IPADS
INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

操作系统初始化

上海交通大学

https://www.sjtu.edu.cn

版权声明

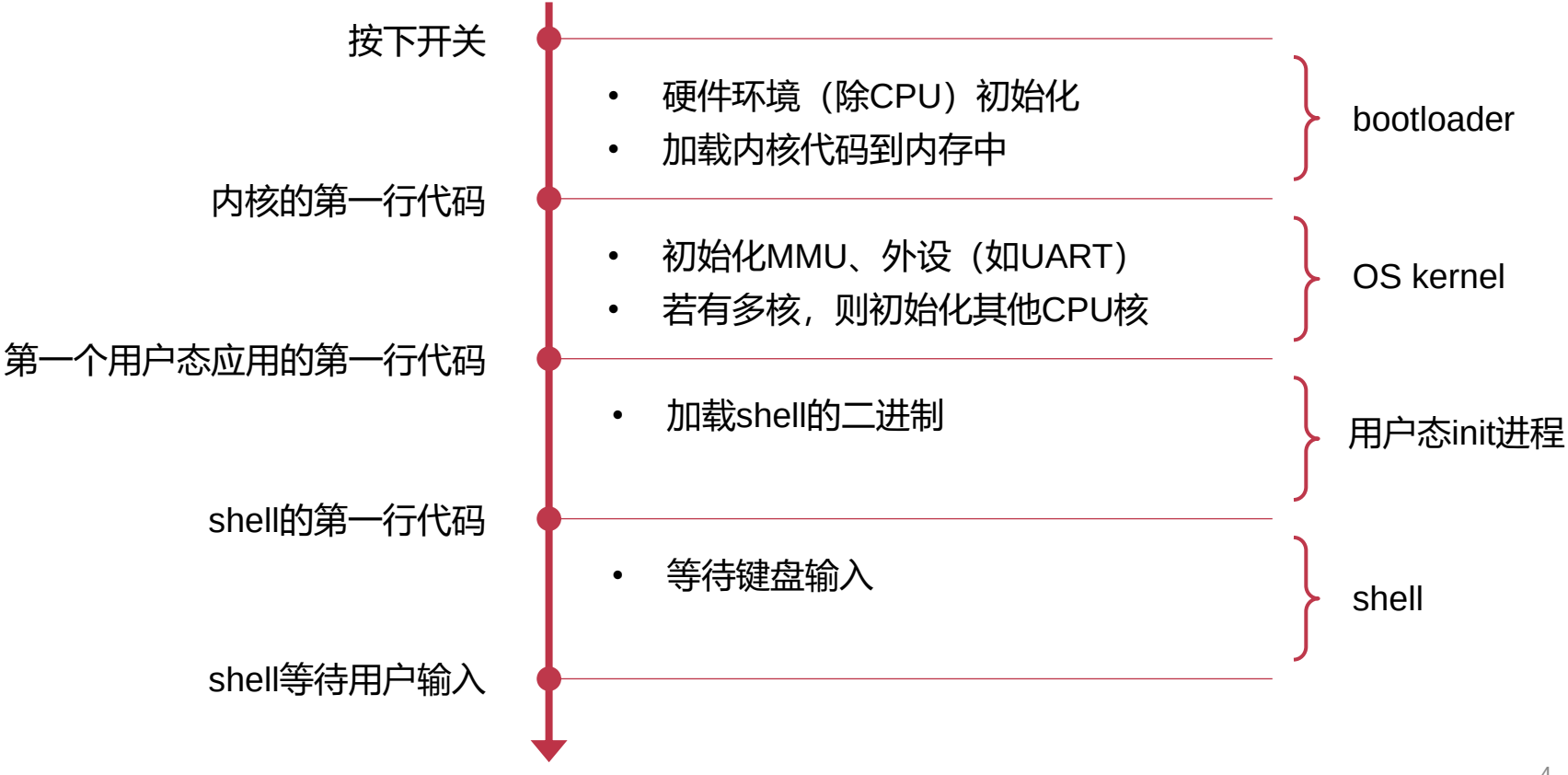
- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源
  - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
  - 完整文本：https://creativecommons.org/licenses/by/4.0/legalcode

启动：一个复杂的过程

- **为什么用“boot”表示启动?**
  - Boot源自bootstrap – 鞋带
  - "pull oneself up by one's bootstraps"
    - 通过拽鞋带，把自己拉起来

The computer term bootstrap began as a metaphor in the 1950s. In computers, pressing a bootstrap button caused a hardwired program to read a bootstrap program from an input unit. The computer would then execute the bootstrap program, which caused it to read more program instructions. It became a self-sustaining process that proceeded without external help from manually entered instructions. As a computing term, bootstrap has been used since at least 1953.

启动流程：从上电到等待用户输入



从计算机上电到内核开始运行

1. **上电后，开始执行BIOS ROM中的代码**
   - 自检（POST: Power-On Self Test）
   - 找到第一个可启动设备（如第一块磁盘）
   - 将可启动设备的第一个块（Master Boot Record ）加载到内存固定地址中
   - 跳转到bootloader的内存地址（MBR中包含bootloader）并继续执行
2. **bootloader开始执行**
   - 将内核的二进制文件从启动设备加载到内存中
   - 若内核文件是压缩包，则对其进行解压
   - 跳转到（解压后的）内核加载地址（物理地址）并继续执行
3. **内核代码开始执行**

实例：树莓派bootloader

- **树莓派：上电后真正运行的第一行代码**
  - 板子上电后固定从0x0地址运行firmware（也称 bootloader）
    - firmware放在SD卡中
  - 然后再由这段代码去初始化CPU、SDRAM等
  - 最后再加载内核、根文件系统到内存，实现系统启动
- **不同主板厂商的实现可以不同**
  - 这部分代码由主板厂商提供，使用人员通常不用关心
  - 树莓派的启动比较特殊：第一行代码由GPU运行
  - 不同版本的树莓派也可能不一样

内核入口函数位置

- **CPU从预定义的RAM地址读取第一行代码，由硬件厂商决定**
  - 树莓派：32位为`0x8000`，64位为`0x80000`

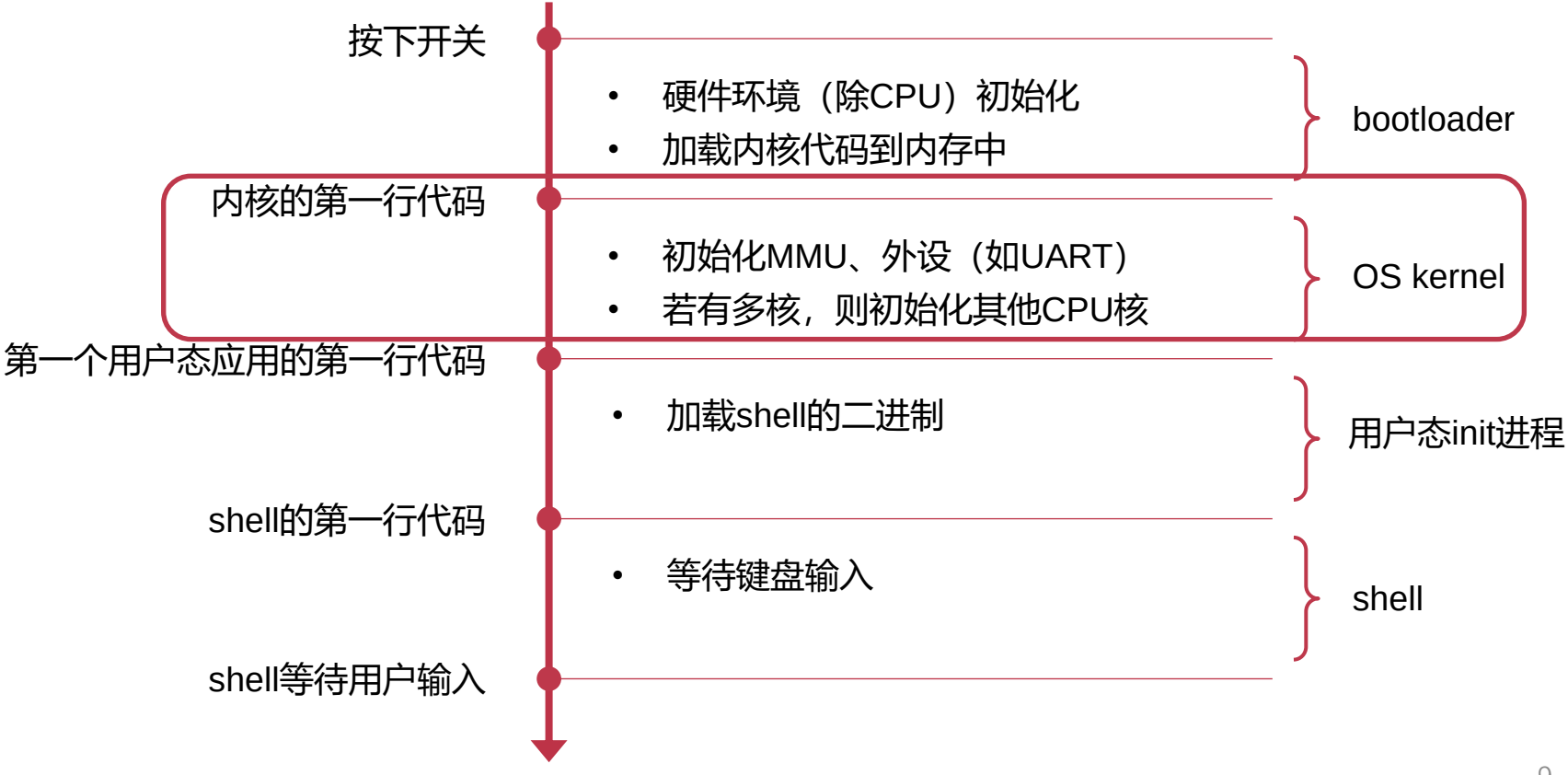
Q：这是虚拟地址还是物理地址?

kernel_address

`kernel_address` is the memory address to which the kernel image should be loaded. 32-bit kernels are loaded to address `0x8000` by default, and 64-bit kernels to address `0x80000`. If `kernel_old` is set, kernels are loaded to the address `0x0`.

https://www.raspberrypi.org/documentation/configuration/config-txt/boot.md

启动流程：从上电到等待用户输入



ARMv8架构下内核启动的3个主要任务

- **设置CPU异常级别（特权级别）**
  - 内核通常运行在EL1，CPU上电后的异常级别不一定是EL1，如何切换异常级别？
- **设置页表并开启虚拟内存机制**
  - 页表该如何配置？
  - 难点：开启地址翻译的前一行指令使用物理地址，开启后立即使用虚拟地址，前后如何衔接？
- **设置异常向量表并打开中断**
  - 异常向量表如何配置？
  - 打开后，异常处理的指令流如何流动？

课程实验内核启动

ChCore启动代码

```
➜ chcore-lab $ cd kernel/arch/aarch64
➜  aarch64 $ ls
CMakeLists.txt      boot      head.S      main.c      plat      tools.S
```

```
➜  kernel $  cd arch/aarch64/boot/raspi3
➜  raspi3 $ ls
CMakeLists.txt      firmware      include      init      peripherals
```

两个涉及到的目录：boot/raspi3 和 kernel

- boot/raspi3目录：firmware（bootloader），其余编译后放在 .init 段（低地址范围）
- kernel目录：编译后放在 .text 段（高地址范围），kernel初始化代码入口在arch/aarch64

ChCore内核的起始地址（编译脚本）

kernel/arch/aarch64/boot/raspi3/include/image.h

```
#pragma once

#define SZ_16K              0x4000
#define SZ_64K              0x10000

#define KERNEL_VADDR        0xffffff0000000000
#define TEXT_OFFSET         0x80000

```

kernel/arch/aarch64/boot/linker.tpl.ld

```
#include "../boot/image.h"

SECTIONS
{
        . = TEXT_OFFSET;
        img_start = .;
        init : {
                ${init_object}
        }

        . = ALIGN(SZ_16K);

        init_end = ABSOLUTE(.);

```

```
list(
    APPEND
    _init_sources
    init/start.S
    init/mmu.c
    init/tools.S
    init/init_c.c
    peripherals/uart.c)

set(init_objects
    ${init_objects}
    PARENT_SCOPE)
```

内核启动入口

kernel/arch/aarch64/boot/raspi3/CMakeLists.txt

内核运行的第一行代码：准备进入EL1

```
 9
10 BEGIN_FUNC(_start)
11     mrs x8, mpidr_el1 /* move core ID to x8 */
12     and x8, x8, #0xFF /* mask */
13     cbz x8, primary /* compare branch zero */
14
```

初始时CPU运行在EL3
（由硬件厂商决定）

```
45
46 primary:
47
48     /* Turn to el1 from other exception levels. */
49     bl  arm64_elX_to_el1
50
51     /* Prepare stack pointer and jump to C. */
52     adr     x0, boot_cpu_stack // only used for boot
53     add     x0, x0, #0x1000
54     mov     sp, x0
55
56     bl  init_c
57
58     /* Should never be here */
59     b   .
60 END_FUNC(_start)
```

设置当前EL为EL1（内核的运行级）

设置启动时用的栈（用于C的函数调用）

跳转到C代码（不再返回到_start函数）

boot/raspi3/init/start.S

```
64
65 BEGIN_FUNC(arm64_elX_to_el1)
66     mrs x9, CurrentEL // read from a reg, decided by the board
67
68     // Check the current exception level.
69     cmp x9, CURRENTEL_EL1
70     beq .Ltarget // if EL1, no need to eret, just ret
71     cmp x9, CURRENTEL_EL2
72     beq .Lin_el2 // if EL1, need to eret from EL2
73     // Otherwise, we are in EL3.
74
75     mrs x9, scr_el3 // scr: secure configure reg
76     mov x10, SCR_EL3_NS | SCR_EL3_HCE | SCR_EL3_RW
77     orr x9, x9, x10
78     msr scr_el3, x9
79
80     // Set the return address and exception level.
81     adr x9, .Ltarget
82     msr elr_el3, x9 // elr: exeception link reg
83     mov x9, SPSR_ELX_DAIF | SPSR_ELX_EL1H
84     msr spsr_el3, x9
85
```

```
128
129     isb
130     eret
131
132 .Ltarget:
133     ret // retaddr in x30
134 END_FUNC(arm64_elX_to_el1)
135
```

树莓派启动后，CPU运行在EL3

设置scr_el3寄存器：NS、HCE、RW（后一页）

为eret做准备：

1. 设置EL3的exception link register（返回地址）
2. 设置EL3的状态寄存器SPSR
   （D: debug; A: error; I: interrupt; F: fast interrupt）

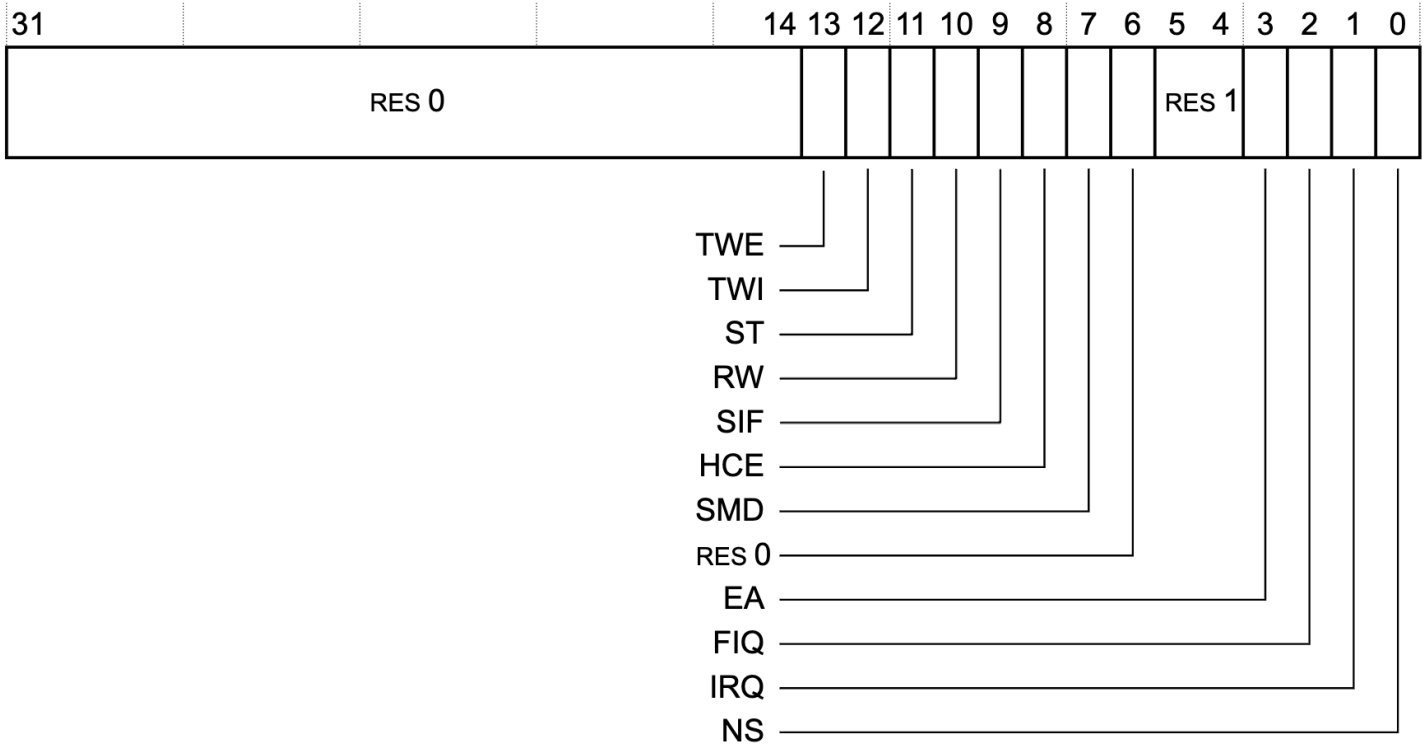
isb: memory barrier，保证顺序执行

eret，跳到 .Ltarget，同时进入EL1

ret: 返回到start.S的50行

https://github.com/s-matyukevich/raspberry-pi-os/blob/master/docs/lesson02/rpi-os.md

Figure 4.38. SCR_EL3 bit assignments



https://developer.arm.com/documentation/ddi0500/j/System-Control/AArch64-register-descriptions/Secure-Configuration-Register

小知识

Bit	V	Semantic
[10] RW		Register Width control for lower exception levels. The possible values are:
	0: 1:	Lower levels are all AArch32. This is the reset value. The next lower level is AArch64.
[8] HCE		Hypervisor Call Enable. This bit enables the use of HVC instructions. The possible values are:
	0: 1:	The HVC instruction is undefined at all exception levels. This is the reset value. The HVC instruction is enabled at EL1, EL2 or EL3.
[0] NS		Non-Secure bit. The possible values are. The possible values are:
	0: 1:	EL0 and EL1 are in Secure state, memory accesses from those exception levels can access Secure memory. This is the reset value. EL0 and EL1 are in Non-secure state, memory accesses from those exception levels cannot access Secure memory.

boot/raspi3/init/start.S

```
45
46 primary:
47
48     /* Turn to el1 from other exception levels. */
49     bl  arm64_elX_to_el1
50
51     /* Prepare stack pointer and jump to C. */
52     adr     x0, boot_cpu_stack // only used for boot
53     add     x0, x0, #0x1000
54     mov     sp, x0
55
56     bl  init_c
57
58     /* Should never be here */
59     b   .
60 END_FUNC(_start)
```

boot/raspi3/init/init_c.c

```
1 #include "boot.h"
2 #include "image.h"
3
4 typedef unsigned long u64;
5
6 #define INIT_STACK_SIZE 0x1000
7 char boot_cpu_stack[PLAT_CPU_NUMBER][INIT_STACK_SIZE] ALIGN(16);
8
```

```
65
66 void init_c(void)
67 {
68     /* Clear the bss area for the kernel image */
69     clear_bss();
70
71     /* Initialize UART before enabling MMU. */
72     early_uart_init();
73     uart_send_string("boot: init_c\r\n");
74
75     wakeup_other_cores(); // no need for qemu
76
77     /* Initialize Boot Page Table. */
78     uart_send_string("[BOOT] Install boot page table\r\n");
79     init_boot_pt();
80
81     /* Enable MMU. */
82     el1_mmu_activate();
83     uart_send_string("[BOOT] Enable el1 MMU\r\n");
84
85     /* Call Kernel Main. */
86     uart_send_string("[BOOT] Jump to kernel main\r\n");
87     start_kernel(secondary_boot_flag);
88
89     /* Never reach here */
90 }
```

设置栈为boot_cpu_stack，之后就可以调用C函数了。

- 问：为什么调C函数之前要设置栈?
- 问：栈的大小是多少？为什么够?

跳转到高地址运行

```
1 #pragma once
2
3 #define SZ_16K          0x4000
4 #define SZ_64K          0x10000
5
6 #define KERNEL_VADDR    0xffffff0000000000
7 #define TEXT_OFFSET     0x80000
8
```

kernel/arch/aarch64/head.S

```
12
13 #include <common/asm.h>
14 #include <common/vars.h>
15
16 BEGIN_FUNC(start_kernel) // high memory addr
17     /*
18      * Code in bootloader specified only the primary
19      * cpu with MPIDR = 0 can be boot here. So we directly
20      * set the TPIDR_EL1 to 0, which represent the logical
21      * cpuid in the kernel
22      */
23     mov     x3, #0
24     msr     TPIDR_EL1, x3 // set CPU ID, only the primary will run this code
25
26     ldr     x2, =kernel_stack // high memory addr
27     add     x2, x2, KERNEL_STACK_SIZE
28     mov     sp, x2 // switch stack, important
29     bl      main
30 END_FUNC(start_kernel)
31
```

换栈：高地址区域

kernel/arch/aarch64/boot/linker.tpl.ld

```
1 #include "../boot/image.h"
2
3 SECTIONS
4 {
5     . = TEXT_OFFSET;
6     img_start = .;
7     init : {
8         ${init_object}
9     }
10
11     . = ALIGN(SZ_16K);
12
13     init_end = ABSOLUTE(.);
14
15     .text KERNEL_VADDR + init_end : AT(init_end) {
16         *(.text*)
17     }
18
```

start_kernel位于高虚拟地址：0xffffff0000000000 + init_end

问：start_kernel函数在物理内存中实际上紧邻着init（在低地址），为什么可以跳到高地址段执行它？

页表初始化

回到 init_c：页表初始化

```
65
66 void init_c(void)
67 {
68     /* Clear the bss area for the kernel image */
69     clear_bss();
70
71     /* Initialize UART before enabling MMU. */
72     early_uart_init();
73     uart_send_string("boot: init_c\r\n");
74
75     wakeup_other_cores(); // no need for qemu
76
77     /* Initialize Boot Page Table. */
78     uart_send_string("[BOOT] Install boot page table\r\n");
79     init_boot_pt();
80
81     /* Enable MMU. */
82     el1_mmu_activate();
83     uart_send_string("[BOOT] Enable el1 MMU\r\n");
84
85     /* Call Kernel Main. */
86     uart_send_string("[BOOT] Jump to kernel main\r\n");
87     start_kernel(secondary_boot_flag);
88
89     /* Never reach here */
90 }
91
```

```
12 /* The number of entries in one page table page */
13 #define PTP_ENTRIES 512
14 /* The size of one page table page */
15 #define PTP_SIZE 4096
16 #define ALIGN(n) __attribute__((__aligned__(n)))
17 u64 boot_ttbr0_l0[PTP_ENTRIES] ALIGN(PTP_SIZE);
18 u64 boot_ttbr0_l1[PTP_ENTRIES] ALIGN(PTP_SIZE);
19 u64 boot_ttbr0_l2[PTP_ENTRIES] ALIGN(PTP_SIZE);
20
21 u64 boot_ttbr1_l0[PTP_ENTRIES] ALIGN(PTP_SIZE);
22 u64 boot_ttbr1_l1[PTP_ENTRIES] ALIGN(PTP_SIZE);
23 u64 boot_ttbr1_l2[PTP_ENTRIES] ALIGN(PTP_SIZE);
```

```
 6 /* Physical memory address space: 0-1G */
 7 #define PHYSMEM_START   (0x0UL)
 8 #define PHYSMEM_BOOT_END (0x10000000UL)
 9 #define PERIPHERAL_BASE (0x20000000UL)
10 #define PHYSMEM_END (0x40000000UL)
```

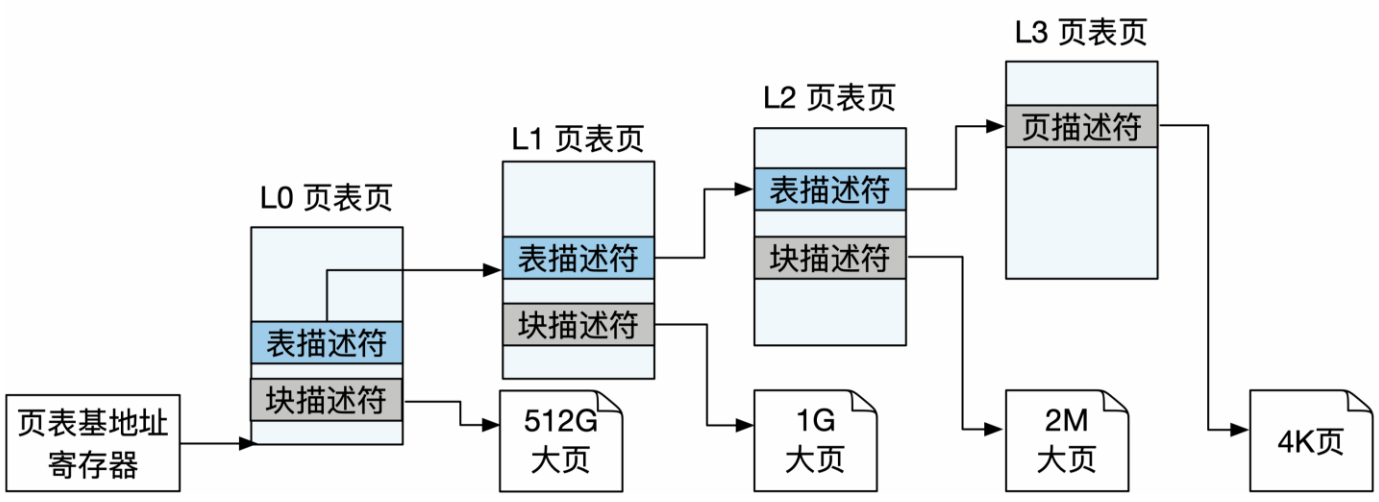
```
39
40 void init_boot_pt(void)
41 {
42     u32 start_entry_idx;
43     u32 end_entry_idx;
44     u32 idx;
45     u64 kva;
46
47     /* TTBR0_EL1 0-1G */
48     boot_ttbr0_l0[0] = ((u64) boot_ttbr0_l1) | IS_TABLE | IS_VALID;
49     boot_ttbr0_l1[0] = ((u64) boot_ttbr0_l2) | IS_TABLE | IS_VALID;
50
51     /* Usable memory: PHYSMEM_START ~ PERIPHERAL_BASE */
52     start_entry_idx = PHYSMEM_START / SIZE_2M;
53     end_entry_idx = PERIPHERAL_BASE / SIZE_2M;
54
55     /* Map each 2M page */
56     for (idx = start_entry_idx; idx < end_entry_idx; ++idx) {
57         boot_ttbr0_l2[idx] = (PHYSMEM_START + idx * SIZE_2M)
58             | UXN    /* Unprivileged execute never */
59             | ACCESSED  /* Set access flag */
60             | INNER_SHARABLE    /* Sharebility */
61             | NORMAL_MEMORY /* Normal memory */
62             | IS_VALID;
63     }
```

为什么是2M?

设置TTBR0页表（低地址使用）

boot/raspi3/init/mmu.c

回顾：2M大页与L2页表项



块描述符：指向大页

页表初始化

```
39
40 void init_boot_pt(void)
41 {
42     u32 start_entry_idx;
43     u32 end_entry_idx;
44     u32 idx;
45     u64 kva;
46
47     /* TTBR0_EL1 0-1G */
48     boot_ttbr0_l0[0] = ((u64) boot_ttbr0_l1) | IS_TABLE | IS_VALID;
49     boot_ttbr0_l1[0] = ((u64) boot_ttbr0_l2) | IS_TABLE | IS_VALID;
50
51     /* Usable memory: PHYSMEM_START ~ PERIPHERAL_BASE */
52     start_entry_idx = PHYSMEM_START / SIZE_2M;
53     end_entry_idx = PERIPHERAL_BASE / SIZE_2M;
54
55     /* Map each 2M page */
56     for (idx = start_entry_idx; idx < end_entry_idx; ++idx) {
57         boot_ttbr0_l2[idx] = (PHYSMEM_START + idx * SIZE_2M)
58             | UXN    /* Unprivileged execute never */
59             | ACCESSED  /* Set access flag */
60             | INNER_SHARABLE    /* Sharebility */
61             | NORMAL_MEMORY /* Normal memory */
62             | IS_VALID;
63     }
```

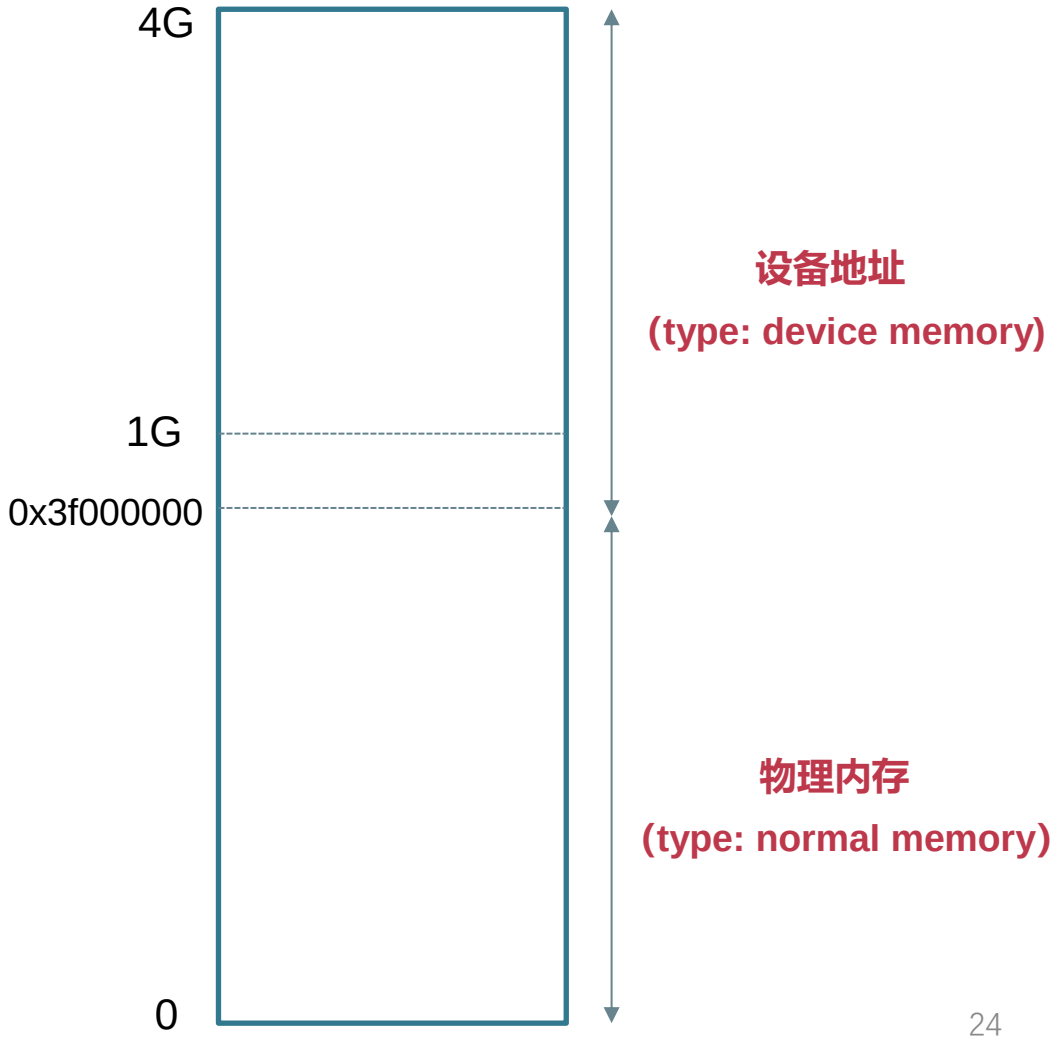
```
17 #define INNER_SHAREABLE  (0x3)
18 /* Please search mair_el1 for these memory types. */
19 #define NORMAL_MEMORY    (0x4)
20 #define DEVICE_MEMORY    (0x0)
```

```
63     }
64
65     /* Peripheral memory: PERIPHERAL_BASE ~ PHYSMEM_END */
66
67     /* Raspi3b/3b+ Peripherals: 0x3f 00 00 00 - 0x3f ff ff ff */
68     start_entry_idx = end_entry_idx;
69     end_entry_idx = PHYSMEM_END / SIZE_2M;
70
71     /* Map each 2M page */
72     for (idx = start_entry_idx; idx < end_entry_idx; ++idx) {
73         boot_ttbr0_l2[idx] = (PHYSMEM_START + idx * SIZE_2M)
74             | UXN    /* Unprivileged execute never */
75             | ACCESSED  /* Set access flag */
76             | DEVICE_MEMORY /* Device memory */ // non-cachable
77             | IS_VALID;
78     }
```

映射完内存地址（左）后，映射设备地址（右）

- 问：设备内存与物理内存有什么区别？

树莓派3b+


4G
1G
0x3f000000
0
设备地址
(type: device memory)
物理内存
(type: normal memory)

页表初始化

```
1 #pragma once
2
3 #define SZ_16K          0x4000
4 #define SZ_64K          0x10000
5
6 #define KERNEL_VADDR    0xffffff0000000000
7 #define TEXT_OFFSET     0x80000
8
```

```
80      /*
81       * TTBR1_EL1 0-1G
82       * KERNEL_VADDR: L0 pte index: 510; L1 pte index: 0; L2 pte index: 0.
83       */
84      kva = KERNEL_VADDR;
85      boot_ttbr1_l0[GET_L0_INDEX(kva)] = ((u64) boot_ttbr1_l1)
86          | IS_TABLE | IS_VALID;
87      boot_ttbr1_l1[GET_L1_INDEX(kva)] = ((u64) boot_ttbr1_l2)
88          | IS_TABLE | IS_VALID;
89
90      start_entry_idx = GET_L2_INDEX(kva);
91      /* Note: assert(start_entry_idx == 0) */
92      end_entry_idx = start_entry_idx + PHYSMEM_BOOT_END / SIZE_2M;
93      /* Note: assert(end_entry_idx < PTP_ENTIRES) */
94
95      /*
96       * Map each 2M page
97       * Usuable memory: PHYSMEM_START ~ PERIPHERAL_BASE
98       */
99      for (idx = start_entry_idx; idx < end_entry_idx; ++idx) {
100         boot_ttbr1_l2[idx] = (PHYSMEM_START + idx * SIZE_2M)
101             | UXN    /* Unprivileged execute never */
102             | ACCESSED  /* Set access flag */
103             | INNER_SHARABLE    /* Sharebility */
104             | NORMAL_MEMORY /* Normal memory */
105             | IS_VALID;
106     }
107
```

设置TTBR1页表（高地址使用）

```
36 #define GET_L0_INDEX(x) (((x) >> (12 + 9 + 9 + 9)) & 0x1ff)
37 #define GET_L1_INDEX(x) (((x) >> (12 + 9 + 9)) & 0x1ff)
38 #define GET_L2_INDEX(x) (((x) >> (12 + 9)) & 0x1ff)
```

问：GET_L0_INDEX 等宏的作用?

对比设置 TTBR0：方式相同，虚拟地址不同

```
55      /* Map each 2M page */
56      for (idx = start_entry_idx; idx < end_entry_idx; ++idx) {
57          boot_ttbr0_l2[idx] = (PHYSMEM_START + idx * SIZE_2M)
58              | UXN    /* Unprivileged execute never */
59              | ACCESSED  /* Set access flag */
60              | INNER_SHARABLE    /* Sharebility */
61              | NORMAL_MEMORY /* Normal memory */
62              | IS_VALID;
63      }
```

设置TTBR0页表（低地址使用）

页表设置完，开启翻译

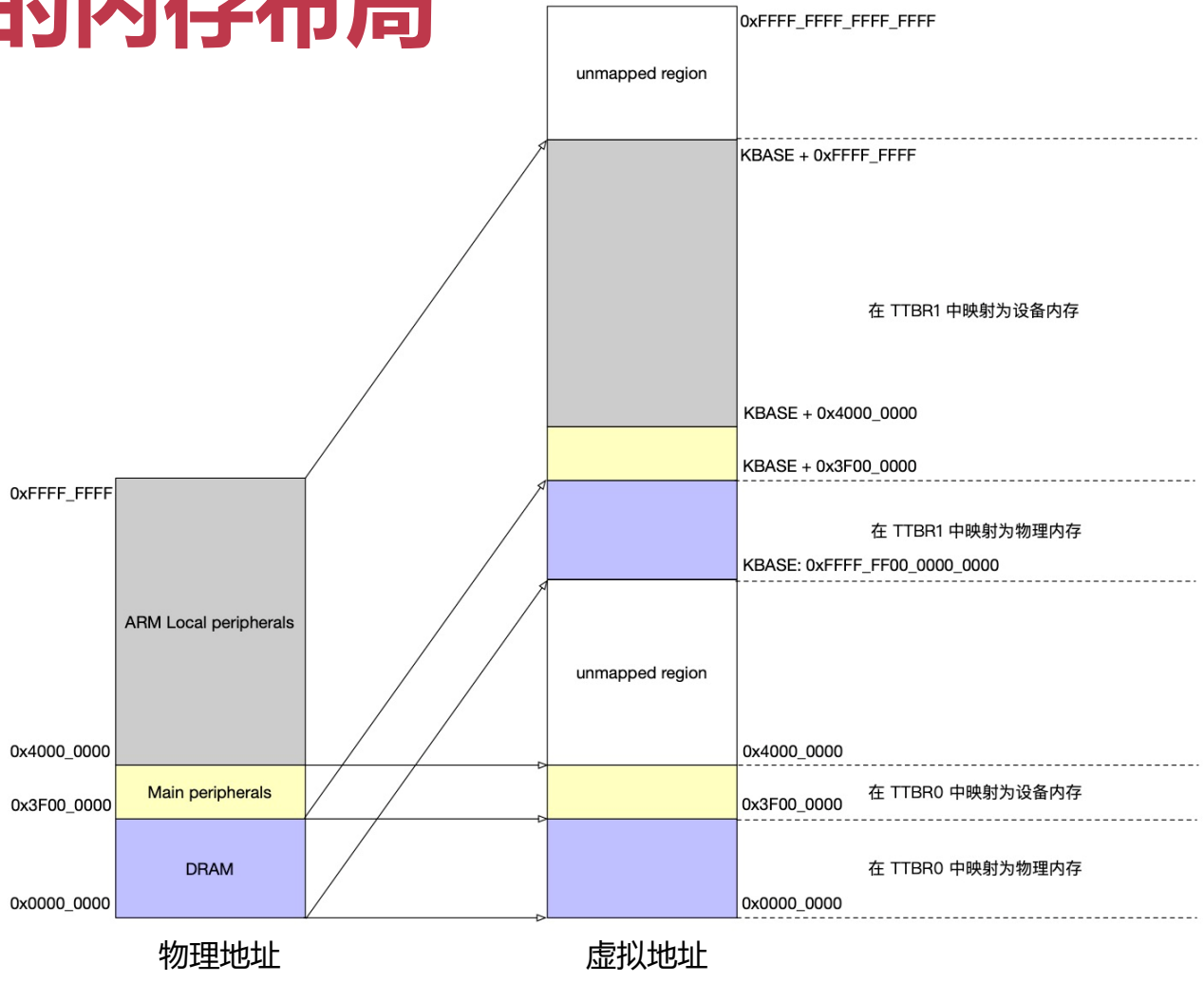
```
66 void init_c(void)
67 {
68     /* Clear the bss area for the kernel image */
69     clear_bss();
70
71     /* Initialize UART before enabling MMU. */
72     early_uart_init();
73     uart_send_string("boot: init_c\r\n");
74
75     wakeup_other_cores(); // no need for qemu
76
77     /* Initialize Boot Page Table. */
78     uart_send_string("[BOOT] Install boot page table\r\n");
79     init_boot_pt();
80
81     /* Enable MMU. */
82     el1_mmu_activate();
83     uart_send_string("[BOOT] Enable el1 MMU\r\n");
84
85     /* Call Kernel Main. */
86     uart_send_string("[BOOT] Jump to kernel main\r\n");
87     start_kernel(secondary_boot_flag);
88
89     /* Never reach here */
90 }
```

```
224 BEGIN_FUNC(el1_mmu_activate)
225     stp     x29, x30, [sp, #-16]!
226     mov     x29, sp
227
228     bl  invalidate_cache_all
229
230     /* Invalidate TLB */
231     tlbi    vmalle1is
232     isb
233     dsb     sy
234
235     /* Initialize Memory Attribute Indirection Register */
236     ldr     x8, =MMU_MAIR_ATTR1 | MMU_MAIR_ATTR2 | MMU_MAIR_ATTR3
237     msr     mair_el1, x8
238
239     /* Initialize TCR_EL1 */
240     /* set cacheable attributes on translation walk */
241     /* (SMP extensions) non-shareable, inner write-back write-allocate */
242     ldr     x8, =MMU_TCR_FLAGS1 | MMU_TCR_FLAGS0 | MMU_TCR_IPS | MMU_TCR_AS
243     msr     tcr_el1, x8 // translation control reg
244     isb
245
246     /* Write ttbr with phys addr of the translation table */
247     adrp    x8, boot_ttbr0_l0
248     msr     ttbr0_el1, x8
249     adrp    x8, boot_ttbr1_l0
250     msr     ttbr1_el1, x8
251     isb
252
253     mrs     x8, sctlr_el1
254     /* Enable MMU */
255     orr     x8, x8, #SCTLR_EL1_M // set bit of MMU
256     /* Disable alignment checking */
257     bic     x8, x8, #SCTLR_EL1_A
258     bic     x8, x8, #SCTLR_EL1_SA0
259     bic     x8, x8, #SCTLR_EL1_SA
260     orr     x8, x8, #SCTLR_EL1_nAA
261     /* Data accesses Cacheable */
262     orr     x8, x8, #SCTLR_EL1_C
263     /* Instruction access Cacheable */
264     orr     x8, x8, #SCTLR_EL1_I
265     msr     sctlr_el1, x8 // commit point
266
267     ldp     x29, x30, [sp], #16 // opposite to 226, push/pop
268     ret
269 END_FUNC(el1_mmu_activate)
```

将页表的物理地址，
写入 TTBR0 和 TTBR1

1. 将sctlr_el1寄存器写入x8
2. 设置x8某些位（开关）
3. 将x8写回sctlr_el1寄存器

此时的内存布局


0xFFFF_FFFF_FFFF_FFFF
unmapped region
KBASE + 0xFFFF_FFFF
在 TTBR1 中映射为设备内存
KBASE + 0x4000_0000
KBASE + 0x3F00_0000
在 TTBR1 中映射为物理内存
KBASE: 0xFFFF_FF00_0000_0000
unmapped region
0xFFFF_FFFF
ARM Local peripherals
0x4000_0000
0x4000_0000
Main peripherals
0x3F00_0000
0x3F00_0000
在 TTBR0 中映射为设备内存
DRAM
在 TTBR0 中映射为物理内存
0x0000_0000
0x0000_0000
物理地址
虚拟地址

开启页表前后

```
224 BEGIN_FUNC(el1_mmu_activate)
225     stp     x29, x30, [sp, #-16]!
226     mov     x29, sp
227
252
253     mrs     x8, sctlr_el1
254     /* Enable MMU */
255     orr     x8, x8, #SCTLR_EL1_M // set bit of MMU
256     /* Disable alignment checking */
257     bic     x8, x8, #SCTLR_EL1_A
258     bic     x8, x8, #SCTLR_EL1_SA0
259     bic     x8, x8, #SCTLR_EL1_SA
260     orr     x8, x8, #SCTLR_EL1_nAA
261     /* Data accesses Cacheable */
262     orr     x8, x8, #SCTLR_EL1_C
263     /* Instruction access Cacheable */
264     orr     x8, x8, #SCTLR_EL1_I
265     msr     sctlr_el1, x8 // commit point
266
267     ldp     x29, x30, [sp], #16 // opposite to 226, push/pop
268     ret
269 END_FUNC(el1_mmu_activate)
```

265时：尚未使用页表

267时：PC等地址已经过MMU翻译

```
65
66 void init_c(void)
67 {
68     /* Clear the bss area for the kernel image */
69     clear_bss();
70
71     /* Initialize UART before enabling MMU. */
72     early_uart_init();
73     uart_send_string("boot: init_c\r\n");
74
75     wakeup_other_cores(); // no need for qemu
76
77     /* Initialize Boot Page Table. */
78     uart_send_string("[BOOT] Install boot page table\r\n");
79     init_boot_pt();
80
81     /* Enable MMU. */
82     el1_mmu_activate();
83     uart_send_string("[BOOT] Enable el1 MMU\r\n");
84
85     /* Call Kernel Main. */
86     uart_send_string("[BOOT] Jump to kernel main\r\n");
87     start_kernel(secondary_boot_flag);
88
89     /* Never reach here */
90 }
```

执行267行，为何能顺利执行?

- 低地址范围：虚拟地址和物理地址完全相同
- 回想下页表中的映射（TTBR0）

start_kernel位于高地址段：

- 0xffffff0000000000 + init_end
- 从 init_c （低地址范围）跳过去后
- 高地址范围的地址已经被映射
- 栈在 start_kernel 已经换成了高地址

异常向量表初始化

异常向量表初始化

```
50 void main(paddr_t boot_flag)
51 {
52         u32 ret = 0;
53
54         /* Init big kernel lock */
55         kernel_lock_init();
56         kinfo("[ChCore] lock init finished\n");
57         BUG_ON(ret != 0);
58
59         /* Init uart: no need to init the uart again */
60         uart_init();
61         kinfo("[ChCore] uart init finished\n");
62
63 #ifdef CHCORE_KERNEL_TEST
64         lab2_test_kernel_vaddr();
65 #endif /* CHCORE_KERNEL_TEST */
66
67         /* Init mm */
68         mm_init();
69         kinfo("[ChCore] mm init finished\n");
70
71 #ifdef CHCORE_KERNEL_TEST
72         void lab2_test_kmalloc(void);
73         lab2_test_kmalloc();
74         void lab2_test_page_table(void);
75         lab2_test_page_table();
76 #endif /* CHCORE_KERNEL_TEST */
77
78         /* Init exception vector */
79         arch_interrupt_init();
```

```
22 void arch_interrupt_init_per_cpu(void)
23 {
24         disable_irq();
25
26         /* platform dependent init */
27         set_exception_vector();
28         plat_interrupt_init();
29 }
30
31 void arch_interrupt_init(void)
32 {
33         arch_interrupt_init_per_cpu();
34         memset(irq_handle_type, HANDLE_KERNEL, MAX_IRQ_NUM);
35 }
```

```
16
17 BEGIN_FUNC(set_exception_vector)
18     adr x0, el1_vector
19     msr vbar_el1, x0
20     ret
21 END_FUNC(set_exception_vector)
22
```

el1_vector（异常向量表）

回顾：异常向量表基地址寄存器

异常向量表

```
.macro  exception_entry label
        /* Each entry should be 0x80 aligned */
        .align  7
        b       \label
.endm
```

```
EXPORT(el1_vector)
/* LAB 3 TODO BEGIN */
/* BLANK BEGIN */
        exception_entry sync_el1t               // Synchronous EL1t
        exception_entry irq_el1t                // IRQ EL1t
        exception_entry fiq_el1t                // FIQ EL1t
        exception_entry error_el1t              // Error EL1t

        exception_entry sync_el1h               // Synchronous EL1h
        exception_entry irq_el1h                // IRQ EL1h
        exception_entry fiq_el1h                // FIQ EL1h
        exception_entry error_el1h              // Error EL1h

        exception_entry sync_el0_64             // Synchronous 64-bit EL0
        exception_entry irq_el0_64              // IRQ 64-bit EL0
        exception_entry fiq_el0_64              // FIQ 64-bit EL0
        exception_entry error_el0_64            // Error 64-bit EL0

        exception_entry sync_el0_32             // Synchronous 32-bit EL0
        exception_entry irq_el0_32              // IRQ 32-bit EL0
        exception_entry fiq_el0_32              // FIQ 32-bit EL0
        exception_entry error_el0_32            // Error 32-bit EL0
```

异常处理函数示例

```
sync_el0_64:
        /* Since we cannot touch x0-x7, we need some extra work here */
        exception_enter
        mrs     x25, esr_el1
        lsr     x24, x25, #ESR_EL1_EC_SHIFT
        cmp     x24, #ESR_EL1_EC_SVC_64
        b.eq    el0_syscall
        /* Not supported exception */
        mov     x0, SYNC_EL0_64
        mrs     x1, esr_el1
        mrs     x2, elr_el1
        bl      handle_entry_c
        /* BLANK BEGIN */
        bl      unlock_kernel
        /* BLANK END */
        exception_exit
```

上下文保存

```
39 .macro  exception_enter
40     sub sp, sp, #ARCH_EXEC_CONT_SIZE
41     stp x0, x1, [sp, #16 * 0]
42     stp x2, x3, [sp, #16 * 1]
43     stp x4, x5, [sp, #16 * 2]
44     stp x6, x7, [sp, #16 * 3]
45     stp x8, x9, [sp, #16 * 4]
46     stp x10, x11, [sp, #16 * 5]
47     stp x12, x13, [sp, #16 * 6]
48     stp x14, x15, [sp, #16 * 7]
49     stp x16, x17, [sp, #16 * 8]
50     stp x18, x19, [sp, #16 * 9]
51     stp x20, x21, [sp, #16 * 10]
52     stp x22, x23, [sp, #16 * 11]
53     stp x24, x25, [sp, #16 * 12]
54     stp x26, x27, [sp, #16 * 13]
55     stp x28, x29, [sp, #16 * 14]
56     mrs x10, sp_el0
57     mrs x11, elr_el1
58     mrs x12, spsr_el1
59     stp x30, x10, [sp, #16 * 15]
60     stp x11, x12, [sp, #16 * 16]
61 .endm
```

```
63 .macro  exception_exit
64     ldp x11, x12, [sp, #16 * 16]
65     ldp x30, x10, [sp, #16 * 15]
66     msr sp_el0, x10
67     msr elr_el1, x11
68     msr spsr_el1, x12
69     ldp x0, x1, [sp, #16 * 0]
70     ldp x2, x3, [sp, #16 * 1]
71     ldp x4, x5, [sp, #16 * 2]
72     ldp x6, x7, [sp, #16 * 3]
73     ldp x8, x9, [sp, #16 * 4]
74     ldp x10, x11, [sp, #16 * 5]
75     ldp x12, x13, [sp, #16 * 6]
76     ldp x14, x15, [sp, #16 * 7]
77     ldp x16, x17, [sp, #16 * 8]
78     ldp x18, x19, [sp, #16 * 9]
79     ldp x20, x21, [sp, #16 * 10]
80     ldp x22, x23, [sp, #16 * 11]
81     ldp x24, x25, [sp, #16 * 12]
82     ldp x26, x27, [sp, #16 * 13]
83     ldp x28, x29, [sp, #16 * 14]
84     add sp, sp, #ARCH_EXEC_CONT_SIZE
85     eret
86 .endm
```

系统调用

```
sync_el0_64:
        /* Since we cannot touch x0-x7, we need some extra work here */
        exception_enter
        mrs     x25, esr_el1
        lsr     x24, x25, #ESR_EL1_EC_SHIFT
        cmp     x24, #ESR_EL1_EC_SVC_64
        b.eq    el0_syscall
        /* Not supported exception */
        mov     x0, SYNC_EL0_64
        mrs     x1, esr_el1
        mrs     x2, elr_el1
        bl      handle_entry_c
        /* BLANK BEGIN */
        bl      unlock_kernel
        /* BLANK END */
        exception_exit
```

回顾：异常症状寄存器 ESR_EL1

系统调用

```
sync_el0_64:
        /* Since we cannot touch x0-x7, we need some extra work here */
        exception_enter
        mrs     x25, esr_el1
        lsr     x24, x25, #ESR_EL1_EC_SHIFT
        cmp     x24, #ESR_EL1_EC_SVC_64
        b.eq    el0_syscall
        /* Not supported exception */
        mov     x0, SYNC_EL0_64
        mrs     x1, esr_el1
        mrs     x2, elr_el1
        bl      handle_entry_c
        /* BLANK BEGIN */
        bl      unlock_kernel
        /* BLANK END */
        exception_exit
```

```
el0_syscall:
        adr     x27, syscall_table              // syscall table in x27
        uxtw    x16, w8                         // syscall number in x16
        ldr     x16, [x27, x16, lsl #3]         // find the syscall entry
        blr     x16

        /* Ret from syscall */
        str     x0, [sp]
        /* BLANK BEGIN */
        bl      unlock_kernel
        /* BLANK END */
```

```
55 const void *syscall_table[NR_SYSCALL] = {
56     [0 ... NR_SYSCALL - 1] = sys_debug,
57     /* lab3 syscalls finished */
58
59     [SYS_getc] = sys_getc,
60     [SYS_yield] = sys_yield,
61     [SYS_create_device_pmo] = sys_create_device_pmo,
62     [SYS_unmap_pmo] = sys_unmap_pmo,
63     [SYS_create_thread] = sys_create_thread,
64     [SYS_create_process] = sys_create_process,
65     [SYS_register_server] = sys_register_server,
66     [SYS_register_client] = sys_register_client,
67     [SYS_ipc_call] = sys_ipc_call,
68     [SYS_ipc_return] = sys_ipc_return,
69     [SYS_cap_copy_to] = sys_cap_copy_to,
70     [SYS_cap_copy_from] = sys_cap_copy_from,
71     [SYS_set_affinity] = sys_set_affinity,
72     [SYS_get_affinity] = sys_get_affinity,
73     /* ... */
74     [SYS_get_cpu_id] = sys_get_cpu_id,
75
76     [SYS_create_pmos] = sys_create_pmos,
77     [SYS_map_pmos] = sys_map_pmos,
78     [SYS_write_pmo] = sys_write_pmo,
79     [SYS_read_pmo] = sys_read_pmo,
80     [SYS_transfer_caps] = sys_transfer_caps,
81
82     /* TMP FS */
83     [SYS_fs_load_cpio] = sys_fs_load_cpio,
84
85     [SYS_top] = sys_top,
86     [SYS_debug] = sys_debug
87 };
```

```
267 /* syscalls */
268 int sys_create_process(void)
269 {
270     struct process *new_process;
271     struct vmspace *vmspace;
272     int cap, r;
273
274     /* cap current process */
275     new_process = obj_alloc(TYPE_PROCESS, sizeof(*new_process));
276     if (!new_process) {
277         r = -ENOMEM;
278         goto out_fail;
279     }
```

小结：从实验内核看系统初始化

- **设置CPU异常级别为EL1**
- **设置页表并开启虚拟内存机制**
  - TTBR0_EL1: 虚拟地址 = 物理地址
  - TTBR1_EL1: 虚拟地址 = 物理地址 + OFFSET
- **设置异常向量表**
  - 每个异常向量表项跳转到对应的异常处理函数
  - 处理异常前保存进程上下文、返回进程前恢复其上下文