cpu_data` `X86_FEATURE_TSC_DEADLINE_TIMER` `Time Stamp Counter` `calibrate_tsc` `TSC` MSRprogrammable interval timer

```
tsc_khz = x86_platform.calibrate_tsc();
cpu_khz = tsc_khz;

for_each_possible_cpu(cpu) {
    cyc2ns_init(cpu);
    set_cyc2ns_scale(cpu_khz, cpu);
}
```

`tsc_init` `TSC`

```
if (tsc_disabled > 0)
    return;
...
...
...
check_system_tsc_reliable();
```

`check_system_tsc_reliable` bootstrap `X86_FEATURE_TSC_RELIABLE` `tsc_clocksource_reliable` `tsc_init` `TSC` :

```
static int __init init_tsc_clocksource(void)
{
    if (!cpu_has_tsc || tsc_disabled > 0 || !tsc_khz)
        return 0;
    ...
    ...
    ...
    if (boot_cpu_has(X86_FEATURE_TSC_RELIABLE)) {
        clocksource_register_khz(&clocksource_tsc, tsc_khz);
        return 0;
    }
```

`device` initcall `TSC` HPET `clocksource` `TSC`

```
static struct clocksource clocksource_tsc = {
    .name                   = "tsc",
    .rating                 = 300,
    .read                   = read_tsc,
    .mask                   = CLOCKSOURCE_MASK(64),
    .flags                  = CLOCK_SOURCE_IS_CONTINUOUS | CLOCK_SOURCE_MUST_VERIFY,
    .archdata               = { .vclock_mode = VCLOCK_TSC },
};
```

Conclusion

Linux `clockevents` Linux x86 Linux twitter 0xAX emailissue **PR** **linux-insides**

- x86
- sysfs
- Time Stamp Counter
- High Precision Event Timer
- ACPI Power Management Timer (PDF)
- frequency.
- dmesg
- programmable interval timer
- IRQ
- IA-PC HPET (High Precision Event Timers) Specification
- IRQ0
- i8259
- initcall
- previous part

Timers and time management in the Linux kernel. Part 7.

Time related system calls in the Linux kernel

This is the seventh and last part chapter which describes timers and time management related stuff in the Linux kernel. In the previous part we saw some x86_64 like High Precision Event Timer and Time Stamp Counter. Internal time management is interesting part of the Linux kernel, but of course not only the kernel needs in the `time` concept. Our programs need to know time too. In this part, we will consider implementation of some time management related system calls. These system calls are:

- `clock_gettime` ;
- `gettimeofday` ;
- `nanosleep` .

We will start from simple userspace C program and see all way from the call of the standard library function to the implementation of certain system call. As each architecture provides its own implementation of certain system call, we will consider only x86_64 specific implementations of system calls, as this book is related to this architecture.

Additionally we will not consider concept of system calls in this part, but only implementations of these three system calls in the Linux kernel. If you are interested in what is it a `system call` , there is special chapter about this.

So, let's from the `gettimeofday` system call.

Implementation of the `gettimeofday` system call

As we can understand from the name of the `gettimeofday` , this function returns current time. First of all, let's look on the following simple example:

```
#include <time.h>
#include <sys/time.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    char buffer[40];
    struct timeval time;

    gettimeofday(&time, NULL);

    strftime(buffer, 40, "Current date/time: %m-%d-%Y/%T", localtime(&time.tv_sec));
    printf("%s\n",buffer);

    return 0;
}
```

As you can see, here we call the `gettimeofday` function which takes two parameters: pointer to the `timeval` structure which represents an elapsed tim:

```
struct timeval {
    time_t      tv_sec;     /* seconds */
    suseconds_t tv_usec;    /* microseconds */
};
```

The second parameter of the `gettimeofday` function is pointer to the `timezone` structure which represents a timezone. In our example, we pass address of the `timeval time` to the `gettimeofday` function, the Linux kernel fills the given `timeval` structure and returns it back to us. Additionally, we format the time with the `strftime` function to get something more human readable than elapsed microseconds. Let's see on result:

```
~$ gcc date.c -o date
~$ ./date
Current date/time: 03-26-2016/16:42:02
```

As you already may know, an userspace application does not call a system call directly from the kernel space. Before the actual system call entry will be called, we call a function from the standard library. In my case it is glibc, so I will consider this case. The implementation of the `gettimeofday` function is located in the sysdeps/unix/sysv/linux/x86/gettimeofday.c source code file. As you already may know, the `gettimeofday` is not usual system call. It is located in the special area which is called `vDSO` (you can read more about it in the part which describes this concept).

The `glibc` implementation of the `gettimeofday` tries to resolve the given symbol, in our case this symbol is `__vdso_gettimeofday` by the call of the `_dl_vdso_vsym` internal function. If the symbol will not be resolved, it returns `NULL` and we fallback to the call of the usual system call:

```
return (_dl_vdso_vsym ("__vdso_gettimeofday", &linux26)
  ?: (void*) (&__gettimeofday_syscall));
```

The `gettimeofday` entry is located in the arch/x86/entry/vdso/vclock_gettime.c source code file. As we can see the `gettimeofday` is weak alias of the `__vdso_gettimeofday` :

```
int gettimeofday(struct timeval *, struct timezone *)
    __attribute__((weak, alias("__vdso_gettimeofday")));
```

The `__vdso_gettimeofday` is defined in the same source code file and calls the `do_realtime` function if the given `timeval` is not null:

```
notrace int __vdso_gettimeofday(struct timeval *tv, struct timezone *tz)
{
    if (likely(tv != NULL)) {
        if (unlikely(do_realtime((struct timespec *)tv) == VCLOCK_NONE))
            return vdso_fallback_gtod(tv, tz);
        tv->tv_usec /= 1000;
    }
    if (unlikely(tz != NULL)) {
        tz->tz_minuteswest = gtod->tz_minuteswest;
        tz->tz_dsttime = gtod->tz_dsttime;
    }

    return 0;
}
```

If the `do_realtime` will fail, we fallback to the real system call via call the `syscall` instruction and passing the `__NR_gettimeofday` system call number and the given `timeval` and `timezone` :

```
notrace static long vdso_fallback_gtod(struct timeval *tv, struct timezone *tz)
{
    long ret;

    asm("syscall" : "=a" (ret) :
        "0" (__NR_gettimeofday), "D" (tv), "S" (tz) : "memory");
    return ret;
}
```

The `do_realtime` function gets the time data from the `vsyscall_gtod_data` structure which is defined in the arch/x86/include/asm/vgtod.h header file and contains mapping of the `timespec` structure and a couple of fields which are related to the current clock source in the system. This function fills the given `timeval` structure with values from the `vsyscall_gtod_data` which contains a time related data which is updated via timer interrupt.

First of all we try to access the `gtod` or `global time of day` the `vsyscall_gtod_data` structure via the call of the `gtod_read_begin` and will continue to do it until it will be successful:

```
do {
    seq = gtod_read_begin(gtod);
    mode = gtod->vclock_mode;
    ts->tv_sec = gtod->wall_time_sec;
    ns = gtod->wall_time_snsec;
    ns += vgetsns(&mode);
    ns >>= gtod->shift;
} while (unlikely(gtod_read_retry(gtod, seq)));

ts->tv_sec += __iter_div_u64_rem(ns, NSEC_PER_SEC, &ns);
ts->tv_nsec = ns;
```

As we got access to the `gtod` , we fill the `ts->tv_sec` with the `gtod->wall_time_sec` which stores current time in seconds gotten from the real time clock during initialization of the timekeeping subsystem in the Linux kernel and the same value but in nanoseconds. In the end of this code we just fill the given `timespec` structure with the resulted values.

That's all about the `gettimeofday` system call. The next system call in our list is the `clock_gettime` .

Implementation of the clock_gettime system call

The `clock_gettime` function gets the time which is specified by the second parameter. Generally the `clock_gettime` function takes two parameters:

- `clk_id` - clock identifier;
- `timespec` - address of the `timespec` structure which represent elapsed time.

Let's look on the following simple example:

```
#include <time.h>
#include <sys/time.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    struct timespec elapsed_from_boot;

    clock_gettime(CLOCK_BOOTTIME, &elapsed_from_boot);

    printf("%d - seconds elapsed from boot\n", elapsed_from_boot.tv_sec);

    return 0;
}
```

which prints `uptime` information:

```
~$ gcc uptime.c -o uptime
~$ ./uptime
14180 - seconds elapsed from boot
```

We can easily check the result with the help of the uptime util:

```
~$ uptime
up  3:56
```

The `elapsed_from_boot.tv_sec` represents elapsed time in seconds, so:

```
>>> 14180 / 60
236
>>> 14180 / 60 / 60
3
>>> 14180 / 60 % 60
```

```
56
```

The `clock_id` maybe one of the following:

- `CLOCK_REALTIME` - system wide clock which measures real or wall-clock time;
- `CLOCK_REALTIME_COARSE` - faster version of the `CLOCK_REALTIME` ;
- `CLOCK_MONOTONIC` - represents monotonic time since some unspecified starting point;
- `CLOCK_MONOTONIC_COARSE` - faster version of the `CLOCK_MONOTONIC` ;
- `CLOCK_MONOTONIC_RAW` - the same as the `CLOCK_MONOTONIC` but provides non NTP adjusted time.
- `CLOCK_BOOTTIME` - the same as the `CLOCK_MONOTONIC` but plus time that the system was suspended;
- `CLOCK_PROCESS_CPUTIME_ID` - per-process time consumed by all threads in the process;
- `CLOCK_THREAD_CPUTIME_ID` - thread-specific clock.

The `clock_gettime` is not usual syscall too, but as the `gettimeofday` , this system call is placed in the `vDSO` area. Entry of this system call is located in the same source code file - arch/x86/entry/vdso/vclock_gettime.c) as for `gettimeofday` .

The Implementation of the `clock_gettime` depends on the clock id. If we have passed the `CLOCK_REALTIME` clock id, the `do_realtime` function will be called:

```
notrace int __vdso_clock_gettime(clockid_t clock, struct timespec *ts)
{
    switch (clock) {
    case CLOCK_REALTIME:
        if (do_realtime(ts) == VCLOCK_NONE)
            goto fallback;
        break;
    ...
    ...
    ...
fallback:
    return vdso_fallback_gettime(clock, ts);
}
```

In other cases, the `do_{name_of_clock_id}` function is called. Implementations of some of them is similar. For example if we will pass the `CLOCK_MONOTONIC` clock id:

```
...
...
...
case CLOCK_MONOTONIC:
    if (do_monotonic(ts) == VCLOCK_NONE)
        goto fallback;
    break;
...
...
...
```

the `do_monotonic` function will be called which is very similar on the implementation of the `do_realtime` :

```
notrace static int __always_inline do_monotonic(struct timespec *ts)
{
    do {
        seq = gtod_read_begin(gtod);
        mode = gtod->vclock_mode;
        ts->tv_sec = gtod->monotonic_time_sec;
        ns = gtod->monotonic_time_snsec;
        ns += vgetsns(&mode);
        ns >>= gtod->shift;
    } while (unlikely(gtod_read_retry(gtod, seq)));

    ts->tv_sec += __iter_div_u64_rem(ns, NSEC_PER_SEC, &ns);
    ts->tv_nsec = ns;

    return mode;
```

```
}
```

We already saw a little about the implementation of this function in the previous paragraph about the `gettimeofday`. There is only one difference here, that the `sec` and `nsec` of our `timespec` value will be based on the `gtod->monotonic_time_sec` instead of `gtod->wall_time_sec` which maps the value of the `tk->tkr_mono.xtime_nsec` or number of nanoseconds elapsed.

That's all.

Implementation of the `nanosleep` system call

The last system call in our list is the `nanosleep`. As you can understand from its name, this function provides `sleeping` ability. Let's look on the following simple example:

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>

int main (void)
{
   struct timespec ts = {5,0};

   printf("sleep five seconds\n");
   nanosleep(&ts, NULL);
   printf("end of sleep\n");

   return 0;
}
```

If we will compile and run it, we will see the first line

```
~$ gcc sleep_test.c -o sleep
~$ ./sleep
sleep five seconds
end of sleep
```

and the second line after five seconds.

The `nanosleep` is not located in the `vDSO` area like the `gettimeofday` and the `clock_gettime` functions. So, let's look how the `real` system call which is located in the kernel space will be called by the standard library. The implementation of the `nanosleep` system call will be called with the help of the syscall instruction. Before the execution of the `syscall` instruction, parameters of the system call must be put in processor registers according to order which is described in the System V Application Binary Interface or in other words:

- `rdi` - first parameter;
- `rsi` - second parameter;
- `rdx` - third parameter;
- `r10` - fourth parameter;
- `r8` - fifth parameter;
- `r9` - sixth parameter.

The `nanosleep` system call has two parameters - two pointers to the `timespec` structures. The system call suspends the calling thread until the given timeout has elapsed. Additionally it will finish if a signal interrupts its execution. It takes two parameters, the first is `timespec` which represents timeout for the sleep. The second parameter is the pointer to the `timespec` structure too and it contains remainder of time if the call of the `nanosleep` was interrupted.

As `nanosleep` has two parameters:

```
int nanosleep(const struct timespec *req, struct timespec *rem);
```

To call system call, we need put the `req` to the `rdi` register, and the `rem` parameter to the `rsi` register. The glibc does these job in the `INTERNAL_SYSCALL` macro which is located in the sysdeps/unix/sysv/linux/x86_64/sysdep.h header file.

```
# define INTERNAL_SYSCALL(name, err, nr, args...) \
  INTERNAL_SYSCALL_NCS (__NR_##name, err, nr, ##args)
```

which takes the name of the system call, storage for possible error during execution of system call, number of the system call (all `x86_64` system calls you can find in the system calls table) and arguments of certain system call. The `INTERNAL_SYSCALL` macro just expands to the call of the `INTERNAL_SYSCALL_NCS` macro, which prepares arguments of system call (puts them into the processor registers in correct order), executes `syscall` instruction and returns the result:

```
# define INTERNAL_SYSCALL_NCS(name, err, nr, args...)      \
  ({                                                                \
    unsigned long int resultvar;                                    \
    LOAD_ARGS_##nr (args)                                            \
    LOAD_REGS_##nr                                                    \
    asm volatile (                                                    \
    "syscall\n\t"                                                    \
    : "=a" (resultvar)                                                \
    : "0" (name) ASM_ARGS_##nr : "memory", REGISTERS_CLOBBERED_BY_SYSCALL);   \
    (long int) resultvar; })
```

The `LOAD_ARGS_##nr` macro calls the `LOAD_ARGS_N` macro where the `N` is number of arguments of the system call. In our case, it will be the `LOAD_ARGS_2` macro. Ultimately all of these macros will be expanded to the following:

```
# define LOAD_REGS_TYPES_1(t1, a1)                      \
  register t1 _a1 asm ("rdi") = __arg1;                      \
  LOAD_REGS_0

# define LOAD_REGS_TYPES_2(t1, a1, t2, a2)                   \
  register t2 _a2 asm ("rsi") = __arg2;                       \
  LOAD_REGS_TYPES_1(t1, a1)
...
...
...
```

After the `syscall` instruction will be executed, the context switch will occur and the kernel will transfer execution to the system call handler. The system call handler for the `nanosleep` system call is located in the kernel/time/hrtimer.c source code file and defined with the `SYSCALL_DEFINE2` macro helper:

```
SYSCALL_DEFINE2(nanosleep, struct timespec __user *, rqtp,
        struct timespec __user *, rmtp)
{
    struct timespec tu;

    if (copy_from_user(&tu, rqtp, sizeof(tu)))
        return -EFAULT;

    if (!timespec_valid(&tu))
        return -EINVAL;

    return hrtimer_nanosleep(&tu, rmtp, HRTIMER_MODE_REL, CLOCK_MONOTONIC);
}
```

More about the `SYSCALL_DEFINE2` macro you may read in the chapter about system calls. If we look at the implementation of the `nanosleep` system call, first of all we will see that it starts from the call of the `copy_from_user` function. This function copies the given data from the userspace to kernelspace. In our case we copy timeout value to sleep to the kernelspace `timespec` structure and check that the given `timespec` is valid by the call of the `timesc_valid` function:

```
static inline bool timespec_valid(const struct timespec *ts)
{
    if (ts->tv_sec < 0)
```

```
        return false;
    if ((unsigned long)ts->tv_nsec >= NSEC_PER_SEC)
        return false;
    return true;
}
```

which just checks that the given `timespec` does not represent date before `1970` and nanoseconds does not overflow `1` second. The `nanosleep` function ends with the call of the `hrtimer_nanosleep` function from the same source code file. The `hrtimer_nanosleep` function creates a timer and calls the `do_nanosleep` function. The `do_nanosleep` does main job for us. This function provides loop:

```
do {
    set_current_state(TASK_INTERRUPTIBLE);
    hrtimer_start_expires(&t->timer, mode);

    if (likely(t->task))
        freezable_schedule();

} while (t->task && !signal_pending(current));

__set_current_state(TASK_RUNNING);
return t->task == NULL;
```

Which freezes current task during sleep. After we set `TASK_INTERRUPTIBLE` flag for the current task, the `hrtimer_start_expires` function starts the give high-resolution timer on the current processor. As the given high resolution timer will expire, the task will be again running.

That's all.

Conclusion

This is the end of the seventh part of the chapter that describes timers and timer management related stuff in the Linux kernel. In the previous part we saw x86_64 specific clock sources. As I wrote in the beginning, this part is the last part of this chapter. We saw important time management related concepts like `clocksource` and `clockevents` frameworks, `jiffies` counter and etc., in this chpater. Of course this does not cover all of the time management in the Linux kernel. Many parts of this mostly related to the scheduling which we will see in other chapter.

If you have questions or suggestions, feel free to ping me in twitter 0xAX, drop me email or just create issue.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-insides.

Links

- system call
- C programming language
- standard library
- glibc
- real time clock
- NTP
- nanoseconds
- register
- System V Application Binary Interface
- context switch
- Introduction to timers in the Linux kernel
- uptime
- system calls table for x86_64
- High Precision Event Timer

- Time Stamp Counter
- x86_64
- previous part

Linux

- - Linux
- - -
- - this part describes impmentation of `semaphore` synchronization primitive in the Linux kernel. Linux `semaphore`

- - Linux `mutex`
- / - - `reader/writer`
- - Linux .

Linux . .

Introduction

linux-insides Linux

```
mutex_lock(&clocksource_mutex);
...
...
...
clocksource_enqueue(cs);
clocksource_enqueue_watchdog(cs);
clocksource_select();
...
...
...
mutex_unlock(&clocksource_mutex);
```

kernel/time/clocksource.c `__clocksource_register_scale` clocksource `clocksource_enqueue` —— `clocksource_list` `mutex_lock` `mutex_unlock` —— `clocksource_mutex`

(mutex) `mutex_lock` `mute_unlock` `clocksource_list` `clocksource_enqueue` `clocksource_list`

```
static void clocksource_enqueue(struct clocksource *cs)
{
    struct list_head *entry = &clocksource_list;
    struct clocksource *tmp;

    list_for_each_entry(tmp, &clocksource_list, list)
        if (tmp->rating >= cs->rating)
            entry = &tmp->list;
    list_add(&cs->list, entry);
}
```

`(entry)` (race condition) `list_add`

Linux [] (/Timers/) Linux `mutex` Linux Linux

- `mutex` ;
- `semaphores` ;
- `seqlocks` ;
- `atomic operations` ;
-

`(spinlock)`

Linux

- `acquired` ;
- `released` .

`(spinlock acquire)` `(spinlock released)` (atomic) Linux `spinlock_t` Linux (widely) `spinlock_t`

```
typedef struct spinlock {
        union {
              struct raw_spinlock rlock;

#ifdef CONFIG_DEBUG_LOCK_ALLOC
# define LOCK_PADSIZE (offsetof(struct raw_spinlock, dep_map))
                struct {
                        u8 __padding[LOCK_PADSIZE];
                        struct lockdep_map dep_map;
                };
#endif
        };
} spinlock_t;
```

include/linux/spinlock_types.h `CONFIG_DEBUG_LOCK_ALLOC` `CONFIG_DEBUG_LOCK_ALLOC` `spinlock_t` (union) ——— `raw_spinlock`

```
typedef struct spinlock {
        union {
              struct raw_spinlock rlock;
        };
} spinlock_t;
```

`raw_spinlock` (normal) `raw_spinlock`

```
typedef struct raw_spinlock {
        arch_spinlock_t raw_lock;
#ifdef CONFIG_GENERIC_LOCKBREAK
        unsigned int break_lock;
#endif
} raw_spinlock_t;
```

`arch_spinlock_t` `break_lock` ——— `1` (SMP) x86_64 `arch_spinlock_t` arch/x86/include/asm/spinlock_types.h

```
#ifdef CONFIG_QUEUED_SPINLOCKS
#include <asm-generic/qspinlock_types.h>
#else
typedef struct arch_spinlock {
        union {
                __ticketpair_t head_tail;
                struct __raw_tickets {
                        __ticket_t head, tail;
                } tickets;
        };
} arch_spinlock_t;
```

`arch_spinlock` `CONFIG_QUEUED_SPINLOCKS` Linux `acquired` `released` `CONFIG_QUEUED_SPINLOCKS` `arch_spinlock_t`

```
typedef struct qspinlock {
    atomic_t    val;
} arch_spinlock_t;
```

include/asm-generic/qspinlock_types.h

`arch_spinlock` `qspinlock` Linux

- `spin_lock_init` ———
- `spin_lock` ———
- `spin_lock_bh` ———
- `spin_lock_irqsave` `spin_lock_irq` ——— `(flag)`

- `spin_unlock` ——— ;
- `spin_unlock_bh` ———
- `spin_is_locked` -
-

`spin_lock_init` include/linux/spinlock.h `spin_lock_init`

```
#define spin_lock_init(_lock)        \
do {                                          \
    spinlock_check(_lock);                      \
    raw_spin_lock_init(&(_lock)->rlock);        \
} while (0)
```

`spin_lock_init` `raw_spin_lock_init` `spinlock_check` `raw_spinlock_t` (normal)

```
static __always_inline raw_spinlock_t *spinlock_check(spinlock_t *lock)
{
    return &lock->rlock;
}
```

`raw_spin_lock_init` :

```
# define raw_spin_lock_init(lock)        \
do {                                              \
    *(lock) = __RAW_SPIN_LOCK_UNLOCKED(lock);         \
} while (0)                                       \
```

`__RAW_SPIN_LOCK_UNLOCKED` `raw_spinlock_t` `__RAW_SPIN_LOCK_UNLOCKED` (released)
include/linux/spinlock_types.h

```
#define __RAW_SPIN_LOCK_UNLOCKED(lockname)      \
        (raw_spinlock_t) __RAW_SPIN_LOCK_INITIALIZER(lockname)

#define __RAW_SPIN_LOCK_INITIALIZER(lockname)   \
        {                                                   \
            .raw_lock = __ARCH_SPIN_LOCK_UNLOCKED,          \
            SPIN_DEBUG_INIT(lockname)                       \
            SPIN_DEP_MAP_INIT(lockname)                     \
        }
```

`SPIN_DEBUG_INIT` `SPIN_DEP_MAP_INIT` `__RAW_SPINLOCK_UNLOCKED`

```
*(&(_lock)->rlock) = __ARCH_SPIN_LOCK_UNLOCKED;
```

`__ARCH_SPIN_LOCK_UNLOCKED` :

```
#define __ARCH_SPIN_LOCK_UNLOCKED       { { 0 } }
```

:

```
#define __ARCH_SPIN_LOCK_UNLOCKED       { ATOMIC_INIT(0) }
```

[x86_64] `CONFIG_QUEUED_SPINLOCKS` `spin_lock_init` ——— (unlocked)

Linux API

```
static __always_inline void spin_lock(spinlock_t *lock)
{
    raw_spin_lock(&lock->rlock);
```

```
}
```

`raw_spin_lock` `_raw_spin_lock`

```
#define raw_spin_lock(lock)    _raw_spin_lock(lock)
```

include/linux/spinlock.h `_raw_spin_lock` `CONFIG_SMP`

```
#if defined(CONFIG_SMP) || defined(CONFIG_DEBUG_SPINLOCK)
# include <linux/spinlock_api_smp.h>
#else
# include <linux/spinlock_api_up.h>
#endif
```

Linux SMP `_raw_spin_lock` arch/x86/include/asm/spinlock.h

```
#define _raw_spin_lock(lock) __raw_spin_lock(lock)
```

`__raw_spin_lock` :

```
static inline void __raw_spin_lock(raw_spinlock_t *lock)
{
        preempt_disable();
        spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
        LOCK_CONTENDED(lock, do_raw_spin_trylock, do_raw_spin_lock);
}
```

include/linux/preempt.h (Linux) `preempt_disable`

```
static inline void __raw_spin_unlock(raw_spinlock_t *lock)
{
        ...
        ...
        ...
        preempt_enable();
}
```

`spin_acquire`

```
#define spin_acquire(l, s, t, i)                lock_acquire_exclusive(l, s, t, NULL, i)
#define lock_acquire_exclusive(l, s, t, n, i)           lock_acquire(l, s, t, 0, 1, n, i)
```

`lock_acquire` :

```
void lock_acquire(struct lockdep_map *lock, unsigned int subclass,
                  int trylock, int read, int check,
                  struct lockdep_map *nest_lock, unsigned long ip)
{
         unsigned long flags;

         if (unlikely(current->lockdep_recursion))
                return;

         raw_local_irq_save(flags);
         check_flags(flags);

         current->lockdep_recursion = 1;
         trace_lock_acquire(lock, subclass, trylock, read, check, nest_lock, ip);
         __lock_acquire(lock, subclass, trylock, read, check,
                        irqs_disabled_flags(flags), nest_lock, ip, 0, 0);
         current->lockdep_recursion = 0;
```

```
        raw_local_irq_restore(flags);
}
```

`lock_acquire` `raw_local_irq_save` `lock_acquire` `raw_local_irq_restore` `__lock_acquire` kernel/locking/lockdep.c

`__lock_acquire` Linux (lock validator) `__raw_spin_lock`

```
LOCK_CONTENDED(lock, do_raw_spin_trylock, do_raw_spin_lock);
```

`LOCK_CONTENDED` include/linux/lockdep.h :

```
#define LOCK_CONTENDED(_lock, try, lock) \
        lock(_lock)
```

`lock` include/linux/spinlock.h `do_raw_spin_lock` `_lock` `raw_spinlock_t`

```
static inline void do_raw_spin_lock(raw_spinlock_t *lock) __acquires(lock)
{
        __acquire(lock);
        arch_spin_lock(&lock->raw_lock);
}
```

`__acquire` [(sparse)] `arch_spin_lock` (queued spinlocks) `x86_64` `arch_spin_lock` include/asm-generic/qspinlock.h

```
#define arch_spin_lock(l)               queued_spin_lock(l)
```

`arch_spin_lock` arch/x86/include/asm/spinlock.h `arch_spinlock` `arch_spin_lock`

```
typedef struct arch_spinlock {
        union {
                __ticketpair_t head_tail;
                struct __raw_tickets {
                        __ticket_t head, tail;
                } tickets;
        };
} arch_spinlock_t;
```

—— (ticket spinlock) (tail) 1 `arch_spin_lock`

```
static __always_inline void arch_spin_lock(arch_spinlock_t *lock)
{
        register struct __raw_tickets inc = { .tail = TICKET_LOCK_INC };

        inc = xadd(&lock->tickets, inc);

        if (likely(inc.head == inc.tail))
                goto out;

        for (;;) {
                unsigned count = SPIN_THRESHOLD;

                do {
                        inc.head = READ_ONCE(lock->tickets.head);
                        if (__tickets_equal(inc.head, inc.tail))
                                goto clear_slowpath;
                        cpu_relax();
                } while (--count);
                __ticket_lock_spinning(lock, inc.tail);
        }
clear_slowpath:
```

```
        __ticket_check_and_clear_slowpath(lock, inc.head);
out:
        barrier();
}
```

arch_spin_lock ——— 1 __raw_tickets

```
#define __TICKET_LOCK_INC       1
```

inc lock->tickets xadd inc (tickets) tickets.tail inc 1 1

out arch_spin_lock barrier (barrier instruction) ((documentatic

arch_spin_lock `` 1 cpu_relax NOP

```
#define cpu_relax()     asm volatile("rep; nop")
```

spin_unlock spin_lock unlock arch_spin_unlock arch_spin_lock lock tickets

```
__add(&lock->tickets.head, TICKET_LOCK_INC, UNLOCK_LOCK_PREFIX);
```

spin_lock spin_unlock

```
     +-------+       +-------+
     |       |       |       |
head |   7   | - - - |   7   | tail
     |       |       |       |
     +-------+       +-------+
                         |
                     +-------+
                     |       |
                     |   8   |
                     |       |
                     +-------+
                         |
                     +-------+
                     |       |
                     |   9   |
                     |       |
                     +-------+
```

API

Linux Linux

twitter 0xAX email issue

PR **linux-insides**

- Concurrent computing
- Synchronization
- Clocksource framework

- Mutex
- Race condition
- Atomic operations
- SMP
- x86_64
- Interrupts
- Preemption
- Linux kernel lock validator
- Sparse
- xadd instruction
- NOP
- Memory barriers
- Previous chapter

Linux . .

Linux Linux -

API:

- `spin_lock_init` -
- `spin_lock` -
- `spin_lock_bh` -
- `spin_lock_irqsave` `spin_lock_irq` - /
- `spin_unlock` -
- `spin_unlock_bh` -
- `spin_is_locked` -
-

include/linux/spinlock.h x86_64 arch/x86/include/asm/spinlock.h `arch_spin_.*`
(`arch_spin_is_locked` `arch_spin_lock` `arch_spin_unlock`) `CONFIG_QUEUED_SPINLOCKS`

```
#ifdef CONFIG_QUEUED_SPINLOCKS
#include <asm/qspinlock.h>
#else
static __always_inline void arch_spin_lock(arch_spinlock_t *lock)
{
    ...
    ...
    ...
}
...
...
...
#endif
```

arch/x86/include/asm/qspinlock.h include/asm-generic/qspinlock.h

```
#define arch_spin_is_locked(l)          queued_spin_is_locked(l)
#define arch_spin_is_contended(l)       queued_spin_is_contended(l)
#define arch_spin_value_unlocked(l)     queued_spin_value_unlocked(l)
#define arch_spin_lock(l)               queued_spin_lock(l)
#define arch_spin_trylock(l)            queued_spin_trylock(l)
#define arch_spin_unlock(l)             queued_spin_unlock(l)
#define arch_spin_lock_flags(l, f)      queued_spin_lock(l)
#define arch_spin_unlock_wait(l)        queued_spin_unlock_wait(l)
```

API

Linux x86_64 - kernel/Kconfig.locks

```
config ARCH_USE_QUEUED_SPINLOCKS
    bool

config QUEUED_SPINLOCKS
    def_bool y if ARCH_USE_QUEUED_SPINLOCKS
    depends on SMP
```

`ARCH_USE_QUEUED_SPINLOCKS` `CONFIG_QUEUED_SPINLOCKS` `ARCH_USE_QUEUED_SPINLOCKS` `x86_64` - arch/x86/Kconfig

```
config X86
    ...
    ...
    ...
    select ARCH_USE_QUEUED_SPINLOCKS
    ...
    ...
    ...
```

test and set

```
int lock(lock)
{
    while (test_and_set(lock) == 1)
        ;
    return 0;
}

int unlock(lock)
{
    lock=0;

    return lock;
}
```

`test_and_set` `lock` `1` `lock` `while` `unlock` `lock` `0` `lock` `test_and_set` `lock=1` `lock` `1`

- `(ticket spinlock)`

MCS `MCS`

`MCS` Linux per-cpu

`next` `next`

```
+---------+
|         |
|  Queue  |
|         |
+---------+
```

```
+---------+     +----------------------------+
|         |     |                            |
|  Queue  |---->| First thread acquired lock |
|         |     |                            |
+---------+     +----------------------------+
```

:

```
+---------+     +----------------------------------------+     +-------------------------+
|         |     |                                        |     |                         |
|  Queue  |---->|  Second thread waits for first thread  |<----| First thread holds lock |
|         |     |                                        |     |                         |
+---------+     +----------------------------------------+     +-------------------------+
```

```
void lock(...)
{
    lock.next = NULL;
    ancestor = put_lock_to_queue_and_return_ancestor(queue, lock);

    // if we have ancestor, the lock already acquired and we
    // need to wait until it will be released
    if (ancestor)
    {
        lock.locked = 1;
        ancestor.next = lock;

        while (lock.is_locked == true)
            ;
    }

    // in other way we are owner of the lock and may exit
}

void unlock(...)
{
    // do we need to notify somebody or we are alonw in the
    // queue?
    if (lock.next != NULL) {
        // the while loop from the lock() function will be
        // finished
        lock.next.is_locked = false;
        // delete ourself from the queue and exit
        ...
        ...
        ...
        return;
    }

    // So, we have no next threads in the queue to notify about
    // lock releasing event. Let's just put `0` to the lock, will
    // delete ourself from the queue and exit.
}
```

Linux 32(32-bit) (word) MCS spinlock_t Linux (widely) Linux Linux 32

Linux

API

Linux include/asm-generic/qspinlock.h API

```
#define arch_spin_is_locked(l)          queued_spin_is_locked(l)
#define arch_spin_is_contended(l)       queued_spin_is_contended(l)
#define arch_spin_value_unlocked(l)     queued_spin_value_unlocked(l)
#define arch_spin_lock(l)               queued_spin_lock(l)
#define arch_spin_trylock(l)            queued_spin_trylock(l)
#define arch_spin_unlock(l)             queued_spin_unlock(l)
#define arch_spin_lock_flags(l, f)      queued_spin_lock(l)
#define arch_spin_unlock_wait(l)        queued_spin_unlock_wait(l)
```

include/asm-generic/qspinlock_types.h qspinlock Linux

```
typedef struct qspinlock {
    atomic_t    val;
} arch_spinlock_t;
```

`qspinlock` - `val` `4` 4

- `0-7` - (locked byte);
- `8` - (pending bit);
- `16-17` - `MCS` `per_cpu` ()
- `18-31` -

`9-15`

```
struct mcs_spinlock {
       struct mcs_spinlock *next;
       int locked;
       int count;
};
```

kernel/locking/mcs_spinlock.h `1` `0` `mcs_spinlock` (nested locks) `mcs_spinlock`

```
static DEFINE_PER_CPU_ALIGNED(struct mcs_spinlock, mcs_nodes[4]);
```

(This array allows to make four attempts of a lock acquisition for the four events in following contexts:)

-
-
-
-

`qspinlock` `API` `API` `qspinlock` `val` - `atomic_t` (one operation at a time variable) `val` API

```
static __always_inline int queued_spin_is_locked(struct qspinlock *lock)
{
    return atomic_read(&lock->val);
}
```

Ok Linux API `main`

```
#define arch_spin_lock(l)               queued_spin_lock(l)
```

- `queued_spin_lock` include/asm-generic/qspinlock_types.h

```
static __always_inline void queued_spin_lock(struct qspinlock *lock)
{
        u32 val;

        val = atomic_cmpxchg_acquire(&lock->val, 0, _Q_LOCKED_VAL);
        if (likely(val == 0))
                return;
        queued_spin_lock_slowpath(lock, val);
}
```

`queued_spin_lock_slowpath` `queued_spin_lock` `atomic_cmpxchg_acquire` CMPXCHG `_Q_LOCKED_VAL` `&lock->val`

`atomic_cmpxchg_acquire` include/linux/atomic.h `atomic_cmpxchg`

```
#define  atomic_cmpxchg_acquire         atomic_cmpxchg
```

x86_64 arch/x86/include/asm/atomic.h `atomic_cmpxchg` `cmpxchg`

```
static __always_inline int atomic_cmpxchg(atomic_t *v, int old, int new)
{
        return cmpxchg(&v->counter, old, new);
}
```

arch/x86/include/asm/cmpxchg.h

```
#define cmpxchg(ptr, old, new) \
    __cmpxchg(ptr, old, new, sizeof(*(ptr)))

#define __cmpxchg(ptr, old, new, size) \
    __raw_cmpxchg((ptr), (old), (new), (size), LOCK_PREFIX)
```

`cmpxchg` `__cpmxchg` `__cpmxchg` `LOCK_PREFIX` `__raw_cmpxchg` `LOCK_PREFIX` LOCK `__raw_cmpxchg`

```
#define __raw_cmpxchg(ptr, old, new, size, lock) \
({
    ...
    ...
    ...
    volatile u32 *__ptr = (volatile u32 *)(ptr);            \
    asm volatile(lock "cmpxchgl %2,%1"                      \
                 : "=a" (__ret), "+m" (*__ptr)              \
                 : "r" (__new), "" (__old)                  \
                 : "memory");                               \
    ...
    ...
    ...
})
```

`atomic_cmpxchg_acquire` `val` `queued_spin_lock`

```
val = atomic_cmpxchg_acquire(&lock->val, 0, _Q_LOCKED_VAL);
if (likely(val == 0))
    return;
```

`MCS` Linux `MCS`

`queued_spin_lock` `lock->val` `1` `_Q_LOCKED_VAL` `queued_spin_lock_slowpath` `queued_spin_lock_slowpath` kernel/locking/qspinlock.c

```
void queued_spin_lock_slowpath(struct qspinlock *lock, u32 val)
{
    if (pv_enabled())
        goto queue;

    if (virt_spin_lock(lock))
        return;

    ...
    ...
    ...
}
```

`pvqspinlock` `pvqspinlock` paravirtualized Linux `_Q_PENDING_VAL` After these checks we compare our value which represents lock with the value of the `_Q_PENDING_VAL` macro and do nothing while this is true

```
if (val == _Q_PENDING_VAL) {
    while ((val = atomic_read(&lock->val)) == _Q_PENDING_VAL)
```

```
        cpu_relax();
}
```

`cpu_relax` NOP - `pending` `pending` (touched) `mcs_spinlock` This is done for optimization, because there are no need in unnecessary latency which will be caused by the cache invalidation in a touching of own `mcs_spinlock` array.

```
for (;;) {
    if (val & ~_Q_LOCKED_MASK)
        goto queue;

    new = _Q_LOCKED_VAL;
    if (val == new)
        new |= _Q_PENDING_VAL;

    old = atomic_cmpxchg_acquire(&lock->val, val, new);
    if (old == val)
        break;

    val = old;
}
```

`if` (`val`) (pending) `val` `&lock->val` `atomic_cmpxchg_acquire` `1` `new` `val` `&lock->val` `val` `atomic_cmpxchg_acquire` `lock->val` `1`

```
smp_cond_acquire(!(atomic_read(&lock->val) & _Q_LOCKED_MASK));
clear_pending_set_locked(lock);
return;
```

`lock->val` `_Q_LOCKED_VAL | _Q_PENDING_VAL` `mcs_nodes`

```
node = this_cpu_ptr(&mcs_nodes[0]);
idx = node->count++;
tail = encode_tail(smp_processor_id(), idx);
```

`mcs_nodes` `tail` `node` `mcs_nodes` `locked` `next` `NULL`

```
node += idx;
node->locked = 0;
node->next = NULL;
```

`cpuper-cpu` `queued_spin_trylock`

```
if (queued_spin_trylock(lock))
        goto release;
```

`queued_spin_trylock` include/asm-generic/qspinlock.h `queued_spin_lock`

```
static __always_inline int queued_spin_trylock(struct qspinlock *lock)
{
    if (!atomic_read(&lock->val) &&
       (atomic_cmpxchg_acquire(&lock->val, 0, _Q_LOCKED_VAL) == 0))
        return 1;
    return 0;
}
```

```
release:
    this_cpu_dec(mcs_nodes[0].count);
```

`queued_spin_trylock`

```
old = xchg_tail(lock, tail);
```

```
if (old & _Q_TAIL_MASK) {
    prev = decode_tail(old);
    WRITE_ONCE(prev->next, node);

    arch_mcs_spin_lock_contended(&node->locked);
}
```

```
next = READ_ONCE(node->next);
if (next)
    prefetchw(next);
```

PREFETCHW cache line `MCS`

```
smp_cond_acquire(!((val = atomic_read(&lock->val)) & _Q_LOCKED_PENDING_MASK));
```

Linux Linux `ticket spinlock` - Linux

twitter 0xAX email issue.

PR **linux-insides**

- spinlock
- interrupt
- interrupt handler
- API
- Test and Set
- MCS
- per-cpu variables
- atomic instruction
- CMPXCHG instruction
- LOCK instruction
- NOP instruction
- PREFETCHW instruction
- x86_64

- Previous part

• •

chapter, -

Linux

Linux

- Linux -

0 1

- ;
- .

1 0 1 1

API

Linux API include/linux/semaphore.h

```
struct semaphore {
    raw_spinlock_t        lock;
    unsigned int        count;
    struct list_head    wait_list;
};
```

- `lock` - ;
- `count` -;
- `wait_list` -.

Linux API Linux

- ;
- .

`DEFINE_SEMAPHORE`

```
#define DEFINE_SEMAPHORE(name)  \
        struct semaphore name = __SEMAPHORE_INITIALIZER(name, 1)
```

`DEFINE_SEMAPHORE` `DEFINE_SEMAPHORE` `__SEMAPHORE_INITIALIZER`

```
#define __SEMAPHORE_INITIALIZER(name, n)            \
```

```
{                                                               \
        .lock           = __RAW_SPIN_LOCK_UNLOCKED((name).lock),        \
        .count          = n,                                    \
        .wait_list      = LIST_HEAD_INIT((name).wait_list),             \
}
```

`__SEMAPHORE_INITIALIZER` `__RAW_SPIN_LOCK_UNLOCKED` `__RAW_SPIN_LOCK_UNLOCKED` include/linux/spinlock_types.h `__ARCH_SPIN_LOCK_UNLOCKED` `__ARCH_SPIN_LOCK_UNLOCKED`

```
#define __ARCH_SPIN_LOCK_UNLOCKED       { { 0 } }
```

`count` `wait_list` `sema_init` include/linux/semaphore.h

```
static inline void sema_init(struct semaphore *sem, int val)
{
       static struct lock_class_key __key;
       *sem = (struct semaphore) __SEMAPHORE_INITIALIZER(*sem, val);
       lockdep_init_map(&sem->lock.dep_map, "semaphore->lock", &__key, 0);
}
```

`__SEMAPHORE_INITIALIZER` Linux Linux API

```
void down(struct semaphore *sem);
void up(struct semaphore *sem);
int  down_interruptible(struct semaphore *sem);
int  down_killable(struct semaphore *sem);
int  down_trylock(struct semaphore *sem);
int  down_timeout(struct semaphore *sem, long jiffies);
```

`down` `up` `down_interruptible` `TASK_INTERRUPTIBLE` `TASK_INTERRUPTIBLE`

`down_killable` `down_interruptible` `TASK_KILLABLE`

`down_trylock` `spin_trylock` `down_timeout` jiffies

API `down` kernel/locking/semaphore.c

```
void down(struct semaphore *sem)
{
        unsigned long flags;

        raw_spin_lock_irqsave(&sem->lock, flags);
        if (likely(sem->count > 0))
                sem->count--;
        else
                __down(sem);
        raw_spin_unlock_irqrestore(&sem->lock, flags);
}
EXPORT_SYMBOL(down);
```

`down` `flags` `raw_spin_lock_irqsave` `raw_spin_lock_irqrestore` include/linux/spinlock.h `spin_lock` `spin_unlock` /

`down` `raw_spin_lock_irqsave` `raw_spin_unlock_irqrestore` `__down` `__down`)

```
static noinline void __sched __down(struct semaphore *sem)
{
        __down_common(sem, TASK_UNINTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
}
```

`__down` `__down_common`

- `semaphore` ;
- `flag` -;
- `timeout` - .

`__down_common` `down_trylock` , `down_timeout` `down_killable` `__down_common`

```
static noinline int __sched __down_interruptible(struct semaphore *sem)
{
        return __down_common(sem, TASK_INTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
}
```

`__down_killable`

```
static noinline int __sched __down_killable(struct semaphore *sem)
{
        return __down_common(sem, TASK_KILLABLE, MAX_SCHEDULE_TIMEOUT);
}
```

`__down_timeout` :

```
static noinline int __sched __down_timeout(struct semaphore *sem, long timeout)
{
        return __down_common(sem, TASK_UNINTERRUPTIBLE, timeout);
}
```

`__down_common` kernel/locking/semaphore.c

```
struct task_struct *task = current;
struct semaphore_waiter waiter;
```

`current` arch/x86/include/asm/current.h

```
#define current get_current()
```

`get_current` `current_task` per-cpu

```
DECLARE_PER_CPU(struct task_struct *, current_task);

static __always_inline struct task_struct *get_current(void)
{
        return this_cpu_read_stable(current_task);
}
```

`waiter` `semaphore.wait_list`

```
struct semaphore_waiter {
        struct list_head list;
        struct task_struct *task;
        bool up;
};
```

`wait_list` `waiter`

```
list_add_tail(&waiter.list, &sem->wait_list);
waiter.task = task;
waiter.up = false;
```

```
for (;;) {
        if (signal_pending_state(state, task))
            goto interrupted;

        if (unlikely(timeout <= 0))
            goto timed_out;

        __set_task_state(task, state);

        raw_spin_unlock_irq(&sem->lock);
        timeout = schedule_timeout(timeout);
        raw_spin_lock_irq(&sem->lock);

        if (waiter.up)
            return 0;
}
```

`waiter.up` `false` `up` `true` `pending` `TASK_INTERRUPTIBLE` `TASK_WAKEKILL` (https://en.wikipedia.org/wiki/Unix_signal) `signal_pending_state` include/linux/sched.h

```
static inline int signal_pending_state(long state, struct task_struct *p)
{
         if (!(state & (TASK_INTERRUPTIBLE | TASK_WAKEKILL)))
                 return 0;
         if (!signal_pending(p))
                 return 0;

         return (state & TASK_INTERRUPTIBLE) || __fatal_signal_pending(p);
}
```

`state` `TASK_INTERRUPTIBLE` `TASK_WAKEKILL` `state` `TASK_INTERRUPTIBLE` `interrupted`

```
interrupted:
    list_del(&waiter.list);
    return -EINTR;
```

`-EINTR`

```
if (unlikely(timeout <= 0))
    goto timed_out;
```

`timed_out`

```
timed_out:
    list_del(&waiter.list);
    return -ETIME;
```

`interrupted` `-ETIME` `state`

```
__set_task_state(task, state);
```

`schedule_timeout`

```
raw_spin_unlock_irq(&sem->lock);
timeout = schedule_timeout(timeout);
raw_spin_lock_irq(&sem->lock);
```

kernel/time/timer.c `schedule_timeout`

`__down_common` `up`

`up` `down`

```
void up(struct semaphore *sem)
{
        unsigned long flags;

        raw_spin_lock_irqsave(&sem->lock, flags);
        if (likely(list_empty(&sem->wait_list)))
                sem->count++;
        else
                __up(sem);
        raw_spin_unlock_irqrestore(&sem->lock, flags);
}
EXPORT_SYMBOL(up);
```

`down` `semaphore` `__up`

```
static noinline void __sched __up(struct semaphore *sem)
{
        struct semaphore_waiter *waiter = list_first_entry(&sem->wait_list,
                                                struct semaphore_waiter, list);
        list_del(&waiter->list);
        waiter->up = true;
        wake_up_process(waiter->task);
}
```

`waiter-up` `__down_common` `wake_up_process` `__up` `__down_common` `schedule_timeout` `schedule_timeout` `schedule_timeout` kernel/sched/core.c `wake_up_process`

Linux Linux `ticket spinlock` Linux

twitter 0xAX email issue

- spinlocks
- synchronization primitive
- semaphore
- context switch
- preemption
- deadlocks
- scheduler
- Doubly linked list in the Linux kernel
- jiffies
- interrupts
- per-cpu
- bitmask
- SIGKILL
- errno
- API

- mutex
- Previous part

• •

chapter (mutex) `MUTual EXclusion`

Linux API

```
struct semaphore {
    raw_spinlock_t        lock;
    unsigned int        count;
    struct list_head    wait_list;
};
```

`count` `semaphore` API

() `API`

Linux

```
struct mutex {
        atomic_t                count;
        spinlock_t              wait_lock;
        struct list_head        wait_list;
#if defined(CONFIG_DEBUG_MUTEXES) || defined(CONFIG_MUTEX_SPIN_ON_OWNER)
        struct task_struct      *owner;
#endif
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
        struct optimistic_spin_queue osq;
#endif
#ifdef CONFIG_DEBUG_MUTEXES
        void                    *magic;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
        struct lockdep_map      dep_map;
#endif
};
```

include/linux/mutex.h - `count` `count` `1` `count`

`count`

- `wait_lock` `wait_list` Linux

- `owner` `mutex` `CONFIG_DEBUG_MUTEXES` `CONFIG_MUTEX_SPIN_ON_OWNER` `osq` `(optimistic spinning)` - `magic` `dep_map` `magic` - `lockdep_map` Linux (lock validator)

Linux `mutex->count` Linux

mutex

- `fastpath` ;
- `midpath` ;
- `slowpath` .

`fastpath` `mutex` `count` `count`

- `midpath` `midpath` MCS lock .

`fastpath` `midpath` - `slowpath`

```
struct mutex_waiter {
        struct list_head        list;
        struct task_struct      *task;
#ifdef CONFIG_DEBUG_MUTEXES
        void                    *magic;
#endif
};
```

include/linux/mutex.h Linux API `mutex_waiter` `mutex_waiter` kernel/locking/semaphore.c `semaphore_waiter`

```
struct semaphore_waiter {
        struct list_head list;
        struct task_struct *task;
        bool up;
};
```

`list` `task` `mutex_waiter` `up` `CONFIG_DEBUG_MUTEXES` `magic`

LinuxLinux API

API

Linux API API include/linux/mutex.h

Linux

```
#define DEFINE_MUTEX(mutexname) \
        struct mutex mutexname = __MUTEX_INITIALIZER(mutexname)
```

`DEFINE_MUTEX` `mutex` `__MUTEX_INITIALIZER` `__MUTEX_INITIALIZER`

```
#define __MUTEX_INITIALIZER(lockname)           \
{                                                       \
       .count = ATOMIC_INIT(1),                         \
       .wait_lock = __SPIN_LOCK_UNLOCKED(lockname.wait_lock), \
       .wait_list = LIST_HEAD_INIT(lockname.wait_list)  \
}
```

`mutex` `count` `1` `wait_lock` `wait_list`

kernel/locking/mutex.c `__mutex_init` `__mutex_init` `mutex_init` :

```
# define mutex_init(mutex)                \
do {                                                    \
        static struct lock_class_key __key;             \
                                                        \
        __mutex_init((mutex), #mutex, &__key);          \
} while (0)
```

`mutex_init` `lock_class_key` `__mutex_init` :

```
void
__mutex_init(struct mutex *lock, const char *name, struct lock_class_key *key)
{
```

```
        atomic_set(&lock->count, 1);
        spin_lock_init(&lock->wait_lock);
        INIT_LIST_HEAD(&lock->wait_list);
        mutex_clear_owner(lock);
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
        osq_lock_init(&lock->osq);
#endif
        debug_mutex_init(lock, name, key);
}
```

`__mutex_init` :

- `lock` -;
- `name` -;
- `key` - key.

`__mutex_init` `atomic_set` include/linux/osq_lock.h `osq_lock_init` (optimistic queue) tail:

```
static inline bool osq_is_locked(struct optimistic_spin_queue *lock)
{
        return atomic_read(&lock->tail) != OSQ_UNLOCKED_VAL;
}
```

`__mutex_init` `debug_mutex_init`

API `mutex_lock` `mutex_unlock` kernel/locking/mutex.c `mutex_lock` :

```
void __sched mutex_lock(struct mutex *lock)
{
        might_sleep();
        __mutex_fastpath_lock(&lock->count, __mutex_lock_slowpath);
        mutex_set_owner(lock);
}
```

include/linux/kernel.h `mutex_lock` `might_sleep` `CONFIG_DEBUG_ATOMIC_SLEEP`

`might_sleep` `__mutex_fastpath_lock` x86_64 \ `__mutex_fastpath_lock` arch/x86/include/asm/mutex_64.h `__mutex_fastpath_lock` fast path `count`

`__mutex_fastpath_lock` :

```
asm_volatile_goto(LOCK_PREFIX "   decl %0\n"
                              "   jns %l[exit]\n"
                              : : "m" (v->counter)
                              : "memory", "cc"
                              : exit);
```

`asm_volatile_goto` include/linux/compiler-gcc.h

```
#define asm_volatile_goto(x...) do { asm goto(x); asm (""); } while (0)
```

`goto` (barrier) `LOCK` lock

```
#define LOCK_PREFIX LOCK_PREFIX_HERE "\n\tlock; "
```

`mutex->counter` `mutex->counter` jns `exit` `exit` `__mutex_fastpath_lock`

```
exit:
        return;
```

`__mutex_fastpath_lock` `mutex->counter`

```
fail_fn(v);
```

`fail_fn` `__mutex_fastpath_lock` `midpath/slowpath` `fail_fn` `__mutex_lock_slowpath` `__mutex_lock_slowpath` `mutex_lock` `__mutex_fastpath_lock` mutex_lock

```
mutex_set_owner(lock);
```

`mutex_set_owner` kernel/locking/mutex.h

```
static inline void mutex_set_owner(struct mutex *lock)
{
        lock->owner = current;
}
```

`__mutex_lock_slowpath` kernel/locking/mutex.c `container_of` `__mutex_fastpath_lock`

```
__visible void __sched
__mutex_lock_slowpath(atomic_t *lock_count)
{
        struct mutex *lock = container_of(lock_count, struct mutex, count);

        __mutex_lock_common(lock, TASK_UNINTERRUPTIBLE, 0,
                            NULL, _RET_IP_, NULL, 0);
}
```

`__mutex_lock_common` `__mutex_lock_common`

```
preempt_disable();
```

`CONFIG_MUTEX_SPIN_ON_OWNER` - `slowpath`

```
if (mutex_optimistic_spin(lock, ww_ctx, use_ww_ctx)) {
        preempt_enable();
        return 0;
}
```

`mutex_optimistic_spin` (spinner) `MCS`

```
osq_lock(&lock->osq)
```

:

```
while (true) {
    owner = READ_ONCE(lock->owner);

    if (owner && !mutex_spin_on_owner(lock, owner))
        break;

    if (mutex_try_to_acquire(lock)) {
        lock_acquired(&lock->dep_map, ip);

        mutex_set_owner(lock);
        osq_unlock(&lock->osq);
        return true;
    }
```

```
}
```

() `mutex_spin_on_owner` `mutex_try_to_acquired` `MCS` `mutex_optimistic_spin` `__mutex_lock_common`

```
if (mutex_optimistic_spin(lock, ww_ctx, use_ww_ctx)) {
    preempt_enable();
    return 0;
}
```

`mutex_optimistic_spin` `mutex_optimistic_spin` `CONFIG_MUTEX_SPIN_ON_OWNER` `mutex_optimistic_spin`

```
#ifndef CONFIG_MUTEX_SPIN_ON_OWNER
static bool mutex_optimistic_spin(struct mutex *lock,
                                  struct ww_acquire_ctx *ww_ctx, const bool use_ww_ctx)
{
    return false;
}
#endif
```

`__mutex_lock_common`

```
if (!mutex_is_locked(lock) &&
   (atomic_xchg_acquire(&lock->count, 0) == 1))
      goto skip_wait;
```

```
list_add_tail(&waiter.list, &lock->wait_list);
waiter.task = task;
```

`__mutex_lock_common`

```
skip_wait:
        mutex_set_owner(lock);
        preempt_enable();
        return 0;
```

```
for (;;) {

    if (atomic_read(&lock->count) >= 0 && (atomic_xchg_acquire(&lock->count, -1) == 1))
        break;

    if (unlikely(signal_pending_state(state, task))) {
        ret = -EINTR;
        goto err;
    }

    __set_task_state(task, state);

     schedule_preempt_disabled();
}
```

pending `TASK_UNINTERRUPTIBLE` `schedule_preempt_disabled`

`mutex_unlock` `mutex_unlock` `__mutex_fastpath_unlock` arch/x86/include/asm/mutex_64.h

```
void __sched mutex_unlock(struct mutex *lock)
{
    __mutex_fastpath_unlock(&lock->count, __mutex_unlock_slowpath);
}
```

`__mutex_fastpath_unlock` `__mutex_fastpath_lock`

```
static inline void __mutex_fastpath_unlock(atomic_t *v,
                                           void (*fail_fn)(atomic_t *))
{
       asm_volatile_goto(LOCK_PREFIX "   incl %0\n"
                         "   jg %l[exit]\n"
                         : : "m" (v->counter)
                         : "memory", "cc"
                         : exit);
       fail_fn(v);
exit:
       return;
}
```

, `mutex->count` `fail_fn`
`__mutex_unlock_slowpath` `__mutex_unlock_slowpath` `mutex->count` `mutex` `__mutex_unlock_common_slowpath`

```
__mutex_unlock_slowpath(atomic_t *lock_count)
{
      struct mutex *lock = container_of(lock_count, struct mutex, count);

      __mutex_unlock_common_slowpath(lock, 1);
}
```

`__mutex_unlock_common_slowpath`

```
if (!list_empty(&lock->wait_list)) {
    struct mutex_waiter *waiter =
           list_entry(lock->wait_list.next, struct mutex_waiter, list);
                wake_up_process(waiter->task);
}
```

. API `mutex_lock` `mutex_unlock` Linux API

- `mutex_lock_interruptible` ;
- `mutex_lock_killable` ;
- `mutex_trylock` .

`unlock` . `API` , `API`

Linux - Linux Linux

twitter 0xAX email issue

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-insides.

- Mutex
- Spinlock
- Semaphore
- Synchronization primitives
- API
- Locking mechanism
- Context switches
- lock validator
- Atomic
- MCS lock
- Doubly linked list
- x86_64
- Inline assembly
- Memory barrier
- Lock instruction
- JNS instruction
- preemption
- Unix signals
- Previous part

Synchronization primitives in the Linux kernel. Part 5.

Introduction

This is the fifth part of the chapter which describes synchronization primitives in the Linux kernel and in the previous parts we finished to consider different types spinlocks, semaphore and mutex synchronization primitives. We will continue to learn synchronization primitives in this part and start to consider special type of synchronization primitives - readers–writer lock.

The first synchronization primitive of this type will be already familiar for us - semaphore. As in all previous parts of this book, before we will consider implementation of the `reader/writer semaphores` in the Linux kernel, we will start from the theoretical side and will try to understand what is the difference between `reader/writer semaphores` and `normal semaphores`.

So, let's start.

Reader/Writer semaphore

Actually there are two types of operations may be performed on the data. We may read data and make changes in data. Two fundamental operations - `read` and `write`. Usually (but not always), `read` operation is performed more often than `write` operation. In this case, it would be logical to we may lock data in such way, that some processes may read locked data in one time, on condition that no one will not change the data. The readers/writer lock allows us to get this lock.

When a process which wants to write something into data, all other `writer` and `reader` processes will be blocked until the process which acquired a lock, will not release it. When a process reads data, other processes which want to read the same data too, will not be locked and will be able to do this. As you may guess, implementation of the `reader/writer semaphore` is based on the implementation of the `normal semaphore`. We already familiar with the semaphore synchronization primitive from the third part of this chapter. From the theoretical side everything looks pretty simple. Let's look how `reader/writer semaphore` is represented in the Linux kernel.

The `semaphore` is represented by the:

```
struct semaphore {
    raw_spinlock_t        lock;
    unsigned int          count;
    struct list_head      wait_list;
};
```

structure. If you will look in the include/linux/rwsem.h header file, you will find definition of the `rw_semaphore` structure which represents `reader/writer semaphore` in the Linux kernel. Let's look at the definition of this structure:

```
#ifdef CONFIG_RWSEM_GENERIC_SPINLOCK
#include <linux/rwsem-spinlock.h>
#else
struct rw_semaphore {
        long count;
        struct list_head wait_list;
        raw_spinlock_t wait_lock;
#ifdef CONFIG_RWSEM_SPIN_ON_OWNER
        struct optimistic_spin_queue osq;
        struct task_struct *owner;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
        struct lockdep_map      dep_map;
#endif
};
```

Before we will consider fields of the `rw_semaphore` structure, we may notice, that declaration of the `rw_semaphore` structure depends on the `CONFIG_RWSEM_GENERIC_SPINLOCK` kernel configuration option. This option is disabled for the x86_64 architecture by default. We can be sure in this by looking at the corresponding kernel configuration file. In our case, this configuration file is - arch/x86/um/Kconfig:

```
config RWSEM_XCHGADD_ALGORITHM
    def_bool 64BIT

config RWSEM_GENERIC_SPINLOCK
    def_bool !RWSEM_XCHGADD_ALGORITHM
```

So, as this book describes only x86_64 architecture related stuff, we will skip the case when the `CONFIG_RWSEM_GENERIC_SPINLOCK` kernel configuration is enabled and consider definition of the `rw_semaphore` structure only from the include/linux/rwsem.h header file.

If we will take a look at the definition of the `rw_semaphore` structure, we will notice that first three fields are the same that in the `semaphore` structure. It contains `count` field which represents amount of available resources, the `wait_list` field which represents doubly linked list of processes which are waiting to acquire a lock and `wait_lock` spinlock for protection of this list. Notice that `rw_semaphore.count` field is `long` type unlike the same field in the `semaphore` structure.

The `count` field of a `rw_semaphore` structure may have following values:

- `0x0000000000000000` - `reader/writer semaphore` is in unlocked state and no one is waiting for a lock;
- `0x000000000000000X` - `X` readers are active or attempting to acquire a lock and no writer waiting;
- `0xffffffff0000000X` - may represent different cases. The first is - `X` readers are active or attempting to acquire a lock with waiters for the lock. The second is - one writer attempting a lock, no waiters for the lock. And the last - one writer is active and no waiters for the lock;
- `0xffffffff00000001` - may represented two different cases. The first is - one reader is active or attempting to acquire a lock and exist waiters for the lock. The second case is one writer is active or attempting to acquire a lock and no waiters for the lock;
- `0xffffffff00000000` - represents situation when there are readers or writers are queued, but no one is active or is in the process of acquire of a lock;
- `0xfffffffe00000001` - a writer is active or attempting to acquire a lock and waiters are in queue.

So, besides the `count` field, all of these fields are similar to fields of the `semaphore` structure. Last three fields depend on the two configuration options of the Linux kernel: the `CONFIG_RWSEM_SPIN_ON_OWNER` and `CONFIG_DEBUG_LOCK_ALLOC`. The first two fields may be familiar us by declaration of the mutex structure from the previous part. The first `osq` field represents MCS lock spinner for `optimistic spinning` and the second represents process which is current owner of a lock.

The last field of the `rw_semaphore` structure is - `dep_map` - debugging related, and as I already wrote in previous parts, we will skip debugging related stuff in this chapter.

That's all. Now we know a little about what is it `reader/writer lock` in general and `reader/writer semaphore` in particular. Additionally we saw how a `reader/writer semaphore` is represented in the Linux kernel. In this case, we may go ahead and start to look at the API which the Linux kernel provides for manipulation of `reader/writer semaphores`.

Reader/Writer semaphore API

So, we know a little about `reader/writer semaphores` from theoretical side, let's look on its implementation in the Linux kernel. All `reader/writer semaphores` related API is located in the include/linux/rwsem.h header file.

As always Before we will consider an API of the `reader/writer semaphore` mechanism in the Linux kernel, we need to know how to initialize the `rw_semaphore` structure. As we already saw in previous parts of this chapter, all synchronization primitives may be initialized in two ways:

- `statically`;
- `dynamically`.

And `reader/writer semaphore` is not an exception. First of all, let's take a look at the first approach. We may initialize `rw_semaphore` structure with the help of the `DECLARE_RWSEM` macro in compile time. This macro is defined in the include/linux/rwsem.h header file and looks:

```
#define DECLARE_RWSEM(name) \
        struct rw_semaphore name = __RWSEM_INITIALIZER(name)
```

As we may see, the `DECLARE_RWSEM` macro just expands to the definition of the `rw_semaphore` structure with the given name. Additionally new `rw_semaphore` structure is initialized with the value of the `__RWSEM_INITIALIZER` macro:

```
#define __RWSEM_INITIALIZER(name)              \
{                                                       \
        .count = RWSEM_UNLOCKED_VALUE,                  \
        .wait_list = LIST_HEAD_INIT((name).wait_list),  \
        .wait_lock = __RAW_SPIN_LOCK_UNLOCKED(name.wait_lock)  \
         __RWSEM_OPT_INIT(name)                         \
         __RWSEM_DEP_MAP_INIT(name)
}
```

and expands to the initialization of fields of `rw_semaphore` structure. First of all we initialize `count` field of the `rw_semaphore` structure to the `unlocked` state with `RWSEM_UNLOCKED_VALUE` macro from the arch/x86/include/asm/rwsem.h architecture specific header file:

```
#define RWSEM_UNLOCKED_VALUE            0x00000000L
```

After this we initialize list of a lock waiters with the empty linked list and spinlock for protection of this list with the `unlocked` state too. The `__RWSEM_OPT_INIT` macro depends on the state of the `CONFIG_RWSEM_SPIN_ON_OWNER` kernel configuration option and if this option is enabled it expands to the initialization of the `osq` and `owner` fields of the `rw_semaphore` structure. As we already saw above, the `CONFIG_RWSEM_SPIN_ON_OWNER` kernel configuration option is enabled by default for x86_64 architecture, so let's take a look at the definition of the `__RWSEM_OPT_INIT` macro:

```
#ifdef CONFIG_RWSEM_SPIN_ON_OWNER
    #define __RWSEM_OPT_INIT(lockname) , .osq = OSQ_LOCK_UNLOCKED, .owner = NULL
#else
    #define __RWSEM_OPT_INIT(lockname)
#endif
```

As we may see, the `__RWSEM_OPT_INIT` macro initializes the MCS lock lock with `unlocked` state and initial `owner` of a lock with `NULL`. From this moment, a `rw_semaphore` structure will be initialized in a compile time and may be used for data protection.

The second way to initialize a `rw_semaphore` structure is `dynamically` or use the `init_rwsem` macro from the include/linux/rwsem.h header file. This macro declares an instance of the `lock_class_key` which is related to the lock validator of the Linux kernel and to the call of the `__init_rwsem` function with the given `reader/writer semaphore`:

```
#define init_rwsem(sem)                         \
do {                                                    \
        static struct lock_class_key __key;             \
                                                        \
        __init_rwsem((sem), #sem, &__key);              \
} while (0)
```

If you will start definition of the `__init_rwsem` function, you will notice that there are couple of source code files which contain it. As you may guess, sometimes we need to initialize additional fields of the `rw_semaphore` structure, like the `osq` and `owner`. But sometimes not. All of this depends on some kernel configuration options. If we will look at the kernel/locking/Makefile makefile, we will see following lines:

```
obj-$(CONFIG_RWSEM_GENERIC_SPINLOCK) += rwsem-spinlock.o
obj-$(CONFIG_RWSEM_XCHGADD_ALGORITHM) += rwsem-xadd.o
```

As we already know, the Linux kernel for `x86_64` architecture has enabled `CONFIG_RWSEM_XCHGADD_ALGORITHM` kernel configuration option by default:

```
config RWSEM_XCHGADD_ALGORITHM
    def_bool 64BIT
```

in the arch/x86/um/Kconfig kernel configuration file. In this case, implementation of the `__init_rwsem` function will be located in the kernel/locking/rwsem-xadd.c source code file for us. Let's take a look at this function:

```
void __init_rwsem(struct rw_semaphore *sem, const char *name,
                  struct lock_class_key *key)
{
#ifdef CONFIG_DEBUG_LOCK_ALLOC
        debug_check_no_locks_freed((void *)sem, sizeof(*sem));
        lockdep_init_map(&sem->dep_map, name, key, 0);
#endif
        sem->count = RWSEM_UNLOCKED_VALUE;
        raw_spin_lock_init(&sem->wait_lock);
        INIT_LIST_HEAD(&sem->wait_list);
#ifdef CONFIG_RWSEM_SPIN_ON_OWNER
        sem->owner = NULL;
        osq_lock_init(&sem->osq);
#endif
}
```

We may see here almost the same as in `__RWSEM_INITIALIZER` macro with difference that all of this will be executed in runtime.

So, from now we are able to initialize a `reader/writer semaphore` let's look at the `lock` and `unlock` API. The Linux kernel provides following primary API to manipulate `reader/writer semaphores` :

- `void down_read(struct rw_semaphore *sem)` - lock for reading;
- `int down_read_trylock(struct rw_semaphore *sem)` - try lock for reading;
- `void down_write(struct rw_semaphore *sem)` - lock for writing;
- `int down_write_trylock(struct rw_semaphore *sem)` - try lock for writing;
- `void up_read(struct rw_semaphore *sem)` - release a read lock;
- `void up_write(struct rw_semaphore *sem)` - release a write lock;

Let's start as always from the locking. First of all let's consider implementation of the `down_write` function which executes a try of acquiring of a lock for `write` . This function is kernel/locking/rwsem.c source code file and starts from the call of the macro from the include/linux/kernel.h header file:

```
void __sched down_write(struct rw_semaphore *sem)
{
        might_sleep();
        rwsem_acquire(&sem->dep_map, 0, 0, _RET_IP_);

        LOCK_CONTENDED(sem, __down_write_trylock, __down_write);
        rwsem_set_owner(sem);
}
```

We already met the `might_sleep` macro in the previous part. In short words, Implementation of the `might_sleep` macro depends on the `CONFIG_DEBUG_ATOMIC_SLEEP` kernel configuration option and if this option is enabled, this macro just prints a stack trace if it was executed in atomic context. As this macro is mostly for debugging purpose we will skip it and will go ahead. Additionally we will skip the next macro from the `down_read` function - `rwsem_acquire` which is related to the lock validator of the Linux kernel, because this is topic of other part.

The only two things that remained in the `down_write` function is the call of the `LOCK_CONTENDED` macro which is defined in the include/linux/lockdep.h header file and setting of owner of a lock with the `rwsem_set_owner` function which sets owner to currently running process:

```
static inline void rwsem_set_owner(struct rw_semaphore *sem)
{
        sem->owner = current;
}
```

As you already may guess, the `LOCK_CONTENDED` macro does all job for us. Let's look at the implementation of the `LOCK_CONTENDED` macro:

```
#define LOCK_CONTENDED(_lock, try, lock) \
        lock(_lock)
```

As we may see it just calls the `lock` function which is third parameter of the `LOCK_CONTENDED` macro with the given `rw_semaphore`. In our case the third parameter of the `LOCK_CONTENDED` macro is the `__down_write` function which is architecture specific function and located in the arch/x86/include/asm/rwsem.h header file. Let's look at the implementation of the `__down_write` function:

```
static inline void __down_write(struct rw_semaphore *sem)
{
        __down_write_nested(sem, 0);
}
```

which just executes a call of the `__down_write_nested` function from the same source code file. Let's take a look at the implementation of the `__down_write_nested` function:

```
static inline void __down_write_nested(struct rw_semaphore *sem, int subclass)
{
        long tmp;

        asm volatile("# beginning down_write\n\t"
                     LOCK_PREFIX "  xadd      %1,(%2)\n\t"
                     "  test " __ASM_SEL(%w1,%k1) "," __ASM_SEL(%w1,%k1) "\n\t"
                     "  jz        1f\n"
                     "  call call_rwsem_down_write_failed\n"
                     "1:\n"
                     "# ending down_write"
                     : "+m" (sem->count), "=d" (tmp)
                     : "a" (sem), "1" (RWSEM_ACTIVE_WRITE_BIAS)
                     : "memory", "cc");
}
```

As for other synchronization primitives which we saw in this chapter, usually `lock/unlock` functions consists only from an inline assembly statement. As we may see, in our case the same for `__down_write_nested` function. Let's try to understand what does this function do. The first line of our assembly statement is just a comment, let's skip it. The second like contains `LOCK_PREFIX` which will be expanded to the LOCK instruction as we already know. The next xadd instruction executes `add` and `exchange` operations. In other words, `xadd` instruction adds value of the `RWSEM_ACTIVE_WRITE_BIAS`:

```
#define RWSEM_ACTIVE_WRITE_BIAS         (RWSEM_WAITING_BIAS + RWSEM_ACTIVE_BIAS)

#define RWSEM_WAITING_BIAS              (-RWSEM_ACTIVE_MASK-1)
#define RWSEM_ACTIVE_BIAS               0x00000001L
```

or `0xffffffff00000001` to the `count` of the given `reader/writer semaphore` and returns previous value of it. After this we check the active mask in the `rw_semaphore->count`. If it was zero before, this means that there were no-one writer before, so we acquired a lock. In other way we call the `call_rwsem_down_write_failed` function from the arch/x86/lib/rwsem.S assembly file. The the `call_rwsem_down_write_failed` function just calls the `rwsem_down_write_failed` function from the kernel/locking/rwsem-xadd.c source code file anticipatorily save general purpose registers:

```
ENTRY(call_rwsem_down_write_failed)
    FRAME_BEGIN
    save_common_regs
```

```
    movq %rax,%rdi
    call rwsem_down_write_failed
    restore_common_regs
    FRAME_END
    ret
    ENDPROC(call_rwsem_down_write_failed)
```

The `rwsem_down_write_failed` function starts from the atomic update of the `count` value:

```
 __visible
struct rw_semaphore __sched *rwsem_down_write_failed(struct rw_semaphore *sem)
{
    count = rwsem_atomic_update(-RWSEM_ACTIVE_WRITE_BIAS, sem);
    ...
    ...
    ...
}
```

with the `-RWSEM_ACTIVE_WRITE_BIAS` value. The `rwsem_atomic_update` function is defined in the arch/x86/include/asm/rwsem.h header file and implement exchange and add logic:

```
static inline long rwsem_atomic_update(long delta, struct rw_semaphore *sem)
{
        return delta + xadd(&sem->count, delta);
}
```

This function atomically adds the given delta to the `count` and returns old value of the count. After this it just returns sum of the given `delta` and old value of the `count` field. In our case we undo write bias from the `count` as we didn't acquire a lock. After this step we try to do `optimistic spinning` by the call of the `rwsem_optimistic_spin` function:

```
if (rwsem_optimistic_spin(sem))
      return sem;
```

We will skip implementation of the `rwsem_optimistic_spin` function, as it is similar on the `mutex_optimistic_spin` function which we saw in the previous part. In short words we check existence other tasks ready to run that have higher priority in the `rwsem_optimistic_spin` function. If there are such tasks, the process will be added to the MCS `waitqueue` and start to spin in the loop until a lock will be able to be acquired. If `optimistic spinning` is disabled, a process will be added to the and marked as waiting for write:

```
waiter.task = current;
waiter.type = RWSEM_WAITING_FOR_WRITE;

if (list_empty(&sem->wait_list))
    waiting = false;

list_add_tail(&waiter.list, &sem->wait_list);
```

waiters list and start to wait until it will successfully acquire the lock. After we have added a process to the waiters list which was empty before this moment, we update the value of the `rw_semaphore->count` with the `RWSEM_WAITING_BIAS`:

```
count = rwsem_atomic_update(RWSEM_WAITING_BIAS, sem);
```

with this we mark `rw_semaphore->counter` that it is already locked and exists/waits one `writer` which wants to acquire the lock. In other way we try to wake `reader` processes from the `wait queue` that were queued before this `writer` process and there are no active readers. In the end of the `rwsem_down_write_failed` a `writer` process will go to sleep which didn't acquire a lock in the following loop:

```
while (true) {
    if (rwsem_try_write_lock(count, sem))
```

```
            break;
        raw_spin_unlock_irq(&sem->wait_lock);
        do {
            schedule();
            set_current_state(TASK_UNINTERRUPTIBLE);
        } while ((count = sem->count) & RWSEM_ACTIVE_MASK);
        raw_spin_lock_irq(&sem->wait_lock);
    }
```

I will skip explanation of this loop as we already met similar functional in the previous part.

That's all. From this moment, our `writer` process will acquire or not acquire a lock depends on the value of the `rw_semaphore->count` field. Now if we will look at the implementation of the `down_read` function which executes a try of acquiring of a lock. We will see similar actions which we saw in the `down_write` function. This function calls different debugging and lock validator related functions/macros:

```
void __sched down_read(struct rw_semaphore *sem)
{
        might_sleep();
        rwsem_acquire_read(&sem->dep_map, 0, 0, _RET_IP_);

        LOCK_CONTENDED(sem, __down_read_trylock, __down_read);
}
```

and does all job in the `__down_read` function. The `__down_read` consists of inline assembly statement:

```
static inline void __down_read(struct rw_semaphore *sem)
{
         asm volatile("# beginning down_read\n\t"
                      LOCK_PREFIX _ASM_INC "(%1)\n\t"
                      "  jns        1f\n"
                      "  call call_rwsem_down_read_failed\n"
                      "1:\n\t"
                      "# ending down_read\n\t"
                      : "+m" (sem->count)
                      : "a" (sem)
                      : "memory", "cc");
}
```

which increments value of the given `rw_semaphore->count` and call the `call_rwsem_down_read_failed` if this value is negative. In other way we jump at the label `1:` and exit. After this `read` lock will be successfully acquired. Notice that we check a sign of the `count` value as it may be negative, because as you may remember most significant word of the `rw_semaphore->count` contains negated number of active writers.

Let's consider case when a process wants to acquire a lock for `read` operation, but it is already locked. In this case the `call_rwsem_down_read_failed` function from the arch/x86/lib/rwsem.S assembly file will be called. If you will look at the implementation of this function, you will notice that it does the same that `call_rwsem_down_read_failed` function does. Except it calls the `rwsem_down_read_failed` function instead of `rwsem_dow_write_failed`. Now let's consider implementation of the `rwsem_down_read_failed` function. It starts from the adding a process to the `wait queue` and updating of value of the `rw_semaphore->counter`:

```
long adjustment = -RWSEM_ACTIVE_READ_BIAS;

waiter.task = tsk;
waiter.type = RWSEM_WAITING_FOR_READ;

if (list_empty(&sem->wait_list))
    adjustment += RWSEM_WAITING_BIAS;
list_add_tail(&waiter.list, &sem->wait_list);

count = rwsem_atomic_update(adjustment, sem);
```

Notice that if the `wait queue` was empty before we clear the `rw_semaphore->counter` and undo `read` bias in other way. At the next step we check that there are no active locks and we are first in the `wait queue` we need to join currently active `reader` processes. In other way we go to sleep until a lock will not be able to acquired.

That's all. Now we know how `reader` and `writer` processes will behave in different cases during a lock acquisition. Now let's take a short look at `unlock` operations. The `up_read` and `up_write` functions allows us to unlock a `reader` or `writer` lock. First of all let's take a look at the implementation of the `up_write` function which is defined in the kernel/locking/rwsem.c source code file:

```
void up_write(struct rw_semaphore *sem)
{
        rwsem_release(&sem->dep_map, 1, _RET_IP_);

        rwsem_clear_owner(sem);
        __up_write(sem);
}
```

First of all it calls the `rwsem_release` macro which is related to the lock validator of the Linux kernel, so we will skip it now. And at the next line the `rwsem_clear_owner` function which as you may understand from the name of this function, just clears the `owner` field of the given `rw_semaphore` :

```
static inline void rwsem_clear_owner(struct rw_semaphore *sem)
{
    sem->owner = NULL;
}
```

The `__up_write` function does all job of unlocking of the lock. The `_up_write` is architecture-specific function, so for our case it will be located in the arch/x86/include/asm/rwsem.h source code file. If we will take a look at the implementation of this function, we will see that it does almost the same that `__down_write` function, but conversely. Instead of adding of the `RWSEM_ACTIVE_WRITE_BIAS` to the `count` , we subtract the same value and check the `sign` of the previous value.

If the previous value of the `rw_semaphore->count` is not negative, a writer process released a lock and now it may be acquired by someone else. In other case, the `rw_semaphore->count` will contain negative values. This means that there is at least one `writer` in a wait queue. In this case the `call_rwsem_wake` function will be called. This function acts like similar functions which we already saw above. It store general purpose registers at the stack for preserving and call the `rwsem_wake` function.

First of all the `rwsem_wake` function checks if a spinner is present. In this case it will just acquire a lock which is just released by lock owner. In other case there must be someone in the `wait queue` and we need to wake or writer process if it exists at the top of the `wait queue` or all `reader` processes. The `up_read` function which release a `reader` lock acts in similar way like `up_write` , but with a little difference. Instead of subtracting of `RWSEM_ACTIVE_WRITE_BIAS` from the `rw_semaphore->count` , it subtracts `1` from it, because less significant word of the `count` contains number active locks. After this it checks `sign` of the `count` and calls the `rwsem_wake` like `__up_write` if the `count` is negative or in other way lock will be successfully released.

That's all. We have considered API for manipulation with `reader/writer semaphore` : `up_read/up_write` and `down_read/down_write` . We saw that the Linux kernel provides additional API, besides this functions, like the `,` and etc. But I will not consider implementation of these function in this part because it must be similar on that we have seen in this part of except few subtleties.

Conclusion

This is the end of the fifth part of the synchronization primitives chapter in the Linux kernel. In this part we met with special type of `semaphore` - `readers/writer` semaphore which provides access to data for multiply process to read or for one process to writer. In the next part we will continue to dive into synchronization primitives in the Linux kernel.

If you have questions or suggestions, feel free to ping me in twitter 0xAX, drop me email or just create issue.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-insides.

Links

- Synchronization primitives
- Readers/Writer lock
- Spinlocks
- Semaphore
- Mutex
- x86_64 architecture
- Doubly linked list
- MCS lock
- API
- Linux kernel lock validator
- Atomic operations
- Inline assembly
- XADD instruction
- LOCK instruction
- Previous part

Synchronization primitives in the Linux kernel. Part 6.

Introduction

This is the sixth part of the chapter which describes synchronization primitives) in the Linux kernel and in the previous parts we finished to consider different readers-writer lock synchronization primitives. We will continue to learn synchronization primitives in this part and start to consider a similar synchronization primitive which can be used to avoid the `writer starvation` problem. The name of this synchronization primitive is - `seqlock` or `sequential locks` .

We know from the previous part that readers-writer lock is a special lock mechanism which allows concurrent access for read-only operations, but an exclusive lock is needed for writing or modifying data. As we may guess, it may lead to a problem which is called `writer starvation` . In other words, a writer process can't acquire a lock as long as at least one reader process which aqcuired a lock holds it. So, in the situation when contention is high, it will lead to situation when a writer process which wants to acquire a lock will wait for it for a long time.

The `seqlock` synchronization primitive can help solve this problem.

As in all previous parts of this book, we will try to consider this synchronization primitive from the theoretical side and only than we will consider API provided by the Linux kernel to manipulate with `seqlocks` .

So, let's start.

Sequential lock

So, what is a `seqlock` synchronization primitive and how does it work? Let's try to answer on these questions in this paragraph. Actually `sequential locks` were introduced in the Linux kernel 2.6.x. Main point of this synchronization primitive is to provide fast and lock-free access to shared resources. Since the heart of `sequential lock` synchronization primitive is spinlock synchronization primitive, `sequential locks` work in situations where the protected resources are small and simple. Additionally write access must be rare and also should be fast.

Work of this synchronization primitive is based on the sequence of events counter. Actually a `sequential lock` allows free access to a resource for readers, but each reader must check existence of conflicts with a writer. This synchronization primitive introduces a special counter. The main algorithm of work of `sequential locks` is simple: Each writer which acquired a sequential lock increments this counter and additionally acquires a spinlock. When this writer finishes, it will release the acquired spinlock to give access to other writers and increment the counter of a sequential lock again.

Read only access works on the following principle, it gets the value of a `sequential lock` counter before it will enter into critical section and compares it with the value of the same `sequential lock` counter at the exit of critical section. If their values are equal, this means that there weren't writers for this period. If their values are not equal, this means that a writer has incremented the counter during the critical section. This conflict means that reading of protected data must be repeated.

That's all. As we may see principle of work of `sequential locks` is simple.

```
unsigned int seq_counter_value;

do {
    seq_counter_value = get_seq_counter_val(&the_lock);
    //
    // do as we want here
    //
} while (__retry__);
```

Actually the Linux kernel does not provide `get_seq_counter_val()` function. Here it is just a stub. Like a `__retry__` too. As I already wrote above, we will see actual the API for this in the next paragraph of this part.

Ok, now we know what a `seqlock` synchronization primitive is and how it is represented in the Linux kernel. In this case, we may go ahead and start to look at the API which the Linux kernel provides for manipulation of synchronization primitives of this type.

Sequential lock API

So, now we know a little about `sequentional lock` synchronization primitive from theoretical side, let's look at its implementation in the Linux kernel. All `sequentional locks` API are located in the include/linux/seqlock.h header file.

First of all we may see that the a `sequential lock` machanism is represented by the following type:

```
typedef struct {
    struct seqcount seqcount;
    spinlock_t lock;
} seqlock_t;
```

As we may see the `seqlock_t` provides two fields. These fields represent a sequential lock counter, description of which we saw above and also a spinlock which will protect data from other writers. Note that the `seqcount` counter represented as `seqcount` type. The `seqcount` is structure:

```
typedef struct seqcount {
    unsigned sequence;
#ifdef CONFIG_DEBUG_LOCK_ALLOC
    struct lockdep_map dep_map;
#endif
} seqcount_t;
```

which holds counter of a sequential lock and lock validator related field.

As always in previous parts of this chapter, before we will consider an API of `sequential lock` mechanism in the Linux kernel, we need to know how to initialize an instance of `seqlock_t` .

We saw in the previous parts that often the Linux kernel provides two approaches to execute initialization of the given synchronization primitive. The same situation with the `seqlock_t` structure. These approaches allows to initialize a `seqlock_t` in two following:

- `statically` ;
- `dynamically` .

ways. Let's look at the first approach. We are able to intialize a `seqlock_t` statically with the `DEFINE_SEQLOCK` macro:

```
#define DEFINE_SEQLOCK(x) \
        seqlock_t x = __SEQLOCK_UNLOCKED(x)
```

which is defined in the include/linux/seqlock.h header file. As we may see, the `DEFINE_SEQLOCK` macro takes one argument and expands to the definition and initialization of the `seqlock_t` structure. Initialization occurs with the help of the `__SEQLOCK_UNLOCKED` macro which is defined in the same source code file. Let's look at the implementation of this macro:

```
#define __SEQLOCK_UNLOCKED(lockname)           \
    {                                  \
        .seqcount = SEQCNT_ZERO(lockname),    \
        .lock =    __SPIN_LOCK_UNLOCKED(lockname)    \
    }
```

As we may see the, `__SEQLOCK_UNLOCKED` macro executes initialization of fields of the given `seqlock_t` structure. The first field is `seqcount` initialized with the `SEQCNT_ZERO` macro which expands to the:

```
#define SEQCNT_ZERO(lockname) { .sequence = 0, SEQCOUNT_DEP_MAP_INIT(lockname)}
```

So we just initialize counter of the given sequential lock to zero and additionally we can see lock validator related initialization which depends on the state of the `CONFIG_DEBUG_LOCK_ALLOC` kernel configuration option:

```
#ifdef CONFIG_DEBUG_LOCK_ALLOC
# define SEQCOUNT_DEP_MAP_INIT(lockname) \
    .dep_map = { .name = #lockname } \
    ...
    ...
    ...
#else
# define SEQCOUNT_DEP_MAP_INIT(lockname)
    ...
    ...
    ...
#endif
```

As I already wrote in previous parts of this chapter we will not consider debugging and lock validator related stuff in this part. So for now we just skip the `SEQCOUNT_DEP_MAP_INIT` macro. The second field of the given `seqlock_t` is `lock` initialized with the `__SPIN_LOCK_UNLOCKED` macro which is defined in the include/linux/spinlock_types.h header file. We will not consider implementation of this macro here as it just initialize rawspinlock with architecture-specific methods (More abot spinlocks you may read in first parts of this chapter).

We have considered the first way to initialize a sequential lock. Let's consider second way to do the same, but do it dynamically. We can initialize a sequentional lock with the `seqlock_init` macro which is defined in the same include/linux/seqlock.h header file.

Let's look at the implementation of this macro:

```
#define seqlock_init(x)                    \
    do {                        \
        seqcount_init(&(x)->seqcount);        \
        spin_lock_init(&(x)->lock);        \
    } while (0)
```

As we may see, the `seqlock_init` expands into two macros. The first macro `seqcount_init` takes counter of the given sequential lock and expands to the call of the `__seqcount_init` function:

```
# define seqcount_init(s)                \
    do {                        \
        static struct lock_class_key __key;    \
        __seqcount_init((s), #s, &__key);    \
    } while (0)
```

from the same header file. This function

```
static inline void __seqcount_init(seqcount_t *s, const char *name,
                     struct lock_class_key *key)
{
    lockdep_init_map(&s->dep_map, name, key, 0);
    s->sequence = 0;
}
```

just initializes counter of the given `seqcount_t` with zero. The second call from the `seqlock_init` macro is the call of the `spin_lock_init` macro which we saw in the first part of this chapter.

So, now we know how to initialize a `sequential lock`, now let's look at how to use it. The Linux kernel provides following API to manipulate `sequential locks`:

```
static inline unsigned read_seqbegin(const seqlock_t *sl);
static inline unsigned read_seqretry(const seqlock_t *sl, unsigned start);
static inline void write_seqlock(seqlock_t *sl);
static inline void write_sequnlock(seqlock_t *sl);
static inline void write_seqlock_irq(seqlock_t *sl);
```

```
static inline void write_sequnlock_irq(seqlock_t *sl);
static inline void read_seqlock_excl(seqlock_t *sl)
static inline void read_sequnlock_excl(seqlock_t *sl)
```

and others. Before we move on to considering the implementation of this API, we must know that actually there are two types of readers. The first type of reader never blocks a writer process. In this case writer will not wait for readers. The second type of reader which can lock. In this case, the locking reader will block the writer as it will wait while reader will not release its lock.

First of all let's consider the first type of readers. The `read_seqbegin` function begins a seq-read critical section.

As we may see this function just returns value of the `read_seqcount_begin` function:

```
static inline unsigned read_seqbegin(const seqlock_t *sl)
{
    return read_seqcount_begin(&sl->seqcount);
}
```

In its turn the `read_seqcount_begin` function calls the `raw_read_seqcount_begin` function:

```
static inline unsigned read_seqcount_begin(const seqcount_t *s)
{
    return raw_read_seqcount_begin(s);
}
```

which just returns value of the `sequential lock` counter:

```
static inline unsigned raw_read_seqcount(const seqcount_t *s)
{
    unsigned ret = READ_ONCE(s->sequence);
    smp_rmb();
    return ret;
}
```

After we have the initial value of the given `sequential lock` counter and did some stuff, we know from the previous paragraph of this function, that we need to compare it with the current value of the counter the same `sequential lock` before we will exit from the critical section. We can achieve this by the call of the `read_seqretry` function. This function takes a `sequential lock`, start value of the counter and through a chain of functions:

```
static inline unsigned read_seqretry(const seqlock_t *sl, unsigned start)
{
    return read_seqcount_retry(&sl->seqcount, start);
}

static inline int read_seqcount_retry(const seqcount_t *s, unsigned start)
{
    smp_rmb();
    return __read_seqcount_retry(s, start);
}
```

it calls the `__read_seqcount_retry` function:

```
static inline int __read_seqcount_retry(const seqcount_t *s, unsigned start)
{
    return unlikely(s->sequence != start);
}
```

which just compares value of the counter of the given `sequential lock` with the initial value of this counter. If the initial value of the counter which is obtained from `read_seqbegin()` function is odd, this means that a writer was in the middle of updating the data when our reader began to act. In this case the value of the data can be in inconsistent state, so we need to try to read it again.

This is a common pattern in the Linux kernel. For example, you may remember the `jiffies` concept from the first part of the timers and time management in the Linux kernel chapter. The sequential lock is used to obtain value of `jiffies` at x86_64 architecture:

```
u64 get_jiffies_64(void)
{
    unsigned long seq;
    u64 ret;

    do {
        seq = read_seqbegin(&jiffies_lock);
        ret = jiffies_64;
    } while (read_seqretry(&jiffies_lock, seq));
    return ret;
}
```

Here we just read the value of the counter of the `jiffies_lock` sequential lock and then we write value of the `jiffies_64` system variable to the `ret` . As here we may see `do/while` loop, the body of the loop will be executed at least one time. So, as the body of loop was executed, we read and compare the current value of the counter of the `jiffies_lock` with the initial value. If these values are not equal, execution of the loop will be repeated, else `get_jiffies_64` will return its value in `ret` .

We just saw the first type of readers which do not block writer and other readers. Let's consider second type. It does not update value of a `sequential lock` counter, but just locks `spinlock` :

```
static inline void read_seqlock_excl(seqlock_t *sl)
{
    spin_lock(&sl->lock);
}
```

So, no one reader or writer can't access protected data. When a reader finishes, the lock must be unlocked with the:

```
static inline void read_sequnlock_excl(seqlock_t *sl)
{
    spin_unlock(&sl->lock);
}
```

function.

Now we know how `sequential lock` work for readers. Let's consider how does writer act when it wants to acquire a `sequential lock` to modify data. To acquire a `sequential lock` , writer should use `write_seqlock` function. If we look at the implementation of this function:

```
static inline void write_seqlock(seqlock_t *sl)
{
    spin_lock(&sl->lock);
    write_seqcount_begin(&sl->seqcount);
}
```

We will see that it acquires `spinlock` to prevent access from other writers and calls the `write_seqcount_begin` function. This function just increments value of the `sequential lock` counter:

```
static inline void raw_write_seqcount_begin(seqcount_t *s)
{
    s->sequence++;
    smp_wmb();
}
```

When a writer process will finish to modify data, the `write_sequnlock` function must be called to release a lock and give access to other writers or readers. Let's consider at the implementation of the `write_sequnlock` function. It looks pretty simple:

```
static inline void write_sequnlock(seqlock_t *sl)
```

```
{
    write_seqcount_end(&sl->seqcount);
    spin_unlock(&sl->lock);
}
```

First of all it just calls `write_seqcount_end` function to increase value of the counter of the `sequential` lock again:

```
static inline void raw_write_seqcount_end(seqcount_t *s)
{
    smp_wmb();
    s->sequence++;
}
```

and in the end we just call the `spin_unlock` macro to give access for other readers or writers.

That's all about `sequential lock` mechanism in the Linux kernel. Of course we did not consider full API of this mechanism in this part. But all other functions are based on these which we described here. For example, Linux kernel also provides some safe macros/functions to use `sequential lock` mechanism in interrupt handlers of softirq: `write_seqclock_irq` and `write_sequnlock_irq` :

```
static inline void write_seqlock_irq(seqlock_t *sl)
{
    spin_lock_irq(&sl->lock);
    write_seqcount_begin(&sl->seqcount);
}

static inline void write_sequnlock_irq(seqlock_t *sl)
{
    write_seqcount_end(&sl->seqcount);
    spin_unlock_irq(&sl->lock);
}
```

As we may see, these functions differ only in the initialization of spinlock. They call `spin_lock_irq` and `spin_unlock_irq` instead of `spin_lock` and `spin_unlock` .

Or for example `write_seqlock_irqsave` and `write_sequnlock_irqrestore` functions which are the same but used `spin_lock_irqsave` and `spin_unlock_irqsave` macro to use in IRQ) handlers.

That's all.

Conclusion

This is the end of the sixth part of the synchronization primitives chapter in the Linux kernel. In this part we met with new synchronization primitive which is called - `sequential lock` . From the theoretical side, this synchronization primitive very similar on a readers-writer lock synchronization primitive, but allows to avoid `writer-starving` issue.

If you have questions or suggestions, feel free to ping me in twitter 0xAX, drop me email or just create issue.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-insides.

Links

- synchronization primitives)
- readers-writer lock
- spinlock
- critical section
- lock validator

- debugging
- API
- x86_64
- Timers and time management in the Linux kernel
- interrupt handlers
- softirq
- IRQ)
- Previous part

Linux

Linux Linux

- - `memblock`
- ioremap - `ioremap`
- kmemcheck - `kmemcheck`

• •

() `start_kernel` `start_kernel` `init` () `start_kernel` API `memblock`

Yinghai Lu `memblock` `x86_64`

`memblock` include/linux/memblock.h

`memblock`

```
struct memblock {
         bool bottom_up;
         phys_addr_t current_limit;
         struct memblock_type memory;   --> array of memblock_region
         struct memblock_type reserved; --> array of memblock_region
#ifdef CONFIG_HAVE_MEMBLOCK_PHYS_MAP
         struct memblock_type physmem;
#endif
};
```

`bottom_up` `true` `current_limit` (`CONFIG_HAVE_MEMBLOCK_PHYS_MAP`)- `memblock_type`

```
struct memblock_type {
    unsigned long cnt;
    unsigned long max;
    phys_addr_t total_size;
    struct memblock_region *regions;
};
```

`memblock_region` `memblock_region`

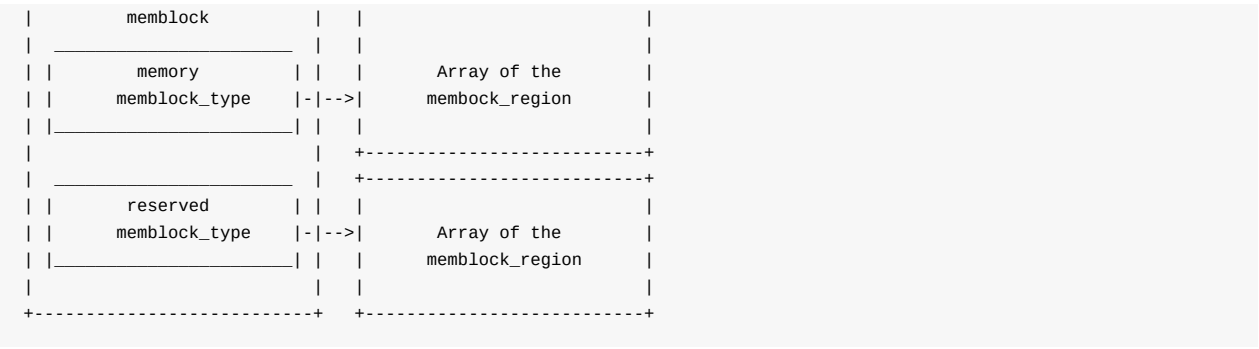
```
struct memblock_region {
        phys_addr_t base;
        phys_addr_t size;
        unsigned long flags;
#ifdef CONFIG_HAVE_MEMBLOCK_NODE_MAP
        int nid;
#endif
};
```

`memblock_region` `flags`

```
#define MEMBLOCK_ALLOC_ANYWHERE    (~(phys_addr_t)0)
#define MEMBLOCK_ALLOC_ACCESSIBLE    0
#define MEMBLOCK_HOTPLUG    0x1
```

`CONFIG_HAVE_MEMBLOCK_NODE_MAP` `memblock_region` - numa

```
+---------------------------+   +---------------------------+
```



`memblock` `memblock_type` `memblock_region` `Memblock` `Memblock`

`memblock` API include/linux/memblock.h , mm/memblock.c `memblock`

```
struct memblock memblock __initdata_memblock = {
    .memory.regions        = memblock_memory_init_regions,
    .memory.cnt            = 1,
    .memory.max            = INIT_MEMBLOCK_REGIONS,

    .reserved.regions    = memblock_reserved_init_regions,
    .reserved.cnt        = 1,
    .reserved.max        = INIT_MEMBLOCK_REGIONS,

#ifdef CONFIG_HAVE_MEMBLOCK_PHYS_MAP
    .physmem.regions    = memblock_physmem_init_regions,
    .physmem.cnt        = 1,
    .physmem.max        = INIT_PHYSMEM_REGIONS,
#endif
    .bottom_up            = false,
    .current_limit        = MEMBLOCK_ALLOC_ANYWHERE,
};
```

`memblock` `__initdata_memblock`

```
#ifdef CONFIG_ARCH_DISCARD_MEMBLOCK
    #define __init_memblock __meminit
    #define __initdata_memblock __meminitdata
#else
    #define __init_memblock
    #define __initdata_memblock
#endif
```

`CONFIG_ARCH_DISCARD_MEMBLOCK` `.init`

`memblock_type memory` `memblock_type reserved` `memblock_type physmem` `memblock_type.regions` `memblock_type` `memblock_region`

```
static struct memblock_region memblock_memory_init_regions[INIT_MEMBLOCK_REGIONS] __initdata_memblock;
static struct memblock_region memblock_reserved_init_regions[INIT_MEMBLOCK_REGIONS] __initdata_memblock;
#ifdef CONFIG_HAVE_MEMBLOCK_PHYS_MAP
static struct memblock_region memblock_physmem_init_regions[INIT_PHYSMEM_REGIONS] __initdata_memblock;
#endif
```

128 `INIT_MEMBLOCK_REGIONS`

```
#define INIT_MEMBLOCK_REGIONS   128
```

`memblock` `__initdata_memblock` ()

`bottom_up`

```
#define MEMBLOCK_ALLOC_ANYWHERE (~(phys_addr_t)0)
```

`0xffffffffffffffff` .

On this step the initialization of the `memblock` structure has been finished and we can look on the Memblock API. `memblock` API

`memblock` API `memblock` mm/memblock.c `memblock` arch/x86/kernel/e820.c `memblock_x86_fill` e820 `memblock_add` `memblock` `memblock_add`

`memblock` `memblock_add`

```
memblock_add_range(&memblock.memory, base, size, MAX_NUMNODES, 0);
```

- `memory` `CONFIG_NODES_SHIFT` 1 `1 << CONFIG_NODES_SHIFT` `memblock_add_range` 0 `memblock_type` `memblock` `memory_region` ( ) `memblock_type` `memblock_type` `memblock`

```
phys_addr_t end = base + memblock_cap_size(base, &size);
```

`memblock_cap_size` `size` `base + size`

```
static inline phys_addr_t memblock_cap_size(phys_addr_t base, phys_addr_t *size)
{
    return *size = min(*size, (phys_addr_t)ULLONG_MAX - base);
}
```

`memblock_cap_size` `ULLONG_MAX - base`

`memblock_add_range` `memblock`

-
-

```
    for (i = 0; i < type->cnt; i++) {
        struct memblock_region *rgn = &type->regions[i];
        phys_addr_t rbase = rgn->base;
        phys_addr_t rend = rbase + rgn->size;

        if (rbase >= end)
            break;
        if (rend <= base)
            continue;
        ...
        ...
        ...
    }
```

`memblock_double_array`

```
while (type->cnt + nr_new > type->max)
    if (memblock_double_array(type, obase, size) < 0)
        return -ENOMEM;
    insert = true;
    goto repeat;
```

`memblock_double_array` `insert` `true` `repeat` `repeat` `memblock_insert_region`

```
    if (base < end) {
        nr_new++;
        if (insert)
            memblock_insert_region(type, i, base, end - base,
                       nid, flags);
    }
```

`insert` `true` `memblock_insert_region` `memblock_insert_region` `memblock_type` ()

```
struct memblock_region *rgn = &type->regions[idx];
```

`memmove`

```
memmove(rgn + 1, rgn, (type->cnt - idx) * sizeof(*rgn));
```

`memblock_region` `memblock_type` `memblock_add_range` `memblock_merge_regions`

`memblock` `region1` :

```
0                   0x1000
+-----------------------+
|                       |
|                       |
|        region1        |
|                       |
|                       |
+-----------------------+
```

`memblock` `region2`

```
0x100                0x2000
+-----------------------+
|                       |
|                       |
|        region2        |
|                       |
|                       |
+-----------------------+
```

```
base = min(rend, end);
```

`0x1000`

```
if (base < end) {
    nr_new++;
    if (insert)
        memblock_insert_region(type, i, base, end - base, nid, flags);
}
```

`overlapping portion` () `memblock_merge_regions` `memblock_type` - `type->regions[i]` `type->regions[i + 1]`

```
while (i < type->cnt - 1) {
    struct memblock_region *this = &type->regions[i];
    struct memblock_region *next = &type->regions[i + 1];
    if (this->base + this->size != next->base ||
        memblock_get_region_node(this) !=
        memblock_get_region_node(next) ||
        this->flags != next->flags) {
        BUG_ON(this->base + this->size > next->base);
        i++;
        continue;
    }
```

```
this->size += next->size;
```

`memmove`

```
memmove(next, next + 1, (type->cnt - (i + 2)) * sizeof(*next));
```

`memblock_type`

```
type->cnt--;
```

```
0                                           0x2000
+-----------------------------------------------+
|                                               |
|                                               |
|                   region1                     |
|                                               |
|                                               |
+-----------------------------------------------+
```

`memblock_add_range`

`memblock_reserve` `memblock_add` `memblock_reserve` `memblock_type.reserved` `memblock_type.memory`

API `memory` `reserved`

- memblock_remove -
- memblock_find_in_range -
- memblock_free -
- for_each_mem_range -

.....

`memblock` API

- get_allocated_memblock_memory_regions_info -
- get_allocated_memblock_reserved_regions_info -

`get_allocated_memblock_reserved_regions_info`

```
phys_addr_t __init_memblock get_allocated_memblock_reserved_regions_info(
                    phys_addr_t *addr)
{
    if (memblock.reserved.regions == memblock_reserved_init_regions)
        return 0;

    *addr = __pa(memblock.reserved.regions);

    return PAGE_ALIGN(sizeof(struct memblock_region) *
              memblock.reserved.max);
}
```

`memblock` 0 `PAGE_ALIGN`

```
#define PAGE_ALIGN(addr) ALIGN(addr, PAGE_SIZE)
```

`get_allocated_memblock_memory_regions_info` `get_allocated_memblock_memory_regions_info`
`memblock_type.memory` `memblock_type.reserved`

`memblock_dbg` `memblock=debug` `memblock_dbg` `printk`

```
#define memblock_dbg(fmt, ...) \
         if (memblock_debug) printk(KERN_INFO pr_fmt(fmt), ##__VA_ARGS__)
```

`memblock_reserve`

```
memblock_dbg("memblock_reserve: [%#016llx-%#016llx] flags %#02lx %pF\n",
             (unsigned long long)base,
             (unsigned long long)base + size - 1,
             flags, (void *)_RET_IP_);
```

```
Kernel command line: root=/dev/sdb earlyprintk=ttyS0 loglevel=7 debug rdinit=/sbin/init root=/dev/ram memblock=debug
memblock_virt_alloc_try_nid_nopanic: 32768 bytes align=0x0 nid=-1 from=0x0 max_addr=0x0 alloc_large_system_hash+0x144/0x228
memblock_reserve: [0x0000023ff38e00-0x0000023ff40dff] flags 0x0 memblock_virt_alloc_internal+0xfd/0x13f
PID hash table entries: 4096 (order: 3, 32768 bytes)
memblock_virt_alloc_try_nid_nopanic: 67108864 bytes align=0x1000 nid=-1 from=0x0 max_addr=0xffffffff swiotlb_init+0x4c/0xad
memblock_reserve: [0x000000bbfe0000-0x000000bffdffff] flags 0x0 memblock_virt_alloc_internal+0xfd/0x13f
memblock_virt_alloc_try_nid_nopanic: 32768 bytes align=0x1000 nid=-1 from=0x0 max_addr=0xffffffff swiotlb_init_with_tbl+0x69/0x147
memblock_reserve: [0x000000bbfd8000-0x000000bbfdffff] flags 0x0 memblock_virt_alloc_internal+0xfd/0x13f
memblock_virt_alloc_try_nid: 131072 bytes align=0x1000 nid=-1 from=0x0 max_addr=0x0 swiotlb_init_with_tbl+0xb9/0x147
memblock_reserve: [0x0000023ff18000-0x0000023ff37fff] flags 0x0 memblock_virt_alloc_internal+0xfd/0x13f
memblock_virt_alloc_try_nid: 262144 bytes align=0x1000 nid=-1 from=0x0 max_addr=0x0 swiotlb_init_with_tbl+0xe8/0x147
memblock_reserve: [0x0000023fed8000-0x0000023ff17fff] flags 0x0 memblock_virt_alloc_internal+0xfd/0x13f
```

debugfs `X86`

- /sys/kernel/debug/memblock/memory
- /sys/kernel/debug/memblock/reserved
- /sys/kernel/debug/memblock/physmem

`memblock`

twitter issue

PR **linux-insides**.

- e820
- numa
- debugfs
-

• •

`__START_KERNEL_map` “” `level2_fixmap_pgt`

```
NEXT_PAGE(level2_fixmap_pgt)
    .fill    506,8,0
    .quad    level1_fixmap_pgt - __START_KERNEL_map + _PAGE_TABLE
    .fill    5,8,0

NEXT_PAGE(level1_fixmap_pgt)
    .fill    512,8,0
```

`level2_fixmap_pgt` `level2_kernel_pgt` code+data+bss arch/x86/include/asm/fixmap.h `fixed_addresses` `VSYSCALL_PAGE` - vsyscall apic `FIX_APIC_BASE`

```
       +-----------+-----------------+---------------+------------------+
       |           |                 |               |                  |
       |kernel text|      kernel     |               |    vsyscalls     |
       | mapping   |       text      |    Modules    |    fix-mapped    |
       |from phys 0|       data      |               |    addresses     |
       |           |                 |               |                  |
       +-----------+-----------------+---------------+------------------+
__START_KERNEL_map   __START_KERNEL    MODULES_VADDR            0xffffffffffffffff
```

```
#define FIXADDR_SIZE    (__end_of_permanent_fixed_addresses << PAGE_SHIFT)
#define FIXADDR_START        (FIXADDR_TOP - FIXADDR_SIZE)
```

`__end_of_permanent_fixed_addresses` `fixed_addresses` `fixed_addresses` `PAGE_SHIFT` `1 <<` `PAGE_SHIFT` `__end_of_permanent_fixed_addresses` `536` KB

The second `FIXADDR_START` macro just substracts fix-mapped area size from the last address of the fix-mapped area to get its base virtual address. `FIXADDR_TOP` is a rounded up address from the base address of the vsyscall space: `FIXADDR_START` `FIXADDR_TOP` vsyscall

```
#define FIXADDR_TOP     (round_up(VSYSCALL_ADDR + PAGE_SIZE, 1<<PMD_SHIFT) - PAGE_SIZE)
```

`fixed_addresses` `fix_to_virt`

```
static __always_inline unsigned long fix_to_virt(const unsigned int idx)
{
        BUILD_BUG_ON(idx >= __end_of_fixed_addresses);
        return __fix_to_virt(idx);
}
```

`BUILD_BUG_ON` `fixed_addresses` `__end_of_fixed_addresses` `__fix_to_virt`

```
#define __fix_to_virt(x)        (FIXADDR_TOP - ((x) << PAGE_SHIFT))
```

`PAGE_SHIFT` `FIXADDR_TOP` `FIXADDR_TOP`

```
static inline unsigned long virt_to_fix(const unsigned long vaddr)
{
```

```
        BUG_ON(vaddr >= FIXADDR_TOP || vaddr < FIXADDR_START);
        return __virt_to_fix(vaddr);
}
```

`virt_to_fix` `FIXADDR_START` `FIXADDR_TOP` `__virt_to_fix`

```
#define __virt_to_fix(x)        ((FIXADDR_TOP - ((x)&PAGE_MASK)) >> PAGE_SHIFT)
```

PFN PFN (page_phys_addr >> PAGE_SHIFT)

`__virt_to_fix` 12 (`FIXADDR_TOP`) `PAGE_SHIFT` 12 `x & PAGE_MASK` 12 `FIXADDR_TOP` `FIXADDR_TOP` 12 12 `PAGE_SHIFT` `Page frame number` 12 `IDT` UUID `FIX_TBOOT_BASE` Xen `ioremap`

`I/O`

- I/O
-

`in` `out` `/proc/ioports` I/O

```
$ cat /proc/ioports
0000-0cf7 : PCI Bus 0000:00
  0000-001f : dma1
  0020-0021 : pic1
  0040-0043 : timer0
  0050-0053 : timer1
  0060-0060 : keyboard
  0064-0064 : keyboard
  0070-0077 : rtc0
  0080-008f : dma page reg
  00a0-00a1 : pic2
  00c0-00df : dma2
  00f0-00ff : fpu
    00f0-00f0 : PNP0C04:00
  03c0-03df : vesafb
  03f8-03ff : serial
  04d0-04d1 : pnp 00:06
  0800-087f : pnp 00:01
  0a00-0a0f : pnp 00:04
  0a20-0a2f : pnp 00:04
  0a30-0a3f : pnp 00:04
0cf8-0cff : PCI conf1
0d00-ffff : PCI Bus 0000:00
...
...
...
```

`/proc/ioports` I/O `0000-0cf7` include/linux/ioport.h `request_region` `request_region`

```
#define request_region(start,n,name)   __request_region(&ioport_resource, (start), (n), (name), 0)
```

- `start` -;
- `n` -;
- `name` -

`request_region` I/O `request_region` `check_region` `release_region` `request_region` `resource` `resource`

```
struct resource {
        resource_size_t start;
        resource_size_t end;
        const char *name;
        unsigned long flags;
        struct resource *parent, *sibling, *child;
};
```

`resource` `parent` `slibling` `child` I/O `ioport_resource`

```
struct resource ioport_resource = {
         .name   = "PCI IO",
         .start  = 0,
         .end    = IO_SPACE_LIMIT,
        .flags  = IORESOURCE_IO,
};
EXPORT_SYMBOL(ioport_resource);
```

`iomem` `iomem_resource`

```
struct resource iomem_resource = {
        .name   = "PCI mem",
        .start  = 0,
        .end    = -1,
        .flags  = IORESOURCE_MEM,
};
```

`request_region` I/O drivers/char/rtc.c `rtc` `module_init`

```
module_init(rtc_init);
```

`rtc_init` `rtc` `rtc.c` `rtc_init` `rtc_request_region` `request_region`

```
r = rtc_request_region(RTC_IO_EXTENT);
```

`rtc_request_region` :

```
r = request_region(RTC_PORT(0), size, "rtc");
```

`RTC_TO_EXTENT` `0x8` `"rtc"` `RTC_PORT`

```
#define RTC_PORT(x)     (0x70 + (x))
```

`request_region(RTC_PORT(0), size, "rtc")` `0x70` `0x8` `/proc/ioports` :

```
~$ sudo cat /proc/ioports | grep rtc
0070-0077 : rtc0
```

I/O I/O CPU I/O `ioremap` `ioremap` I/O

-
-

I/O API I/O API

- `request_mem_region`

- `release_mem_region`
- `check_mem_region`

```
~$ sudo cat /proc/iomem
...
...
...
be826000-be82cfff : ACPI Non-volatile Storage
be82d000-bf744fff : System RAM
bf745000-bfff4fff : reserved
bfff5000-dc041fff : System RAM
dc042000-dc0d2fff : reserved
dc0d3000-dc138fff : System RAM
dc139000-dc27dfff : ACPI Non-volatile Storage
dc27e000-deffefff : reserved
defff000-defffffff : System RAM
df000000-dfffffff : RAM buffer
e0000000-feaffffff : PCI Bus 0000:00
  e0000000-efffffff : PCI Bus 0000:01
    e0000000-efffffff : 0000:01:00.0
  f7c00000-f7cfffff : PCI Bus 0000:06
    f7c00000-f7c0ffff : 0000:06:00.0
    f7c10000-f7c101ff : 0000:06:00.0
      f7c10000-f7c101ff : ahci
  f7d00000-f7dfffff : PCI Bus 0000:03
    f7d00000-f7d3ffff : 0000:03:00.0
      f7d00000-f7d3ffff : alx
...
...
...
```

`e820_reserve_resources` arch/x86/kernel/setup.c arch/x86/kernel/e820.c e820 `iomen` `e820` `iomem`

```
static inline const char *e820_type_to_string(int e820_type)
{
    switch (e820_type) {
    case E820_RESERVED_KERN:
    case E820_RAM:    return "System RAM";
    case E820_ACPI:    return "ACPI Tables";
    case E820_NVS:    return "ACPI Non-volatile Storage";
    case E820_UNUSABLE:    return "Unusable memory";
    default:    return "reserved";
    }
}
```

`/proc/iomem`

`ioremap` `ioremap` arch/x86/mm/ioremap.c `early_ioremap_init` `ioremap` `ioremap` `vmalloc` `paging_init` `ioremap` `vmalloc` `early_ioremap_init`

```
BUILD_BUG_ON((fix_to_virt(0) + PAGE_SIZE) & ((1 << PMD_SHIFT) - 1));
```

`BUILD_BUG_ON` `BUILD_BUG_ON` `early_ioremap_setup` mm/early_ioremap.c `ioremap` `early_ioremap_setup` `slot_virt` `__end_of_permanent_fixed_addresses` `FIX_BITMAP_BEGIN` `FIX_BITMAP_END` `ioremap` `512`

```
#define NR_FIX_BTMAPS        64
#define FIX_BTMAPS_SLOTS    8
#define TOTAL_FIX_BTMAPS    (NR_FIX_BTMAPS * FIX_BTMAPS_SLOTS)
```

`early_ioremap_setup`

```
void __init early_ioremap_setup(void)
{
        int i;

        for (i = 0; i < FIX_BTMAPS_SLOTS; i++)
                if (WARN_ON(prev_map[i]))
                        break;

        for (i = 0; i < FIX_BTMAPS_SLOTS; i++)
                slot_virt[i] = __fix_to_virt(FIX_BTMAP_BEGIN - NR_FIX_BTMAPS*i);
}
```

`slot_virt`

```
static void __iomem *prev_map[FIX_BTMAPS_SLOTS] __initdata;
static unsigned long prev_size[FIX_BTMAPS_SLOTS] __initdata;
static unsigned long slot_virt[FIX_BTMAPS_SLOTS] __initdata;
```

`slot_virt` `prev_map` `ioremap` `ioremap 512` `__initdata` `early_ioremap_setup`
`early_ioremap_pmd` `ioremap early_ioremap_pmd`

```
static inline pmd_t * __init early_ioremap_pmd(unsigned long addr)
{
    pgd_t *base = __va(read_cr3());
    pgd_t *pgd = &base[pgd_index(addr)];
    pud_t *pud = pud_offset(pgd, addr);
    pmd_t *pmd = pmd_offset(pud, addr);
    return pmd;
}
```

0 `bm_pte` (`ioremap`) `pmd_populate_kernel`

```
pmd = early_ioremap_pmd(fix_to_virt(FIX_BTMAP_BEGIN));
memset(bm_pte, 0, sizeof(bm_pte));
pmd_populate_kernel(&init_mm, pmd, bm_pte);
```

`pmd_populate_kernel` :

- `init_mm` - `init` ( )
- `pmd` - `ioremap`
- `bm_pte` - `ioremap`

```
static pte_t bm_pte[PAGE_SIZE/sizeof(pte_t)] __page_aligned_bss;
```

`pmd_popularte_kernel` arch/x86/include/asm/pgalloc.h (`bm_pte`)(`pmd`):

```
static inline void pmd_populate_kernel(struct mm_struct *mm,
                                       pmd_t *pmd, pte_t *pte)
{
        paravirt_alloc_pte(mm, __pa(pte) >> PAGE_SHIFT);
        set_pmd(pmd, __pmd(__pa(pte) | _PAGE_TABLE));
}
```

`set_pmd`

```
#define set_pmd(pmdp, pmd)              native_set_pmd(pmdp, pmd)
```

`native_set_pmd`

```
static inline void native_set_pmd(pmd_t *pmdp, pmd_t pmd)
{
        *pmdp = pmd;
}
```

`ioremap` `early_ioremap_init` `ioremap`

`ioremap`

- early_ioremap
- early_iounmap

IO / `CONFIG_MMU` `cr3` (pgd) `CONFIG_MMU` `n` `early_ioremap` `early_iounmap` `y` `early_ioremap` `__early_ioremap`

- `phys_addr` - I/O
- `size` - I/O
- `prot` -

`__early_ioremap` `ioremap` `prev_map` `slot`

```
slot = -1;
for (i = 0; i < FIX_BTMAPS_SLOTS; i++) {
    if (!prev_map[i]) {
        slot = i;
        break;
    }
}
...
...
...
prev_size[slot] = size;
last_addr = phys_addr + size - 1;
```

```
offset = phys_addr & ~PAGE_MASK;
phys_addr &= PAGE_MASK;
size = PAGE_ALIGN(last_addr + 1) - phys_addr;
```

`PAGE_MASK` 12 `phys_addr` `PAGE_MASK`

```
#define PAGE_MASK       (~(PAGE_SIZE-1))
```

4096 `1000000000000` `PAGE_SIZE - 1` `111111111111` `~` `000000000000` `~PAGE_MASK` `111111111111` 12 `ioremap`

```
nrpages = size >> PAGE_SHIFT;
idx = FIX_BTMAP_BEGIN - NR_FIX_BTMAPS*slot;
```

arch/x86/mm/ioremap.c `__early_set_fixmap` `4096`

```
while (nrpages > 0) {
    __early_set_fixmap(idx, phys_addr, prot);
    phys_addr += PAGE_SIZE;
    --idx;
    --nrpages;
```

```
}
```

`__early_set_fixmap` (`bm_pte`)

```
pte = early_ioremap_pte(addr);
```

`early_ioremap_pte` `pgprot_val` `set_pte` `pte_clear`

```
if (pgprot_val(flags))
        set_pte(pte, pfn_pte(phys >> PAGE_SHIFT, flags));
    else
        pte_clear(&init_mm, addr, pte);
```

As you can see above, we passed `FIXMAP_PAGE_IO` as flags to the `__early_ioremap` . `FIXMPA_PAGE_IO` expands to the:
`FIXMAP_PAGE_IO` `__early_ioremap` `FIXMPA_PAGE_IO`

```
(__PAGE_KERNEL_EXEC | _PAGE_NX)
```

`set_pte` `set_pmd` `PTE` () `PTE` `__flush_tlb_one`

```
__flush_tlb_one(addr);
```

arch/x86/include/asm/tlbflush.h `cpu_has_invlpg` `__flush_tlb_single` `__flush_tlb`

```
static inline void __flush_tlb_one(unsigned long addr)
{
        if (cpu_has_invlpg)
                __flush_tlb_single(addr);
        else
                __flush_tlb();
}
```

`__flush_tlb_one` TLB `TLB` `cr3` `__flush_tlb`

```
native_write_cr3(native_read_cr3());
```

`invlpg` `TLB` `__flush_tlb_one` `cpu_has_invlpg`

```
#if defined(CONFIG_X86_INVLPG) || defined(CONFIG_X86_64)
# define cpu_has_invlpg         1
#else
# define cpu_has_invlpg         (boot_cpu_data.x86 > 3)
#endif
```

CPU `invlpg` `__flush_tlb_single` `__native_flush_tlb_single`

```
static inline void __native_flush_tlb_single(unsigned long addr)
{
        asm volatile("invlpg (%0)" ::"r" (addr) : "memory");
}
```

`__flush_tlb` `cr3` `__early_set_fixmap` `__early_ioremap` I/O `slot` `prev_map`

```
prev_map[slot] = (void __iomem *)(offset + slot_virt[slot]);
```

`early_iounmap` I/O I/O `early_ioremap` `after_paging_init` `__late_clear_fixmap` `__early_set_fixmap` 0 `__early_set_fixmap` I/O `NULL`

```
prev_map[slot] = NULL;
```

`fixmap` `ioremap` `ioremap` `ioremap` `ioremap`

twitter issue

PR **linux-insides**.

- apic
- vsyscall
- Intel Trusted Execution Technology
- Xen
- Real Time Clock
- e820
- Memory management unit
- TLB
- Paging
-

Linux

kmemcheck

LinuxLinux

- ;
- .

/

`/proc/iomem`

```
$ sudo cat /proc/iomem

00000000-00000fff : reserved
00001000-0009d7ff : System RAM
0009d800-0009ffff : reserved
000a0000-000bffff : PCI Bus 0000:00
000c0000-000cffff : Video ROM
000d0000-000d3fff : PCI Bus 0000:00
000d4000-000d7fff : PCI Bus 0000:00
000d8000-000dbfff : PCI Bus 0000:00
000dc000-000dffff : PCI Bus 0000:00
000e0000-000fffff : reserved
...
...
...
```

`iomem`

```
$ sudo cat /proc/ioports

0000-0cf7 : PCI Bus 0000:00
  0000-001f : dma1
  0020-0021 : pic1
  0040-0043 : timer0
  0050-0053 : timer1
  0060-0060 : keyboard
  0064-0064 : keyboard
  0070-0077 : rtc0
  0080-008f : dma page reg
  00a0-00a1 : pic2
  00c0-00df : dma2
  00f0-00ff : fpu
    00f0-00f0 : PNP0C04:00
  03c0-03df : vga+
  03f8-03ff : serial
  04d0-04d1 : pnp 00:06
  0800-087f : pnp 00:01
  0a00-0a0f : pnp 00:04
  0a20-0a2f : pnp 00:04
  0a30-0a3f : pnp 00:04
...
...
...
```

`ioports` I/O/ `io remap` `io remap` `io remap`

Linux Linux

kmemcheck `kmemcheck` Linux

Linux `kmemcheck` `kmemcheck` `kmemcheck` C

```
#include <stdlib.h>
#include <stdio.h>

struct A {
        int a;
};

int main(int argc, char **argv) {
        struct A *a = malloc(sizeof(struct A));
        printf("a->a = %d\n", a->a);
        return 0;
}
```

`A` `a`

```
gcc test.c -o test
```

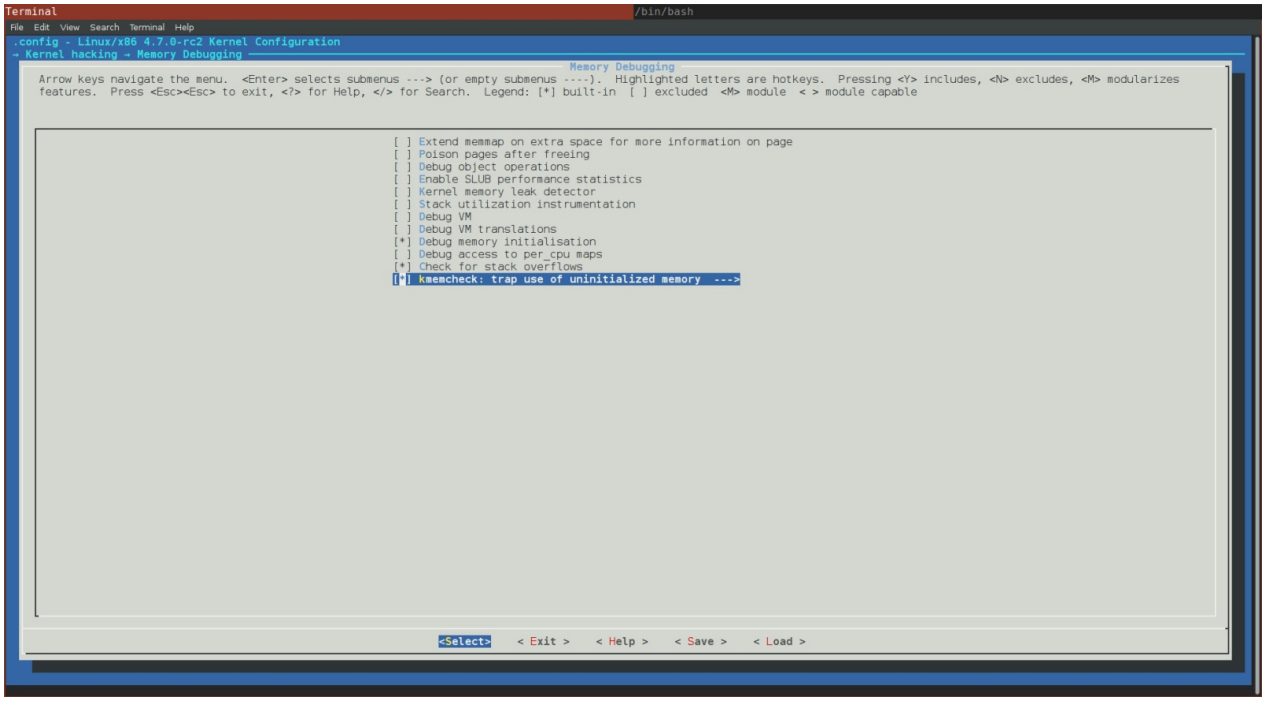
`a` valgrind

```
~$   valgrind --leak-check=yes ./test
==28469== Memcheck, a memory error detector
==28469== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==28469== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==28469== Command: ./test
==28469==
==28469== Conditional jump or move depends on uninitialised value(s)
==28469==    at 0x4E820EA: vfprintf (in /usr/lib64/libc-2.22.so)
==28469==    by 0x4E88D48: printf (in /usr/lib64/libc-2.22.so)
==28469==    by 0x4005B9: main (in /home/alex/test)
==28469==
==28469== Use of uninitialised value of size 8
==28469==    at 0x4E7E0BB: _itoa_word (in /usr/lib64/libc-2.22.so)
==28469==    by 0x4E8262F: vfprintf (in /usr/lib64/libc-2.22.so)
==28469==    by 0x4E88D48: printf (in /usr/lib64/libc-2.22.so)
==28469==    by 0x4005B9: main (in /home/alex/test)
...
...
...
```

`kmemcheck` `valgrind`

`CONFIG_KMEMCHECK`

```
Kernel hacking
  -> Memory Debugging
```



`kmemcheck` `kmemcheck` x86_64 `x86` arch/x86/Kconfig

```
config X86
  ...
  ...
  ...
  select HAVE_ARCH_KMEMCHECK
  ...
  ...
  ...
```

`kmemcheck`

`kmemcheck` `kmemcheck`

```
struct my_struct *my_struct = kmalloc(sizeof(struct my_struct), GFP_KERNEL);
```

page `kmemcheck` Linux `kmemcheck` `kmemcheck` `kmemcheck` `present` `kmemcheck` `not present`

`kmemcheck` Linux

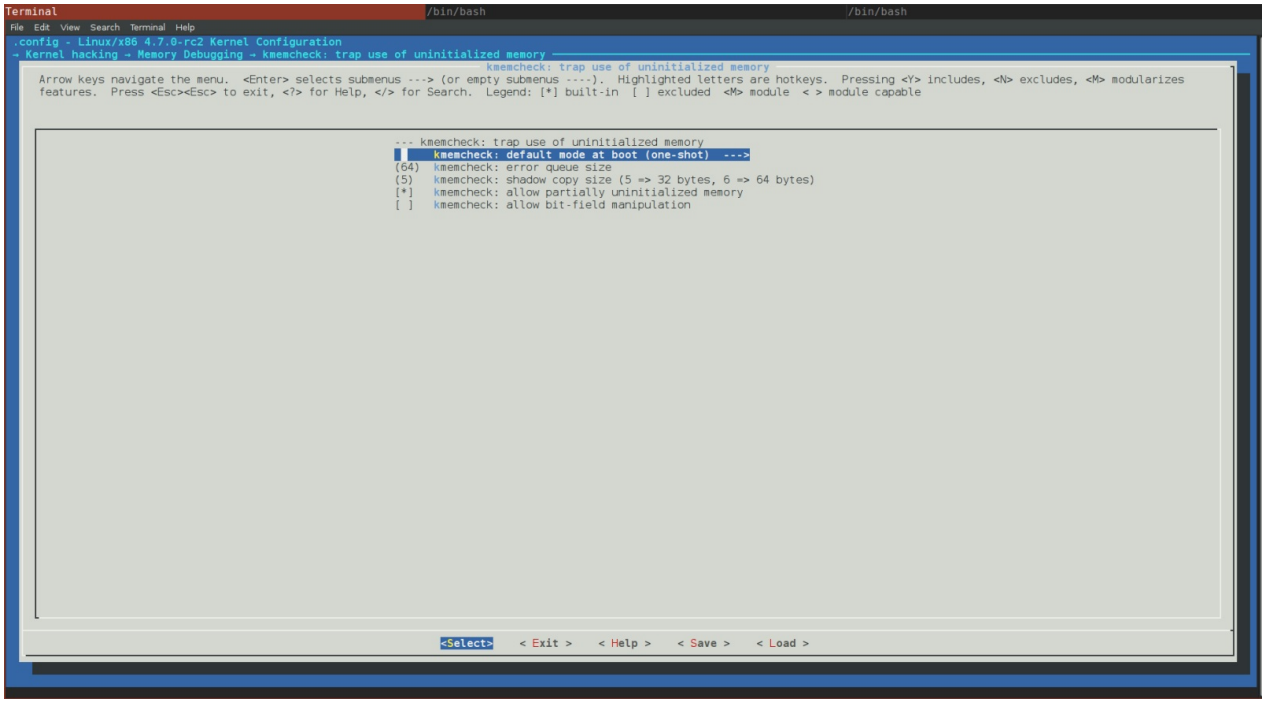
`kmemcheck` Linux

`kmemcheck` LinuxLinux `kmemcheck` mm/kmemcheck.c x86_64arch/x86/mm/kmemcheck

`kmemcheck` `CONFIG_KMEMCHECK` command line `kmemcheck`

- kmemcheck=0 (disabled)
- kmemcheck=1 (enabled)
- kmemcheck=2 (one-shot mode)

`kmemcheck` `kmemcheck`



Linux part `do_initcall_level` , `do_early_param` command line `k` `param_kmemcheck` command line:

```
static int __init param_kmemcheck(char *str)
{
    int val;
    int ret;

    if (!str)
        return -EINVAL;

    ret = kstrtoint(str, 0, &val);
    if (ret)
        return ret;
    kmemcheck_enabled = val;
    return 0;
}

early_param("kmemcheck", param_kmemcheck);
```

`param_kmemcheck` `0` (), `1` () or `2` () `param_kmemcheck` command line `kmemcheck` `kmemcheck_enabled`

initcalls `kmemcheck_init` :

```
int __init kmemcheck_init(void)
{
    ...
    ...
    ...
}

early_initcall(kmemcheck_init);
```

`kmemcheck_init` `kmemcheck_selftest`

```
if (!kmemcheck_selftest()) {
    printk(KERN_INFO "kmemcheck: self-tests failed; disabling\n");
    kmemcheck_enabled = 0;
    return -EINVAL;
}
```

```
printk(KERN_INFO "kmemcheck: Initialized\n");
```

`kmemcheck_init` `EINVAL` `kmemcheck_selftest` `rep movsb` , `movzwq`) `kmemcheck_selftest` `true` `false`

:

```
struct my_struct *my_struct = kmalloc(sizeof(struct my_struct), GFP_KERNEL);
```

`kmem_getpages` mm/slab.c

```
if (kmemcheck_enabled && !(cachep->flags & SLAB_NOTRACK)) {
    kmemcheck_alloc_shadow(page, cachep->gfporder, flags, nodeid);

    if (cachep->ctor)
        kmemcheck_mark_uninitialized_pages(page, nr_pages);
    else
        kmemcheck_mark_unallocated_pages(page, nr_pages);
}
```

`kmemcheck` `SLAB_NOTRACK` `non-present` `SLAB_NOTRACK` `kmemcheck_alloc_shadow` mm/kmemcheck.c

```
void kmemcheck_alloc_shadow(struct page *page, int order, gfp_t flags, int node)
{
    struct page *shadow;

       shadow = alloc_pages_node(node, flags | __GFP_NOTRACK, order);

       for(i = 0; i < pages; ++i)
        page[i].shadow = page_address(&shadow[i]);

       kmemcheck_hide_pages(page, pages);
}
```

shadow bits shadow `kmemcheck` `kmemcheck_hide_pages` arch/x86/mm/kmemcheck/kmemcheck.c `non-present`

```
void kmemcheck_hide_pages(struct page *p, unsigned int n)
{
    unsigned int i;

    for (i = 0; i < n; ++i) {
        unsigned long address;
        pte_t *pte;
        unsigned int level;

        address = (unsigned long) page_address(&p[i]);
        pte = lookup_address(address, &level);
        BUG_ON(!pte);
        BUG_ON(level != PG_LEVEL_4K);

        set_pte(pte, __pte(pte_val(*pte) & ~_PAGE_PRESENT));
        set_pte(pte, __pte(pte_val(*pte) | _PAGE_HIDDEN));
        __flush_tlb_one(address);
    }
}
```

present hidden TLB , `kmemcheck` `present` `kmalloc`

Linux arch/x86/mm/fault.c `do_page_fault`

```
static noinline void
__do_page_fault(struct pt_regs *regs, unsigned long error_code,
        unsigned long address)
{
    ...
    ...
    ...
    if (kmemcheck_active(regs))
        kmemcheck_hide(regs);
    ...
    ...
    ...
}
```

`kmemcheck_active` `kmemcheck_context` per-cpu `balance` 0

```
bool kmemcheck_active(struct pt_regs *regs)
{
    struct kmemcheck_context *data = this_cpu_ptr(&kmemcheck_context);

    return data->balance > 0;
}
```

`kmemcheck_context` `kmemcheck` `balance` `kmemcheck` `balance` `kmemcheck` `data->balance` 0 `kmemcheck_hide` `kmemecheck` `present` `kmemcheck_hide` `present` `kmemcheck` `data->balance` 0 `kmemcheck_active` false `kmemcheck_hide` `do_page_fault`

```
if (kmemcheck_fault(regs, address, error_code))
        return;
```

`kmemcheck_fault`

```
if (regs->flags & X86_VM_MASK)
        return false;
if (regs->cs != __KERNEL_CS)
        return false;
```

`kmemcheck` `kmemcheck_fault` false false:

```
pte = kmemcheck_pte_lookup(address);
if (!pte)
    return false;
```

`kmemcheck_fault` `kmemcheck_access` present `kmemcheck_access`

```
static struct kmemcheck_error error_fifo[CONFIG_KMEMCHECK_QUEUE_SIZE];
```

`kmemcheck` tasklet :

```
static DECLARE_TASKLET(kmemcheck_tasklet, &do_wakeup, 0);
```

tasklet `do_wakeup` arch/x86/mm/kmemcheck/error.c

`do_wakeup` `kmemcheck_error_recall` `kmemcheck`

```
kmemcheck_show(regs);
```

`kmemcheck_fault` `kmemcheck_show` present

```
if (unlikely(data->balance != 0)) {
    kmemcheck_show_all();
    kmemcheck_error_save_bug(regs);
    data->balance = 0;
    return;
}
```

`kmemcheck_show_all` `kmemcheck_show_addr`

```
static unsigned int kmemcheck_show_all(void)
{
    struct kmemcheck_context *data = this_cpu_ptr(&kmemcheck_context);
    unsigned int i;
    unsigned int n;

    n = 0;
    for (i = 0; i < data->n_addrs; ++i)
        n += kmemcheck_show_addr(data->addr[i]);

    return n;
}
```

`kmemcheck_show_addr` :

```
int kmemcheck_show_addr(unsigned long address)
{
    pte_t *pte;

    pte = kmemcheck_pte_lookup(address);
    if (!pte)
        return 0;

    set_pte(pte, __pte(pte_val(*pte) | _PAGE_PRESENT));
    __flush_tlb_one(address);
    return 1;
}
```

`kmemcheck_show` TF

```
if (!(regs->flags & X86_EFLAGS_TF))
    data->flags = regs->flags;
```

`TF` `debug` `kmemcheck` /

`kmemcheck`

Linux 0xAX issue - `kmemleak`

PR **linux-insides.**

Links

- memory management
- debugging
- memory leaks
- kmemcheck documentation

- valgrind
- page fault
- initcalls
- opcode
- translation lookaside buffer
- per-cpu variables
- flags register
- tasklet
- Paging
- Previous part

Linux

-

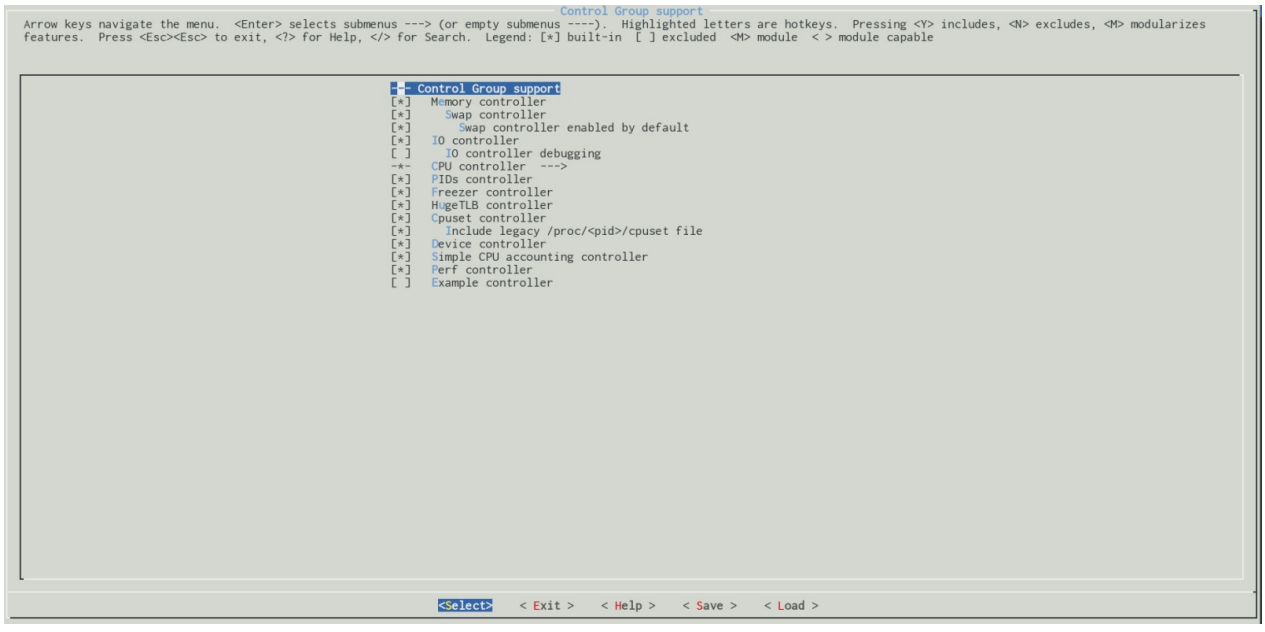
linux - Linux `cgroups`

`Cgroups` Linux `cgroup` `Cgroups` `cgroups` `cgroups` `cgroup` `cgroup` `cgroup` `cgroup` ""

`cgroup` `cgroup` pid Linux 12 `cgroup`

- `cpuset` - `cgroup`
- `cpu` - `cgroup` CPU
- `cpuacct` - `cgroup`
- `io` -;
- `memory` - `cgroup` ;
- `devices` - `cgroup`
- `freezer` - `cgroup` /
- `net_cls` - `cgroup`
- `net_prio` - `cgroup`
- `perf_event` - `cgroup`);
- `hugetlb` - `cgroup` ;
- `pid` - `cgroup`

`cgroup` `cpuset` `CONFIG_CPUSETS` `io` `CONFIG_BLK_CGROUP` `General setup → Control Group support`



proc `cgroup`

```
$ cat /proc/cgroups
#subsys_name    hierarchy    num_cgroups    enabled
cpuset    8    1    1
cpu    7    66    1
cpuacct    7    66    1
blkio    11    66    1
memory    9    94    1
devices    6    66    1
freezer    2    1    1
```

```
net_cls    4    1    1
perf_event    3    1    1
net_prio    4    1    1
hugetlb    10    1    1
pids    5    69    1
```

sysfs :

```
$ ls -l /sys/fs/cgroup/
total 0
dr-xr-xr-x 5 root root  0 Dec  2 22:37 blkio
lrwxrwxrwx 1 root root 11 Dec  2 22:37 cpu -> cpu,cpuacct
lrwxrwxrwx 1 root root 11 Dec  2 22:37 cpuacct -> cpu,cpuacct
dr-xr-xr-x 5 root root  0 Dec  2 22:37 cpu,cpuacct
dr-xr-xr-x 2 root root  0 Dec  2 22:37 cpuset
dr-xr-xr-x 5 root root  0 Dec  2 22:37 devices
dr-xr-xr-x 2 root root  0 Dec  2 22:37 freezer
dr-xr-xr-x 2 root root  0 Dec  2 22:37 hugetlb
dr-xr-xr-x 5 root root  0 Dec  2 22:37 memory
lrwxrwxrwx 1 root root 16 Dec  2 22:37 net_cls -> net_cls,net_prio
dr-xr-xr-x 2 root root  0 Dec  2 22:37 net_cls,net_prio
lrwxrwxrwx 1 root root 16 Dec  2 22:37 net_prio -> net_cls,net_prio
dr-xr-xr-x 2 root root  0 Dec  2 22:37 perf_event
dr-xr-xr-x 5 root root  0 Dec  2 22:37 pids
dr-xr-xr-x 5 root root  0 Dec  2 22:37 systemd
```

`cgroup` Linux `cgroup`

`/sys/fs/cgroup` pid `tasks`

`libcgroup` // `cgroups` (Fedora `libcgroup-tools`)

bash

```
#!/bin/bash

while :
do
    echo "print line" > /dev/tty
    sleep 5
done
```

```
$ sudo chmod +x cgroup_test_script.sh
~$ ./cgroup_test_script.sh
print line
print line
print line
...
...
...
```

`cgroupfs` `/sys/fs/cgroup`

```
$ cd /sys/fs/cgroup
```

`devices` `cgroup`

```
# cd devices
```

`cgroup_test_group`

```
# mkdir cgroup_test_group
```

`cgroup_test_group`

```
/sys/fs/cgroup/devices/cgroup_test_group$ ls -l
total 0
-rw-r--r-- 1 root root 0 Dec  3 22:55 cgroup.clone_children
-rw-r--r-- 1 root root 0 Dec  3 22:55 cgroup.procs
--w------- 1 root root 0 Dec  3 22:55 devices.allow
--w------- 1 root root 0 Dec  3 22:55 devices.deny
-r--r--r-- 1 root root 0 Dec  3 22:55 devices.list
-rw-r--r-- 1 root root 0 Dec  3 22:55 notify_on_release
-rw-r--r-- 1 root root 0 Dec  3 22:55 tasks
```

`tasks` `devices.deny` `tasks` `cgroup_test_group` `cgroup` pid `devices.deny` `cgroup` (`/dev/tty`) `devices.deny`

```
# echo "c 5:0 w" > devices.deny
```

`c` `/dev/tty` “” `ls`

```
~$ ls -l /dev/tty
crw-rw-rw- 1 root tty 5, 0 Dec  3 22:48 /dev/tty
```

`c` `5:0` `ls` `w` `cgroups` `cgroup_test_script.sh`

```
~$ ./cgroup_test_script.sh
print line
print line
print line
...
...
```

pid `cgroup` `devices/tasks`

```
# echo $(pidof -x cgroup_test_script.sh) > /sys/fs/cgroup/devices/cgroup_test_group/tasks
```

```
~$ ./cgroup_test_script.sh
print line
print line
print line
print line
print line
print line
./cgroup_test_script.sh: line 5: /dev/tty: Operation not permitted
```

docker)

```
~$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
             NAMES
fa2d2085cd1c        mariadb:10          "docker-entrypoint..."   12 days ago         Up 4 minutes        0.0.0.0:3306
->3306/tcp   mysql-work

~$ cat /sys/fs/cgroup/devices/docker/fa2d2085cd1c8d797002c77387d2061f56fefb470892f140d0dc511bd4d9bb61/tasks | head -3
5501
5584
5585
```

```
...
...
...
```

`docker` `docker` `cgroup`

```
$ docker exec -it mysql-work /bin/bash
$ top
 PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
                                         1 mysql     20   0  963996 101268  15744 S   0.0  0.6   0:00.46 mysqld
                                                                                  71 root      20   0   20248   3028   2732
S   0.0  0.0   0:00.01 bash                                                                                              77 roo
t      20   0   21948   2424   2056 R   0.0  0.0   0:00.00 top
```

`cgroup`

```
$ systemd-cgls

Control group /:
-.slice
├─docker
| └─fa2d2085cd1c8d797002c77387d2061f56fefb470892f140d0dc511bd4d9bb61
|   ├─5501 mysqld
|   └─6404 /bin/bash
```

`cgroup` Linux

cgroup

Linux `cgroup` Linux `cgroup` Linux `cgroups` “”“”

`Cgroups` Linux init/main.c

```
cgroup_init_early();
```

kernel/cgroup.c

```
int __init cgroup_init_early(void)
{
    static struct cgroup_sb_opts __initdata opts;
    struct cgroup_subsys *ss;
    ...
    ...
    ...
}
```

`cgroup_sb_opts`

```
struct cgroup_sb_opts {
    u16 subsys_mask;
    unsigned int flags;
    char *release_agent;
    bool cpuset_clone_children;
    char *name;
    bool none;
};
```

`cgroupfs` `name=` cgroup (`my_cgrp`)

```
$ mount -t cgroup -oname=my_cgrp,none /mnt/cgroups
```

- `ss` `cgroup_subsys` include/linux/cgroup-defs.h `cgroup`

```
struct cgroup_subsys {
    int (*css_online)(struct cgroup_subsys_state *css);
    void (*css_offline)(struct cgroup_subsys_state *css);
    ...
    ...
    ...
    bool early_init:1;
    int id;
    const char *name;
    struct cgroup_root *root;
    ...
    ...
    ...
}
```

`css_online` `css_offline` cgroup cgroup `early_init` `id` `name` cgroup ”“ `root` cgroup

`cgroup_subsys` `cgroups` `cgroup_init_early` ”“ `cgroup_subsys->early_init` `1`

```
init_cgroup_root(&cgrp_dfl_root, &opts);
cgrp_dfl_root.cgrp.self.flags |= CSS_NO_REF;
```

`init_cgroup_root` `cgroup` `CSS_NO_REF` CSS `cgrp_dfl_root`

```
struct cgroup_root cgrp_dfl_root;
```

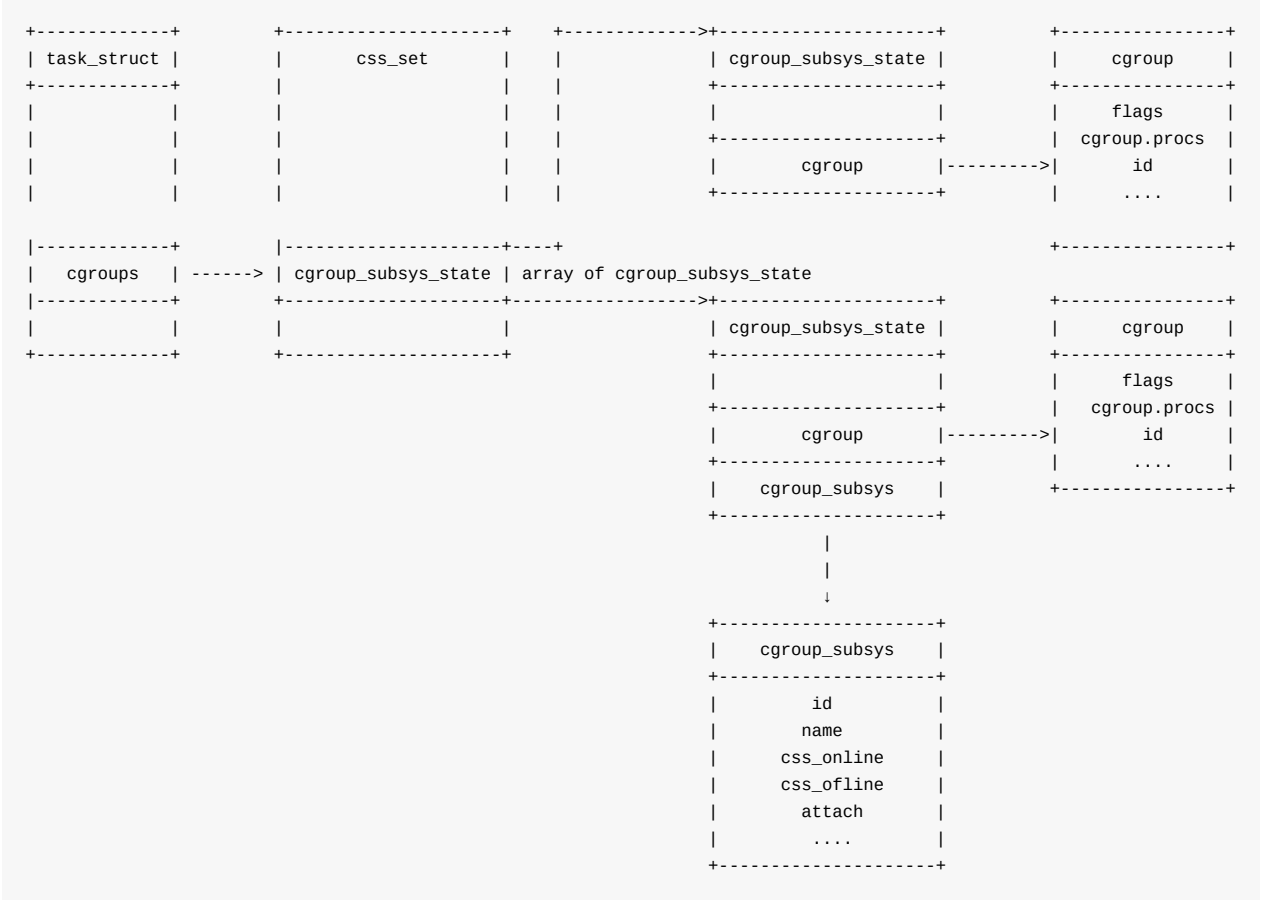
`cgrp` `cgroup` `cgroup` `cgroup` include/linux/cgroup-defs.h Linux `task_struct` `task_struct` `cgroup` `task_struct` `css_set` `css_set`

```
struct css_set {
    ...
    ...
    ....
    struct cgroup_subsys_state *subsys[CGROUP_SUBSYS_COUNT];
    ...
    ...
    ...
}
```

`cgroup_subsys_state` `cgroup`

```
struct cgroup_subsys_state {
    ...
    ...
    ...
    struct cgroup *cgroup;
    ...
    ...
    ...
}
```

`cgroups`



`init_cgroup_root` `cgrp_dfl_root` `css_set` `init_task`

```
RCU_INIT_POINTER(init_task.cgroups, &init_css_set);
```

`cgroup_init_early` `early cgroups` `cgroup_init_subsys`

```
for_each_subsys(ss, i) {
        ss->id = i;
        ss->name = cgroup_subsys_name[i];

        if (ss->early_init)
            cgroup_init_subsys(ss, true);
}
```

`for_each_subsys` kernel/cgroup.c `cgroup_subsys` for

```
#define SUBSYS(_x) [_x ## _cgrp_id] = &_x ## _cgrp_subsys,
    static struct cgroup_subsys *cgroup_subsys[] = {
        #include <linux/cgroup_subsys.h>
};
#undef SUBSYS
```

`SUBSYS` () cgroup `cgroup_subsys` linux/cgroup_subsys.h `SUBSYS`

```
#if IS_ENABLED(CONFIG_CPUSETS)
SUBSYS(cpuset)
#endif

#if IS_ENABLED(CONFIG_CGROUP_SCHED)
SUBSYS(cpu)
#endif
...
```

```
...
...
```

`SUBSYS` `#undef` `&_x ## _cgrp_subsys` `C` `##` `cpuset cpu` `SUBSYS` `cpuset_cgrp_subsys cp_cgrp_subsys` kernel/cpuset.c

```
struct cgroup_subsys cpuset_cgrp_subsys = {
    ...
    ...
    ...
    .early_init    = true,
};
```

`cgroup_init_early` `cgroup_init_subsys`

- `cpuset` ;
- `cpu` ;
- `cpuacct` .

`cgroup_init_subsys` `css_alloc`

Linux `cgroup` `cgroup` `cgroup`

twitter

PR **linux-insides.**

- control groups
- PID
- cpuset
- block devices
- huge pages
- sysfs
- proc
- cgroups kernel documentation
- cgroups v2
- bash
- docker)
- perf events)
- Previous chapter

Linux

- CPU
- CPU
- initcall
- Linux

Per-cpu

Per-cpu CPU

per-cpu API - `DEFINE_PER_CPU`

```
#define DEFINE_PER_CPU(type, name) \
        DEFINE_PER_CPU_SECTION(type, name, "")
```

per-cpu include/linux/percpu-defs.h

`DECLARE_PER_CPU` 2 `type` `name` per-cpu

```
DEFINE_PER_CPU(int, per_cpu_n)
```

`DEFINE_PER_CPU` `DEFINE_PER_CPU_SECTION` `DEFINE_PER_CPU_SECTION`

```
#define DEFINE_PER_CPU_SECTION(type, name, sec)    \
         __PCPU_ATTRS(sec) PER_CPU_DEF_ATTRIBUTES  \
         __typeof__(type) name
```

```
#define __PCPU_ATTRS(sec)                                           \
         __percpu __attribute__((section(PER_CPU_BASE_SECTION sec)))     \
         PER_CPU_ATTRIBUTES
```

`section` :

```
#define PER_CPU_BASE_SECTION ".data..percpu"
```

per-cpu

```
__attribute__((section(".data..percpu"))) int per_cpu_n
```

`.data..percpu` `per_cpu_n` `vmlinux`

```
.data..percpu 00013a58  0000000000000000  0000000001a5c000  00e00000  2**12
              CONTENTS, ALLOC, LOAD, DATA
```

`DEFINE_PER_CPU` `.data..percpu` per-cpu `setup_per_cpu_areas` `.data..percpu` CPU

per-cpu init/main.c `setup_per_cpu_areas` arch/x86/kernel/setup_percpu.c

```
pr_info("NR_CPUS:%d nr_cpumask_bits:%d nr_cpu_ids:%d nr_node_ids:%d\n",
        NR_CPUS, nr_cpumask_bits, nr_cpu_ids, nr_node_ids);
```

`setup_per_cpu_areas` `CONFIG_NR_CPUS` CPUs CPU `nr_cpumask_bits` `cpumask` `NR_CPUS` `NUMA`

`dmesg`

```
$ dmesg | grep percpu
[    0.000000] setup_percpu: NR_CPUS:8 nr_cpumask_bits:8 nr_cpu_ids:8 nr_node_ids:1
```

`per-cpu` per-cpu per-cpu Linux `percpu_alloc`

```
percpu_alloc=     per-cpu
         "embed"  "page"

         mm/percpu.c
```

mm/percpu.c

```
early_param("percpu_alloc", percpu_alloc_setup);
```

`percpu_alloc_setup` `percpu_alloc` `pcpu_chosen_fc` `auto`

```
enum pcpu_fc pcpu_chosen_fc __initdata = PCPU_FC_AUTO;
```

`percpu_alloc` `embed` per-cpu memblock bootmem `page` `PAGE_SIZE`

`setup_per_cpu_areas` page

```
if (pcpu_chosen_fc != PCPU_FC_PAGE) {
    ...
    ...
    ...
}
```

`PCPU_FC_PAGE` `embed` `pcpu_embed_first_chunk`

```
rc = pcpu_embed_first_chunk(PERCPU_FIRST_CHUNK_RESERVE,
                        dyn_size, atom_size,
                        pcpu_cpu_distance,
                        pcpu_fc_alloc, pcpu_fc_free);
```

`pcpu_embed_first_chunk` per-cpu bootmen `pcpu_embed_first_chunk`

- `PERCPU_FIRST_CHUNK_RESERVE` - `per-cpu`
- `dyn_size` -
- `atom_size` -
- `pcpu_cpu_distance` - cpus
- `pcpu_fc_alloc` - `percpu`
- `pcpu_fc_free` - `percpu`

`pcpu_embed_first_chunk`

```
const size_t dyn_size = PERCPU_MODULE_RESERVE + PERCPU_DYNAMIC_RESERVE - PERCPU_FIRST_CHUNK_RESERVE;
size_t atom_size;
#ifdef CONFIG_X86_64
        atom_size = PMD_SIZE;
#else
        atom_size = PAGE_SIZE;
#endif
```

`PCPU_FC_PAGE` `pcpu_page_first_chunk` `pcpu_embed_first_chunk` `per-cpu` `setup_percpu_segment` `per-cpu` `x86` `per-cpu` `x86_cpu_to_apicid` , `irq_stack_ptr` N `.data..percpu` N CPU bootstrap `DEFINE_PER_CPU`

per-cpu API

- get_cpu_var(var)

- put_cpu_var(var)

`get_cpu_var`

```
#define get_cpu_var(var)     \
(*({                         \
         preempt_disable();  \
         this_cpu_ptr(&var); \
}))
```

Linux per-cpu per-cpu CPU `preempt_disable` `this_cpu_ptr`

```
#define this_cpu_ptr(ptr) raw_cpu_ptr(ptr)
```

```
#define raw_cpu_ptr(ptr)        per_cpu_ptr(ptr, 0)
```

`per_cpu_ptr` CPU 2 per-cpu per-cpu `put_cpu_var` `preempt_enable` per-cpu

```
get_cpu_var(var);
...
// 'var'
...
put_cpu_var(var);
```

`per_cpu_ptr`

```
#define per_cpu_ptr(ptr, cpu)                            \
({                                                       \
        __verify_pcpu_ptr(ptr);                          \
         SHIFT_PERCPU_PTR((ptr), per_cpu_offset((cpu))); \
})
```

cpu per-cpu `__verify_pcpu_ptr`

```
#define __verify_pcpu_ptr(ptr)
do {
    const void __percpu *__vpp_verify = (typeof((ptr) + 0))NULL;
    (void)__vpp_verify;
} while (0)
```

`ptr` `const void __percpu *`

`SHIFT_PERCPU_PTR` `per_cpu_offset` CPU

```
#define per_cpu_offset(x) (__per_cpu_offset[x])
```

`x` `__per_cpu_offset`

```
extern unsigned long __per_cpu_offset[NR_CPUS];
```

`NR_CPUS` CPU `__per_cpu_offset` CPU per-cpu `X` `__per_cpu_offset[Y]` `X*Y` `SHIFT_PERCPU_PTR`

```
#define SHIFT_PERCPU_PTR(__p, __offset)                                 \
         RELOC_HIDE((typeof(*(__p)) __kernel __force *)(__p), (__offset))
```

`RELOC_HIDE` `(typeof(ptr)) (__ptr + (off))`

API per-cpu include/linux/percpu-defs.h

per-cpu

- `.data..percpu` per-cpu
- `DEFINE_PER_CPU` CPU0
- `__per_cpu_offset` (`BOOT_PERCPU_OFFSET`) `.data..percpu`
- `per_cpu_ptr` per-cpu CPU `__per_cpu_offset` CPU

CPU masks

`Cpumasks` LinuxCPU `Cpumasks` API

- include/linux/cpumask.h
- lib/cpumask.c
- kernel/cpu.c

include/linux/cpumask.h Cpumasks CPU CPU Kernel entry point `boot_cpu_init` cpumask cpu ……

```
set_cpu_online(cpu, true);
set_cpu_active(cpu, true);
set_cpu_present(cpu, true);
set_cpu_possible(cpu, true);
```

`set_cpu_possible` cpu ID `cpu_present` CPUs `cpu_online` `cpu_present` CPUs `CONFIG_HOTPLUG_CPU` `possible == present` `active == online` `true` `cpumask_set_cpu` `cpumask_clear_cpu`

`cpumask` `cpumask_t`

```
typedef struct cpumask { DECLARE_BITMAP(bits, NR_CPUS); } cpumask_t;
```

`cpumask` `bits` `DECLARE_BITMAP`

- bitmap name;
- number of bits.

`unsigned long`

```
#define DECLARE_BITMAP(name,bits) \
        unsigned long name[BITS_TO_LONGS(bits)]
```

`BITS_TO_LONGS`

```
#define BITS_TO_LONGS(nr)       DIV_ROUND_UP(nr, BITS_PER_BYTE * sizeof(long))
#define DIV_ROUND_UP(n,d) (((n) + (d) - 1) / (d))
```

`x86_64` `unsigned long` 8

```
(((8) + (8) - 1) / (8)) = 1
```

`NR_CPUS` CPU include/linux/threads.h `CONFIG_NR_CPUS`

```
#ifndef CONFIG_NR_CPUS
        #define CONFIG_NR_CPUS  1
#endif

#define NR_CPUS         CONFIG_NR_CPUS
```

cpumask `DECLARE_BITMAP` `to_cpumask` `struct cpumask *`

```
#define to_cpumask(bitmap)                                              \
        ((struct cpumask *)(1 ? (bitmap)                                \
                            : (void *)sizeof(__check_is_bitmap(bitmap))))
```

`true` `__check_is_bitmap`

```
static inline int __check_is_bitmap(const unsigned long *bitmap)
{
        return 1;
}
```

`1` `bitmap` `bitmap` `unsigned long *` `cpu_possible_bits` `to_cpumask` `unsigned long` `struct cpumask *`

cpumask API

cpumaskLinux API cpumask `set_cpu_online`

- CPU ;
- CPU ;

```
void set_cpu_online(unsigned int cpu, bool online)
{
    if (online) {
        cpumask_set_cpu(cpu, to_cpumask(cpu_online_bits));
        cpumask_set_cpu(cpu, to_cpumask(cpu_active_bits));
    } else {
        cpumask_clear_cpu(cpu, to_cpumask(cpu_online_bits));
    }
}
```

`state` `cpumask_set_cpu` `cpumask_clear_cpu` `cpumask_set_cpu` `struct cpumask *` `cpu_online_bits`

```
static DECLARE_BITMAP(cpu_online_bits, CONFIG_NR_CPUS) __read_mostly;
```

`cpumask_set_cpu` `set_bit`

```
static inline void cpumask_set_cpu(unsigned int cpu, struct cpumask *dstp)
{
        set_bit(cpumask_check(cpu), cpumask_bits(dstp));
}
```

`set_bit` `cpu_online_bits` `set_bit`

- cpumask_check;
- cpumask_bits.

`cpumask_check` `cpumask_bits` `struct cpumask *` `bits`

```
#define cpumask_bits(maskp) ((maskp)->bits)
```

`set_bit`

```
 static __always_inline void
 set_bit(long nr, volatile unsigned long *addr)
 {
         if (IS_IMMEDIATE(nr)) {
```

```
                asm volatile(LOCK_PREFIX "orb %1,%0"
                        : CONST_MASK_ADDR(nr, addr)
                        : "iq" ((u8)CONST_MASK(nr))
                        : "memory");
        } else {
                asm volatile(LOCK_PREFIX "bts %1,%0"
                        : BITOP_ADDR(addr) : "Ir" (nr) : "memory");
        }
 }
```

`nr` `IS_IMMEDIATE` GCC `__builtin_constant_p`

```
#define IS_IMMEDIATE(nr)    (__builtin_constant_p(nr))
```

`__builtin_constant_p` `cpu` `else`

```
asm volatile(LOCK_PREFIX "bts %1,%0" : BITOP_ADDR(addr) : "Ir" (nr) : "memory");
```

`LOCK_PREFIX` x86 `lock` CPU CPU - DMA cell

`BITOP_ADDR` `(*(volatile long *)` `+m` `+` `m` `BITOP_ADDR`

```
#define BITOP_ADDR(x) "+m" (*(volatile long *) (x))
```

`memory`

`Ir` -

`bts` `CF` cpu 0 `set_bit` `cpu_online_bits` cpumask 0 cpu

`set_cpu_*` API cpumask cpumasks API

cpumask API

cpumaks CPUs

```
#define num_online_cpus()    cpumask_weight(cpu_online_mask)
```

`online` CPUs `cpu_online_mask` `cpumask_weight` `cpumask_weight` `bitmap_weight`

- cpumask bitmap;
- `nr_cpumask_bits` - `NR_CPUS`

```
static inline unsigned int cpumask_weight(const struct cpumask *srcp)
{
    return bitmap_weight(cpumask_bits(srcp), nr_cpumask_bits);
}
```

`num_online_cpus` cpumask CPU

- num_possible_cpus;
- num_active_cpus;
- cpu_online;
- cpu_possible.

Linux `cpumask` API

- `for_each_cpu` - mask cpu;
- `for_each_cpu_not` - cpu;
- `cpumask_clear_cpu` - cpumask cpu;
- `cpumask_test_cpu` - mask cpu;
- `cpumask_setall` - mask cpu;
- `cpumask_size` - 'struct cpumask' ;

- cpumask documentation

initcall

Linux `initcall` Linux

```
early_param("debug", debug_kernel);
```

```
arch_initcall(init_pit_clocksource);
```

Linux Linux `initcall`

```
static int __init nmi_warning_debugfs(void)
{
    debugfs_create_u64("nmi_longest_ns", 0644,
                       arch_debugfs_dir, &nmi_longest_ns);
    return 0;
}
```

arch/x86/kernel/nmi.c `arch_debugfs_dir` `nmi_longest_ns` debugfs `arch_debugfs_dir` `debugfs`
Linux arch/x86/kernel/kdebugfs.c `arch_kdebugfs_init`

```
arch_initcall(arch_kdebugfs_init);
```

Linux `fs` `initcalls` `initcalls` `arch_kdebugfs_dir` `nmi_longest_ns` Linux `initcalls`

- `early` ;
- `core` ;
- `postcore` ;
- `arch` ;
- `susys` ;
- `fs` ;
- `device` ;
- `late` .

`initcall_level_names` init/main.c

```
static char *initcall_level_names[] __initdata = {
    "early",
    "core",
    "postcore",
    "arch",
    "subsys",
    "fs",
    "device",
    "late",
};
```

`initcall` `early initcalls` `core initcalls` `initcall` Linux

initcall Linux

Linux include/linux/init.h `initcall`

```
#define early_initcall(fn)        __define_initcall(fn, early)
#define core_initcall(fn)        __define_initcall(fn, 1)
#define postcore_initcall(fn)        __define_initcall(fn, 2)
#define arch_initcall(fn)        __define_initcall(fn, 3)
#define subsys_initcall(fn)        __define_initcall(fn, 4)
#define fs_initcall(fn)            __define_initcall(fn, 5)
#define device_initcall(fn)        __define_initcall(fn, 6)
#define late_initcall(fn)        __define_initcall(fn, 7)
```

`__define_initcall` `__define_initcall`

- `fn` - `initcalls`
- `id` - `initcall` `initcalls`

`__define_initcall`

```
#define __define_initcall(fn, id) \
    static initcall_t __initcall_##fn##id __used \
    __attribute__((__section__(".initcall" #id ".init"))) = fn; \
    LTO_REFERENCE_INITCALL(__initcall_##fn##id)
```

`__define_initcall` `initcall_t` `initcall`

```
typedef int (*initcall_t)(void);
```

`_-define_initcall` ## `__define_initcall` `.initcall id .init` ELF gcc `__initcall_function_name_id` `__used` include/asm-generic/vmlinux.lds.h `initcalls` `.data`

```
#define INIT_CALLS                    \
        VMLINUX_SYMBOL(__initcall_start) = .;    \
        *(.initcallearly.init)                    \
        INIT_CALLS_LEVEL(0)                        \
        INIT_CALLS_LEVEL(1)                        \
        INIT_CALLS_LEVEL(2)                        \
        INIT_CALLS_LEVEL(3)                        \
        INIT_CALLS_LEVEL(4)                        \
        INIT_CALLS_LEVEL(5)                        \
        INIT_CALLS_LEVEL(rootfs)                \
        INIT_CALLS_LEVEL(6)                        \
        INIT_CALLS_LEVEL(7)                        \
        VMLINUX_SYMBOL(__initcall_end) = .;

#define INIT_DATA_SECTION(initsetup_align)    \
    .init.data : AT(ADDR(.init.data) - LOAD_OFFSET) {       \
        ...                                                 \
        INIT_CALLS                                            \
        ...                                                 \
    }
```

- `__used` include/linux/compiler-gcc.h `gcc`

```
#define __used   __attribute__((__used__))
```

`__define_initcall`

```
LTO_REFERENCE_INITCALL(__initcall_#fn##id)
```

`CONFIG_LTO`

```
#ifdef CONFIG_LTO
#define LTO_REFERENCE_INITCALL(x) \
        static __used __exit void *reference_##x(void)  \
        {                                               \
                return &x;                              \
        }
#else
#define LTO_REFERENCE_INITCALL(x)
#endif
```

`__define_initcall` `*_initcall` Linux `initcalls` `.data` Linux `initcall`

Linux `initcalls` Linux init/main.c `do_basic_setup`

```
static void __init do_basic_setup(void)
{
    ...
    ...
    ...
       do_initcalls();
    ...
    ...
    ...
}
```

Linux `CPU` `do_initcalls` `initcall` `do_initcall_level`

```
static void __init do_initcalls(void)
{
    int level;

    for (level = 0; level < ARRAY_SIZE(initcall_levels) - 1; level++)
        do_initcall_level(level);
}
```

`initcall_levels` `__define_initcall`

```
static initcall_t *initcall_levels[] __initdata = {
    __initcall0_start,
    __initcall1_start,
    __initcall2_start,
    __initcall3_start,
    __initcall4_start,
    __initcall5_start,
    __initcall6_start,
    __initcall7_start,
    __initcall_end,
};
```

Linux `arch/x86/kernel/vmlinux.lds`

```
.init.data : AT(ADDR(.init.data) - 0xffffffff80000000) {
    ...
    ...
    ...
    ...
    __initcall_start = .;
    *(.initcallearly.init)
    __initcall0_start = .;
    *(.initcall0.init)
    *(.initcall0s.init)
    __initcall1_start = .;
    ...
    ...
```

```
}
```

`do_initcall_level` - `initcall` `initcall_command_line` kernel/params.c `parse_args` `do_on_initcall`

```
for (fn = initcall_levels[level]; fn < initcall_levels[level+1]; fn++)
        do_one_initcall(*fn);
```

`do_on_initcall` `initcall`

```
int __init_or_module do_one_initcall(initcall_t fn)
{
    int count = preempt_count();
    int ret;
    char msgbuf[64];

    if (initcall_blacklisted(fn))
        return -EPERM;

    if (initcall_debug)
        ret = do_one_initcall_debug(fn);
    else
        ret = fn();

    msgbuf[0] = 0;

    if (preempt_count() != count) {
        sprintf(msgbuf, "preemption imbalance ");
        preempt_count_set(count);
    }
    if (irqs_disabled()) {
        strlcat(msgbuf, "disabled interrupts ", sizeof(msgbuf));
        local_irq_enable();
    }
    WARN(msgbuf[0], "initcall %pF returned with %s\n", fn, msgbuf);

    return ret;
}
```

`do_on_initcall` preemption `initcall_backlist` `initcalls` `blacklisted_initcalls` `initcall`

```
list_for_each_entry(entry, &blacklisted_initcalls, next) {
    if (!strcmp(fn_name, entry->buf)) {
        pr_debug("initcall %s blacklisted\n", fn_name);
        kfree(fn_name);
        return true;
    }
}
```

`initcalls` `blacklisted_initcalls` Linux Linux

`initcalls` `initcall`

```
if (initcall_debug)
    ret = do_one_initcall_debug(fn);
else
    ret = fn();
```

`initcall_debug` `do_one_initcall_debug` `initcall` `fn()` `initcall_debug`

```
bool initcall_debug;
```

`initcall_debug` Linux

```
initcall_debug    [KNL] Trace initcalls as they are executed.  Useful
                      for working out where the kernel is dying during
                      startup.
```

`do_one_initcall_debug` `do_one_initcall` `do_one_initcall_debug` `initcall` `initcall` pid `initcall`

```
static int __init_or_module do_one_initcall_debug(initcall_t fn)
{
    ktime_t calltime, delta, rettime;
    unsigned long long duration;
    int ret;

    printk(KERN_DEBUG "calling  %pF @ %i\n", fn, task_pid_nr(current));
    calltime = ktime_get();
    ret = fn();
    rettime = ktime_get();
    delta = ktime_sub(rettime, calltime);
    duration = (unsigned long long) ktime_to_ns(delta) >> 10;
    printk(KERN_DEBUG "initcall %pF returned %d after %lld usecs\n",
         fn, ret, duration);

    return ret;
}
```

`initcall` `do_one_initcall` `do_one_initcall_debug` `do_one_initcall` initcall `__preempt_count_add` `__preempt_count_sub` `preemption imbalance`

```
if (preempt_count() != count) {
    sprintf(msgbuf, "preemption imbalance ");
    preempt_count_set(count);
}
```

IRQs `disabled interrupts` `IRQs` `IRQs` `initcall`

```
if (irqs_disabled()) {
    strlcat(msgbuf, "disabled interrupts ", sizeof(msgbuf));
    local_irq_enable();
}
```

Linux Linux `initcall` `initcall`

`initcalls` `rootfs initcalls` include/linux/init.h `rootfs_initcall`

```
#define rootfs_initcall(fn)        __define_initcall(fn, rootfs)
```

rootfs init/initramfs.c `populate_rootfs` initramfs

```
rootfs_initcall(populate_rootfs);
```

```
[    0.199960] Unpacking initramfs...
```

`rootfs_initcall` `console_initcall` `security_initcall` `initcall` `*_initcall_sync` `*_initcall` `_sync`

```
#define core_initcall_sync(fn)        __define_initcall(fn, 1s)
#define postcore_initcall_sync(fn)    __define_initcall(fn, 2s)
#define arch_initcall_sync(fn)        __define_initcall(fn, 3s)
#define subsys_initcall_sync(fn)    __define_initcall(fn, 4s)
#define fs_initcall_sync(fn)        __define_initcall(fn, 5s)
#define device_initcall_sync(fn)    __define_initcall(fn, 6s)
#define late_initcall_sync(fn)        __define_initcall(fn, 7s)
```

Linux Linux

twitter 0xAX email issue

PR **linux-insides**.

- callback
- debugfs
- integer type
- symbols concatenation
- GCC
- Link time optimization
- Introduction to linkers
- Linux kernel command line
- Process identifier
- IRQs
- rootfs
- previous part

Notification Chains in Linux Kernel

Introduction

The Linux kernel is huge piece of C) code which consists from many different subsystems. Each subsystem has its own purpose which is independent of other subsystems. But often one subsystem wants to know something from other subsystem(s). There is special mechanism in the Linux kernel which allows to solve this problem partly. The name of this mechanism is - `notification chains` and its main purpose to provide a way for different subsystems to subscribe on asynchronous events from other subsystems. Note that this mechanism is only for communication inside kernel, but there are other mechanisms for communication between kernel and userspace.

Before we will consider `notification chains` API and implementation of this API, let's look at `Notification chains` mechanism from theoretical side as we did it in other parts of this book. Everything which is related to `notification chains` mechanism is located in the include/linux/notifier.h header file and kernel/notifier.c source code file. So let's open them and start to dive.

Notification Chains related data structures

Let's start to consider `notification chains` mechanism from related data structures. As I wrote above, main data structures should be located in the include/linux/notifier.h header file, so the Linux kernel provides generic API which does not depend on certain architecture. In general, the `notification chains` mechanism represents a list (that's why it named `chains`) of callback) functions which are will be executed when an event will be occurred.

All of these callback functions are represented as `notifier_fn_t` type in the Linux kernel:

```
typedef    int (*notifier_fn_t)(struct notifier_block *nb, unsigned long action, void *data);
```

So we may see that it takes three following arguments:

- `nb` - is linked list of function pointers (will see it now);
- `action` - is type of an event. A notification chain may support multiple events, so we need this parameter to distinguish an event from other events;
- `data` - is storage for private information. Actually it allows to provide additional data information about an event.

Additionally we may see that `notifier_fn_t` returns an integer value. This integer value maybe one of:

- `NOTIFY_DONE` - subscriber does not interested in notification;
- `NOTIFY_OK` - notification was processed correctly;
- `NOTIFY_BAD` - something went wrong;
- `NOTIFY_STOP` - notification is done, but no further callbacks should be called for this event.

All of these results defined as macros in the include/linux/notifier.h header file:

```
#define NOTIFY_DONE        0x0000
#define NOTIFY_OK        0x0001
#define NOTIFY_BAD        (NOTIFY_STOP_MASK|0x0002)
#define NOTIFY_STOP        (NOTIFY_OK|NOTIFY_STOP_MASK)
```

Where `NOTIFY_STOP_MASK` represented by the:

```
#define NOTIFY_STOP_MASK    0x8000
```

macro and means that callbacks will not be called during next notifications.

Each part of the Linux kernel which wants to be notified on a certain event will should provide own `notifier_fn_t` callback function. Main role of the `notification chains` mechanism is to call certain callbacks when an asynchronous event occurred.

The main building block of the `notification chains` mechanism is the `notifier_block` structure:

```
struct notifier_block {
    notifier_fn_t notifier_call;
    struct notifier_block __rcu *next;
    int priority;
};
```

which is defined in the include/linux/notifier.h file. This struct contains pointer to callback function - `notifier_call`, link to the next notification callback and `priority` of a callback function as functions with higher priority are executed first.

The Linux kernel provides notification chains of four following types:

- Blocking notifier chains;
- SRCU notifier chains;
- Atomic notifier chains;
- Raw notifier chains.

Let's consider all of these types of notification chains by order:

In the first case for the `blocking notifier chains`, callbacks will be called/executed in process context. This means that the calls in a notification chain may be blocked.

The second `SRCU notifier chains` represent alternative form of `blocking notifier chains`. In the first case, blocking notifier chains uses `rw_semaphore` synchronization primitive to protect chain links. `SRCU` notifier chains run in process context too, but uses special form of RCU mechanism which is permissible to block in an read-side critical section.

In the third case for the `atomic notifier chains` runs in interrupt or atomic context and protected by spinlock synchronization primitive. The last `raw notifier chains` provides special type of notifier chains without any locking restrictions on callbacks. This means that protection rests on the shoulders of caller side. It is very useful when we want to protect our chain with very specific locking mechanism.

If we will look at the implementation of the `notifier_block` structure, we will see that it contains pointer to the `next` element from a notification chain list, but we have no head. Actually a head of such list is in separate structure depends on type of a notification chain. For example for the `blocking notifier chains`:

```
struct blocking_notifier_head {
    struct rw_semaphore rwsem;
    struct notifier_block __rcu *head;
};
```

or for `atomic notification chains`:

```
struct atomic_notifier_head {
    spinlock_t lock;
    struct notifier_block __rcu *head;
};
```

Now as we know a little about `notification chains` mechanism let's consider implementation of its API.

Notification Chains

Usually there are two sides in a publish/subscriber mechanisms. One side who wants to get notifications and other side(s) who generates these notifications. We will consider notification chains mechanism from both sides. We will consider `blocking notification chains` in this part, because of other types of notification chains are similar to it and differs mostly in protection mechanisms.

Before a notification producer is able to produce notification, first of all it should initialize head of a notification chain. For example let's consider notification chains related to kernel loadable modules. If we will look in the kernel/module.c source code file, we will see following definition:

```
static BLOCKING_NOTIFIER_HEAD(module_notify_list);
```

which defines head for loadable modules blocking notifier chain. The `BLOCKING_NOTIFIER_HEAD` macro is defined in the include/linux/notifier.h header file and expands to the following code:

```
#define BLOCKING_INIT_NOTIFIER_HEAD(name) do {    \
        init_rwsem(&(name)->rwsem);                \
        (name)->head = NULL;                       \
    } while (0)
```

So we may see that it takes name of a name of a head of a blocking notifier chain and initializes read/write semaphore and set head to `NULL` . Besides the `BLOCKING_INIT_NOTIFIER_HEAD` macro, the Linux kernel additionally provides `ATOMIC_INIT_NOTIFIER_HEAD` , `RAW_INIT_NOTIFIER_HEAD` macros and `srcu_init_notifier` function for initialization atomic and other types of notification chains.

After initialization of a head of a notification chain, a subsystem which wants to receive notification from the given notification chain it should register with certain function which is depends on type of notification. If you will look in the include/linux/notifier.h header file, you will see following four function for this:

```
extern int atomic_notifier_chain_register(struct atomic_notifier_head *nh,
        struct notifier_block *nb);

extern int blocking_notifier_chain_register(struct blocking_notifier_head *nh,
        struct notifier_block *nb);

extern int raw_notifier_chain_register(struct raw_notifier_head *nh,
        struct notifier_block *nb);

extern int srcu_notifier_chain_register(struct srcu_notifier_head *nh,
        struct notifier_block *nb);
```

As I already wrote above, we will cover only blocking notification chains in the part, so let's consider implementation of the `blocking_notifier_chain_register` function. Implementation of this function is located in the kernel/notifier.c source code file and as we may see the `blocking_notifier_chain_register` takes two parameters:

- `nh` - head of a notification chain;
- `nb` - notification descriptor.

Now let's look at the implementation of the `blocking_notifier_chain_register` function:

```
int raw_notifier_chain_register(struct raw_notifier_head *nh,
        struct notifier_block *n)
{
    return notifier_chain_register(&nh->head, n);
}
```

As we may see it just returns result of the `notifier_chain_register` function from the same source code file and as we may understand this function does all job for us. Definition of the `notifier_chain_register` function looks:

```
int blocking_notifier_chain_register(struct blocking_notifier_head *nh,
        struct notifier_block *n)
{
    int ret;

    if (unlikely(system_state == SYSTEM_BOOTING))
        return notifier_chain_register(&nh->head, n);

    down_write(&nh->rwsem);
    ret = notifier_chain_register(&nh->head, n);
    up_write(&nh->rwsem);
    return ret;
}
```

As we may see implementation of the `blocking_notifier_chain_register` is pretty simple. First of all there is check which check current system state and if a system in rebooting state we just call the `notifier_chain_register`. In other way we do the same call of the `notifier_chain_register` but as you may see this call is protected with read/write semaphores. Now let's look at the implementation of the `notifier_chain_register` function:

```
static int notifier_chain_register(struct notifier_block **nl,
        struct notifier_block *n)
{
    while ((*nl) != NULL) {
        if (n->priority > (*nl)->priority)
            break;
        nl = &((*nl)->next);
    }
    n->next = *nl;
    rcu_assign_pointer(*nl, n);
    return 0;
}
```

This function just inserts new `notifier_block` (given by a subsystem which wants to get notifications) to the notification chain list. Besides subscribing on an event, subscriber may unsubscribe from a certain events with the set of `unsubscribe` functions:

```
extern int atomic_notifier_chain_unregister(struct atomic_notifier_head *nh,
        struct notifier_block *nb);

extern int blocking_notifier_chain_unregister(struct blocking_notifier_head *nh,
        struct notifier_block *nb);

extern int raw_notifier_chain_unregister(struct raw_notifier_head *nh,
        struct notifier_block *nb);

extern int srcu_notifier_chain_unregister(struct srcu_notifier_head *nh,
        struct notifier_block *nb);
```

When a producer of notifications wants to notify subscribers about an event, the `*.notifier_call_chain` function will be called. As you already may guess each type of notification chains provides own function to produce notification:

```
extern int atomic_notifier_call_chain(struct atomic_notifier_head *nh,
        unsigned long val, void *v);

extern int blocking_notifier_call_chain(struct blocking_notifier_head *nh,
        unsigned long val, void *v);

extern int raw_notifier_call_chain(struct raw_notifier_head *nh,
        unsigned long val, void *v);

extern int srcu_notifier_call_chain(struct srcu_notifier_head *nh,
        unsigned long val, void *v);
```

Let's consider implementation of the `blocking_notifier_call_chain` function. This function is defined in the kernel/notifier.c source code file:

```
int blocking_notifier_call_chain(struct blocking_notifier_head *nh,
        unsigned long val, void *v)
{
    return __blocking_notifier_call_chain(nh, val, v, -1, NULL);
}
```

and as we may see it just returns result of the `__blocking_notifier_call_chain` function. As we may see, the `blocking_notifer_call_chain` takes three parameters:

- `nh` - head of notification chain list;
- `val` - type of a notification;

- `v` - input parameter which may be used by handlers.

But the `__blocking_notifier_call_chain` function takes five parameters:

```
int __blocking_notifier_call_chain(struct blocking_notifier_head *nh,
                   unsigned long val, void *v,
                   int nr_to_call, int *nr_calls)
{
    ...
    ...
    ...
}
```

Where `nr_to_call` and `nr_calls` are number of notifier functions to be called and number of sent notifications. As you may guess the main goal of the `__blocking_notifer_call_chain` function and other functions for other notification types is to call callback function when an event occurred. Implementation of the `__blocking_notifier_call_chain` is pretty simple, it just calls the `notifier_call_chain` function from the same source code file protected with read/write semaphore:

```
int __blocking_notifier_call_chain(struct blocking_notifier_head *nh,
                   unsigned long val, void *v,
                   int nr_to_call, int *nr_calls)
{
    int ret = NOTIFY_DONE;

    if (rcu_access_pointer(nh->head)) {
        down_read(&nh->rwsem);
        ret = notifier_call_chain(&nh->head, val, v, nr_to_call,
                    nr_calls);
        up_read(&nh->rwsem);
    }
    return ret;
}
```

and returns its result. In this case all job is done by the `notifier_call_chain` function. Main purpose of this function informs registered notifiers about an asynchronous event:

```
static int notifier_call_chain(struct notifier_block **nl,
                   unsigned long val, void *v,
                   int nr_to_call, int *nr_calls)
{
    ...
    ...
    ...
    ret = nb->notifier_call(nb, val, v);
    ...
    ...
    ...
    return ret;
}
```

That's all. In generall all looks pretty simple.

Now let's consider on a simple example related to loadable modules. If we will look in the kernel/module.c. As we already saw in this part, there is:

```
static BLOCKING_NOTIFIER_HEAD(module_notify_list);
```

definition of the `module_notify_list` in the kernel/module.c source code file. This definition determines head of list of blocking notifier chains related to kernel modules. There are at least three following events:

- MODULE_STATE_LIVE
- MODULE_STATE_COMING
- MODULE_STATE_GOING

in which maybe interested some subsystems of the Linux kernel. For example tracing of kernel modules states. Instead of direct call of the `atomic_notifier_chain_register`, `blocking_notifier_chain_register` and etc., most notification chains come with a set of wrappers used to register to them. Registatrion on these modules events is going with the help of such wrapper:

```
int register_module_notifier(struct notifier_block *nb)
{
    return blocking_notifier_chain_register(&module_notify_list, nb);
}
```

If we will look in the kernel/tracepoint.c source code file, we will see such registration during initialization of tracepoints:

```
static __init int init_tracepoints(void)
{
    int ret;

    ret = register_module_notifier(&tracepoint_module_nb);
    if (ret)
        pr_warn("Failed to register tracepoint module enter notifier\n");

    return ret;
}
```

Where `tracepoint_module_nb` provides callback function:

```
static struct notifier_block tracepoint_module_nb = {
    .notifier_call = tracepoint_module_notify,
    .priority = 0,
};
```

When one of the `MODULE_STATE_LIVE`, `MODULE_STATE_COMING` or `MODULE_STATE_GOING` events occurred. For example the `MODULE_STATE_LIVE` the `MODULE_STATE_COMING` notifications will be sent during execution of the init_module system call. Or for example `MODULE_STATE_GOING` will be sent during execution of the delete_module `system call`:

```
SYSCALL_DEFINE2(delete_module, const char __user *, name_user,
        unsigned int, flags)
{
    ...
    ...
    ...
    blocking_notifier_call_chain(&module_notify_list,
                    MODULE_STATE_GOING, mod);
    ...
    ...
    ...
}
```

Thus when one of these system call will be called from userspace, the Linux kernel will send certain notification depends on a system call and the `tracepoint_module_notify` callback function will be called.

That's all.

Links

- C programming langauge)
- API
- callback)
- RCU
- spinlock
- loadable modules

- semaphore
- tracepoints
- system call
- init_module system call
- delete_module
- previous part

Linux

LinuxB+

-
-
-

Linux ——

Linux include/linux/list.h free-electrons.com

include/linux/types.h

```
struct list_head {
    struct list_head *next, *prev;
};
```

glib

```
struct GList {
  gpointer data;
  GList *next;
  GList *prev;
};
```

-

```
struct nmi_desc {
    spinlock_t lock;
    struct list_head head;
};
```

`list_head` drivers/char/misc.c API

```
#define MISC_MAJOR              10
```

```
ls -l /dev |  grep 10
crw-------   1 root root     10, 235 Mar 21 12:01 autofs
drwxr-xr-x  10 root root         200 Mar 21 12:01 cpu
crw-------   1 root root     10,  62 Mar 21 12:01 cpu_dma_latency
crw-------   1 root root     10, 203 Mar 21 12:01 cuse
drwxr-xr-x   2 root root         100 Mar 21 12:01 dri
crw-rw-rw-   1 root root     10, 229 Mar 21 12:01 fuse
crw-------   1 root root     10, 228 Mar 21 12:01 hpet
crw-------   1 root root     10, 183 Mar 21 12:01 hwrng
crw-rw----+  1 root kvm      10, 232 Mar 21 12:01 kvm
crw-rw----   1 root disk     10, 237 Mar 21 12:01 loop-control
crw-------   1 root root     10, 227 Mar 21 12:01 mcelog
crw-------   1 root root     10,  59 Mar 21 12:01 memory_bandwidth
crw-------   1 root root     10,  61 Mar 21 12:01 network_latency
crw-------   1 root root     10,  60 Mar 21 12:01 network_throughput
crw-r-----   1 root kmem     10, 144 Mar 21 12:01 nvram
brw-rw----   1 root disk      1,  10 Mar 21 12:01 ram10
crw--w----   1 root tty       4,  10 Mar 21 12:01 tty10
crw-rw----   1 root dialout   4,  74 Mar 21 12:01 ttyS10
crw-------   1 root root     10,  63 Mar 21 12:01 vga_arbiter
crw-------   1 root root     10, 137 Mar 21 12:01 vhci
```

`miscdevice`

```
struct miscdevice
{
      int minor;
      const char *name;
      const struct file_operations *fops;
      struct list_head list;
      struct device *parent;
      struct device *this_device;
      const char *nodename;
      mode_t mode;
};
```

`list`

```
static LIST_HEAD(misc_list);
```

`list_head`

```
#define LIST_HEAD(name) \
    struct list_head name = LIST_HEAD_INIT(name)
```

`LIST_HEAD_INIT` `name` `prev` `next`

```
#define LIST_HEAD_INIT(name) { &(name), &(name) }
```

`misc_register` `INIT_LIST_HEAD` `miscdevice->list`

```
INIT_LIST_HEAD(&misc->list);
```

`LIST_HEAD_INIT`

```
static inline void INIT_LIST_HEAD(struct list_head *list)
{
    list->next = list;
    list->prev = list;
}
```

`device_create`

```
list_add(&misc->list, &misc_list);
```

`list.h`

```
static inline void list_add(struct list_head *new, struct list_head *head)
{
    __list_add(new, head, head->next);
}
```

3 `__list_add`

- new -
- head - `head` .
- head->next - `head`

`__list_add`

```
static inline void __list_add(struct list_head *new,
                  struct list_head *prev,
                  struct list_head *next)
```

```
{
    next->prev = new;
    new->next = next;
    new->prev = prev;
    prev->next = new;
}
```

`prev` `next` `LIST_HEAD_INIT` `misc` `miscdevice->list`

```
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
```

- ptr -
- type - ;
- member - `list_head`

```
const struct miscdevice *p = list_entry(v, struct miscdevice, list)
```

`p->minor` `p->name` `miscdevice` `list_entry`

```
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
```

`container_of`

```
#define container_of(ptr, type, member) ({                      \
    const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
    (type *)( (char *)__mptr - offsetof(type,member) );})
```

```
#include <stdio.h>

int main() {
    int i = 0;
    printf("i = %d\n", ({++i; ++i;}));
    return 0;
}
```

`2`

`typeof` , `container_of` `container_of` 0 `0`

```
#include <stdio.h>

struct s {
        int field1;
        char field2;
    char field3;
};

int main() {
    printf("%p\n", &((struct s*)0)->field3);
    return 0;
```

```
}
```

`0x5`

`offsetof`

```
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
```

`container_of type list_head member ptr )` `__mptr` `ptr` `list_head` `type` `member`

> `ptr struct list_head *` `imcompatible pointer types` warning `((type *)0)->member type member`

`offsetof`

`list_add` `list_entry` `<linux/list.h>` API

- list_add
- list_add_tail
- list_del
- list_replace
- list_move
- list_is_last
- list_empty
- list_cut_position
- list_splice
- list_for_each
- list_for_each_entry

API

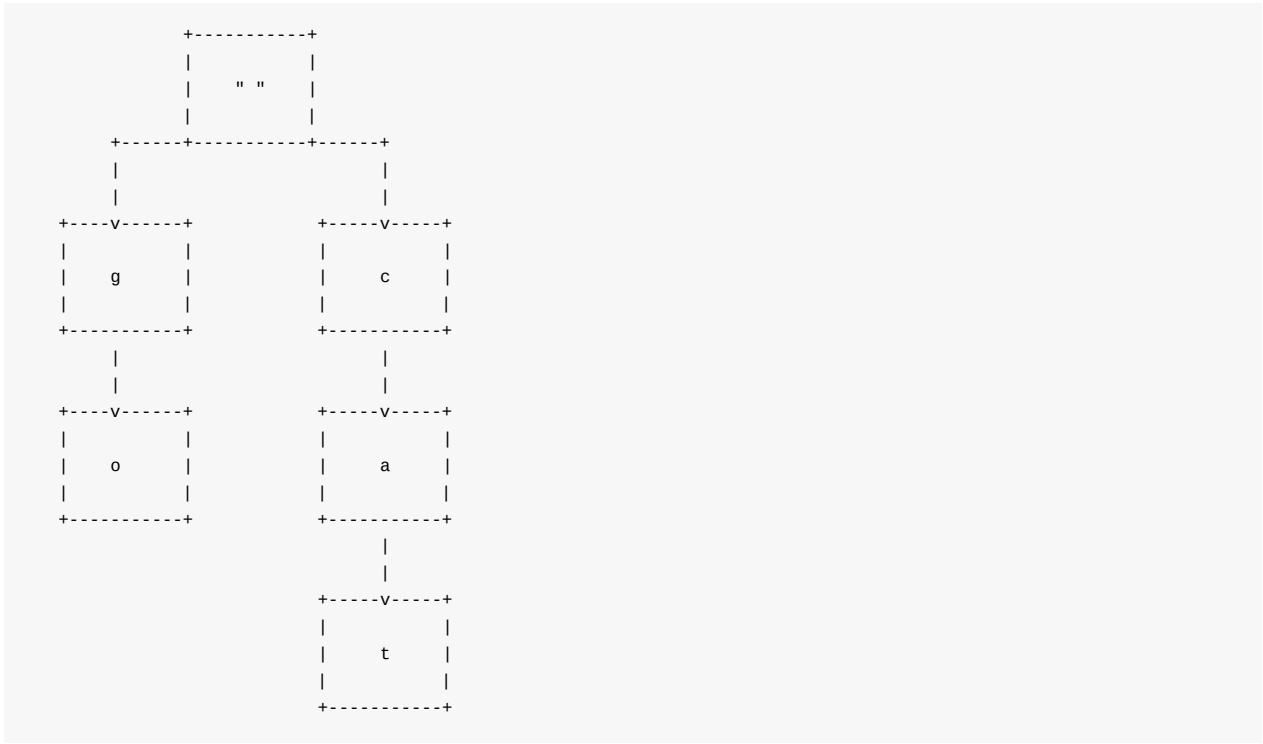
Linux

Linux Radix treeLinux `radix tree` API

- include/linux/radix-tree.h
- lib/radix-tree.c

`radix tree` Radix tree `trie` trie associative array `-key-value`

trie `n-tree`



`trie` `go` `cat` trie `radix tree` `trie`

Linux Radix Radix include/linux/radix-tree.h :

```
struct radix_tree_root {
        unsigned int            height;
        gfp_t                   gfp_mask;
        struct radix_tree_node  __rcu *rnode;
};
```

radix root

- `height` -
- `gfp_mask` -
- `rnode` -

`gfp_mask` :

Linux flag - `gfp_mask` `GFP_` `GFP_NOIO` IO `__GFP_HIGHMEM` `GFP_ATOMIC`

`rnode`

```
struct radix_tree_node {
        unsigned int    path;
        unsigned int    count;
        union {
                struct {
                        struct radix_tree_node *parent;
                        void *private_data;
                };
                struct rcu_head rcu_head;
        };
        /* For tree user */
        struct list_head private_list;
        void __rcu      *slots[RADIX_TREE_MAP_SIZE];
        unsigned long   tags[RADIX_TREE_MAX_TAGS][RADIX_TREE_TAG_LONGS];
};
```

- `path` -
- `count` -
- `parent` -
- `private_data` -
- `rcu_head` - RCU
- `private_list` -

`radix_tree_node` `tags` `slots` Radix slots slots NULL Linux Radix `tags` Radix

radix radix API

Linux API

radix

`RADIX_TREE`

```
RADIX_TREE(name, gfp_mask);
`
```

`name` `RADIX_TREE` radix `RADIX_TREE`

```
#define RADIX_TREE(name, mask) \
         struct radix_tree_root name = RADIX_TREE_INIT(mask)

#define RADIX_TREE_INIT(mask)   { \
        .height = 0,              \
        .gfp_mask = (mask),       \
        .rnode = NULL,            \
}
```

`RADIX_TREE` `name` `radix_tree_root` `RADIX_TREE_INIT` `mask` `RADIX_TREE_INIT` `radix_tree_root` gfp_mask `mask` `radix_tree_root` `mask` `INIT_RADIX_TREE`

```
struct radix_tree_root my_radix_tree;
INIT_RADIX_TREE(my_tree, gfp_mask_for_my_radix_tree);
```

`INIT_RADIX_TREE`

```
#define INIT_RADIX_TREE(root, mask)  \
do {                                 \
```

```
        (root)->height = 0;          \
        (root)->gfp_mask = (mask);   \
        (root)->rnode = NULL;        \
} while (0)
```

`INIT_RADIX_TREE` `RADIX_TREE_INIT`

radix

- `radix_tree_insert` ;
- `radix_tree_delete` .

`radix_tree_insert`

- radix root
-
-

`radix_tree_delete` `radix_tree_insert`

radix The search in a radix tree implemented in two ways:

- `radix_tree_lookup` ;
- `radix_tree_gang_lookup` ;
- `radix_tree_lookup_slot` .

`radix_tree_lookup`

- radix root
-

radix

`radix_tree_gang_lookup`

```
unsigned int radix_tree_gang_lookup(struct radix_tree_root *root,
                                    void **results,
                                    unsigned long first_index,
                                    unsigned int max_items);
```

`max_items`

`radix_tree_lookup_slot` slot

- Radix tree
- Trie

Linux ——

Linux

Linux bitmap API Linux `API`

- lib/bitmap.c
- include/linux/bitmap.h

  x86_64

- arch/x86/include/asm/bitops.h

  Linux / CPU cpumasks bit array Linux

bit array Linux

`API` Linux `unsigned long`

```
unsigned long my_bitmap[8]
```

`DECLARE_BITMAP` include/linux/types.h

```
#define DECLARE_BITMAP(name,bits) \
    unsigned long name[BITS_TO_LONGS(bits)]
```

`DECLARE_BITMAP`

- `name` -;
- `bits` -;

`BITS_TO_LONGS(bits)` `unsigned long` `BITS_TO_LONGS` `long` `bits` `8`

```
#define BITS_PER_BYTE           8
#define DIV_ROUND_UP(n,d) (((n) + (d) - 1) / (d))
#define BITS_TO_LONGS(nr)       DIV_ROUND_UP(nr, BITS_PER_BYTE * sizeof(long))
```

`DECLARE_BITMAP(my_bitmap, 64)`

```
>>> (((64) + (64) - 1) / (64))
1
```

```
unsigned long my_bitmap[1];
```

API API arch/x86/include/asm/bitops.h

- `set_bit` ;
- `clear_bit` .

arch/x86/include/asm/bitops.h atomic

`x86` xchgcmpxchg lock `set_bit` `clear_bit`

non-atomic `set_bit` `clear_bit` arch/x86/include/asm/bitops.h `__set_bit` :

```
static inline void __set_bit(long nr, volatile unsigned long *addr)
{
    asm volatile("bts %1,%0" : ADDR : "Ir" (nr) : "memory");
}
```

- `nr` - LCTT 0
- `addr` -

`addr` `volatile` `__set_bit` bts `nr` CF LCTT `nr` 1

`nr` `addr` `ADDR` `ADDR` `+m`

```
#define ADDR                BITOP_ADDR(addr)
#define BITOP_ADDR(x) "+m" (*(volatile long *) (x))
```

`+m` `__set_bit`

- `+m` - `+`
- `I` -
- `r` -

`memory` atomicnon-atomic

```
static __always_inline void
set_bit(long nr, volatile unsigned long *addr)
{
    if (IS_IMMEDIATE(nr)) {
        asm volatile(LOCK_PREFIX "orb %1,%0"
            : CONST_MASK_ADDR(nr, addr)
            : "iq" ((u8)CONST_MASK(nr))
            : "memory");
    } else {
        asm volatile(LOCK_PREFIX "bts %1,%0"
            : BITOP_ADDR(addr) : "Ir" (nr) : "memory");
    }
}
```

LCTT BITOP_ADDR `#define BITOP_ADDR(x) "=m" (*(volatile long *) (x))` ORB

`__set_bit` `__always_inline` `__always_inline` include/linux/compiler-gcc.h `always_inline`

```
#define __always_inline inline __attribute__((always_inline))
```

Linux `set_bit` `set_bit` `IS_IMMEDIATE` gcc

```
#define IS_IMMEDIATE(nr)        (__builtin_constant_p(nr))
```

`__builtin_constant_p` `1` `0` `bts` `1` 0 `__set_bit` `CONST_MASK_ADDR`

```
#define CONST_MASK_ADDR(nr, addr)    BITOP_ADDR((void *)(addr) + ((nr)>>3))
```

`0x1000` `0x9` `0x9` `+` `addr + 1`：

```
>>> hex(0x1000 + (0x9 >> 3))
'0x1001'
```

`CONST_MASK` `1` `0`

```
#define CONST_MASK(nr)            (1 << ((nr) & 7))
```

```
>>> bin(1 << (0x9 & 7))
'0b10'
```

`0x4097` `9` 1

```
>>> bin(0x4097)
'0b1000000010010111'
>>> bin((0x4097 >> 0x9) | (1 << (0x9 & 7)))
'0b100010'
```

`9` LCTT 9 0 0010 1 1

`LOCK_PREFIX` lock

`set_bit` `__set_bit` Linux `clear_bit` `__clear_bit` `__clear_bit`

```
static inline void __clear_bit(long nr, volatile unsigned long *addr)
{
    asm volatile("btr %1,%0" : ADDR : "Ir" (nr));
}
```

`__clear_bit` btr `bts` `btr` `bts` LCTT btr bts `CF`

`__clear_bit` `clear_bit`

```
static __always_inline void
clear_bit(long nr, volatile unsigned long *addr)
{
    if (IS_IMMEDIATE(nr)) {
        asm volatile(LOCK_PREFIX "andb %1,%0"
            : CONST_MASK_ADDR(nr, addr)
            : "iq" ((u8)~CONST_MASK(nr)));
    } else {
        asm volatile(LOCK_PREFIX "btr %1,%0"
            : BITOP_ADDR(addr)
            : "Ir" (nr));
    }
}
```

`set_bit` `clear_bit` `btr` `set_bit` `bts` `clear_bit` `set_bit`

Linux Linux `test_bit` arch/x86/include/asm/bitops.h `constant_test_bit` `variable_test_bit`

```
#define test_bit(nr, addr)            \
    (__builtin_constant_p((nr))                \
     ? constant_test_bit((nr), (addr))           \
     : variable_test_bit((nr), (addr)))
```

`nr` `test_bit` `constant_test_bit` `variable_test_bit` `variable_test_bit`

```
static inline int variable_test_bit(long nr, volatile const unsigned long *addr)
{
    int oldbit;

    asm volatile("bt %2,%1\n\t"
             "sbb %0,%0"
             : "=r" (oldbit)
             : "m" (*(unsigned long *)addr), "Ir" (nr));

    return oldbit;
}
```

`variable_test_bit` `set_bit` bt sbb `bt` `bit test` CF `sbb` `CF` `CF` `sbb` `00000000 - CF` `oldbit`

`constant_test_bit` `set_bit`

```
static __always_inline int constant_test_bit(long nr, const volatile unsigned long *addr)
{
    return ((1UL << (nr & (BITS_PER_LONG-1))) &
        (addr[nr >> _BITOPS_LONG_SHIFT])) != 0;
}
```

`1` `0` `CONST_MASK`

Linux

- `__change_bit`;
- `change_bit`.

`set_bit` `__set_bit` `__change_bit`

```
static inline void __change_bit(long nr, volatile unsigned long *addr)
{
    asm volatile("btc %1,%0" : ADDR : "Ir" (nr));
}
```

`__change_bit` `__set_bit` btc `bts` `CF` `1` `0`

```
>>> int(not 1)
0
>>> int(not 0)
1
```

`__change_bit` `change_bit`

```
static inline void change_bit(long nr, volatile unsigned long *addr)
{
    if (IS_IMMEDIATE(nr)) {
        asm volatile(LOCK_PREFIX "xorb %1,%0"
            : CONST_MASK_ADDR(nr, addr)
            : "iq" ((u8)CONST_MASK(nr)));
```

```
    } else {
        asm volatile(LOCK_PREFIX "btc %1,%0"
            : BITOP_ADDR(addr)
            : "Ir" (nr));
    }
}
```

`set_bit` `xor` `or` `btc` LCTT `bts` `bts`

API

arch/x86/include/asm/bitops.h API Linux API include/linux/bitmap.h lib/bitmap.c include/linux/bitops.h

4

- `for_each_set_bit`
- `for_each_set_bit_from`
- `for_each_clear_bit`
- `for_each_clear_bit_from`

`for_each_set_bit`

```
#define for_each_set_bit(bit, addr, size) \
    for ((bit) = find_first_bit((addr), (size));        \
         (bit) < (size);                    \
         (bit) = find_next_bit((addr), (size), (bit) + 1))
```

`find_first_bit`

arch/x86/include/asm/bitops.h `64-bit` `32-bit` API

API

- `bitmap_zero` ;
- `bitmap_fill` .

`1` `bitmap_zero`

```
static inline void bitmap_zero(unsigned long *dst, unsigned int nbits)
{
    if (small_const_nbits(nbits))
        *dst = 0UL;
    else {
        unsigned int len = BITS_TO_LONGS(nbits) * sizeof(unsigned long);
        memset(dst, 0, len);
    }
}
```

`nbits` `small_const_nbits`

```
#define small_const_nbits(nbits) \
    (__builtin_constant_p(nbits) && (nbits) <= BITS_PER_LONG)
```

`nbits` `BITS_PER_LONG` `64` `long` 0 `long` memset

`bitmap_fill` `biramp_zero` `0xff` `0b11111111`

```
static inline void bitmap_fill(unsigned long *dst, unsigned int nbits)
```

```
{
    unsigned int nlongs = BITS_TO_LONGS(nbits);
    if (!small_const_nbits(nbits)) {
        unsigned int len = (nlongs - 1) * sizeof(unsigned long);
        memset(dst, 0xff,  len);
    }
    dst[nlongs - 1] = BITMAP_LAST_WORD_MASK(nbits);
}
```

`bitmap_fill` `bitmap_zero` include/linux/bitmap.h `bitmap_zero` `bitmap_copy` memcpy memset `bitmap_and` , `bitmap_or` , `bitamp_xor` include/linux/bitmap.h

LCTT Linux

- bitmap
- linked data structures
- tree data structures
- hot-plug
- cpumasks
- IRQs
- API
- atomic operations
- xchg instruction
- cmpxchg instruction
- lock instruction
- bts instruction
- btr instruction
- bt instruction
- sbb instruction
- btc instruction
- man memcpy
- man memset
- CF
- inline assembler
- gcc

-
- Elf64
-

Linux init `initrd` lockdep

Linux

64

Intel

... 64 Linux

- 32
- PAE
- IA-32e

  `IA-32e`

- `CR0.PG`
- `CR4.PAE`
- `IA32_EFER.LME`

arch/x86/boot/compressed/head_64.S

```
movl    $(X86_CR0_PG | X86_CR0_PE), %eax
movl    %eax, %cr0
```

and

```
movl    $MSR_EFER, %ecx
rdmsr
btsl    $_EFER_LME, %eax
wrmsr
```

`x86_64` `4096` `4096` `512` `PAE` `IA32_EFER.LME` Linux `x86_64`
4CPU

```
leal    pgtable(%ebx), %eax
movl    %eax, %cr3
```

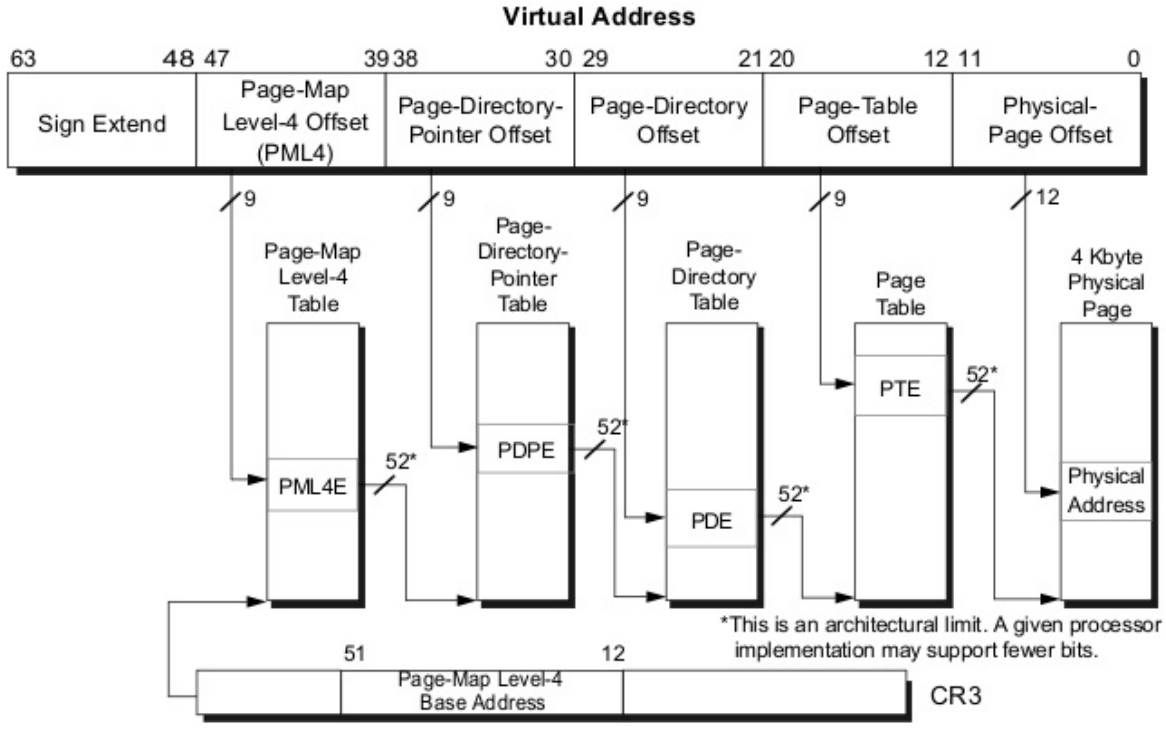
`cr3` `cr3` Linux `PML4` `Page Global Directory` `cr3` 64

```
63                52 51                                                  32
```

```
 ---------------------------------------------------------------------------
|                    |                                                      |
|    Reserved MBZ    |           Address of the top level structure         |
|                    |                                                      |
 ---------------------------------------------------------------------------
31                                 12 11           5     4     3 2          0
 ---------------------------------------------------------------------------
|                                    |              |  P  |  P  |            |
|  Address of the top level structure |   Reserved   |  C  |  W  |  Reserved  |
|                                    |              |  D  |  T  |            |
 ---------------------------------------------------------------------------
```

- 0 2 -
- 12 51 -
- 3 4 - PWT Page-Level Writethrough PCD Page-level Cache Disable
- - 0
- 52 63 - 0

- MMU
- 64 48 `2^48` 256TB
- `cr3`
- 39 47 4 30 38 3 29 21 2 12 20 1 0 11



`CPL (Current Privilege Level)` `CPL < 3`

```
63  62                52 51                                                32
 ---------------------------------------------------------------------------
| N |                    |                                                  |
|   |     Available      |     Address of the paging structure on lower level  |
| X |                    |                                                  |
 ---------------------------------------------------------------------------
31                                          12 11  9 8 7 6 5   4   3 2 1     0
```

```
 --------------------------------------------------------------------------------
|                                           |     | M |I| | P | P |U|W|    |
| Address of the paging structure on lower level | AVL | B |G|A| C | W | | |  P |
|                                           |     | Z |N| | D | T |S|R|    |
 --------------------------------------------------------------------------------
```

- 63 - N/X
- 52 62 - CPU
- 12 51 -
- 9 11 - CPU
- MBZ - 0
-
- A -
- PWT PCD
- U/S - /
- R/W -
- P -

Linux 4

Linux

`x86_64` Linux 4

-
-
-
-

Linux `System.map`

```
$ grep "start_kernel" System.map
ffffffff81efe497 T x86_64_start_kernel
ffffffff81efeaa2 T start_kernel
```

`0xffffffff81efe497` `start_kernel` `x86_64_start_kernel` `x86_64` `2^64` 48
48 64

```
0xffffffffffffffff  +-----------+
                    |           |
                    |           | Kernelspace
                    |           |
 0xffff800000000000 +-----------+
                    |           |
                    |           |
                    |   hole    |
                    |           |
                    |           |
0x00007fffffffffff  +-----------+
                    |           |
                    |           |  Userspace
                    |           |
0x0000000000000000  +-----------+
```

`sign extension` 48 48 63 0 1

-
-

`0x000000000000000` `0x00007fffffffffff` `0xffff8000000000` `0xffffffffffffffff` 48 63 0 1 48

```
0000000000000000 - 00007fffffffffff (=47 bits) user space, different per mm
hole caused by [48:63] sign extension
ffff800000000000 - ffff87ffffffffff (=43 bits) guard hole, reserved for hypervisor
ffff880000000000 - ffffc7ffffffffff (=64 TB) direct mapping of all phys. memory
ffffc80000000000 - ffffc8ffffffffff (=40 bits) hole
ffffc90000000000 - ffffe8ffffffffff (=45 bits) vmalloc/ioremap space
ffffe90000000000 - ffffe9ffffffffff (=40 bits) hole
ffffea0000000000 - ffffeaffffffffff (=40 bits) virtual memory map (1TB)
... unused hole ...
ffffec0000000000 - fffffc0000000000 (=44 bits) kasan shadow memory (16TB)
... unused hole ...
ffffff0000000000 - ffffff7fffffffff (=39 bits) %esp fixup stacks
... unused hole ...
ffffffff80000000 - ffffffffa0000000 (=512 MB)  kernel text mapping, from phys 0
ffffffffa0000000 - ffffffffff5fffff (=1525 MB) module mapping space
ffffffffff600000 - ffffffffffdfffff (=8 MB) vsyscalls
ffffffffffe00000 - ffffffffffffffff (=2 MB) unused hole
```

(hypervisor) (guard hole) arch/x86/include/asm/page_64_types.h

```
#define __PAGE_OFFSET _AC(0xffff880000000000, UL)
```

`__PAGE_OFFSET` `0xffff800000000000` `0xffff80ffffffffff` 3

- `ffff880000000000` vmalloc 1TB `ksan` (shadow memory) commit

`esp` - `0` `__PAGE_OFFSET`

```
#define __START_KERNEL_map      _AC(0xffffffff80000000, UL)
```

`.text` `CONFIG_PHYSICAL_START` ELF64

```
readelf -s vmlinux | grep ffffffff81000000
     1: ffffffff81000000     0 SECTION LOCAL  DEFAULT    1
 65099: ffffffff81000000     0 NOTYPE  GLOBAL DEFAULT    1 _text
 90766: ffffffff81000000     0 NOTYPE  GLOBAL DEFAULT    1 startup_64
```

`CONFIG_PHYSICAL_START` `0x1000000` `vmlinux` - `0xffffffff80000000` - `0x1000000` `0xffffffff80000000 + 1000000 = 0xffffffff81000000`

`vsyscalls` 2M

```
0xffffffff81000000
```

```
1111111111111111 111111111 111111110 000001000 000000000 000000000000
      63:48        47:39     38:30     29:21     20:12      11:0
```

- `48-63` -
- `37-49` - 4
- `30-38` - 3
- `21-29` - 2
- `12-20` - 1

- `0-11` -

Linux

- Paging on Wikipedia
- Intel 64 and IA-32 architectures software developer's manual volume 3A
- MMU
- ELF64
- Documentation/x86/x86_64/mm.txt
- Last part - Kernel booting process

ELF

ELF (Executable and Linkable Format)(core dumps) LinuxUnix 64ELF

ELF

- ELF(ELF header) - CPU
- (Program header table) - (segments)
- (Section header table) - (sections)

ELF(ELF header)

ELF(ELF header)

- ELF - ELF
- - ,...
-
- ELF
-
-
-
- ELF(ELF header)
-
- ...

ELF64 header `elf64_hdr`

```
typedef struct elf64_hdr {
    unsigned char    e_ident[EI_NIDENT];
    Elf64_Half e_type;
    Elf64_Half e_machine;
    Elf64_Word e_version;
    Elf64_Addr e_entry;
    Elf64_Off e_phoff;
    Elf64_Off e_shoff;
    Elf64_Word e_flags;
    Elf64_Half e_ehsize;
    Elf64_Half e_phentsize;
    Elf64_Half e_phnum;
    Elf64_Half e_shentsize;
    Elf64_Half e_shnum;
    Elf64_Half e_shstrndx;
} Elf64_Ehdr;
```

elf.h

(sections)

ELF(sections) (index)(sections)

-
-
-
-
-
-
-

-
-
-

linux `elf64_shdr` :

```
typedef struct elf64_shdr {
    Elf64_Word sh_name;
    Elf64_Word sh_type;
    Elf64_Xword sh_flags;
    Elf64_Addr sh_addr;
    Elf64_Off sh_offset;
    Elf64_Xword sh_size;
    Elf64_Word sh_link;
    Elf64_Word sh_info;
    Elf64_Xword sh_addralign;
    Elf64_Xword sh_entsize;
} Elf64_Shdr;
```

elf.h

(Program header table)

(sections)(segments) (segments)

```
typedef struct elf64_phdr {
    Elf64_Word p_type;
    Elf64_Word p_flags;
    Elf64_Off p_offset;
    Elf64_Addr p_vaddr;
    Elf64_Addr p_paddr;
    Elf64_Xword p_filesz;
    Elf64_Xword p_memsz;
    Elf64_Xword p_align;
} Elf64_Phdr;
```

`elf64_phdr` elf.h .

EFL Documentation `vmlinux` ELF

vmlinux

`vmlinux` ELF `readelf`

```
$ readelf -h  vmlinux
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1000000
  Start of program headers:          64 (bytes into file)
  Start of section headers:          381608416 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         5
  Size of section headers:           64 (bytes)
```

```
  Number of section headers:         73
  Section header string table index: 70
```

`vmlinux` 64 Documentation/x86/x86_64/mm.txt :

```
ffffffff80000000 - ffffffffa0000000 (=512 MB)  kernel text mapping, from phys 0
```

`vmlinux` ELF

```
$ readelf -s vmlinux | grep ffffffff81000000
     1: ffffffff81000000     0 SECTION LOCAL  DEFAULT    1
 65099: ffffffff81000000     0 NOTYPE  GLOBAL DEFAULT    1 _text
 90766: ffffffff81000000     0 NOTYPE  GLOBAL DEFAULT    1 startup_64
```

`startup_64` `ffffffff80000000`, `ffffffff81000000`

arch/x86/kernel/vmlinux.lds.S :

```
    . = __START_KERNEL;
    ...
    ...
    ..
    /* Text and read-only data */
    .text :  AT(ADDR(.text) - LOAD_OFFSET) {
        _text = .;
        ...
        ...
        ...
    }
```

`__START_KERNEL` :

```
#define __START_KERNEL        (__START_KERNEL_map + __PHYSICAL_START)
```

`__START_KERNEL_map` `ffffffff80000000` `__PHYSICAL_START` `0x1000000` `startup_64` `ffffffff81000000`

```
readelf -l vmlinux

Elf file type is EXEC (Executable file)
Entry point 0x1000000
There are 5 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000200000 0xffffffff81000000 0x0000000001000000
                 0x0000000000cfd000 0x0000000000cfd000  R E    200000
  LOAD           0x0000000001000000 0xffffffff81e00000 0x0000000001e00000
                 0x0000000000100000 0x0000000000100000  RW     200000
  LOAD           0x0000000001200000 0x0000000000000000 0x0000000001f00000
                 0x0000000000014d98 0x0000000000014d98  RW     200000
  LOAD           0x0000000001315000 0xffffffff81f15000 0x0000000001f15000
                 0x000000000011d000 0x0000000000279000  RWE    200000
  NOTE           0x0000000000b17284 0xffffffff81917284 0x0000000001917284
                 0x0000000000000024 0x0000000000000024         4

 Section to Segment mapping:
  Segment Sections...
   00     .text .notes __ex_table .rodata __bug_table .pci_fixup .builtin_fw
          .tracedata __ksymtab __ksymtab_gpl __kcrctab __kcrctab_gpl
          __ksymtab_strings __param __modver
```

```
01     .data .vvar
02     .data..percpu
03     .init.text .init.data .x86_cpu_dev.init .altinstructions
       .altinstr_replacement .iommu_table .apicdrivers .exit.text
       .smp_locks .data_nosave .bss .brk
```

(sections)(segments) - `arch/x86/kernel/vmlinux.lds` (sections)

ELF(Executable and Linkable Format) -

Inline assembly

Introduction

While reading source code in the Linux kernel, I often see statements like this:

```
__asm__("andq %%rsp,%0; ":"=r" (ti) : "0" (CURRENT_MASK));
```

Yes, this is inline assembly or in other words assembler code which is integrated in a high level programming language. In this case the high level programming language is C. Yes, the `C` programming language is not very high-level, but still.

If you are familiar with the assembly programming language, you may notice that `inline assembly` is not very different from normal assembler. Moreover, the special form of inline assembly which is called `basic form` is exactly the same. For example:

```
__asm__("movq %rax, %rsp");
```

or:

```
__asm__("hlt");
```

The same code (of course without `__asm__` prefix) you might see in plain assembly code. Yes, this is very similar, but not so simple as it might seem at first glance. Actually, the GCC supports two forms of inline assembly statements:

- `basic` ;
- `extended` .

The basic form consists of only two things: the `__asm__` keyword and the string with valid assembler instructions. For example it may look something like this:

```
__asm__("movq    $3, %rax\t\n"
        "movq    %rsi, %rdi");
```

The `asm` keyword may be used in place of `__asm__` , however `__asm__` is portable whereas the `asm` keyword is a `GNU` extension. In further examples I will only use the `__asm__` variant.

If you know assembly programming language this looks pretty familiar. The main problem is in the second form of inline assembly statements - `extended` . This form allows us to pass parameters to an assembly statement, perform jumps etc. Does not sound difficult, but requires knowledge of special rules in addition to knowledge of the assembly language. Every time I see yet another piece of inline assembly code in the Linux kernel, I need to refer to the official documentation of `GCC` to remember how a particular `qualifier` behaves or what the meaning of `=&r` is for example.

I've decided to write this part to consolidate my knowledge related to the inline assembly, as inline assembly statements are quite common in the Linux kernel and we may see them in linux-insides parts sometimes. I thought that it would be useful if we have a special part which contains information on more important aspects of the inline assembly. Of course you may find comprehensive information about inline assembly in the official documentation, but I like to put everything in one place.

Note: This part will not provide guide for assembly programming. It is not intended to teach you to write programs with assembler or to know what one or another assembler instruction means. Just a little memo for extended asm.

Introduction to extended inline assembly

So, let's start. As I already mentioned above, the `basic` assembly statement consists of the `asm` or `__asm__` keyword and set of assembly instructions. This form is in no way different from "normal" assembly. The most interesting part is inline assembler with operands, or `extended` assembler. An extended assembly statement looks more complicated and consists of more than two parts:

```
__asm__ [volatile] [goto] (AssemblerTemplate
                           [ : OutputOperands ]
                           [ : InputOperands  ]
                           [ : Clobbers       ]
                           [ : GotoLabels     ]);
```

All parameters which are marked with squared brackets are optional. You may notice that if we skip the optional parameters and the modifiers `volatile` and `goto` we obtain the `basic` form.

Let's start to consider this in order. The first optional `qualifier` is `volatile`. This specifier tells the compiler that an assembly statement may produce `side effects`. In this case we need to prevent compiler optimizations related to the given assembly statement. In simple terms the `volatile` specifier instructs the compiler not to modify the statement and place it exactly where it was in the original code. As an example let's look at the following function from the Linux kernel:

```
static inline void native_load_gdt(const struct desc_ptr *dtr)
{
    asm volatile("lgdt %0"::"m" (*dtr));
}
```

Here we see the `native_load_gdt` function which loads a base address from the Global Descriptor Table to the `GDTR` register with the `lgdt` instruction. This assembly statement is marked with `volatile` qualifier. It is very important that the compiler does not change the original place of this assembly statement in the resulting code. Otherwise the `GDTR` register may contain wrong address for the `Global Descriptor Table` or the address may be correct, but the structure has not been filled yet. This can lead to an exception being generated, preventing the kernel from booting correctly.

The second optional `qualifier` is the `goto`. This qualifier tells the compiler that the given assembly statement may perform a jump to one of the labels which are listed in the `GotoLabels`. For example:

```
__asm__ goto("jmp %l[label]" : : : : label);
```

Since we finished with these two qualifiers, let's look at the main part of an assembly statement body. As we have seen above, the main part of an assembly statement consists of the following four parts:

- set of assembly instructions;
- output parameters;
- input parameters;
- clobbers.

The first represents a string which contains a set of valid assembly instructions which may be separated by the `\t\n` sequence. Names of processor registers must be prefixed with the `%%` sequence in `extended` form and other symbols like immediates must start with the `$` symbol. The `OutputOperands` and `InputOperands` are comma-separated lists of C variables which may be provided with "constraints" and the `Clobbers` is a list of registers or other values which are modified by the assembler instructions from the `AssemblerTemplate` beyond those listed in the `OutputOperands`. Before we dive into the examples we have to know a little bit about `constraints`. A constraint is a string which specifies placement of an operand. For example the value of an operand may be written to a processor register or read from memory etc.

Consider the following simple example:

```
#include <stdio.h>

int main(void)
{
        unsigned long a = 5;
        unsigned long b = 10;
        unsigned long sum = 0;
```

```
        __asm__("addq %1,%2" : "=r" (sum) : "r" (a), "0" (b));
        printf("a + b = %lu\n", sum);
        return 0;
}
```

Let's compile and run it to be sure that it works as expected:

```
$ gcc test.c -o test
./test
a + b = 15
```

Ok, great. It works. Now let's look at this example in detail. Here we see a simple `C` program which calculates the sum of two variables placing the result into the `sum` variable and in the end we print the result. This example consists of three parts. The first is the assembly statement with the add instruction. It adds the value of the source operand together with the value of the destination operand and stores the result in the destination operand. In our case:

```
addq %1, %2
```

will be expanded to the:

```
addq a, b
```

Variables and expressions which are listed in the `OutputOperands` and `InputOperands` may be matched in the `AssemblerTemplate`. An input/output operand is designated as `%N` where the `N` is the number of operand from left to right beginning from `zero`. The second part of the our assembly statement is located after the first `:` symbol and contains the definition of the output value:

```
"=r" (sum)
```

Notice that the `sum` is marked with two special symbols: `=r`. This is the first constraint that we have encountered. The actual constraint here is only `r` itself. The `=` symbol is `modifier` which denotes output value. This tells to compiler that the previous value will be discarded and replaced by the new data. Besides the `=` modifier, `GCC` provides support for following three modifiers:

- `+` - an operand is read and written by an instruction;
- `&` - output register shouldn't overlap an input register and should be used only for output;
- `%` - tells the compiler that operands may be commutative.

Now let's go back to the `r` qualifier. As I mentioned above, a qualifier denotes the placement of an operand. The `r` symbol means a value will be stored in one of the general purpose register. The last part of our assembly statement:

```
"r" (a), "0" (b)
```

These are input operands - variables `a` and `b`. We already know what the `r` qualifier does. Now we can have a look at the constraint for the variable `b`. The `0` or any other digit from `1` to `9` is called "matching constraint". With this a single operand can be used for multiple roles. The value of the constraint is the source operand index. In our case `0` will match `sum`. If we look at assembly output of our program:

```
0000000000400400 <main>:
  ...
  ...
  ...
  4004fe:       48 c7 45 f8 05 00 00    movq   $0x5,-0x8(%rbp)
  400506:       48 c7 45 f0 0a 00 00    movq   $0xa,-0x10(%rbp)

  400516:       48 8b 55 f8             mov    -0x8(%rbp),%rdx
  40051a:       48 8b 45 f0             mov    -0x10(%rbp),%rax
  40051e:       48 01 d0                add    %rdx,%rax
```

First of all our values `5` and `10` will be put at the stack and then these values will be moved to the two general purpose registers: `%rdx` and `%rax`.

This way the `%rax` register is used for storing the value of the `b` as well as storing the result of the calculation. **NOTE** that I've used `gcc 6.3.1` version, so the resulted code of your compiler may differ.

We have looked at input and output parameters of an inline assembly statement. Before we move on to other constraints supported by `gcc`, there is one remaining part of the inline assembly statement we have not discussed yet - `clobbers`.

Clobbers

As mentioned above, the "clobbered" part should contain a comma-separated list of registers whose content will be modified by the assembler code. This is useful if our assembly expression needs additional registers for calculation. If we add clobbered registers to the inline assembly statement, the compiler take this into account and the register in question will not simultaneously be used by the compiler.

Consider the example from before, but we will add an additional, simple assembler instruction:

```
__asm__("movq $100, %%rdx\t\n"
        "addq %1,%2" : "=r" (sum) : "r" (a), "0" (b));
```

If we look at the assembly output:

```
0000000000400400 <main>:
  ...
  ...
  ...
  4004fe:       48 c7 45 f8 05 00 00    movq   $0x5,-0x8(%rbp)
  400506:       48 c7 45 f0 0a 00 00    movq   $0xa,-0x10(%rbp)

  400516:       48 8b 55 f8             mov    -0x8(%rbp),%rdx
  40051a:       48 8b 45 f0             mov    -0x10(%rbp),%rax

  40051e:       48 c7 c2 64 00 00 00    mov    $0x64,%rdx
  400525:       48 01 d0                add    %rdx,%rax
```

we will see that the `%rdx` register is overwritten with `0x64` or `100` and the result will be `110` instead of `10`. Now if we add the `%rdx` register to the list of `clobbered` registers:

```
__asm__("movq $100, %%rdx\t\n"
        "addq %1,%2" : "=r" (sum) : "r" (a), "0" (b) : "%rdx");
```

and look at the assembler output again:

```
0000000000400400 <main>:
  4004fe:       48 c7 45 f8 05 00 00    movq   $0x5,-0x8(%rbp)
  400506:       48 c7 45 f0 0a 00 00    movq   $0xa,-0x10(%rbp)

  400516:       48 8b 4d f8             mov    -0x8(%rbp),%rcx
  40051a:       48 8b 45 f0             mov    -0x10(%rbp),%rax

  40051e:       48 c7 c2 64 00 00 00    mov    $0x64,%rdx
  400525:       48 01 c8                add    %rcx,%rax
```

the `%rcx` register will be used for `sum` calculation, preserving the intended semantics of the program. Besides general purpose registers, we may pass two special specifiers. They are:

- `cc`;
- `memory`.

The first - `cc` indicates that an assembler code modifies flags register. This is typically used if the assembly within contains arithmetic or logic instructions:

```
__asm__("incq %0" ::""(variable): "cc");
```

The second `memory` specifier tells the compiler that the given inline assembly statement executes read/write operations on memory not specified by operands in the output list. This prevents the compiler from keeping memory values loaded and cached in registers. Let's take a look at the following example:

```
#include <stdio.h>

int main(void)
{
        unsigned long a[3] = {10000000000, 0, 1};
        unsigned long b = 5;

        __asm__ volatile("incq %0" :: "m" (a[0]));

        printf("a[0] - b = %lu\n", a[0] - b);
        return 0;
}
```

This example may be artificial, but it illustrates the main idea. Here we have an array of integers and one integer variable. The example is pretty simple, we take the first element of `a` and increment its value. After this we subtract the value of `b` from the first element of `a`. In the end we print the result. If we compile and run this simple example the result may surprise you:

```
~$ gcc -O3  test.c -o test
~$ ./test
a[0] - b = 9999999995
```

The result is `a[0] - b = 9999999995` here, but why? We incremented `a[0]` and subtracted `b`, so the result should be `a[0] - b = 9999999996` here.

If we have a look at the assembler output for this example:

```
00000000004004f6 <main>:
  4004b4:       48 b8 00 e4 0b 54 02    movabs $0x2540be400,%rax
  4004be:       48 89 04 24             mov    %rax,(%rsp)
  ...
  ...
  ...
  40050e:       ff 44 24 f0             incq   (%rsp)

  4004d8:       48 be fb e3 0b 54 02    movabs $0x2540be3fb,%rsi
```

we will see that the first element of the `a` contains the value `0x2540be400` (`10000000000`). The last two lines of code are the actual calculations.

We see our increment instruction with `incq` but then just a move of `0x2540be3fb` (`9999999995`) to the `%rsi` register. This looks strange.

The problem is we have passed the `-O3` flag to `gcc`, so the compiler did some constant folding and propagation to determine the result of `a[0] - 5` at compile time and reduced it to a `movabs` with a constant `0x2540be3fb` or `9999999995` in runtime.

Let's now add `memory` to the clobbers list:

```
__asm__ volatile("incq %0" :: "m" (a[0]) : "memory");
```

and the new result of running this is:

```
~$ gcc -O3  test.c -o test
~$ ./test
a[0] - b = 9999999996
```

Now the result is correct. If we look at the assembly output again:

```
00000000004004f6 <main>:
  400404:       48 b8 00 e4 0b 54 02    movabs $0x2540be400,%rax
  40040b:       00 00 00
  40040e:       48 89 04 24             mov    %rax,(%rsp)
  400412:       48 c7 44 24 08 00 00    movq   $0x0,0x8(%rsp)
  400419:       00 00
  40041b:       48 c7 44 24 10 01 00    movq   $0x1,0x10(%rsp)
  400422:       00 00
  400424:       48 ff 04 24             incq   (%rsp)
  400428:       48 8b 04 24             mov    (%rsp),%rax
  400431:       48 8d 70 fb             lea    -0x5(%rax),%rsi
```

we will see one difference here which is in the last two lines:

```
  400428:       48 8b 04 24             mov    (%rsp),%rax
  400431:       48 8d 70 fb             lea    -0x5(%rax),%rsi
```

Instead of constant folding, `GCC` now preserves calculations in the assembly and places the value of `a[0]` in the `%rax` register afterwards. In the end it just subtracts the constant value of `b` from the `%rax` register and puts result to the `%rsi` .

Besides the `memory` specifier, we also see a new constraint here - `m` . This constraint tells the compiler to use the address of `a[0]` , instead of its value. So, now we are finished with `clobbers` and we may continue by looking at other constraints supported by `GCC` besides `r` and `m` which we have already seen.

Constraints

Now that we are finished with all three parts of an inline assembly statement, let's return to constraints. We already saw some constraints in the previous parts, like `r` which represents a `register` operand, `m` which represents a memory operand and `0-9` which represent an reused, indexed operand. Besides these `GCC` provides support for other constraints. For example the `i` constraint represents an `immediate` integer operand with know value:

```
#include <stdio.h>

int main(void)
{
        int a = 0;

        __asm__("movl %1, %0" : "=r"(a) : "i"(100));
        printf("a = %d\n", a);
        return 0;
}
```

The result is:

```
~$ gcc test.c -o test
~$ ./test
a = 100
```

Or for example `I` which represents an immediate 32-bit integer. The difference between `i` and `I` is that `i` is general, whereas `I` is strictly specified to 32-bit integer data. For example if you try to compile the following code:

```
unsigned long test_asm(int nr)
{
```

```
        unsigned long a = 0;

        __asm__("movq %1, %0" : "=r"(a) : "I"(0xfffffffffffff));
        return a;
}
```

you will get an error:

```
$ gcc -O3 test.c -o test
test.c: In function ‘test_asm’:
test.c:7:9: warning: asm operand 1 probably doesn’t match constraints
         __asm__("movq %1, %0" : "=r"(a) : "I"(0xfffffffffffff));
         ^
test.c:7:9: error: impossible constraint in ‘asm’
```

when at the same time:

```
unsigned long test_asm(int nr)
{
        unsigned long a = 0;

        __asm__("movq %1, %0" : "=r"(a) : "i"(0xfffffffffffff));
        return a;
}
```

works perfectly:

```
~$ gcc -O3 test.c -o test
~$ echo $?
0
```

`GCC` also supports `J`, `K`, `N` constraints for integer constants in the range of 0-63 bits, signed 8-bit integer constants and unsigned 8-bit integer constants respectively. The `o` constraint represents a memory operand with an `offsetable` memory address. For example:

```
#include <stdio.h>

int main(void)
{
        static unsigned long arr[3] = {0, 1, 2};
        static unsigned long element;

        __asm__ volatile("movq 16+%1, %0" : "=r"(element) : "o"(arr));
        printf("%lu\n", element);
        return 0;
}
```

The result, as expected:

```
~$ gcc -O3 test.c -o test
~$ ./test
2
```

All of these constraints may be combined (so long as they do not conflict). In this case the compiler will choose the best one for a certain situation. For example:

```
unsigned long a = 10;
unsigned long b = 20;

void main(void)
{
    __asm__ ("movq %1,%0" : "=mr"(b) : "rm"(a));
```

```
}
```

will use a memory operand:

```
main:
        movq a(%rip),b(%rip)
        ret
b:
        .quad   20
a:
        .quad   10
```

instead of direct usage of general purpose registers.

That's about all of the commonly used constraints in inline assembly statements. You can find more in the official documentation.

Architecture specific constraints

Before we finish, let's look at the set of special constraints. These constrains are architecture specific and as this book is specific to the x86_64 architecture, we will look at constraints related to it. First of all the set of `a` ... `d` and also `S` and `D` constraints represent generic purpose registers. In this case the `a` constraint corresponds to `%al`, `%ax`, `%eax` or `%rax` register depending on instruction size. The `S` and `D` constraints are `%si` and `%di` registers respectively. For example let's take our previous example. We can see in its assembly output that value of the `a` variable is stored in the `%eax` register. Now let's look at the assembly output of the same assembly, but with other constraint:

```
#include <stdio.h>

int a = 1;

int main(void)
{
        int b;
        __asm__ ("movq %1,%0" : "=r"(b) : "d"(a));
        return b;
}
```

Now we see that value of the `a` variable will be stored in the `%rax` register:

```
0000000000400400 <main>:
  4004aa:       48 8b 05 6f 0b 20 00    mov    0x200b6f(%rip),%rax        # 601020 <a>
```

The `f` and `t` constraints represent any floating point stack register - `%st` and the top of the floating point stack respectively. The `u` constraint represents the second value from the top of the floating point stack.

That's all. You may find more details about x86_64 and general constraints in the official documentation.

Links

- Linux kernel source code
- assembly programming language
- GCC
- GNU extension
- Global Descriptor Table
- Processor registers
- add instruction
- flags register
- x86_64

- constraints

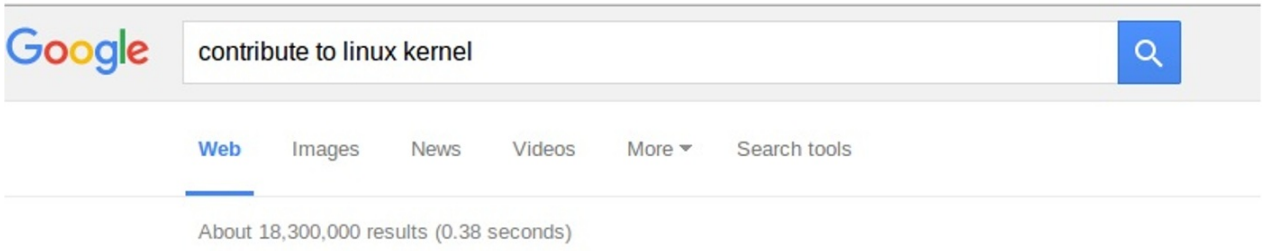
Linux

`x86_64` `Hello World`

`syscall` `printf` Linux Linux Linux

Linux Linux Linux linux-insides `9096`

Linux `linux-insides` Linux Linux



Linux Linux Linux Linux Linux

Linux

Linux

- Linux
- Linux

Linux Linux Linux

Linux Ubuntu (Vivid Vervet) Linux `4.1`

```
$ sudo add-apt-repository ppa:kernel-ppa/ppa
$ sudo apt-get update
```

```
$ apt-cache showpkg linux-headers
```

Linux `${version}`

```
$ sudo apt-get install linux-headers-${version} linux-headers-${version}-generic linux-image-${version}-generic --fix
-missing
```

grub

Linux Linux kernel.org Linux Linux `git` `git` `kernel.org`

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

`github` Linux

```
$ git clone git@github.com:torvalds/linux.git
```

fork

```
$ git checkout master
$ git pull upstream master
```

`upstream` Linux

```
git remote add upstream git@github.com:torvalds/linux.git
```

```
~/dev/linux (master) $ git remote -v
origin    git@github.com:0xAX/linux.git (fetch)
origin    git@github.com:0xAX/linux.git (push)
upstream    https://github.com/torvalds/linux.git (fetch)
upstream    https://github.com/torvalds/linux.git (push)
```

fork (`origin`) (`upstream`)

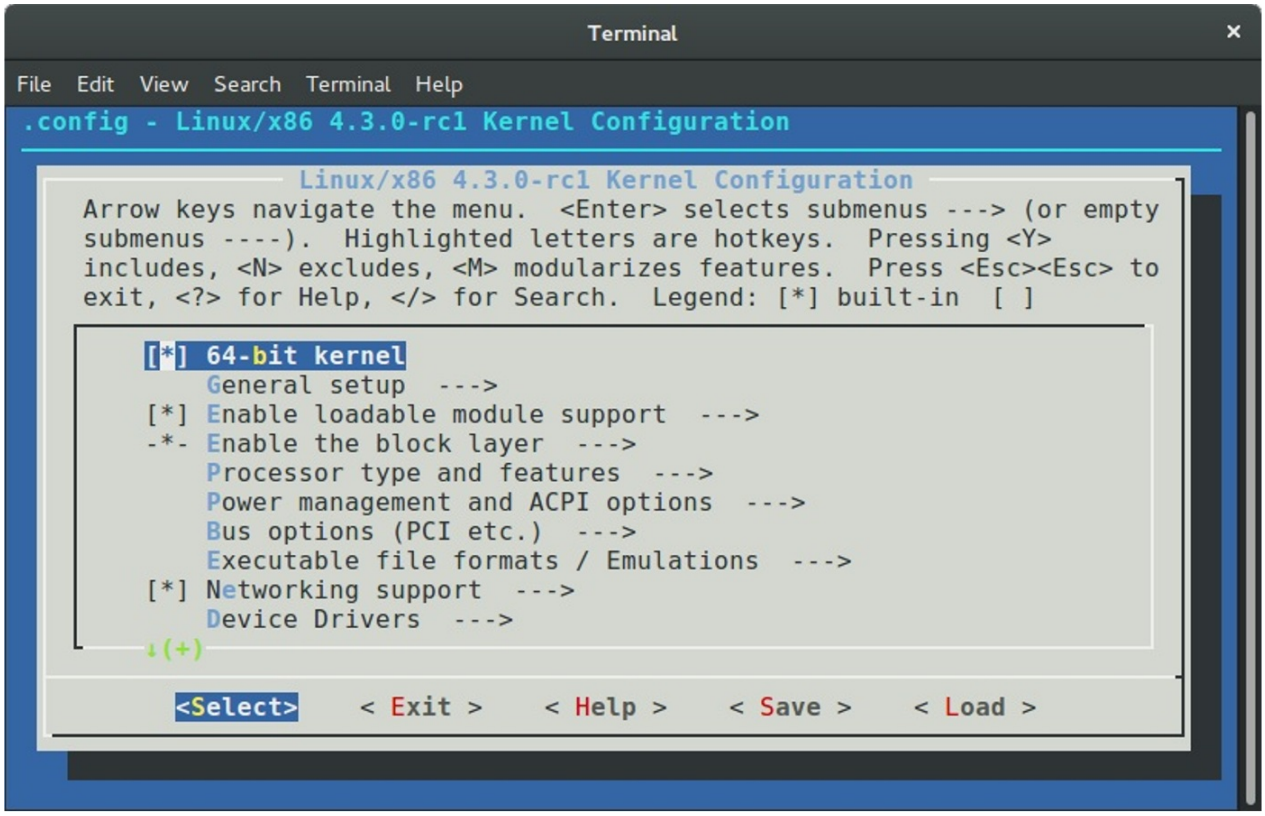
Linux Linux `/boot`

```
$ sudo cp /boot/config-$(uname -r) ~/dev/linux/.config
```

`/proc/config.gz`

```
$ cat /proc/config.gz | gunzip > ~/dev/linux/.config
```

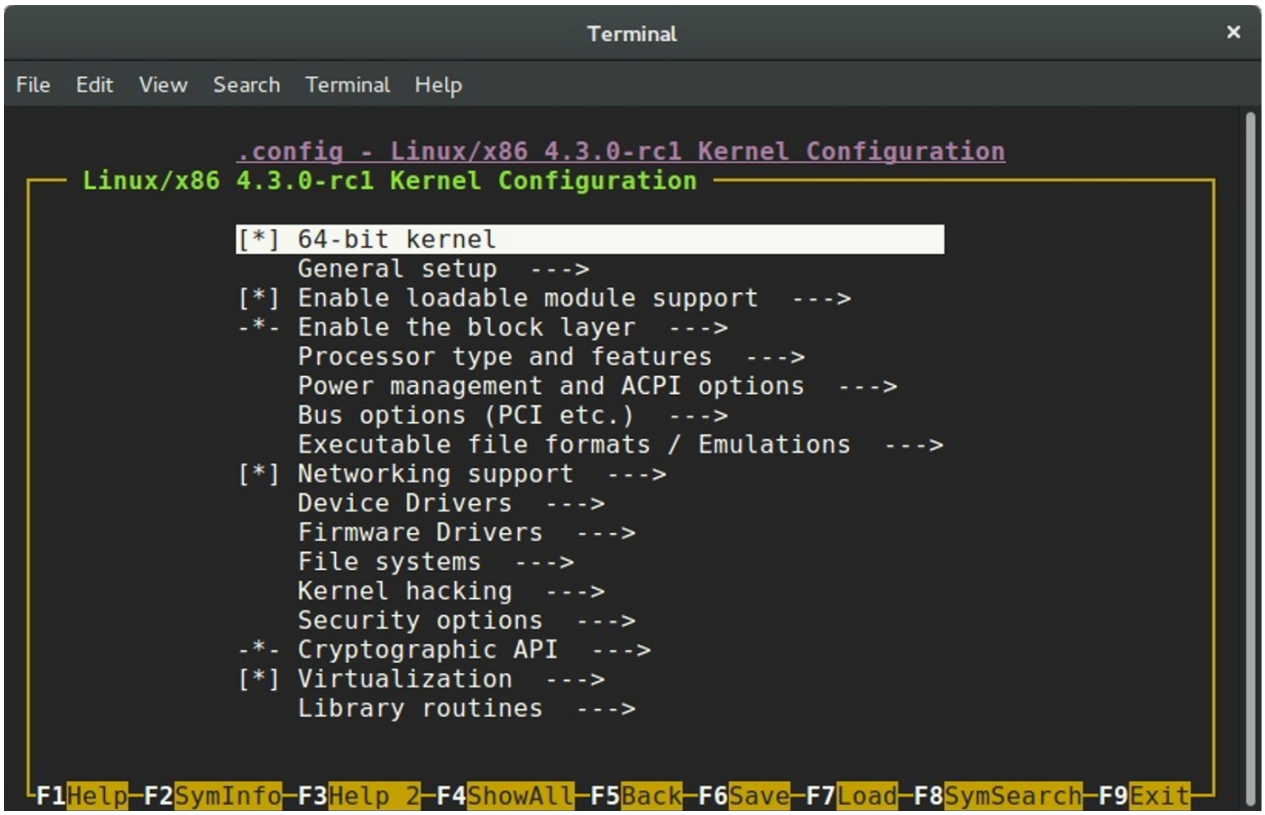
Linux Linux Makefile `menuconfig`



`defconfig` x86_64 defconfig `ARCH` `make` `defconfig`

```
$ make ARCH=arm64 defconfig
```

`allnoconfig` `allyesconfig` `allmodconfig` `nconfig` `ncurses` Linux



`randconfig` Linux Linux

Linux Linux Linux

```
$ make
scripts/kconfig/conf  --silentoldconfig Kconfig
#
# configuration written to .config
#
  CHK     include/config/kernel.release
  UPD     include/config/kernel.release
  CHK     include/generated/uapi/linux/version.h
  CHK     include/generated/utsrelease.h
  ...
  ...
  ...
  OBJCOPY arch/x86/boot/vmlinux.bin
  AS      arch/x86/boot/header.o
  LD      arch/x86/boot/setup.elf
  OBJCOPY arch/x86/boot/setup.bin
  BUILD   arch/x86/boot/bzImage
  Setup is 15740 bytes (padded to 15872 bytes).
System is 4342 kB
CRC 82703414
Kernel: arch/x86/boot/bzImage is ready  (#73)
```

`make` `-jN` `N`

```
$ make -j8
```

- `ARCH`
- `CROSS_COMPILER`

arm64 Linux

```
$ make -j4 ARCH=arm64 CROSS_COMPILER=aarch64-linux-gnu- defconfig
$ make -j4 ARCH=arm64 CROSS_COMPILER=aarch64-linux-gnu-
```

- `arch/x86/boot/bzImage`

Linux

Linux Linux Linux

```
...
...
...
Kernel: arch/x86/boot/bzImage is ready  (#73)
```

bzImage Linux `headers` `modules`

```
$ sudo make headers_install
$ sudo make modules_install
```

```
$ sudo make install
```

Linux `bootloader` `/boot/grub2/grub.cfg` Linux Fedora Ubuntu grub

```
#!/bin/bash
```

```
source "term-colors"

DISTRIBUTIVE=$(cat /etc/*-release | grep NAME | head -1 | sed -n -e 's/NAME\=//p')
echo -e "Distributive: ${Green}${DISTRIBUTIVE}${Color_Off}"

if [[ "$DISTRIBUTIVE" == "Fedora" ]] ;
then
    su -c 'grub2-mkconfig -o /boot/grub2/grub.cfg'
else
    sudo update-grub
fi

echo "${Green}Done.${Color_Off}"
```

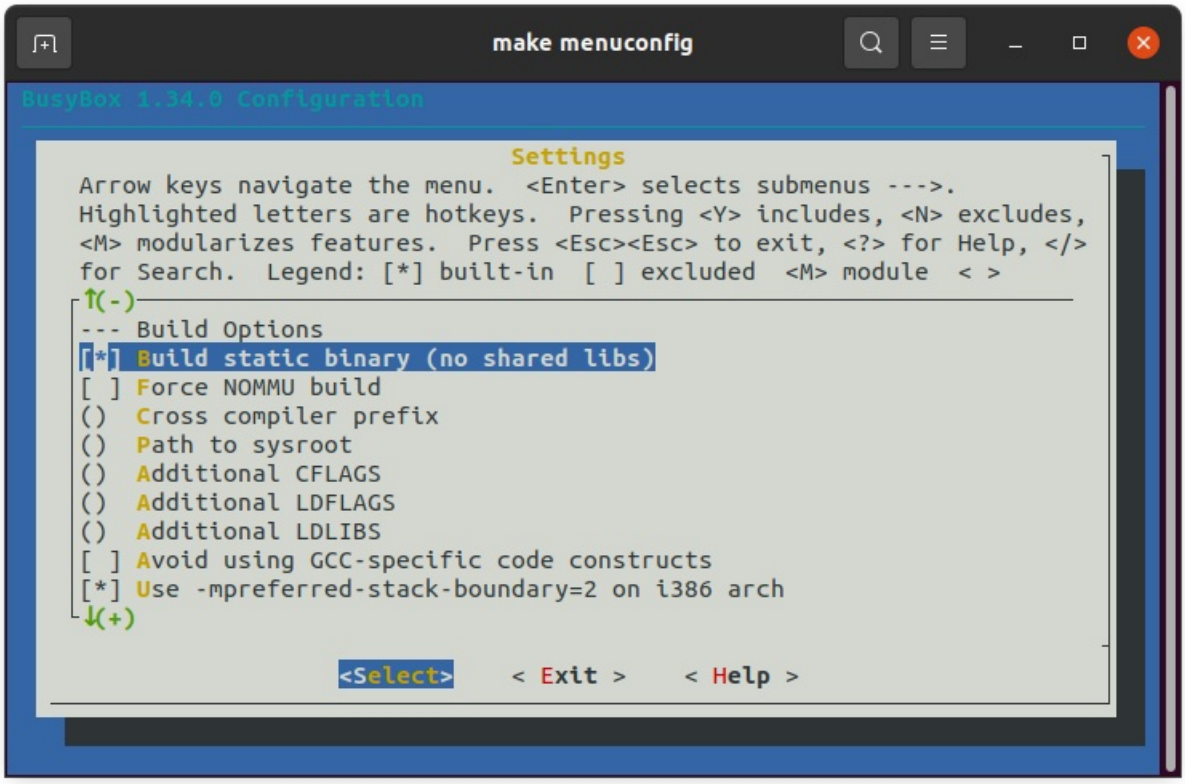
Linux

Linux qemu - initrd `initrd` Linux `initrd`

busybox `menuconfig`

```
$ mkdir initrd
$ cd initrd
$ curl http://busybox.net/downloads/busybox-1.23.2.tar.bz2 | tar xjf -
$ cd busybox-1.23.2/
$ make menuconfig
$ make -j4
```

`busybox` - `/bin/busybox` coreutils `busysbox` `Build BusyBox as a static binary (no shared libs)`




```
Busybox Settings
--> Build Options
```

`busysbox`

```
$ make -j4
$ sudo make install
```

`busybox` `initrd` `initrd`

```
$ cd ..
$ mkdir -p initramfs
$ cd initramfs
$ mkdir -pv {bin,sbin,etc,proc,sys,usr/{bin,sbin}}
$ cp -av ../busybox-1.23.2/_install/* .
```

`busybox` `bin` `sbin` `init` `init` procfs sysfs shell

```
#!/bin/sh

mount -t proc none /proc
mount -t sysfs none /sys

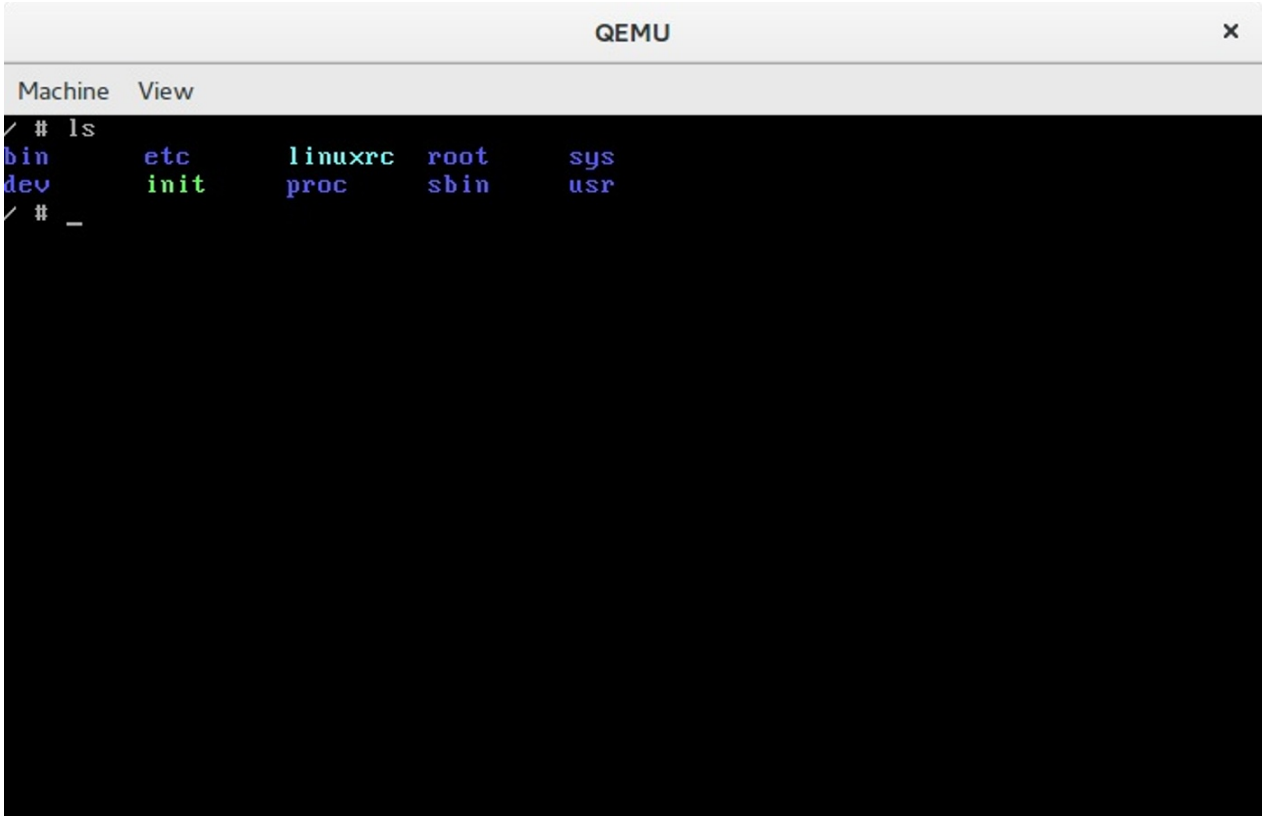
exec /bin/sh
```

`initrd`

```
$ find . -print0 | cpio --null -ov --format=newc | gzip -9 > ~/dev/initrd_x86_64.gz
```

qemu Linux

```
$ qemu-system-x86_64 -snapshot -m 8GB -serial stdio -kernel ~/dev/linux/arch/x86_64/boot/bzImage -initrd ~/dev/initrd
_x86_64.gz -append "root=/dev/sda1 ignore_loglevel"
```



Linux

ivandaviov/minimal `initrd`

Linux

Linux (`to do`) (`not to do`) (`to do`) (`todo`) Linux Linux

Linus

```
$ git checkout master
$ git pull upstream master
```

Linux `staging` drivers/staging `staging` Greg Kroah-Hartman Linux

Digi International EPCA PCI 295 `dgap_sindex`

```
static char *dgap_sindex(char *string, char *group)
{
    char *ptr;

    if (!string || !group)
        return NULL;

    for (; *string; string++) {
        for (ptr = group; *ptr; ptr++) {
            if (*ptr == *string)
                return string;
        }
    }

    return NULL;
}
```

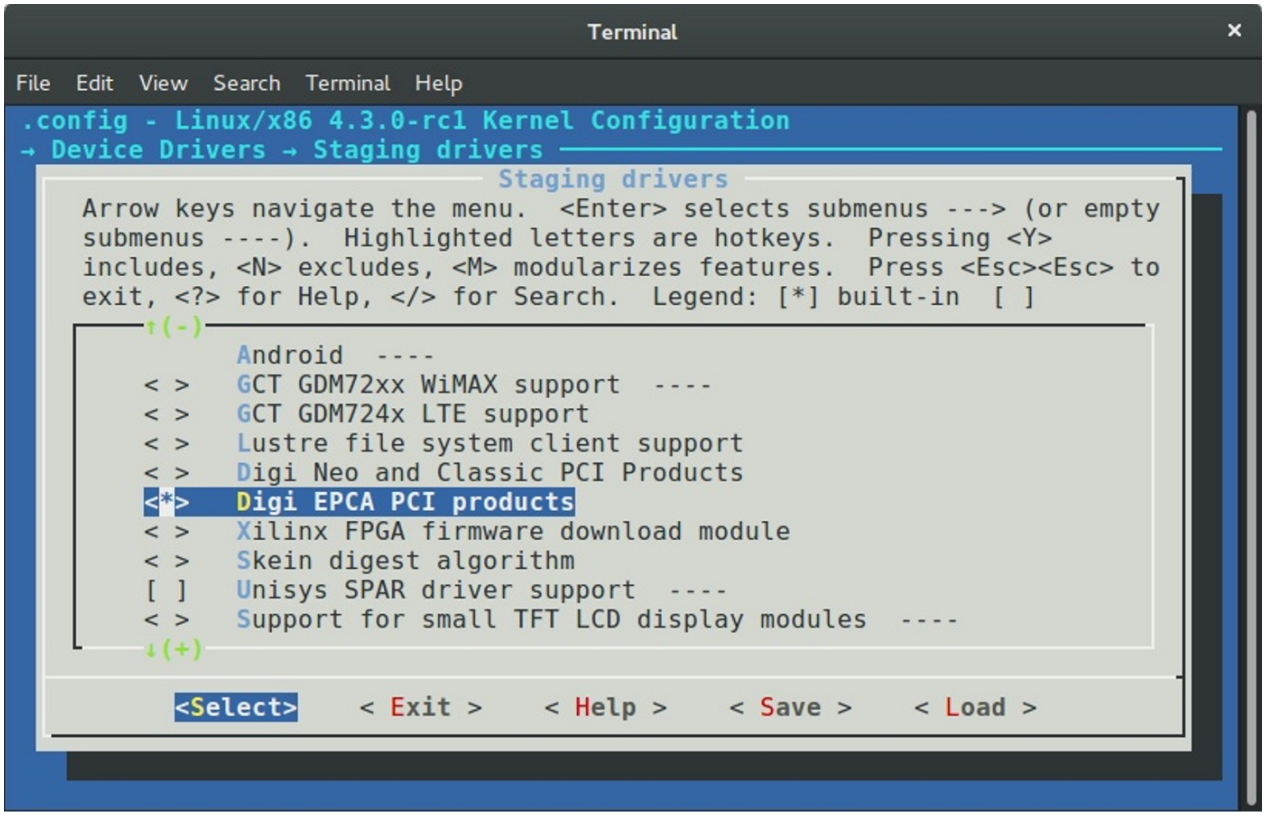
`group` `string` Linux lib/string.c `strpbrk` `dgap_sinidex` drivers/staging/dgap/dgap.c `dgap_sindex` `strpbrk`

`git` Linux

```
$ git checkout -b "dgap-remove-dgap_sindex"
```

`dgap_sindex` `strpbrk` Linux dgap

```
Device Drivers
--> Staging drivers
----> Digi EPCA PCI products
```




```
$ git add .
$ git commit -s -v
```

`$GIT_EDITOR` `$EDITOR` `-s` `Signed-off-by` - 00cc1633 `-v` `HEAD`

`[PATCH]` :

```
[PATCH] staging/dgap: Use strpbrk() instead of dgap_sindex()


The <linux/string.h> provides strpbrk() function that does the same that the
dgap_sindex(). Let's use already defined function instead of writing custom.
```

`Sign-off-by` `80` `Custom function removed` git blame

`format-patch`

```
$ git format-patch master
0001-staging-dgap-Use-strpbrk-instead-of-dgap_sindex.patch
```

(`master`) `format-patch` `dgap-remove-dgap_sindex` `master` `format-patch` `--stdout`

```
$ git format-patch master --stdout > dgap-patch-1.patch
```

Linux `git` `git send-email` `linux-kernel@vger.kernel.org` `get_maintainer.pl`

```
$ ./scripts/get_maintainer.pl -f drivers/staging/dgap/dgap.c
```

```
Lidza Louina <lidza.louina@gmail.com> (maintainer:DIGI EPCA PCI PRODUCTS)
Mark Hounschell <markh@compro.net> (maintainer:DIGI EPCA PCI PRODUCTS)
Daeseok Youn <daeseok.youn@gmail.com> (maintainer:DIGI EPCA PCI PRODUCTS)
Greg Kroah-Hartman <gregkh@linuxfoundation.org> (supporter:STAGING SUBSYSTEM)
driverdev-devel@linuxdriverproject.org (open list:DIGI EPCA PCI PRODUCTS)
devel@driverdev.osuosl.org (open list:STAGING SUBSYSTEM)
linux-kernel@vger.kernel.org (open list)
```

```
$ git send-email --to "Lidza Louina <lidza.louina@gmail.com>" \
  --cc "Mark Hounschell <markh@compro.net>"                   \
  --cc "Daeseok Youn <daeseok.youn@gmail.com>"                \
  --cc "Greg Kroah-Hartman <gregkh@linuxfoundation.org>"      \
  --cc "driverdev-devel@linuxdriverproject.org"               \
  --cc "devel@driverdev.osuosl.org"                           \
  --cc "linux-kernel@vger.kernel.org"
```

Linux ( ) Linus

Linux

-
- Linux -
- Linux - scripts/checkpatch.pl

```
$ ./scripts/checkpatch.pl -f drivers/staging/dgap/dgap.c
WARNING: Block comments use * on subsequent lines
#94: FILE: drivers/staging/dgap/dgap.c:94:
+/*
+     SUPPORTED PRODUCTS

CHECK: spaces preferred around that '|' (ctx:VxV)
#143: FILE: drivers/staging/dgap/dgap.c:143:
+    { PPCM,        PCI_DEV_XEM_NAME,     64, (T_PCXM|T_PCLITE|T_PCIBUS) },
```

`git diff`

```
~/dev/linux (dgap-remove-dgap_sindex) $ git diff
diff --git a/init/main.c b/init/main.c
index 9e64d70..af379a5 100644
--- a/init/main.c
+++ b/init/main.c
@@ -153,6 +153,8 @@ EXPORT_SYMBOL(reset_devices);
 static int __init set_reset_devices(char *str)
 {
        reset_devices = 1;
+
+
        return 1;
 }
```

- Linus github pull requests

- `git format-patch` `vN` `N` `git format-patch` `--cover-letter` `git send-email` `--in-reply-to`

```
|--> cover letter
  |----> patch_1
  |----> patch_2
```

`message-id` `--in-reply-to` `git send-email` `send-email` `format-patch` git send-email git format-patch

-
- scripts Linux `checkpatch.pl` `get_maintainer.pl` stackusage extract-vmlinux `scripts` Lorenzo Stoakes
- Linux `lkml` Linux `lkml` Linux
- Linux `[PATCH vN]` (`N`)

```
[PATCH v2] staging/dgap: Use strpbrk() instead of dgap_sindex()
```

Linux

Happy Hacking!

Linux Twitter

PR

- blog posts about assembly programming for x86_64
- Assembler
- distro
- package manager
- grub
- kernel.org
- version control system
- arm64
- bzImage
- qemu
- initrd
- busybox
- coreutils
- procfs
- sysfs
- Linux kernel mail listing archive
- Linux kernel coding style guide
- How to Get Your Change Into the Linux Kernel
- Linux Kernel Newbies
- plain text

Linux

Linux `make`

Makefile :) Makefile `1591` 4.2.0

Makefile Linux Makefile Makefile tags `make` bzImage

make

`make` `Makefile`

`Makefile` vmlinux Makefile

```
VERSION = 4
PATCHLEVEL = 2
SUBLEVEL = 0
EXTRAVERSION = -rc3
NAME = Hurr durr I'ma sheep
```

`Makefile` `KERNELVERSION`

```
KERNELVERSION = $(VERSION)$(if $(PATCHLEVEL),.$(PATCHLEVEL)$(if $(SUBLEVEL),.$(SUBLEVEL)))$(EXTRAVERSION)
```

`ifeq` `make` `Makefile` `make help` `make` `make V=1` `ifeq` `make` `V=n`

```
ifeq ("$(origin V)", "command line")
  KBUILD_VERBOSE = $(V)
endif
ifndef KBUILD_VERBOSE
  KBUILD_VERBOSE = 0
endif

ifeq ($(KBUILD_VERBOSE),1)
  quiet =
  Q =
else
  quiet=quiet_
  Q = @
endif

export quiet Q KBUILD_VERBOSE
```

`V=n` `make` `KBUILD_VERBOSE` `V` `KBUILD_VERBOSE` `0` `KBUILD_VERBOSE` `quiet` `Q` `@` `CC scripts/mod/empty.o` `Compiling .... scripts/mod/empty.o` LCTT CC Makefile

`ifeq` `make` `O=/dir` `dir`

```
ifeq ($(KBUILD_SRC),)

ifeq ("$(origin O)", "command line")
```

```
  KBUILD_OUTPUT := $(O)
endif

ifneq ($(KBUILD_OUTPUT),)
saved-output := $(KBUILD_OUTPUT)
KBUILD_OUTPUT := $(shell mkdir -p $(KBUILD_OUTPUT) && cd $(KBUILD_OUTPUT) \
                                && /bin/pwd)
$(if $(KBUILD_OUTPUT),, \
     $(error failed to create output directory "$(saved-output)"))

sub-make: FORCE
    $(Q)$(MAKE) -C $(KBUILD_OUTPUT) KBUILD_SRC=$(CURDIR) \
    -f $(CURDIR)/Makefile $(filter-out _all sub-make,$(MAKECMDGOALS))

skip-makefile := 1
endif # ifneq ($(KBUILD_OUTPUT),)
endif # ifeq ($(KBUILD_SRC),)
```

`KBUILD_SRC` makefile `KBUILD_OUTPUT` `O` `KBUILD_OUTPUT`

- `KBUILD_OUTPUT` `saved-output`
-
-
- `make` `-C`

`ifeq` make `C` `M`

```
ifeq ("$(origin C)", "command line")
  KBUILD_CHECKSRC = $(C)
endif
ifndef KBUILD_CHECKSRC
  KBUILD_CHECKSRC = 0
endif

ifeq ("$(origin M)", "command line")
  KBUILD_EXTMOD := $(M)
endif
```

`C` `Makefile` `$CHECK` `c` sparse `M`

`KBUILD_SRC` `KBUILD_SRC` `srctree` `.`

```
ifeq ($(KBUILD_SRC),)
        srctree := .
endif

objtree    := .
src        := $(srctree)
obj        := $(objtree)

export srctree objtree VPATH
```

`Makefile` `make` `objtree` `SUBARCH` LCTT CPU

```
SUBARCH := $(shell uname -m | sed -e s/i.86/x86/ -e s/x86_64/x86/ \
                  -e s/sun4u/sparc64/ \
                  -e s/arm.*/arm/ -e s/sa110/arm/ \
                  -e s/s390x/s390/ -e s/parisc64/parisc/ \
                  -e s/ppc.*/powerpc/ -e s/mips.*/mips/ \
                  -e s/sh[234].*/sh/ -e s/aarch64.*/arm64/ )
```

uname `uname` `SUBARCH` `SUBARCH` `SRCARCH` `hfr-arch` `SRCARCH` `hfr-arch`

```
ifeq ($(ARCH),i386)
        SRCARCH := x86
```

```
endif
ifeq ($(ARCH),x86_64)
        SRCARCH := x86
endif

hdr-arch  := $(SRCARCH)
```

`ARCH` `SUBARCH` `KCONFIG_CONFIG` `.config`

```
KCONFIG_CONFIG    ?= .config
export KCONFIG_CONFIG
```

shell

```
CONFIG_SHELL := $(shell if [ -x "$$BASH" ]; then echo $$BASH; \
      else if [ -x /bin/bash ]; then echo /bin/bash; \
      else echo sh; fi ; fi)
```

`C` `C++`

```
HOSTCC       = gcc
HOSTCXX      = g++
HOSTCFLAGS   = -Wall -Wmissing-prototypes -Wstrict-prototypes -O2 -fomit-frame-pointer -std=gnu89
HOSTCXXFLAGS = -O2
```

`CC` `HOST*` `CC` `HOSTCC` `host`

`KBUILD_MODULES` `KBUILD_BUILTIN`

```
KBUILD_MODULES :=
KBUILD_BUILTIN := 1

ifeq ($(MAKECMDGOALS),modules)
  KBUILD_BUILTIN := $(if $(CONFIG_MODVERSIONS),1)
endif
```

`modules` `make` `KBUILD_BUILTIN` `CONFIG_MODVERSIONS`

```
include scripts/Kbuild.include
```

Kbuild `Kernel Build System` `Kbuild` Makefile scripts/Kbuild.include `Kbuild` `Kbuild` binutils

```
AS        = $(CROSS_COMPILE)as
LD        = $(CROSS_COMPILE)ld
CC        = $(CROSS_COMPILE)gcc
CPP        = $(CC) -E
AR        = $(CROSS_COMPILE)ar
NM        = $(CROSS_COMPILE)nm
STRIP        = $(CROSS_COMPILE)strip
OBJCOPY        = $(CROSS_COMPILE)objcopy
OBJDUMP        = $(CROSS_COMPILE)objdump
AWK        = awk
...
...
...
```

`USERINCLUDE` `LINUXINCLUDE`

```
USERINCLUDE    := \
        -I$(srctree)/arch/$(hdr-arch)/include/uapi \
        -Iarch/$(hdr-arch)/include/generated/uapi \
        -I$(srctree)/include/uapi \
        -Iinclude/generated/uapi \
        -include $(srctree)/include/linux/kconfig.h

LINUXINCLUDE    := \
        -I$(srctree)/arch/$(hdr-arch)/include \
        ...
```

C

```
KBUILD_CFLAGS   := -Wall -Wundef -Wstrict-prototypes -Wno-trigraphs \
           -fno-strict-aliasing -fno-common \
           -Werror-implicit-function-declaration \
           -Wno-format-security \
           -std=gnu89
```

Makefile `arch/` Kbuild Makefile

`RCS_FIND_IGNORE` `RCS_TAR_IGNORE`

```
export RCS_FIND_IGNORE := \( -name SCCS -o -name BitKeeper -o -name .svn -o    \
              -name CVS -o -name .pc -o -name .hg -o -name .git \) \
              -prune -o
export RCS_TAR_IGNORE := --exclude SCCS --exclude BitKeeper --exclude .svn \
             --exclude CVS --exclude .pc --exclude .hg --exclude .git
```

`vmlinux`

Makefile Makefile `make` Makefile 598 `vmlinux`

```
all: vmlinux
    include arch/$(SRCARCH)/Makefile
```

`export RCS_FIND_IGNORE.....` `all: vmlinux.....` Makefile `make *.config`

`all:` Makefile arch/x86/Makefile Makefile `all` Makefile `vmlinux`

```
vmlinux: scripts/link-vmlinux.sh $(vmlinux-deps) FORCE
```

`vmlinux` linux scripts/link-vmlinux.sh vmlinux

`vmlinux-deps`

```
vmlinux-deps := $(KBUILD_LDS) $(KBUILD_VMLINUX_INIT) $(KBUILD_VMLINUX_MAIN)
```

`built-in.o` `Kbuild` `$(obj-y)` `$(LD) -r` `build-in.o` `vmlinux-deps` `vmlinux` `vmlinux-deps`

```
arch/x86/kernel/vmlinux.lds arch/x86/kernel/head_64.o
arch/x86/kernel/head64.o    arch/x86/kernel/head.o
init/built-in.o             usr/built-in.o
arch/x86/built-in.o         kernel/built-in.o
mm/built-in.o               fs/built-in.o
ipc/built-in.o              security/built-in.o
crypto/built-in.o           block/built-in.o
```

```
lib/lib.a                   arch/x86/lib/lib.a
lib/built-in.o              arch/x86/lib/built-in.o
drivers/built-in.o          sound/built-in.o
firmware/built-in.o         arch/x86/pci/built-in.o
arch/x86/power/built-in.o   arch/x86/video/built-in.o
net/built-in.o
```

```
$(sort $(vmlinux-deps)): $(vmlinux-dirs) ;
$(vmlinux-dirs): prepare scripts
    $(Q)$(MAKE) $(build)=$@
```

`vmlinux-dir` `prepare` `scripts` `prepare` `Makefile`

```
prepare: prepare0
prepare0: archprepare FORCE
    $(Q)$(MAKE) $(build)=.
archprepare: archheaders archscripts prepare1 scripts_basic

prepare1: prepare2 $(version_h) include/generated/utsrelease.h \
                   include/config/auto.conf
    $(cmd_crmodverdir)
prepare2: prepare3 outputmakefile asm-generic
```

`prepare0` `archprepare` `archheader` `archscripts` `x86_64` Makefile `x86_64` Makefile (defconfig) 16-bit `BITS` `32` arch/x86/Makefile `i386` `64` `x86_84`

Makefile (syscall table) `archheaders`

```
archheaders:
    $(Q)$(MAKE) $(build)=arch/x86/entry/syscalls all
```

Makefile `archscripts`

```
archscripts: scripts_basic
    $(Q)$(MAKE) $(build)=arch/x86/tools relocs
```

`archscripts` Makefile `scripts_basic` `scripts_basic` scripts/basic Makefile make

```
scripts_basic:
    $(Q)$(MAKE) $(build)=scripts/basic
```

`scripts/basic/Makefile` `fixdep` `bin2`

```
hostprogs-y    := fixdep
hostprogs-$(CONFIG_BUILD_BIN2C)     += bin2c
always         := $(hostprogs-y)

$(addprefix $(obj)/,$(filter-out fixdep,$(always))): $(obj)/fixdep
```

`fixdep` gcc `make` `bin2c` `CONFIG_BUILD_BIN2C` LCTT stdinstdout C C `hostprogs-y` `Kbuild` documentation `hostprogs-y` `Kbuild` `fixed` `Makefile` `fixdep.c`

`make` `Kbuild`

```
$ make
  HOSTCC  scripts/basic/fixdep
```

`script_basic` `archscripts` `make` arch/x86/tools Makefile `relocs` :

```
$(Q)$(MAKE) $(build)=arch/x86/tools relocs
```

`relocs_32.c` `relocs_64.c` `make`

```
  HOSTCC  arch/x86/tools/relocs_32.o
  HOSTCC  arch/x86/tools/relocs_64.o
  HOSTCC  arch/x86/tools/relocs_common.o
  HOSTLD  arch/x86/tools/relocs
```

`relocs.c` `version.h`

```
$(version_h): $(srctree)/Makefile FORCE
    $(call filechk,version.h)
    $(Q)rm -f $(old_version_h)
```

```
CHK     include/config/kernel.release
```

Makefile `arch/x86/include/generated/asm` `asm-generic` `generic` `asm-generic` `archprepare` `prepare0`

```
prepare0: archprepare FORCE
    $(Q)$(MAKE) $(build)=.
```

`build` scripts/Kbuild.include

```
build := -f $(srctree)/scripts/Makefile.build obj
```

- `.`

```
$(Q)$(MAKE) -f $(srctree)/scripts/Makefile.build obj=.
```

scripts/Makefile.build `obj` `Kbuild` `Kbuild`

```
include $(kbuild-file)
```

`.` `kernel/bounds.s` `arch/x86/kernel/asm-offsets.s` Kbuild `prepare` `vmlinux-dirs` `scripts` `filealias` `mk_elfconfig` `modpost` `scripts/host-programs` `vmlinux-dirs` `vmlinux-dirs`

```
init usr arch/x86 kernel mm fs ipc security crypto block
drivers sound firmware arch/x86/pci arch/x86/power
arch/x86/video net lib arch/x86/lib
```

Makefile `vmlinux-dirs`

```
vmlinux-dirs    := $(patsubst %/,%,$(filter %/, $(init-y) $(init-m) \
             $(core-y) $(core-m) $(drivers-y) $(drivers-m) \
             $(net-y) $(net-m) $(libs-y) $(libs-m)))

init-y        := init/
drivers-y     := drivers/ sound/ firmware/
net-y         := net/
```

```
libs-y        := lib/
...
...
...
```

`patsubst` `filter` `/` `vmlinux-dirs` `vmlinux-dirs`

```
$(vmlinux-dirs): prepare scripts
    $(Q)$(MAKE) $(build)=$@
```

`$@` `vmlinux-dirs` `vmlinux-dirs` `make`

```
  CC      init/main.o
  CHK     include/generated/compile.h
  CC      init/version.o
  CC      init/do_mounts.o
  ...
  CC      arch/x86/crypto/glue_helper.o
  AS      arch/x86/crypto/aes-x86_64-asm_64.o
  CC      arch/x86/crypto/aes_glue.o
  ...
  AS      arch/x86/entry/entry_64.o
  AS      arch/x86/entry/thunk_64.o
  CC      arch/x86/entry/syscall_64.o
```

`built-io.o`

```
$ find . -name built-in.o
./arch/x86/crypto/built-in.o
./arch/x86/crypto/sha-mb/built-in.o
./arch/x86/net/built-in.o
./init/built-in.o
./usr/built-in.o
...
...
```

`built-in.o` `vmlinux` `vmlinux` Makefile `vmlinux` samples , Documentation

```
vmlinux: scripts/link-vmlinux.sh $(vmlinux-deps) FORCE
    ...
    ...
    +$(call if_changed,link-vmlinux)
```

scripts/link-vmlinux.sh `built-in.o` System.map

```
  LINK    vmlinux
  LD      vmlinux.o
  MODPOST vmlinux.o
  GEN     .version
  CHK     include/generated/compile.h
  UPD     include/generated/compile.h
  CC      init/version.o
  LD      init/built-in.o
  KSYM    .tmp_kallsyms1.o
  KSYM    .tmp_kallsyms2.o
  LD      vmlinux
  SORTEX  vmlinux
  SYSMAP  System.map
```

`vmlinux` `System.map`

```
$ ls vmlinux System.map
System.map  vmlinux
```

`vmlinux` bzImage.

bzImage

`bzImage` linux `vmlinux` `make bzImage` `bzImage` `make` `bzImage` arch/x86/kernel/Makefile

```
all: bzImage
```

`bzImage` arch/x86/kernel/Makefile

```
bzImage: vmlinux
    $(Q)$(MAKE) $(build)=$(boot) $(KBUILD_IMAGE)
    $(Q)mkdir -p $(objtree)/arch/$(UTS_MACHINE)/boot
    $(Q)ln -fsn ../../x86/boot/bzImage $(objtree)/arch/$(UTS_MACHINE)/boot/$@
```

boot `make`

```
boot := arch/x86/boot
```

`arch/x86/boot` `arch/x86/boot/compressed` `setup.bin` `vmlinux.bin` `bzImage` arch/x86/boot/Makefile `$(obj)/setup.elf`

```
$(obj)/setup.elf: $(src)/setup.ld $(SETUP_OBJS) FORCE
    $(call if_changed,ld)
```

`arch/x86/boot` `setup.ld` `boot` `SETUP_OBJS`

```
  AS      arch/x86/boot/bioscall.o
  CC      arch/x86/boot/cmdline.o
  AS      arch/x86/boot/copy.o
  HOSTCC  arch/x86/boot/mkcpustr
  CPUSTR  arch/x86/boot/cpustr.h
  CC      arch/x86/boot/cpu.o
  CC      arch/x86/boot/cpuflags.o
  CC      arch/x86/boot/cpucheck.o
  CC      arch/x86/boot/early_serial_console.o
  CC      arch/x86/boot/edd.o
```

arch/x86/boot/header.S

```
$(obj)/header.o: $(obj)/voffset.h $(obj)/zoffset.h
```

`voffset.h` `sed` `nm` `vmlinux`

```
#define VO__end 0xffffffff82ab0000
#define VO__text 0xffffffff81000000
```

`zoffset.h` arch/x86/boot/compressed/Makefile `vmlinux`

```
$(obj)/zoffset.h: $(obj)/compressed/vmlinux FORCE
    $(call if_changed,zoffset)
```

`$(obj)/compressed/vmlinux` `vmlinux-objs-y` ——— arch/x86/boot/compressed `vmlinux.bin` `vmlinux.bin.bz2` `mkpiggy`

```
  LDS     arch/x86/boot/compressed/vmlinux.lds
  AS      arch/x86/boot/compressed/head_64.o
  CC      arch/x86/boot/compressed/misc.o
  CC      arch/x86/boot/compressed/string.o
  CC      arch/x86/boot/compressed/cmdline.o
  OBJCOPY arch/x86/boot/compressed/vmlinux.bin
  BZIP2   arch/x86/boot/compressed/vmlinux.bin.bz2
  HOSTCC  arch/x86/boot/compressed/mkpiggy
```

`vmlinux.bin` `vmlinux` `u32` LCTT 4-Byte `vmlinux.bin.all` `vmlinux.bin.bz2` `vmlinux.bin.all` `vmlinux.bin` `vmlinux.relocs` LCTT vmlinux `vmlinux.relocs` `vmlinux` `relocs` `vmlinux` `piggy.S` `mkpiggy`

```
  MKPIGGY arch/x86/boot/compressed/piggy.S
  AS      arch/x86/boot/compressed/piggy.o
```

`zoffset`

```
  ZOFFSET arch/x86/boot/zoffset.h
```

`zoffset.h` `voffset.h` arch/x86/boot

```
  AS      arch/x86/boot/header.o
  CC      arch/x86/boot/main.o
  CC      arch/x86/boot/mca.o
  CC      arch/x86/boot/memory.o
  CC      arch/x86/boot/pm.o
  AS      arch/x86/boot/pmjump.o
  CC      arch/x86/boot/printf.o
  CC      arch/x86/boot/regs.o
  CC      arch/x86/boot/string.o
  CC      arch/x86/boot/tty.o
  CC      arch/x86/boot/video.o
  CC      arch/x86/boot/video-mode.o
  CC      arch/x86/boot/video-vga.o
  CC      arch/x86/boot/video-vesa.o
  CC      arch/x86/boot/video-bios.o
```

`setup.elf`

```
  LD      arch/x86/boot/setup.elf
```

```
ld -m elf_x86_64   -T arch/x86/boot/setup.ld arch/x86/boot/a20.o arch/x86/boot/bioscall.o arch/x86/boot/cmdline.o arc
h/x86/boot/copy.o arch/x86/boot/cpu.o arch/x86/boot/cpuflags.o arch/x86/boot/cpucheck.o arch/x86/boot/early_serial_co
nsole.o arch/x86/boot/edd.o arch/x86/boot/header.o arch/x86/boot/main.o arch/x86/boot/mca.o arch/x86/boot/memory.o ar
ch/x86/boot/pm.o arch/x86/boot/pmjump.o arch/x86/boot/printf.o arch/x86/boot/regs.o arch/x86/boot/string.o arch/x86/b
oot/tty.o arch/x86/boot/video.o arch/x86/boot/video-mode.o arch/x86/boot/version.o arch/x86/boot/video-vga.o arch/x86
/boot/video-vesa.o arch/x86/boot/video-bios.o -o arch/x86/boot/setup.elf
```

`arch/x86/boot/*` `setup.bin`

```
objcopy  -O binary arch/x86/boot/setup.elf arch/x86/boot/setup.bin
```

`vmlinux` `vmlinux.bin`

```
objcopy  -O binary -R .note -R .comment -S arch/x86/boot/compressed/vmlinux arch/x86/boot/vmlinux.bin
```

arch/x86/boot/tools/build.c `setup.bin` `vmlinux.bin` `bzImage`

```
arch/x86/boot/tools/build arch/x86/boot/setup.bin arch/x86/boot/vmlinux.bin arch/x86/boot/zoffset.h arch/x86/boot/bzI
mage
```

`bzImage` `setup.bin` `vmlinux.bin`

```
Setup is 16268 bytes (padded to 16384 bytes).
System is 4704 kB
CRC 94a88f9a
Kernel: arch/x86/boot/bzImage is ready  (#5)
```

`make` `bzImage` linux Makefile linux linux

LCTT Linux

- GNU make util
- Linux kernel top Makefile
- cross-compilation
- Ctags
- sparse
- bzImage
- uname
- shell
- Kbuild
- binutils
- gcc
- Documentation
- System.map
- Relocation

linux-insides

Linker

C `*.o` /

```
*-linkers
*--main.c
*--lib.c
*--lib.h
```

`main.c`

```
#include <stdio.h>

#include "lib.h"

int main(int argc, char **argv) {
    printf("factorial of 5 is: %d\n", factorial(5));
    return 0;
}
```

`lib.c`

```
int factorial(int base) {
    int res,i = 1;

    if (base == 0) {
        return 1;
    }

    while (i <= base) {
        res *= i;
        i++;
    }

    return res;
}
```

`lib.h`

```
#ifndef LIB_H
#define LIB_H

int factorial(int base);

#endif
```

`main.c`

```
$ gcc -c main.c
```

`nm`

```
$ nm -A main.o
main.o:                 U factorial
main.o:0000000000000000 T main
main.o:                 U printf
```

`nm` `U` `T` `.text` `nm` `main.c`

- `factorial` - `lib.c` `main.c` `lib.c`
- `main` -;
- `printf` - glibc `main.c`

`nm` `main.o` `0000000000000000` `main` `main.o`

```
$ objdump -S main.o

main.o:     file format elf64-x86-64
Disassembly of section .text:

0000000000000000 <main>:
   0:    55                      push   %rbp
   1:    48 89 e5                mov    %rsp,%rbp
   4:    48 83 ec 10             sub    $0x10,%rsp
   8:    89 7d fc                mov    %edi,-0x4(%rbp)
   b:    48 89 75 f0             mov    %rsi,-0x10(%rbp)
   f:    bf 05 00 00 00          mov    $0x5,%edi
  14:    e8 00 00 00 00          callq  19 <main+0x19>
  19:    89 c6                   mov    %eax,%esi
  1b:    bf 00 00 00 00          mov    $0x0,%edi
  20:    b8 00 00 00 00          mov    $0x0,%eax
  25:    e8 00 00 00 00          callq  2a <main+0x2a>
  2a:    b8 00 00 00 00          mov    $0x0,%eax
  2f:    c9                      leaveq
  30:    c3                      retq
```

`callq` `callq` `objdump`

```
$ objdump -S -r main.o

...
  14:    e8 00 00 00 00          callq  19 <main+0x19>
  15: R_X86_64_PC32                  factorial-0x4
  19:    89 c6                   mov    %eax,%esi
...
  25:    e8 00 00 00 00          callq  2a <main+0x2a>
  26:   R_X86_64_PC32                printf-0x4
  2a:    b8 00 00 00 00          mov    $0x0,%eax
...
```

`objdump` `-r` `--reloc`

`objdump`

```
  14:    e8 00 00 00 00          callq  19 <main+0x19>
  15:   R_X86_64_PC32                factorial-0x4
  19:    89 c6                   mov    %eax,%esi
```

`e8 00 00 00 00` `e8` `call` `e8 00 00 00` `00 00 00 00` 4 4 `x86_64` 64 8
`-mcmodel=small` `main.c` `gcc`

```
-mcmodel=small

:  2GB  64
```

 gcc gcc 2 GB 4 call main.c

```
$ gcc main.c lib.c -o factorial | objdump -S factorial | grep factorial

factorial:     file format elf64-x86-64
...
...
0000000000400506 <main>:
    40051a:    e8 18 00 00 00           callq  400537 <factorial>
...
...
0000000000400537 <factorial>:
    400550:    75 07                    jne    400559 <factorial+0x22>
    400557:    eb 1b                    jmp    400574 <factorial+0x3d>
    400559:    eb 0e                    jmp    400569 <factorial+0x32>
    40056f:    7e ea                    jle    40055b <factorial+0x24>
...
...
```

 main 0x0000000000400506 0x0 C glibc C -nostdlib gcc main
 .init objdump

```
objdump -S factorial | less

factorial:     file format elf64-x86-64

Disassembly of section .init:

00000000004003a8 <_init>:
  4003a8:       48 83 ec 08             sub    $0x8,%rsp
  4003ac:       48 8b 05 a5 05 20 00    mov    0x2005a5(%rip),%rax        # 600958 <_DYNAMIC+0x1d0>
```

 glibc 0x00000000004003a8 readelf ELF

```
$ readelf -d factorial | grep \(INIT\)
 0x000000000000000c (INIT)               0x4003a8
```

 main 0000000000400506 .init factorial 0x0000000000400537 factorial e8 18 00 00
00 e8 call 18 00 00 00 x86_64 00 00 00 18 callq factorial

```
>>> hex(0x40051a + 0x18 + 0x5) == hex(0x400537)
True
```

 0x18 0x5 call 5 e8 18 00 00 00 0x18 factorial

GNU

GNU ld gcc factorial

```
$ gcc main.c lib.o -o factorial
```

——— `factorial`

```
./factorial
factorial of 5 is: 120
```

`gcc` `GUN ld` ——— `collect2`

```
~$ /usr/lib/gcc/x86_64-linux-gnu/4.9/collect2 --version
collect2 version 4.9.3
/usr/bin/ld --version
GNU ld (GNU Binutils for Debian) 2.25
...
...
...
```

gcc `GUN ld`

```
ld main.o lib.o -o factorial
```

```
$ ld main.o lib.o -o factorial
ld: warning: cannot find entry symbol _start; defaulting to 00000000004000b0
main.o: In function `main':
main.c:(.text+0x26): undefined reference to `printf'
```

- `_start`
- `printf`

`_start` `main` :) `_start` `_start` `crt1.0`

```
$ objdump -S /usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crt1.o

/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crt1.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_start>:
   0:    31 ed                   xor    %ebp,%ebp
   2:    49 89 d1                mov    %rdx,%r9
   ...
   ...
   ...
```

`ld`

```
ld /usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crt1.o \
main.o lib.o -o factorial

/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crt1.o: In function `_start':
/tmp/buildd/glibc-2.19/csu/../sysdeps/x86_64/start.S:115: undefined reference to `__libc_csu_fini'
/tmp/buildd/glibc-2.19/csu/../sysdeps/x86_64/start.S:116: undefined reference to `__libc_csu_init'
/tmp/buildd/glibc-2.19/csu/../sysdeps/x86_64/start.S:122: undefined reference to `__libc_start_main'
main.o: In function `main':
main.c:(.text+0x26): undefined reference to `printf'
```

`printf`

- `__libc_csu_fini`

- `__libc_csu_init`
- `__libc_start_main`

`_start` `glibc` sysdeps/x86_64/start.S

```
mov $__libc_csu_fini, %R8_LP
mov $__libc_csu_init, %RCX_LP
...
call __libc_start_main
```

`.init` `.fini` `main` csu/elf-init.c

- `crtn.o`;
- `crti.o`.

.init .fini `_init` `_fini`

`crtn.o` `.init` `.fini`

```
$ objdump -S /usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crtn.o

0000000000000000 <.init>:
   0:    48 83 c4 08              add    $0x8,%rsp
   4:    c3                       retq

Disassembly of section .fini:

0000000000000000 <.fini>:
   0:    48 83 c4 08              add    $0x8,%rsp
   4:    c3                       retq
```

`crti.o` `_init` `_fini`

```
$ ld \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crt1.o \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crti.o \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crtn.o main.o lib.o \
-o factorial
```

`-lc` `ld` `$LD_LIBRARY_PATH` `-lc`

```
$ ld \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crt1.o \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crti.o \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crtn.o main.o lib.o -lc \
-o factorial
```

```
$ ./factorial
bash: ./factorial: No such file or directory
```

readelf

```
$ readelf -l factorial

Elf file type is EXEC (Executable file)
Entry point 0x4003c0
There are 7 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
```

```
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x0000000000000188 0x0000000000000188  R E    8
  INTERP         0x00000000000001c8 0x00000000004001c8 0x00000000004001c8
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000610 0x0000000000000610  R E    200000
  LOAD           0x0000000000000610 0x0000000000600610 0x0000000000600610
                 0x00000000000001cc 0x00000000000001cc  RW     200000
  DYNAMIC        0x0000000000000610 0x0000000000600610 0x0000000000600610
                 0x0000000000000190 0x0000000000000190  RW     8
  NOTE           0x00000000000001e4 0x00000000004001e4 0x00000000004001e4
                 0x0000000000000020 0x0000000000000020  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .tex
t .fini .rodata .eh_frame
   03     .dynamic .got .got.plt .data
   04     .dynamic
   05     .note.ABI-tag
   06
```

```
  INTERP         0x00000000000001c8 0x00000000004001c8 0x00000000004001c8
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
```

`elf` `.interp` `.interp` `ascii` Linux `readelf` `x86_64` `/lib64/ld-linux-x86-64.so.2` `ld-linux-x86-64.so.2` `-dynamic-linker` `ld`

```
$ gcc -c main.c lib.c

$ ld \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crt1.o \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crti.o \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crtn.o main.o lib.o \
-dynamic-linker /lib64/ld-linux-x86-64.so.2 \
-lc -o factorial
```

```
$ ./factorial

factorial of 5 is: 120
```

`main.c` `lib.c` `gcc` `main.o` `lib.o`

```
$ file lib.o main.o
lib.o:  ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
main.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

`gcc` `GNU ld` C `GNU linker` `-o` `-dynamic-linker` `GNU ld`

GNU

`GNU linker` `-o <output>` - `ld` `output` `-l<name>` - `-dynamic-linker` `ld`

`@file` `file` `linker.ld`

```
$ ld @linker.ld
```

`-b` `--format` `ELF`, `DJGPP/COFF` `--oformat=output-format`

`--defsym` `--defsym=symbol=expression` Linux ARM Makefile - arch/arm/boot/compressed/Makefile

```
LDFLAGS_vmlinux = --defsym _kernel_bss_size=$(KBSS_SZ)
```

`.bss` `_kernel_bss_size`

```
ldr r5, =_kernel_bss_size
```

`-shared` `-M` `-map <filename>`

```
$ ld -M @linker.ld
...
...
...
.text           0x00000000004003c0      0x112
 *(.text.unlikely .text.*_unlikely .text.unlikely.*)
 *(.text.exit .text.exit.*)
 *(.text.startup .text.startup.*)
 *(.text.hot .text.hot.*)
 *(.text .stub .text.* .gnu.linkonce.t.*)
 .text          0x00000000004003c0       0x2a /usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crt1.o
...
...
...
 .text          0x00000000004003ea       0x31 main.o
                0x00000000004003ea                main
 .text          0x000000000040041b       0x3f lib.o
                0x000000000040041b                factorial
```

`GNU` `--help` `--version` `ld` `GNU` `ld` `ld`

`ld` AT&T

-
-
-
-
- ...

`-T` `ld` `SECTIONS` `.` hello world

```
.data
        msg:    .ascii  "hello, world!\n"

.text
```

```
.global _start

_start:
        mov    $1,%rax
        mov    $1,%rdi
        mov    $msg,%rsi
        mov    $14,%rdx
        syscall

        mov    $60,%rax
        mov    $0,%rdi
        syscall
```

```
$ as -o hello.o hello.asm
$ ld -o hello hello.o
```

`.text` `.data` `hello.asm`

```
/*
 * Linker script for the factorial
 */
OUTPUT(hello)
OUTPUT_FORMAT("elf64-x86-64")
INPUT(hello.o)

SECTIONS
{
    . = 0x200000;
    .text : {
          *(.text)
    }

    . = 0x400000;
    .data : {
          *(.data)
    }
}
```

`C` `OUTPUT` `OUTPUT_FORMAT` `INPUT` `ld` `SECTIONS` `SECTIONS` `SECTIONS` `. = 0x200000` `.` `0x200000` `. = 0x400000` `0x400000` `. = 0x200000` `.text` `*(.text)` `*` `*(.text)` `.text` `hello.o(.text)` `. = 0x400000`

```
$ as -o hello.o hello.S && ld -T linker.script && ./hello
hello, world!
```

`objdump` `.text` `0x200000` `.data` `0x400000`

```
$ objdump -D hello

Disassembly of section .text:

0000000000200000 <_start>:
  200000:    48 c7 c0 01 00 00 00     mov    $0x1,%rax
  ...

Disassembly of section .data:

0000000000400000 <msg>:
  400000:    68 65 6c 6c 6f           pushq  $0x6f6c6c65
  ...
```

`ASSERT(exp, message)` linux-insides Linux Linux `0x1f1` Linux

```
. = ASSERT(hdr == 0x1f1, "The setup header has the wrong offset!");
```

`INCLUDE filename` `ld`

- symbol = expression ;
- symbol += expression ;
- symbol -= expression ;
- symbol *= expression ;
- symbol /= expression ;
- symbol <<= expression ;
- symbol >>= expression ;
- symbol &= expression ;
- symbol |= expression ;

C

```
START_ADDRESS = 0x200000;
DATA_OFFSET   = 0x200000;

SECTIONS
{
    . = START_ADDRESS;
    .text : {
          *(.text)
    }

    . = START_ADDRESS + DATA_OFFSET;
    .data : {
          *(.data)
    }
}
```

C

- `ABSOLUTE` -
- `ADDR` -
- `ALIGN` - `.`
- `DEFINED` - `1` `0`
- `MAX` and `MIN` -
- `NEXT` -
- `SIZEOF` -

...

PR linux-insides-zh

- Book about Linux kernel insides
- linker
- object files
- glibc

- opcode
- ELF
- GNU linker
- My posts about assembly programming for x86_64
- readelf

linux-insides-zh

Linux Linux

`C` main

```
int main(int argc, char *argv[]) {
    // Entry point is here
}
```

`main`

```
int main(int argc, char *argv[]) {
    return 0;
}
```

gdb

```
$ gcc -ggdb program.c -o program
$ gdb ./program
The target architecture is assumed to be i386:x86-64:intel
Reading symbols from ./program...done.
```

gdb `info files`

```
(gdb) info files
Symbols from "/home/alex/program".
Local exec file:
    `/home/alex/program', file type elf64-x86-64.
    Entry point: 0x400430
    0x0000000000400238 - 0x0000000000400254 is .interp
    0x0000000000400254 - 0x0000000000400274 is .note.ABI-tag
    0x0000000000400274 - 0x0000000000400298 is .note.gnu.build-id
    0x0000000000400298 - 0x00000000004002b4 is .gnu.hash
    0x00000000004002b8 - 0x0000000000400318 is .dynsym
    0x0000000000400318 - 0x0000000000400357 is .dynstr
    0x0000000000400358 - 0x0000000000400360 is .gnu.version
    0x0000000000400360 - 0x0000000000400380 is .gnu.version_r
    0x0000000000400380 - 0x0000000000400398 is .rela.dyn
    0x0000000000400398 - 0x00000000004003c8 is .rela.plt
    0x00000000004003c8 - 0x00000000004003e2 is .init
    0x00000000004003f0 - 0x0000000000400420 is .plt
    0x0000000000400420 - 0x0000000000400428 is .plt.got
    0x0000000000400430 - 0x00000000004005e2 is .text
    0x00000000004005e4 - 0x00000000004005ed is .fini
    0x00000000004005f0 - 0x0000000000400610 is .rodata
    0x0000000000400610 - 0x0000000000400644 is .eh_frame_hdr
    0x0000000000400648 - 0x000000000040073c is .eh_frame
    0x0000000000600e10 - 0x0000000000600e18 is .init_array
    0x0000000000600e18 - 0x0000000000600e20 is .fini_array
    0x0000000000600e20 - 0x0000000000600e28 is .jcr
    0x0000000000600e28 - 0x0000000000600ff8 is .dynamic
    0x0000000000600ff8 - 0x0000000000601000 is .got
    0x0000000000601000 - 0x0000000000601028 is .got.plt
    0x0000000000601028 - 0x0000000000601034 is .data
    0x0000000000601034 - 0x0000000000601038 is .bss
```

`Entry point: 0x400430`

```
(gdb) break *0x400430
Breakpoint 1 at 0x400430
(gdb) run
Starting program: /home/alex/program

Breakpoint 1, 0x0000000000400430 in _start ()
```

`main` `_start` `main`

`C`

```
// program.c

#include <stdlib.h>
#include <stdio.h>

static int x = 1;

int y = 2;

int main(int argc, char *argv[]) {
    int z = 3;

    printf("x + y + z = %d\n", x + y + z);

    return EXIT_SUCCESS;
}
```

```
$ gcc -Wall program.c -o sum
```

```
$ ./sum
x + y + z = 6
```

- exec*

> The exec() family of functions replaces the current process image with a new process image.

execve files/exec.c

```
SYSCALL_DEFINE3(execve,
        const char __user *, filename,
        const char __user *const __user *, argv,
        const char __user *const __user *, envp)
{
    return do_execve(getname(filename), argv, envp);
}
```

`do_execve` `do_execve` ELF `start_thread`
x86_64 arch/x86/kernel/process_64.c

`start_thread`)

Linux `_start`

`_start`

```
$ gcc -Wall program.c -o sum
```

`_start` gcc `verbose mode` `-v`

`gcc`

```
$ gcc -v -ggdb program.c -o sum
...
...
...
/usr/libexec/gcc/x86_64-redhat-linux/6.1.1/cc1 -quiet -v program.c -quiet -dumpbase program.c -mtune=generic -march=x
86-64 -auxbase test -ggdb -version -o /tmp/ccvUWZkF.s
...
...
...
```

`cc1` `C` `/tmp/ccvUWZkF.s` `GNU as`

```
$ gcc -v -ggdb program.c -o sum
...
...
...
as -v --64 -o /tmp/cc79wZSU.o /tmp/ccvUWZkF.s
...
...
...
```

`collect2`

```
$ gcc -v -ggdb program.c -o sum
...
...
...
/usr/libexec/gcc/x86_64-redhat-linux/6.1.1/collect2 -plugin /usr/libexec/gcc/x86_64-redhat-linux/6.1.1/liblto_plugin.
so -plugin-opt=/usr/libexec/gcc/x86_64-redhat-linux/6.1.1/lto-wrapper -plugin-opt=-fresolution=/tmp/ccLEGYra.res -plu
gin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s -plugin-opt=-pass-through=-lc -plugin-opt=-pass-through
=-lgcc -plugin-opt=-pass-through=-lgcc_s --build-id --no-add-needed --eh-frame-hdr --hash-style=gnu -m elf_x86_64 -dy
namic-linker /lib64/ld-linux-x86-64.so.2 -o test /usr/lib/gcc/x86_64-redhat-linux/6.1.1/../../../../lib64/crt1.o /usr
/lib/gcc/x86_64-redhat-linux/6.1.1/../../../../lib64/crti.o /usr/lib/gcc/x86_64-redhat-linux/6.1.1/crtbegin.o -L/usr/
lib/gcc/x86_64-redhat-linux/6.1.1 -L/usr/lib/gcc/x86_64-redhat-linux/6.1.1/../../../../lib64 -L/lib/../lib64 -L/usr/l
ib/../lib64 -L. -L/usr/lib/gcc/x86_64-redhat-linux/6.1.1/../../.. /tmp/cc79wZSU.o -lgcc --as-needed -lgcc_s --no-as-n
eeded -lc -lgcc --as-needed -lgcc_s --no-as-needed /usr/lib/gcc/x86_64-redhat-linux/6.1.1/crtend.o /usr/lib/gcc/x86_6
4-redhat-linux/6.1.1/../../../../lib64/crtn.o
...
...
...
```

```
$ ldd program
    linux-vdso.so.1 (0x00007ffc9afd2000)
    libc.so.6 => /lib64/libc.so.6 (0x00007f56b389b000)
    /lib64/ld-linux-x86-64.so.2 (0x0000556198231000)
```

`printf` `-nostdlib`

```
$ gcc -nostdlib program.c -o program
/usr/bin/ld: warning: cannot find entry symbol _start; defaulting to 000000000040017c
```

```
/tmp/cc02msGW.o: In function `main':
/home/alex/program.c:11: undefined reference to `printf'
collect2: error: ld returned 1 exit status
```

`_start` `_start`

```
$ gcc -nostdlib -lc -ggdb program.c -o program
/usr/bin/ld: warning: cannot find entry symbol _start; defaulting to 0000000000400350
```

`/usr/lib64/libc.so.6` `_start` `gcc` `collect2` `/lib64/crt1.o`
`objdump` `_start`

```
$ objdump -d /lib64/crt1.o

/lib64/crt1.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_start>:
   0:    31 ed                    xor    %ebp,%ebp
   2:    49 89 d1                 mov    %rdx,%r9
   5:    5e                       pop    %rsi
   6:    48 89 e2                 mov    %rsp,%rdx
   9:    48 83 e4 f0              and    $0xfffffffffffffff0,%rsp
   d:    50                       push   %rax
   e:    54                       push   %rsp
   f:    49 c7 c0 00 00 00 00     mov    $0x0,%r8
  16:    48 c7 c1 00 00 00 00     mov    $0x0,%rcx
  1d:    48 c7 c7 00 00 00 00     mov    $0x0,%rdi
  24:    e8 00 00 00 00           callq  29 <_start+0x29>
  29:    f4                       hlt
```

`crt1.o` `_start` `_start` sysdeps/x86_64/start.S
`_start` `ebp` ABI)

```
xorl %ebp, %ebp
```

`r9`

```
mov %RDX_LP, %R9_LP
```

ELF

> After the dynamic linker has built the process image and performed the relocations, each shared object gets the opportunity to execute some initialization code. ... Similarly, shared objects may have termination functions, which are executed with the atexit (BA_OS) mechanism after the base process begins its termination sequence.

`r9` `__libc_start_main` `rdx` `%rdx` `%rsp` `_start` `__libc_start_main`
`__libc_start_main` csu/libc-start.c

```
STATIC int LIBC_START_MAIN (int (*main) (int, char **, char **),
                            int argc,
                           char **argv,
                            __typeof (main) init,
                           void (*fini) (void),
                           void (*rtld_fini) (void),
                           void *stack_end)
```

It takes address of the `main` function of a program, `argc` and `argv`. `init` and `fini` functions are constructor and destructor of the program. The `rtld_fini` is termination function which will be called after the program will be exited to terminate and free dynamic section. The last parameter of the `__libc_start_main` is the pointer to the stack of the program. Before we can call the `__libc_start_main` function, all of these parameters must be prepared and passed to it. Let's return to the sysdeps/x86_64/start.S assembly file and continue to see what happens before the `__libc_start_main` function will be called from there.

`main` `argc` `argv` `init` `fini` `rtld_fini` `__libc_start_main` `__libc_start_main`
sysdeps/x86_64/start.S `__libc_start_main`

`__libc_start_main` `_start`

```
+-----------------+
|       NULL      |
+-----------------+
|       envp      |
+-----------------+
|       NULL      |
+------------------
|       argv      | <- rsp
+------------------
|       argc      |
+-----------------+
```

`ebp` `r9` `rsi` `rsp` `argv` `rsi`

```
+-----------------+
|       NULL      |
+-----------------+
|       envp      |
+-----------------+
|       NULL      |
+------------------
|       argv      | <- rsp
+-----------------+
```

`argv` `rdx`

```
popq %rsi
mov %RSP_LP, %RDX_LP
```

`argc` `argv` ABI `16` `rax`

```
and  $~15, %RSP_LP
pushq %rax

pushq %rsp
mov $__libc_csu_fini, %R8_LP
mov $__libc_csu_init, %RCX_LP
mov $main, %RDI_LP
```

`r8` `rcx` `main` `rdi` csu/libc-start.c `__libc_start_main`

`__libc_start_main` `/lib64/crt1.o`

```
$ gcc -nostdlib /lib64/crt1.o -lc -ggdb program.c -o program
/lib64/crt1.o: In function `_start':
(.text+0x12): undefined reference to `__libc_csu_fini'
/lib64/crt1.o: In function `_start':
(.text+0x19): undefined reference to `__libc_csu_init'
collect2: error: ld returned 1 exit status
```

- `__libc_csu_fini` `__libc_csu_init` `__libc_start_main` `C` ELF

> After the dynamic linker has built the process image and performed the relocations, each shared object gets the opportunity to execute some initialization code. ... Similarly, shared objects may have termination functions, which are executed with the atexit (BA_OS) mechanism after the base process begins its termination sequence.

`.text` , `.data`

- `.init`
- `.fini`

We can find it with `readelf` util:

`readelf`

```
$ readelf -e test | grep init
  [11] .init             PROGBITS         00000000004003c8  000003c8

$ readelf -e test | grep fini
  [15] .fini             PROGBITS         0000000000400504  00000504
```

/ errno

`main` `.init` `.fini` `/lib64/crti.o`

```
$ gcc -nostdlib /lib64/crt1.o /lib64/crti.o  -lc -ggdb program.c -o program
```

```
$ ./program
Segmentation fault (core dumped)
```

`segmentation fault` `objdump` `lib64/crti.o`

```
$ objdump -D /lib64/crti.o

/lib64/crti.o:     file format elf64-x86-64


Disassembly of section .init:

0000000000000000 <_init>:
   0:    48 83 ec 08              sub    $0x8,%rsp
   4:    48 8b 05 00 00 00 00     mov    0x0(%rip),%rax        # b <_init+0xb>
   b:    48 85 c0                 test   %rax,%rax
   e:    74 05                    je     15 <_init+0x15>
  10:    e8 00 00 00 00           callq  15 <_init+0x15>

Disassembly of section .fini:

0000000000000000 <_fini>:
   0:    48 83 ec 08              sub    $0x8,%rsp
```

`/lib64/crti.o` `.init` `.fini` sysdeps/x86_64/crti.S

```
    .section .init,"ax",@progbits
    .p2align 2
    .globl _init
    .type _init, @function
_init:
    subq $8, %rsp
    movq PREINIT_FUNCTION@GOTPCREL(%rip), %rax
    testq %rax, %rax
    je .Lno_weak_fn
    call *%rax
.Lno_weak_fn:
```

```
    call PREINIT_FUNCTION
```

`.init` 16 `PREINIT_FUNCTION`

```
00000000004003c8 <_init>:
  4003c8:       48 83 ec 08             sub    $0x8,%rsp
  4003cc:       48 8b 05 25 0c 20 00    mov    0x200c25(%rip),%rax        # 600ff8 <_DYNAMIC+0x1d0>
  4003d3:       48 85 c0                test   %rax,%rax
  4003d6:       74 05                   je     4003dd <_init+0x15>
  4003d8:       e8 43 00 00 00          callq  400420 <__libc_start_main@plt+0x10>
  4003dd:       48 83 c4 08             add    $0x8,%rsp
  4003e1:       c3                      retq
```

where the `PREINIT_FUNCTION` is the `__gmon_start__` which does setup for profiling. You may note that we have no return instruction in the sysdeps/x86_64/crti.S. Actually that's why we got segmentation fault. Prolog of `_init` and `_fini` is placed in the sysdeps/x86_64/crtn.S assembly file:

`PREINIT_FUNCTION` `__gmon_start__` sysdeps/x86_64/crti.S `return` `segmentation fault` `_init` `_fini` sysdeps/x86_64/crtn.S

```
.section .init,"ax",@progbits
addq $8, %rsp
ret

.section .fini,"ax",@progbits
addq $8, %rsp
ret
```

```
$ gcc -nostdlib /lib64/crt1.o /lib64/crti.o /lib64/crtn.o  -lc -ggdb program.c -o program

$ ./program
x + y + z = 6
```

`_start` `main`

`_start` `ld` `.text`

```
$ ld --verbose | grep ENTRY
ENTRY(_start)
```

`_start` sysdeps/x86_64/start.S `__libc_start_main` `argc/argv` csu/libc-start.c `__libc_start_main` `stack canary` `main` `main`

```
result = main (argc, argv, __environ MAIN_AUXVEC_PARAM);
exit (result);
```

- system call
- gdb
- execve

- ELF
- x86_64
- segment registers
- context switch
- System V ABI

Linux

`linux-insides-zh` Linux (`Interrupt Descriptor Table`), (`Global Descriptor Table`)

Intel AMD

IDT

- - sync
- - sync
- - async

- - `%rip`
- - `%rip`
- -

RFLAGS.IF = 1 RFLAGS.IF

NMIrFLAGS.IF NMINMIIRET

“”“”25632 arch / x86 / include / asm / traps.h

```
/* / */
enum {
    X86_TRAP_DE = 0,    /*  0,  */
    X86_TRAP_DB,        /*  1,  */
    X86_TRAP_NMI,        /*  2,  */
    X86_TRAP_BP,        /*  3,  */
    X86_TRAP_OF,        /*  4,  */
    X86_TRAP_BR,        /*  5,  */
    X86_TRAP_UD,        /*  6,  */
    X86_TRAP_NM,        /*  7,  */
    X86_TRAP_DF,        /*  8,  */
    X86_TRAP_OLD_MF,    /*  9,  */
    X86_TRAP_TS,        /* 10,  TSS */
    X86_TRAP_NP,        /* 11,  */
    X86_TRAP_SS,        /* 12,  */
    X86_TRAP_GP,        /* 13,  */
    X86_TRAP_PF,        /* 14,  */
    X86_TRAP_SPURIOUS,    /* 15,  */
    X86_TRAP_MF,        /* 16, x87  */
    X86_TRAP_AC,        /* 17,  */
    X86_TRAP_MC,        /* 18,  */
    X86_TRAP_XF,        /* 19, SIMD  */
    X86_TRAP_IRET = 32,    /* 32, IRET  */
};
```

Error code

-
-

```
31                          16 15                                3   2   1   0
+-------------------------------------------------------------------------------+
|                             |                                  | T | I | E |
|          Reserved           |            Selector Index        | - | D | X |
|                             |                                  | I | T | T |
+-------------------------------------------------------------------------------+
```

:

- `EXT` - 1 0
- `IDT` - 1“” 0“”“LDT”“TI”
- `TI` - 1“LDT” 0“GDT”
- `Selector Index` - “GDT‘“LDT”“IDT”“IDT”“TI”

```
31                                                              4   3   2   1   0
+-------------------------------------------------------------------------------+
|                                                             |     | R | U | R | - |
|                       Reserved                              | I/D | S | - | - | P |
|                                                             |     | V | S | W | - |
+-------------------------------------------------------------------------------+
```

:

- `I/D` - 1
- `RSV` - 11;
- `U/S` - 0 `CPL = 0,12` 1CPL = 3
- `R/W` - 0 1
- `P` - 0 1

Interrupt Control Transfers

IDT

- `Task Gate` - TSS
- `Interrupt Gate` -
- `Trap Gate` -

```
127                                                                           96
+-------------------------------------------------------------------------------+
|                                                                               |
|                               Reserved                                        |
|                                                                               |
+-------------------------------------------------------------------------------
95                                                                            64
+-------------------------------------------------------------------------------+
|                                                                               |
|                             Offset 63..32                                     |
|                                                                               |
+-------------------------------------------------------------------------------+
63                              48 47      46  44   42    39             34    32
+-------------------------------------------------------------------------------+
|                                 |       |  D  |   |     |      |   |   |     |
|       Offset 31..16             |   P   |  P  | 0 |Type |0 0 0 | 0 | 0 | IST |
|                                 |       |  L  |   |     |      |   |   |     |
 -------------------------------------------------------------------------------+
31                                16 15                                       0
+-------------------------------------------------------------------------------+
|                                   |                                           |
|          Segment Selector         |                Offset 15..0               |
|                                   |                                           |
+-------------------------------------------------------------------------------+
```

- `Selector` -

- `Offset` -
- `DPL` -
- `P` -
- `IST` -
- `TYPE` - LDTTSS

`IDT` Linux `x86_64`

```
struct gate_struct64 {
    u16 offset_low;
    u16 segment;
    unsigned ist : 3, zero0 : 5, type : 5, dpl : 2, p : 1;
    u16 offset_middle;
    u32 offset_high;
    u32 zero1;
} __attribute__((packed));
```

arch/x86/include/asm/desc_defs.h

`IST` /

```
struct ldttss_desc64 {
    u16 limit0;
    u16 base0;
    unsigned base1 : 8, type : 5, dpl : 2, p : 1;
    unsigned limit1 : 4, zero0 : 3, g : 1, base2 : 8;
    u32 base3;
    u32 zero1;
} __attribute__((packed));
```

Exceptions During a Task Switch

TSSTSS

Nonmaskable interrupt

API

Interrupt Stack Table

Linux

- Linux/x86 boot protocol
- Linux kernel parameters

- 64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf

- 8250 UART Programming
- Serial ports on OSDEV

VGA

- Video Graphics Array (VGA)

IO

- IO port programming

GCC and GAS

- GCC type attributes
- Assembler Directives

- task_struct definition

- PowerPC and Linux Kernel Inside

- Linux x86 Program Start Up
- Memory Layout in Program Execution (32 bits)

()

@xinqiu

@lijiangsheng1

@littleneko

@qianmoke

@icecoobe

@choleraehyq

@mudongliang

@oska874

@cloudusers

@hailincai

@zmj1316

@zhangyangjing

@huxq

@worldwar

@keltoy

@a1ickgu0

@hao-lee

@woodpenker

@tjm-1990

@up2wing

@NeoCui

@narcijie

@biopuppet

@Albertchamberlain

@nannxnann