Lecture4 System Init

计算机启动

- **内核启动的两个任务**

1. 配置页表并开启虚拟内存机制，允许使用虚拟地址

> Q：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？
>
> A：低地址部分物理内存和虚拟内存相等，即可衔接(tricky)

2. 配置异常向量表并打开中断，允许“双循环”

> 1. Exception/syscalls->异常向量表->handler
> 2. Interrupt->异常向量表->handler

内核代码加载与运行

树莓派上电

1. 板子上电之后固定从0x0地址运行firmware(bootloader)
2. 然后再由这段代码去初始化CPU、SDRAM等
3. 最后再加载内核、根文件系统到内存，实现系统启动

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

  x86：0x7C00

ChCore启动代码

> 有`boot`和`kernel`两个目录
>
> ICS得好好复习复习（内存结构方面）！！！

1. boot目录：编译后放在`.init`段，低地址范围(bootloader)
2. kernel目录：编译后放在`.text`段，高地址范围

> `.`代表当前位置，一开始赋值为`TEXT_OFFSET`
>
> `.o`文件的位置不能换，因为一开始必须执行`start.S.o`以初始化

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/start.S

> Arm不一定启动在EL3，所以需要去设置EL等级——by`arm64_elx_to_el1`

```
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
（后面代码起作用的主要是_el3相关部分）

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

> `bl`指令不需要栈也可以ret，因为`x30`寄存器是返回值寄存器(link register: `lr`)

回到 boot/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/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C函数

- 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？

注意，本页代码依然在boot目录

因为栈是从上到下增长的，而启动时内存地址是从下往上分配的，所以得将boot_cpu_stack加上0x1000后作为栈底，即栈的大小为4K

Q1回答：C语言会有压栈操作(参数传递、Caller-saved、Callee-saved)

Q2回答：栈的大小是4K，跑boot程序是够的，且不允许调用递归

从 /boot 到 /kernel

```
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/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

scripts/linker-aarch64.lds.in

```
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位于高地址段：0xffffff0000000000 + init_end

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

完成换栈操作，开始调用main函数

Q：为什么可以跳到高地址执行start_kernel？（最最tricky的地方）

> A：因为完成了虚拟地址的映射

页表初始化

回到 boot/init_c：页表初始化

```c
void init_c(void)
{
    /* Clear the bss area for the kernel image */
    clear_bss();

    /* Initialize UART before enabling MMU. */
    early_uart_init();
    uart_send_string("boot: init_c\r\n");

    wakeup_other_cores(); // no need for qemu

    /* Initialize Boot Page Table. */
    uart_send_string("[BOOT] Install boot page table\r\n");
    init_boot_pt();

    /* Enable MMU. */
    el1_mmu_activate();
    uart_send_string("[BOOT] Enable el1 MMU\r\n");

    /* Call Kernel Main. */
    uart_send_string("[BOOT] Jump to kernel main\r\n");
    start_kernel(secondary_boot_flag);

    /* Never reach here */
}
```

```c
/* The number of entries in one page table page */
#define PTP_ENTRIES 512
/* The size of one page table page */
#define PTP_SIZE 4096
#define ALIGN(n) __attribute__((__aligned__(n)))
u64 boot_ttbr0_l0[PTP_ENTRIES] ALIGN(PTP_SIZE);
u64 boot_ttbr0_l1[PTP_ENTRIES] ALIGN(PTP_SIZE);
u64 boot_ttbr0_l2[PTP_ENTRIES] ALIGN(PTP_SIZE);

u64 boot_ttbr1_l0[PTP_ENTRIES] ALIGN(PTP_SIZE);
u64 boot_ttbr1_l1[PTP_ENTRIES] ALIGN(PTP_SIZE);
u64 boot_ttbr1_l2[PTP_ENTRIES] ALIGN(PTP_SIZE);
```

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

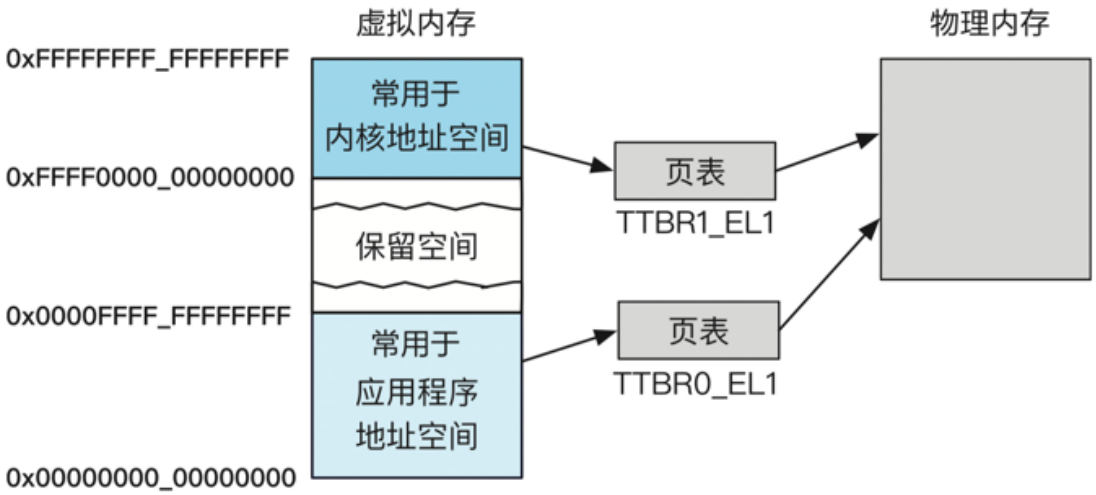
```c
void init_boot_pt(void)
{
    u32 start_entry_idx;
    u32 end_entry_idx;
    u32 idx;
    u64 kva;

    /* TTBR0_EL1 0-1G */
    boot_ttbr0_l0[0] = ((u64) boot_ttbr0_l1) | IS_TABLE | IS_VALID;
    boot_ttbr0_l1[0] = ((u64) boot_ttbr0_l2) | IS_TABLE | IS_VALID;

    /* Usable memory: PHYSMEM_START ~ PERIPHERAL_BASE */
    start_entry_idx = PHYSMEM_START / SIZE_2M;
    end_entry_idx = PERIPHERAL_BASE / SIZE_2M;

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

为什么是2M?

设置TTBR0页表（低地址使用）

boot/raspi3/init/mmu.c

> TTBR就相当于是x86中的CR3（存放Page-Directory Base，页表的基地址）
>
> Q：那为什么要用TTBR0_EL0和TTBR1_EL1两个寄存器呢?
>
> A：因为ARM的虚拟内存只有48位，中间一段空间不用。0x0000大头的地址空间用TTBR0_EL1来翻译，0xffff打头的地址空间用TTBR1_EL1来翻译（相当于CR3被切分成两半）



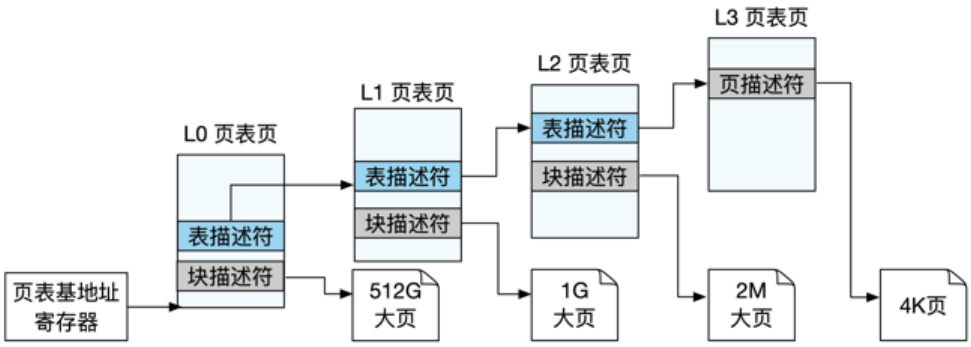
> Q：为什么代码中是`ttbr0_l0`和`ttbr0_l1`?
>
> A：我们有四级页表，l0和l1分别代表L0页表页和L1页表页，零级页表指向一级页表，一级页表指向二级页表

- 树莓派的内存一共4G，0~1G是我们可以使用的物理内存，2G~4G是外设映射的内存

Q：为什么要用2M?

A：因为设置的是L2的大页，大小为2M

2M大页与L2页表项



块描述符：指向大页

回到 init_c：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?
- 问：为什么要设置为 non-cachable?

25

Q：设备内存与物理内存有什么区别?

A：设备内存不是真实的内存，是设备映射在内存上的地址(MMIO)，对应的是设备上的寄存器

Q：为什么要设置成non-cachable?

A：因为设备寄存器的值会变，而如果存入到cache中在CPU看来就不会变（要做轮询）

物理地址并不是百分百都是物理内存，还有可能是设备的地址

回到 init_c：页表初始化

```
#define SZ_16K          0x4000
#define SZ_64K          0x10000

#define KERNEL_VADDR    0xffffff0000000000
#define TEXT_OFFSET     0x80000
```

```
    /*
     * TTBR1_EL1 0-1G                设置TTBR1页表（高地址使用）
     * KERNEL_VADDR: L0 pte index: 510; L1 pte index: 0; L2 pte index: 0.
     */
    kva = KERNEL_VADDR;
    boot_ttbr1_l0[GET_L0_INDEX(kva)] = ((u64) boot_ttbr1_l1)
        | IS_TABLE | IS_VALID;
    boot_ttbr1_l1[GET_L1_INDEX(kva)] = ((u64) boot_ttbr1_l2)
        | IS_TABLE | IS_VALID;

    start_entry_idx = GET_L2_INDEX(kva);
    /* Note: assert(start_entry_idx == 0) */
    end_entry_idx = start_entry_idx + PHYSMEM_BOOT_END / SIZE_2M;
    /* Note: assert(end_entry_idx < PTP_ENTIRES) */

    /*
     * Map each 2M page
     * Usuable memory: PHYSMEM_START ~ PERIPHERAL_BASE
     */
    for (idx = start_entry_idx; idx < end_entry_idx; ++idx) {
        boot_ttbr1_l2[idx] = (PHYSMEM_START + idx * SIZE_2M)
            | UXN    /* Unprivileged execute never */
            | ACCESSED  /* Set access flag */
            | INNER_SHARABLE    /* Sharebility */
            | NORMAL_MEMORY /* Normal memory */
            | IS_VALID;
    }
```

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

对比设置 TTBR0：
设置的方式相同，虚拟地址范围不同

```
    /* Map each 2M page */
    for (idx = start_entry_idx; idx < end_entry_idx; ++idx) {
        boot_ttbr0_l2[idx] = (PHYSMEM_START + idx * SIZE_2M)
            | UXN    /* Unprivileged execute never */
            | ACCESSED  /* Set access flag */
            | INNER_SHARABLE    /* Sharebility */
            | NORMAL_MEMORY /* Normal memory */
            | IS_VALID;          设置TTBR0页表（低地址使用）
    }
```

> 这里是在初始化高地址区域，与低地址基本类似
>
> 然后这里的高地址和低地址映射到同一块物理内存上

页表设置完，开启翻译

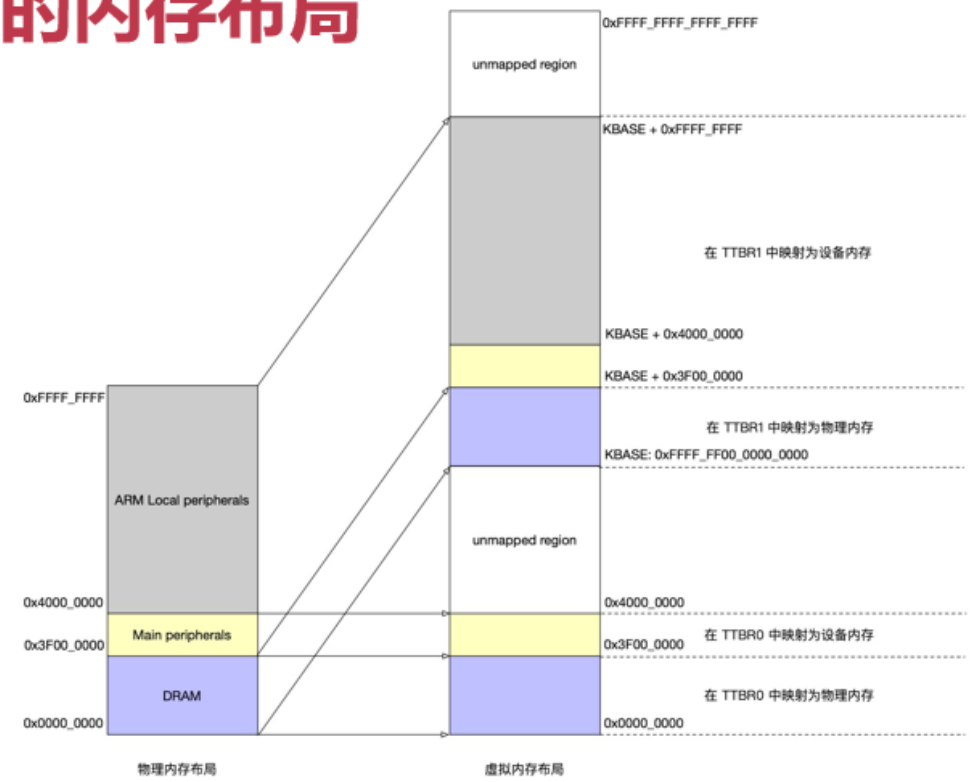
依然是 init_c.c

```
void init_c(void)
{
    /* Clear the bss area for the kernel image */
    clear_bss();

    /* Initialize UART before enabling MMU. */
    early_uart_init();
    uart_send_string("boot: init_c\r\n");

    wakeup_other_cores(); // no need for qemu

    /* Initialize Boot Page Table. */
    uart_send_string("[BOOT] Install boot page table\r\n");
    init_boot_pt();

    /* Enable MMU. */
    el1_mmu_activate();  用汇编写的函数，为什么?
    uart_send_string("[BOOT] Enable el1 MMU\r\n");

    /* Call Kernel Main. */
    uart_send_string("[BOOT] Jump to kernel main\r\n");
    start_kernel(secondary_boot_flag);

    /* Never reach here */
}
```

```
    stp     x29, x30, [sp, #-16]!
    mov     x29, sp

    bl  invalidate_cache_all

    /* Invalidate TLB */
    tlbi    vmalle1is
    isb
    dsb     sy

    /* Initialize Memory Attribute Indirection Register */
    ldr     x8, =MMU_MAIR_ATTR1 | MMU_MAIR_ATTR2 | MMU_MAIR_ATTR3
    msr     mair_el1, x8

    /* Initialize TCR_EL1 */
    /* set cacheable attributes on translation walk */
    /* (SMP extensions) non-shareable, inner write-back write-allocate */
    ldr     x8, =MMU_TCR_FLAGS1 | MMU_TCR_FLAGS0 | MMU_TCR_IPS | MMU_TCR_AS
    msr     tcr_el1, x8 // translation control reg
    isb

    /* Write ttbr with phys addr of the translation table */
    adrp    x8, boot_ttbr0_l0
    msr     ttbr0_el1, x8          将页表的物理地址，
    adrp    x8, boot_ttbr1_l0      写入 TTBR0 和 TTBR1
    msr     ttbr1_el1, x8
    isb

    mrs     x8, sctlr_el1
    /* Enable MMU */
    orr     x8, x8, #SCTLR_EL1_M // set bit of MMU
    /* Disable alignment checking */
    bic     x8, x8, #SCTLR_EL1_A
    bic     x8, x8, #SCTLR_EL1_SA0   1. 将sctlr_el1寄存器写入x8
    bic     x8, x8, #SCTLR_EL1_SA    2. 设置x8某些bit为1（开关）
    orr     x8, x8, #SCTLR_EL1_nAA   3. 将x8写回sctlr_el1寄存器
    /* Data accesses Cacheable */
    orr     x8, x8, #SCTLR_EL1_C
    /* Instruction access Cacheable */
    orr     x8, x8, #SCTLR_EL1_I
    msr     sctlr_el1, x8 // commit point

    ldp     x29, x30, [sp], #16 // opposite to 226, push/pop
    ret
END_FUNC(el1_mmu_activate)
```

> Q：为什么要用汇编写?
>
> A：对于一些System ISA，C语言并不支持（比如控制页表），所以需要回到汇编
>
> - 将x8写回SCLTR_EL1之后开始虚拟地址翻译

此时的内存布局



> 内存分为**物理内存(normal memory)**和**设备虚拟化地址(device memory)**

```
 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)
```

> 树莓派3b+里面物理内存的地址是(0~0x3f000000)，设备地址是(0x3f000000~4G)
>
> > Main peripherals(0x3f000000~1G), ARM Local peripherals(1G~4G)
>
> 配置完页表后，物理内存中所有的地址都会在Kernel中有映射，但只有0~1G的空间（包含normal memory和一定大小的设备地址）会在userspace（低地址)中有映射

开启页表前后

依然是 init_c.c

```
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   265时：尚未使用页表
266
267     ldp     x29, x30, [sp], #16 // opposite to 226, push/pop
268     ret                                   267时：PC等地址已经过MMU翻译
269 END_FUNC(el1_mmu_activate)
```

```
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?

- 页表不是在低地址区域（0-1G）么?
- 虚拟地址和物理地址完全相同
- 回想下页表中的映射（TTBR1）

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

> 看看上面的内存分布图就能明白

异常向量表初始化

异常向量表初始化（kernel/main.c）

```
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   el1_vector (异常向量表)
20     ret
21 END_FUNC(set_exception_vector)
22
```

回顾：异常向量表基地址寄存器

Summary

1. 设置CPU异常级别为EL1
2. 设置初始化时的简单页表，并开启虚拟内存机制

   TTBR0_EL1：vaddr=paddr

   TTBR1_EL1：vaddr=paddr + offset
3. 设置异常向量表

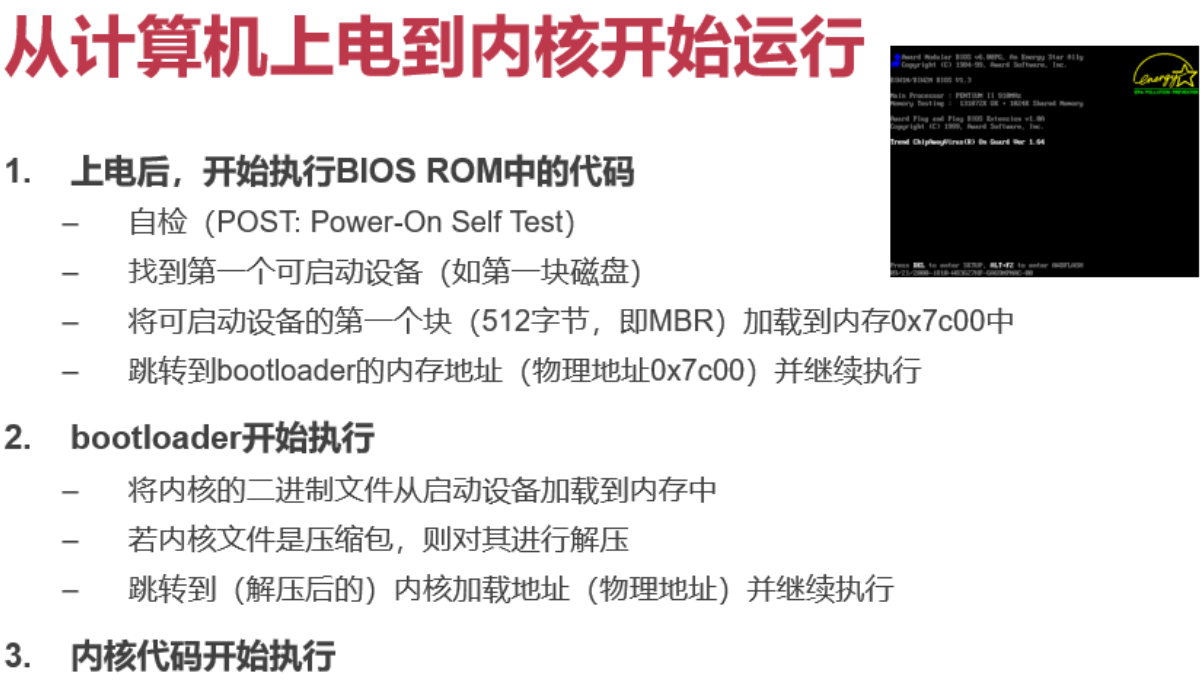
   每个异常向量表项->异常处理函数

   保存&恢复context

> 硬件的本质是同步/异步异常

内核启动前：BIOS的作用

从计算机上电到内核开始运行



> BIOS本质是一个小系统，具有加载内存的能力
>
> 以前内存很小，如果能压缩内核文件几百K是一件很好的事情

BIOS

> Basic Input/Output System

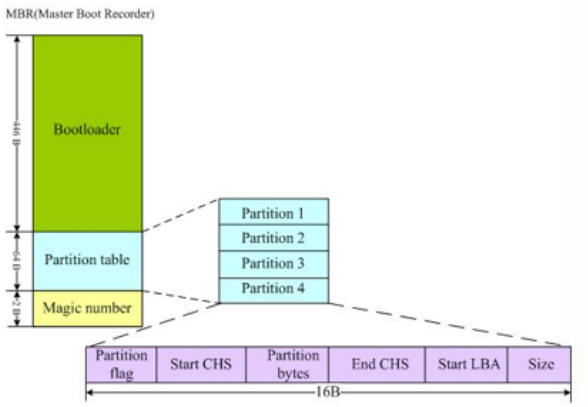
- BIOS保存在主板的只读内存中(ROM：Read-Only Memory)
- CPU负责执行BIOS，x86 CPU在reset后，**PC固定指向0xFFF0（BIOS物理地址）**
- 许多嵌入式设备中并没有BIOS

上电自检(POST)

- BIOS程序首先检查，计算机硬件能否满足运行的基本条件，这叫做“硬件自检”(Power-On Self-Test)
- 如果硬件出现问题，主板会发出不同含义的蜂鸣，启动中止。如果没有问题，屏幕就会显示出CPU、内存、硬盘等信息。

MBR(Master Boot Record)

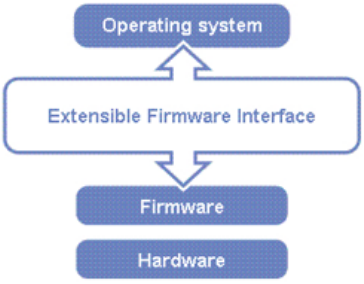
- **MBR：主引导记录**
  - 磁盘的0柱面0磁头0扇区称为主引导扇区
- **三部分组成**
  1. 主引导程序（boot loader）
  2. 硬盘分区表DPT（disk partition table）
  3. 硬盘有效标志（0x55AA）



EFI/UEFI

> Intel做的兼容性工作 硬件+软件->生态

- **Intel提出来EFI取代BIOS interface**
  - EFI（Extensible Firmware Interface）
- **2005年，Intel再次提出UEFI取代EFI**
  - UEFI（Unified Extensible Firmware Interface）



两种启动的对比

- **定制化的主板（常见的ARM开发板，通常不再扩展其他设备）**
  - 需要初始化具体主板相关硬件如GPIO和内存等
  - 初始化的时候就预先知道有哪些设备
  - 一般由厂商提供的BSP完成
- **通用的主板（常见如PC，通常需要再插入其他设备）**
  - 系统配置情况在开机时候是不知道的
  - 需要探测（Probe）、Training(内存和PCIe)和枚举（PCIe等等即插即用设备）
  - BIOS/EFI提供了整个主板、包括主板上外插的设备的软件抽象
    - 通过定义的接口把这些信息传递给OS，使OS不改而能够适配到所有机型和硬件

> 定制化的主板：在BSP写死
>
> 通用主板：使用BIOS/EFI

ARM嵌入式设备启动的特点

- **通常与设备强相关**
  - 厂商提供私有固件（firmware）用于初始化
    - 一般称为BSP： Board Support Package
  - 不同厂商的方案往往相差很大
- **缺点：对可插拔外设的兼容性**
  - ARM嵌入式设备一般不支持PCIe等外设

总结

- **x86平台常见组合**
  - BIOS（在ROM）：传统BIOS、EFI/UEFI、coreboot
  - Bootloader（在磁盘的MBR）：NTLDR、Grub
  - Linux kernel（在磁盘其他位置）
- **ARM嵌入式平台常见组合**
  - ROM code（在ROM）：主板厂商私有
  - Bootloader（在SD卡）：如uboot（非必须）
  - Linux kernel（在SD卡）