Lab1: BOOT!!!

(参考来源：上海交通大学并行与分布式系统研究所+操作系统实验) Creative Commons Attribution 4.0 License

1. Prerequisite

国内大部分的镜像都坏掉了，我们只能自己部署一个加速的镜像。

1. 首先fork这个项目，记得给一颗星：第三方docker加速

2. 然后前往cloudflare官网：workers & pages > overview > Pages > Connect to Git

点击save and deploy即可。

成功后可以看到以下界面：

我们的开发是在ubuntu 22.04.2 LTS上进行，架构为aarch64. 采用snap安装docker。直接在application中搜索docker即可安装。注意赋予其全部的权限。(不是为什么要snap啊)

接下来进行换源。需要进行如下操作：

```
sudo vim /var/snap/docker/current/config/daemon.json
```

注意:wq!强制保存。

Tips: apt 安装则需要进行这一步：

```
sudo vim /etc/docker/daemon.json
```

后将以下这一段加入：

```
{
    "log-level":        "error",
    "storage-driver":   "overlay2",
    "registry-mirrors": [
            "https://docker.fxxk.dedyn.io"
    ]
}
```

即可完成操作。

1. 每次编译的时候，如果遇到问题，则需要执行`sudo chmod -R 777`。不管是make后还是进行调试前。
2. 在debug配置过程中，vscode中的midebuggerpath可以通过`which gdb`获取新的路径进行修改。

Environment Settings:

利用命令返回到我们的实验文件夹下。 进行

```
make build
make qemu
```

进行拉取镜像和编译工作。

```
parallels@ubuntu-linux-22-04-02-desktop:~/Desktop/labs/OS-Course-Lab-ChCore-Lab1$ make build
chbuild: use_cgroup: , tags:
Unable to find image 'ipads/chcore_builder:v1.8.3' locally
v1.8.3: Pulling from ipads/chcore_builder
0fb668748fc8: Pull complete
fe1061f02714: Pull complete
680dac4f8bc0: Downloading [==================>                    ]  193.2MB/456.1MB
dcaf12565bed: Download complete
596b8c862a88: Download complete
ec515d817d11: Download complete
841cdf8aff29: Download complete
b6e6289c990f: Download complete
7a695be17089: Download complete
```

```
[ 57%] Building C object CMakeFiles/kernel.img.dir/arch/aarch64/boot/raspi3/init/init_c.c.obj
[ 71%] Building C object CMakeFiles/kernel.img.dir/arch/aarch64/boot/raspi3/peripherals/uart.c.obj
[ 85%] Building ASM object CMakeFiles/kernel.img.dir/incbin_procmgr_bin.S.obj
[100%] Linking C executable kernel.img
[100%] Built target kernel.img
Scanning dependencies of target kernel8.img
[100%] Built target kernel8.img
[ 87%] Performing install step for 'kernel'
[  0%] Built target touch-inc-procmgr
[  0%] Built target linker.ld
Scanning dependencies of target kernel.img
[ 14%] Building ASM object CMakeFiles/kernel.img.dir/incbin_procmgr_bin.S.obj
[ 28%] Linking C executable kernel.img
[100%] Built target kernel.img
[100%] Built target kernel8.img
Install the project...
-- Install configuration: "Release"
-- Installing: /home/parallels/Desktop/labs/OS-Course-Lab-ChCore-Lab1/build/kernel.img
-- Installing: /home/parallels/Desktop/labs/OS-Course-Lab-ChCore-Lab1/build/kernel8.img
-- Installing: /home/parallels/Desktop/labs/OS-Course-Lab-ChCore-Lab1/build/emulate.sh
-- Installing: /home/parallels/Desktop/labs/OS-Course-Lab-ChCore-Lab1/build/simulate.sh
[100%] Completed 'kernel'
[100%] Built target kernel
Succeeded to build all targets
To use chcore toolchain, please input the commnad
        export CMAKE_TOOLCHAIN_FILE=/home/parallels/Desktop/labs/OS-Course-Lab-ChCore-Lab1/build/toolchain.cmake
```

接下来通过make qemu查看是否可以运行。

```
parallels@ubuntu-linux-22-04-02-desktop:~/Desktop/labs/OS-Course-Lab-ChCore-Lab1$ make qemu
QEMU: Terminated
parallels@ubuntu-linux-22-04-02-desktop:~/Desktop/labs/OS-Course-Lab-ChCore-Lab1$
```

OK. 我们就完成了这份基础的环境搭建。

和先前一样，我们在一个终端`make qemu-gdb`,在另一个终端`make gdb`。

```
问题   输出   调试控制台   终端   端口

                    [ No Source Available ]

   > 0x80000 <_start>        mrs     x8, mpidr_el1
     0x80004 <_start+4>      and     x8, x8, #0xff
     0x80008 <_start+8>      cbz     x8, 0x8005c <_start+92>
     0x8000c <_start+12>     adr     x0, 0x88458 <clear_bss_flag>
     0x80010 <_start+16>     ldr     x1, [x0]
     0x80014 <_start+20>     cmp     x1, #0x0
     0x80018 <_start+24>     b.ne    0x8000c <_start+12>  // b.any
     0x8001c <_start+28>     bl      0x88000 <arm64_elX_to_el1>

remote Thread 1.1 In: _start                                   L??   PC: 0x80000
(gdb)

make
make
```

2. 启动0号核

`_start` 函数（位于 `kernel/arch/aarch64/boot/raspi3/init/start.S`）是 ChCore 内核启动时执行的第一块代码。由于 QEMU 在模拟机器启动时会同时开启 4 个 CPU 核心，于是 4 个核会同时开始执行 `_start` 函数。而在内核的初始化过程中，我们通常需要首先让其中一个核进入初始化流程，待进行了一些基本的初始化后，再让其他核继续执行。

> 思考题 1：阅读 `_start` 函数的开头，尝试说明 ChCore 是如何让其中一个核首先进入初始化流程，并让其他核暂停执行的。

我们仅需要阅读前三行代码即可。寻找到kernel/arch/aarch64/boot/raspi3/init/start.S 文件，我们有：

```
mrs x8, mpidr_el1
and x8, x8, #0xFF
cbz x8, primary
```

`mrs` 读入mpidr_el1系统级寄存器的状态。mpidr_el1是一个64位寄存器，其结构如下：

MPIDR_EL1, Multiprocessor Affinity Register, EL1

The MPIDR_EL1 provides an additional core identification mechanism for scheduling purposes in a cluster.

Bit field descriptions

MPIDR_EL1 is a 64-bit register, and is part of the Other system control registers functional group.

This register is Read Only.

Figure B2-70 MPIDR_EL1 bit assignments

RES0, [63:40]

RES0 Reserved.

Aff3, [39:32]

Affinity level 3. Highest level affinity field.

CLUSTERID
Indicates the value read in the **CLUSTERIDAFF3** configuration signal.

RES1, [31]

RES1 Reserved.

U, [30]

Indicates a single core system, as distinct from core 0 in a cluster. This value is:

`0` Core is part of a multiprocessor system. This is the value for implementations with more than one core, and for implementations with an ACE or CHI master interface.

[29:25]

UNK Reserved.

MT, [24]

Indicates whether the lowest level of affinity consists of logical cores that are implemented using a multithreading type approach. This value is:

`1` Performance of PEs at the lowest affinity level is very interdependent.

Affinity0 represents threads. Cortex-A75 is not multithreaded, but may be in a system with other cores that are multithreaded.

Aff2, [23:16]

Affinity level 2. Second highest level affinity field.

CLUSTERID
Indicates the value read in the **CLUSTERIDAFF2** configuration signal.

Aff1, [15:12]

Part of Affinity level 1. Third highest level affinity field.

RAZ Read-As-Zero.

Aff1, [11:8]

Part of Affinity level 1. Third highest level affinity field.

CPUID. Identification number for each CPU in the Cortex-A75 cluster:

`0x0` MP1: CPUID: 0.

to

`0x7` MP8: CPUID: 7.

Aff0, [7:0]

Affinity level 0. The level identifies individual threads within a multithreaded core. The Cortex-A75 core is single-threaded, so this field has the value `0x00`.

我们可以获得在x8中的值，这里表示采用CPUID=0，单线程核。

在MPIDR_EL1中，亲和力（Affinity）具有{Aff0,Aff1,Aff2,Aff3}四个等级，并且0等级的重要度最高。因此判断是否为单核启动，只需要取其低8位的数字，即将x8与0xFF进行AND操作后，当值为0时，进入`primary`函数中，进行初始化操作。

其余核心则依次进入`wait_for_bss_clear`和`wait_until_smp_enabled`被暂时挂起，等待基本初始化完成后继续执行。

```
    /* Wait for bss clear */
wait_for_bss_clear:
    adr x0, clear_bss_flag
    ldr x1, [x0]
    cmp     x1, #0
    bne wait_for_bss_clear
```

3. 切换异常级

> 练习题 2：在 `arm64_elX_to_el1` 函数的 `LAB 1 TODO 1` 处填写一行汇编代码，获取 CPU 当前异常级别。

首先我们先阅读以下start.S 代码：

```
...
.extern arm64_elX_to_el1
...
BEGIN_FUNC(_start)
    mrs x8, mpidr_el1
    and x8, x8, #0xFF
    cbz x8, primary
...
primary:

    /* Turn to el1 from other exception levels. */
    bl  arm64_elX_to_el1

    /* Prepare stack pointer and jump to C. */
    adr     x0, boot_cpu_stack
    add     x0, x0, #INIT_STACK_SIZE
    mov     sp, x0

    b   init_c
    ...
```

AArch64 架构中，特权级被称为异常级别（Exception Level，EL），四个异常级别分别为 EL0、EL1、EL2、EL3，其中 EL3 为最高异常级别，常用于安全监控器（Secure Monitor），EL2 其次，常用于虚拟机监控器（Hypervisor），EL1 是内核常用的异常级别，也就是通常所说的内核态，EL0 是最低异常级别，也就是通常所说的用户态。

为了使 `arm64_elX_to_el1` 函数具有通用性，我们没有直接写死从 EL3 降至 EL1 的逻辑，而是首先判断当前所在的异常级别，并根据当前异常级别的不同，跳转到相应的代码执行。

因此我们需要从系统寄存器中获取当前的状态级别。我们只需要执行：

```
mrs x9, CURRENTEL
```

即可。

可以看到我们的异常级分别通过EL0->0 EL1->4 EL2->8 EL3->12表示。我们目前位于异常级3. 有关currentEL：仅有两个bit用于标识异常级，也就是EL。分别为0，1，2，3. （为什么64个bit只用了两

个。。。）

CurrentEL, Current Exception Level

The CurrentEL characteristics are:

Purpose

Holds the current Exception level.

Configurations

There are no configuration notes.

Attributes

CurrentEL is a 64-bit register.

Field descriptions

Bits [63:4]

Reserved, RES0.

EL, bits [3:2]

Current Exception level.

0b00 EL0.

0b01 EL1.

0b10 EL2.

0b11 EL3.

When the HCR_EL2.NV bit is 1, EL1 read accesses to the CurrentEL register return the value of 0b10 in this field.

The reset behavior of this field is:

- This field resets to the highest implemented Exception level.

Bits [1:0]

Reserved, RES0.

> 练习题 3：在 `arm64_elX_to_el1` 函数的 `LAB 1 TODO 2` 处填写大约 4 行汇编代码，设置从 EL3 跳转到 EL1 所需的 `elr_el3` 和 `spsr_el3` 寄存器值。

`eret`指令可用于从高异常级别跳到更低的异常级别，在执行它之前我们需要设置 设置 `elr_elx`（异常链接寄存器）和 `spsr_elx`（保存的程序状态寄存器），分别控制`eret`执行后的指令地址（PC）和程序状态（包括异常返回后的异常级别）。

`elr_el3` 的正确设置应使得控制流在 `eret` 后从 `arm64_elX_to_el1` 返回到 `_start` 继续执行初始化。`spsr_el3` 的正确设置应正确屏蔽 DAIF 四类中断，并且将 SP 正确设置为 `EL1h`. 在设置好这两个系统寄存器后，不需要立即 `eret`.

我们需要继续向下阅读源码。我们有`.Lin_el2`函数用于在EL2异常级进行某些进程操作，可以忽略。在`.Lno_gic_sr`和`.Ltarget`中，我们有

```
.Lno_gic_sr:
// Set EL1 to 64bit.
    mov x9, HCR_EL2_RW
    msr hcr_el2, x9

    // Set the return address and exception level.
    adr x9, .Ltarget
    msr elr_el2, x9
    mov x9, SPSR_ELX_DAIF | SPSR_ELX_EL1H
    msr spsr_el2, x9

    isb
    eret

.Ltarget:
    ret
```

可以看到，

1. 首先需要将返回的函数地址加载到x9当中，其次再将返回的地址载入到`elr_el3`中，因为这是在降低异常级时所需要读取和返回的函数地址，也就是`_start`。
2. 其次，我们还需要保存程序状态以屏蔽中断，这个是通过保存`SPSR_ELX_DAIF | SPSR_ELX_EL1H`来进行的，在屏蔽DAIF四类中断信号的情况下载入EL1H，因此采用异或的方式。
3. `SPSR_ELX_DAIF`中存储了DAIF的bit信息，再通过异或的方式就可以“过滤”掉`SPSR_ELX_EL1H`中存在的中断信号信息。

添加代码：

```
// Set the return address and exception level.
/* LAB 1 TODO 2 BEGIN */
/* BLANK BEGIN */
adr x9, .Ltarget
msr elr_el3, x9
mov x9, SPSR_ELX_DAIF | SPSR_ELX_EL1H
msr spsr_el3, x9
/* BLANK END */
/* LAB 1 TODO 2 END */
```

添加代码并且rebuild以后，我们成功返回了`_start`。

4. 跳转到C语言代码

降低异常级别到 EL1 后，我们准备从汇编跳转到 C 代码，在此之前我们先设置栈（SP）。因此，`_start` 函数在执行 `arm64_elX_to_el1` 后，即设置内核启动阶段的栈，并跳转到第一个 C 函数 `init_c`。

> 思考题 4：说明为什么要在进入 C 函数之前设置启动栈。如果不设置，会发生什么？

我们有

```
/* Prepare stack pointer and jump to C. */
    adr     x0, boot_cpu_stack
    add     x0, x0, #INIT_STACK_SIZE
    mov     sp, x0
```

其中，INIT_STACK_SIZE位于`consts.h`的定义中。`#define INIT_STACK_SIZE 0x1000` 分配的栈大小为1000. 如果不设置启动栈，在发生异常或者需要传递参数的情况下，机器将无法获得参数/从错误中恢复，造成无法保存上下文信息和传递参数的问题。

进入 `init_c` 函数后，第一件事首先通过 `clear_bss` 函数清零了 `.bss` 段，该段用于存储未初始化的全局变量和静态变量（具体请参考附录）。

> 思考题 5：在实验 1 中，其实不调用 `clear_bss` 也不影响内核的执行，请思考不清理 `.bss` 段在之后的何种情况下会导致内核无法工作。

我们需要观察`clear_bss`函数。其位于init_c.c中。我们有：

```
static void clear_bss(void)
{
    u64 bss_start_addr;
    u64 bss_end_addr;
    u64 i;

    bss_start_addr = (u64)&_bss_start;
    bss_end_addr = (u64)&_bss_end;
```

```
    for (i = bss_start_addr; i < bss_end_addr; ++i)
        *(char *)i = 0;

    clear_bss_flag = 0;
}
```

在这其中，我们初始化了起始地址和终止地址，并将其指向的内存内容都更改为0.因此以下情况可能会导致未经过clear_bss_flag的内核出错：

1. 当初始化依赖于起始地址和终止地址内的内容时，未清零的情况可能会导致不可预测的错误；
2. 存储了某些错误的值导致系统读取到了错误的内容，引发崩溃。
3. 不兼容问题，某些未清零的值可能会导致跨平台出错。

5. 初始化串口输出

到目前为止我们仍然只能通过 GDB 追踪内核的执行过程，而无法看到任何输出，这无疑是对我们写操作系统的积极性的一种打击。因此在 `init_c` 中，我们启用树莓派的 UART 串口，从而能够输出字符。

在 `kernel/arch/aarch64/boot/raspi3/peripherals/uart.c` 已经给出了 `early_uart_init` 和 `early_uart_send` 函数，分别用于初始化 UART 和发送单个字符（也就是输出字符）。

> 练习题 6：在 `kernel/arch/aarch64/boot/raspi3/peripherals/uart.c` 中 `LAB 1 TODO 3` 处实现通过 UART 输出字符串的逻辑。

我们只需要先初始化后再一个个传入字符即可。

```
void uart_send_string(char *str)
{
    /* LAB 1 TODO 3 BEGIN */
    /* BLANK BEGIN */
    early_uart_init();
    unsigned int index=0;
    while(str[index]!='\0')
    {
        early_uart_send(str[index++]);
    }
    /* BLANK END */
    /* LAB 1 TODO 3 END */
}
```

我们可以看到

成功输出了字符串。

6. 启用 MMU

在内核的启动阶段，还需要配置启动页表（`init_kernel_pt` 函数），并启用 MMU（`el1_mmu_activate` 函数），使可以通过虚拟地址访问内存，从而为之后跳转到高地址作准备（内核通常运行在虚拟地址空间 `0xffffff0000000000` 之后的高地址）。

关于配置启动页表的内容由于包含关于页表的细节，将在本实验下一部分实现，目前直接启用 MMU。

在 EL1 异常级别启用 MMU 是通过配置系统寄存器 `sctlr_el1` 实现的（Arm Architecture Reference Manual D13.2.118）。具体需要配置的字段主要包括：

- 是否启用 MMU（`M` 字段）
- 是否启用对齐检查（`A SA0 SA nAA` 字段）
- 是否启用指令和数据缓存（`C I` 字段）

> 练习题 7：在 `kernel/arch/aarch64/boot/raspi3/init/tools.S` 中 `LAB 1 TODO 4` 处填写一行汇编代码，以启用 MMU。

这里我们从上面的三条规则可以看到需要检查是否启用字段。我们再根据源码的阅读可以发现：

```
mrs     x8, sctlr_el1
/* Enable MMU */
/* LAB 1 TODO 4 BEGIN */
/* BLANK BEGIN */
/* BLANK END */
/* LAB 1 TODO 4 END */
/* Disable alignment checking */
bic     x8, x8, #SCTLR_EL1_A
bic     x8, x8, #SCTLR_EL1_SA0
bic     x8, x8, #SCTLR_EL1_SA
orr     x8, x8, #SCTLR_EL1_nAA
/* Data accesses Cacheable */
orr     x8, x8, #SCTLR_EL1_C
/* Instruction access Cacheable */
orr     x8, x8, #SCTLR_EL1_I
msr     sctlr_el1, x8
```

进行需要的操作时bic或orr，接下来我们再了解两个函数。

1. `bic`

BIC

Bit Clear.

Syntax

`BIC{S}{cond} Rd, Rn, Operand2`

where:

S
is an optional suffix. If `S` is specified, the condition flags are updated on the result of the operation.

cond
is an optional condition code.

Rd
is the destination register.

Rn
is the register holding the first operand.

Operand2
is a flexible second operand.

Operation

The `BIC` (Bit Clear) instruction performs an AND operation on the bits in *Rn* with the complements of the corresponding bits in the value of *Operand2*.

In certain circumstances, the assembler can substitute `BIC` for `AND`, or `AND` for `BIC`. Be aware of this when reading disassembly listings.

2. `orr`

ORR

Logical OR.

Syntax

`ORR{S}{cond} Rd, Rn, Operand2`

where:

S
is an optional suffix. If `S` is specified, the condition flags are updated on the result of the operation.

cond
is an optional condition code.

Rd
is the destination register.

Rn
is the register holding the first operand.

Operand2
is a flexible second operand.

Operation

The `ORR` instruction performs bitwise OR operations on the values in *Rn* and *Operand2*.

In certain circumstances, the assembler can substitute `ORN` for `ORR`, or `ORR` for `ORN`. Be aware of this when reading disassembly listings.

可以看到bic用于清零，也就是禁用某些字段。而orr用于启用字段。因此我们需要启用mmu，则需要

```
orr x8, x8, #SCTLR_EL1_M
```

由于没有配置启动页表，在启用 MMU 后，内核会立即发生地址翻译错误（Translation Fault），进而尝试跳转到异常处理函数（Exception Handler），该异常处理函数的地址为异常向量表基地址（`vbar_el1` 寄存器）加上 `0x200`。此时我们没有设置异常向量表（`vbar_el1` 寄存器的值是0），因此执行流会来到 `0x200` 地址，此处的代码为非法指令，会再次触发异常并跳转到 `0x200` 地址。使用 GDB 调试，在 GDB 中输入 `continue` 后，待内核输出停止后，按 Ctrl-C，可以观察到内核在 `0x200` 处无限循环。

如图所示。

7. 内核启动页表

AArch64地址翻译

在 AArch64 架构的 EL1 异常级别存在两个页表基址寄存器：`ttbr0_el1`[^ttbr0_el1] 和 `ttbr1_el1`[^ttbr1_el1]，分别用作虚拟地址空间低地址和高地址的翻译。那么什么地址范围称为“低地址”，什么地址范围称为“高地址”呢？这由 `tcr_el1` 翻译控制寄存器[^tcr_el1]控制，该寄存器提供了丰富的可配置性，可决定 64 位虚拟地址的高多少位为 `0` 时，使用 `ttbr0_el1` 指向的页表进行翻译，高多少位为 `1` 时，使用 `ttbr1_el1` 指向的页表进行翻译[^ttbr-sel]。一般情况下，我们会将 `tcr_el1` 配置为高低地址各有 48 位的地址范围，即，`0x0000_0000_0000_0000`～`0x0000_ffff_ffff_ffff` 为低地址，`0xffff_0000_0000_0000`～`0xffff_ffff_ffff_ffff` 为高地址。

[^ttbr0_el1]: Arm Architecture Reference Manual, D13.2.144 [^ttbr1_el1]: Arm Architecture Reference Manual, D13.2.147 [^tcr_el1]: Arm Architecture Reference Manual, D13.2.131 [^ttbr-sel]: Arm Architecture Reference Manual, D5.2 Figure D5-13

了解了如何决定使用 `ttbr0_el1` 还是 `ttbr1_el1` 指向的页表，再来看地址翻译过程如何进行。通常我们会将系统配置为使用 4KB 翻译粒度、4 级页表（L0 到 L3），同时在 L1 和 L2 页表中分别允许映射 2MB 和 1GB 大页（或称为块）[^huge-page]，因此地址翻译的过程如下图所示：[^huge-page]: 操作系统：原理与实现

其中，当映射为 1GB 块或 2MB 块时，图中 L2、L3 索引或 L3 索引的位置和低 12 位共同组成块内偏移。

每一级的每一个页表占用一个 4KB 物理页，称为页表页（Page Table Page），其中有 512 个条目，每个条目占 64 位。AArch64 中，页表条目称为描述符（descriptor）[^descriptor]，最低位（bit[0]）为 1 时，描述符有效，否则无效。有效描述符有两种类型，一种指向下一级页表（称为表描述符），另一种指向物理块（大页）或物理页（称为块描述符或页描述符）。在上面所说的地址翻译配置下，描述符结构如下（“Output address”在这里即物理地址，一些地方称为物理页帧号（Page Frame Number，PFN））：

L0、L1、L2 页表描述符

L3 页表描述符

[^descriptor]: Arm Architecture Reference Manual, D5.3

> 思考题 8：请思考多级页表相比单级页表带来的优势和劣势（如果有的话），并计算在 AArch64 页表中分别以 4KB 粒度和 2MB 粒度映射 0～4GB 地址范围所需的物理内存大小（或页表页数量）。

我们有：

1. 4KB粒度 (1) 0~4GB虚拟地址范围需要$2^2\cdot2^{30}B$,也就是32位。 (2) 页表大小为4KB，则具有页内偏移:$2^2\cdot2^{10}B$,也就是12位。 (3) 则需要用20位的物理页号。根据aarch64系统结构，每级列表最多有$2^9=512$个页表项。这样就需要2^{2+9+9}的三级页表结构。 (4) 这样我们就有：1个0级页表（1个页表项），1个1级页表（4个页表项），4个2级页表（满表），$4\cdot 2^9$个3级页表（满表）。($\mathbf{0+0+2+9}+9$)这样我们就有$4KB+4KB+16KB+8MB$的内存要求。
2. 2MB粒度 (1) 0~4GB虚拟地址范围需要$2^2\cdot2^{30}B$,也就是32位。 (2) 页表大小为$2MB$,也就是$2\cdot 2^{20}B$,也就是21位。 (3) 这样就需要11位的物理页号。这样就需要2^{2+9}的二级页表。 (4) 这样的页表空间则为：1个0级页表（1个页表项），1个1级页表（1个页表项），4个二级页表（满表），总共24KB的内存要求。

页表描述符中除了包含下一级页表或物理页/块的地址，还包含对内存访问进行控制的属性（attribute）。这里涉及到太多细节，本文档限于篇幅只介绍最常用的几个页/块描述符中的属性字段：

字段	位	描述
UXN	bit[54]	置为 `1` 表示非特权态无法执行（Unprivileged eXecute-Never）
PXN	bit[53]	置为 `1` 表示特权态无法执行（Privileged eXecute-Never）
nG	bit[11]	置为 `1` 表示该描述符在 TLB 中的缓存只对当前 ASID 有效
AF	bit[10]	置为 `1` 表示该页/块在上一次 AF 置 `0` 后被访问过
SH	bits[9:8]	表示可共享属性[^mem-attr]
AP	bits[7:6]	表示读写等数据访问权限[^mem-access]
AttrIndx	bits[4:2]	表示内存属性索引，间接指向 `mair_el1` 寄存器中配置的属性[^mair_el1]，用于控制将物理页映射为正常内存（normal memory）或设备内存（device memory），以及控制 cache 策略等

[^mem-attr]: Arm Architecture Reference Manual, D5.5 [^mem-access]: Arm Architecture Reference Manual, D5.4 [^mair_el1]: Arm Architecture Reference Manual, D13.2.97

配置内核启动页表

有了关于页表配置的前置知识，我们终于可以开始配置内核的启动页表了。

操作系统内核通常运行在虚拟内存的高地址（如前所述，`0xffff_0000_0000_0000` 之后的虚拟地址）。通过对内核页表的配置，将虚拟内存高地址映射到内核实际所在的物理内存，在执行内核代码时，PC 寄存器的值是高地址，对全局变量、栈等的访问都使用高地址。在内核运行时，除了需要访问内核代码和数据等，往往还需要能够对任意物理内存和外设内存（MMIO）进行读写，这种读写同样通过高地址进行。

因此，在内核启动时，首先需要对内核自身、其余可用物理内存和外设内存进行虚拟地址映射，最简单的映射方式是一对一的映射，即将虚拟地址 `0xffff_0000_0000_0000 + addr` 映射到 `addr`。需要注意的是，在 ChCore 实验中我们使用了 `0xffff_ff00_0000_0000` 作为内核虚拟地址的开始（注意开头 `f` 数量的区别），不过这不影响我们对知识点的理解。

在树莓派 3B+ 机器上，物理地址空间分布如下[^bcm2836]：

[^bcm2836]: bcm2836-peripherals.pdf & Raspberry Pi Hardware - Peripheral Addresses

物理地址范围	对应设备
`0x00000000`~`0x3f000000`	物理内存（SDRAM）
`0x3f000000`~`0x40000000`	共享外设内存
`0x40000000`~`0xffffffff`	本地（每个 CPU 核独立）外设内存

现在将目光转移到 `kernel/arch/aarch64/boot/raspi3/init/mmu.c` 文件，我们需要在 `init_kernel_pt` 为内核配置从 `0x00000000` 到 `0x80000000`（`0x40000000` 后的 1G，ChCore 只需使用这部分地址中的本地外设）的映射，其中 `0x00000000` 到 `0x3f000000` 映射为 normal memory，`0x3f000000` 到 `0x80000000`映射为 device memory，其中 `0x00000000` 到 `0x40000000` 以 2MB 块粒度映射，`0x40000000` 到 `0x80000000` 以 1GB 块粒度映射。

> 思考题 9: 请结合上述地址翻译规则，计算在练习题 10 中，你需要映射几个 L2 页表条目，几个 L1 页表条目，几个 L0 页表条目。页表页需要占用多少物理内存?

1. 我们从`0x0000_0000`到`0x3fff_ffff`的映射需要用到30位的虚拟地址。这其中，页大小是2MB，则其页偏移量为21位。那么我们就需要使用2级页表进行映射。二级页表条目则需要$11=(0+0+2)+9$位。这样我们就有1个0级页表（1个页表项），1个1级页表（4个页表项），4个二级页表（满表）。24KB.
2. 从`0x4000_0000`到`0x7fff_ffff`映射则需要32位的虚拟地址，其中，页大小是1GB，则采用单级页表。页内偏移位30位，则需要一级页表条目2位。这样我们就需要1个0级页表（1个页表项），1个一级页表（4个页表项）。 8KB.

> 练习题 10：在 `init_kernel_pt` 函数的 `LAB 1 TODO 5` 处配置内核高地址页表（`boot_ttbr1_l0`、`boot_ttbr1_l1` 和 `boot_ttbr1_l2`），以 2MB 粒度映射。

我们可以首先看一下MMU.c中的部分：

```
void init_kernel_pt(void)
{
        u64 vaddr = PHYSMEM_START;

        /* TTBR0_EL1 0-1G */
```

```
        boot_ttbr0_l0[GET_L0_INDEX(vaddr)] = ((u64)boot_ttbr0_l1) |
IS_TABLE
                                            | IS_VALID | NG;
        boot_ttbr0_l1[GET_L1_INDEX(vaddr)] = ((u64)boot_ttbr0_l2) |
IS_TABLE
                                            | IS_VALID | NG;

        /* Normal memory: PHYSMEM_START ~ PERIPHERAL_BASE */
        /* Map with 2M granularity */
        for (; vaddr < PERIPHERAL_BASE; vaddr += SIZE_2M) {
                boot_ttbr0_l2[GET_L2_INDEX(vaddr)] =
                        (vaddr) /* low mem, va = pa */
                        | UXN /* Unprivileged execute never */
                        | ACCESSED /* Set access flag */
                        | NG /* Mark as not global */
                        | INNER_SHARABLE /* Sharebility */
                        | NORMAL_MEMORY /* Normal memory */
                        | IS_VALID;
        }

        /* Peripheral memory: PERIPHERAL_BASE ~ PHYSMEM_END */
        /* Map with 2M granularity */
        for (vaddr = PERIPHERAL_BASE; vaddr < PHYSMEM_END; vaddr +=
SIZE_2M) {
                boot_ttbr0_l2[GET_L2_INDEX(vaddr)] =
                        (vaddr) /* low mem, va = pa */
                        | UXN /* Unprivileged execute never */
                        | ACCESSED /* Set access flag */
                        | NG /* Mark as not global */
                        | DEVICE_MEMORY /* Device memory */
                        | IS_VALID;
        }

        /* TTBR1_EL1 0-1G */
        /* LAB 1 TODO 5 BEGIN */
        /* Step 1: set L0 and L1 page table entry */
        /* BLANK BEGIN */
        /* BLANK END */

        /* Step 2: map PHYSMEM_START ~ PERIPHERAL_BASE with 2MB
granularity */
        /* BLANK BEGIN */
        /* BLANK END */

        /* Step 2: map PERIPHERAL_BASE ~ PHYSMEM_END with 2MB granularity
*/
        /* BLANK BEGIN */
        /* BLANK END */
        /* LAB 1 TODO 5 END */

        /*
         * Local peripherals, e.g., ARM timer, IRQs, and mailboxes
         *
         * 0x4000_0000 .. 0xFFFF_FFFF
```

```
         * 1G is enough (for Mini-UART). Map 1G page here.
         */
        vaddr = KERNEL_VADDR + PHYSMEM_END;
        boot_ttbr1_l1[GET_L1_INDEX(vaddr)] = PHYSMEM_END | UXN /*
Unprivileged
                                                                  execute
never
                                                                */
                                                | ACCESSED /* Set access flag
*/
                                                | NG /* Mark as not global */
                                                | DEVICE_MEMORY /* Device
memory */
                                                | IS_VALID;
}
```

> 提示：你只需要将 `addr`(`0x00000000` 到 `0x80000000`) 按照要求的页粒度一一映射到 `KERNEL_VADDR + addr`(`vaddr`) 上。`vaddr` 对应的物理地址是 `vaddr - KERNEL_VADDR`. Attributes 的设置请参考给出的低地址页表配置。

从提示中我们可以看到，我们只需要将下标中的vaddr映射到高地址，即`vaddr=vaddr+KERNEL_VADDR`中，然后将真实的物理地址，即`paddr=vaddr-KERNEL_VADDR`赋给映射页表即可。注意高地址需要的寄存器位ttbr1. 这样我们就有：

```
        /* TTBR1_EL1 0-1G */
        /* LAB 1 TODO 5 BEGIN */
        /* Step 1: set L0 and L1 page table entry */
        /* BLANK BEGIN */
        vaddr = PHYSMEM_START + KERNEL_VADDR;   // the started entry of
the vaddr.
        boot_ttbr1_l0[GET_L0_INDEX(vaddr)] = ((u64)boot_ttbr1_l1) |
IS_TABLE
                                                | IS_VALID | NG;
        boot_ttbr1_l1[GET_L1_INDEX(vaddr)] = ((u64)boot_ttbr1_l2) |
IS_TABLE
                                                | IS_VALID | NG;
        /* BLANK END */
        // 上面的部分我们将高地址负责的ttbr1的l0级页表和l1级页笔哦映射到高地址。
        /* Step 2: map PHYSMEM_START ~ PERIPHERAL_BASE with 2MB
granularity */
        /* BLANK BEGIN */
        for (; vaddr < PERIPHERAL_BASE + KERNEL_VADDR; vaddr += SIZE_2M) {
                boot_ttbr1_l2[GET_L2_INDEX(vaddr)] =
                        (vaddr - KERNEL_VADDR) /* high mem, va - Kva = pa
*/
                        | UXN /* Unprivileged execute never */
                        | ACCESSED /* Set access flag */
                        | NG /* Mark as not global */
                        | INNER_SHARABLE /* Sharebility */
                        | NORMAL_MEMORY /* Normal memory */
                        | IS_VALID;
```

```
        }
        /* BLANK END */
        // 修改Normal Memory映射部分，注意是高地址映射，需要修改起始虚拟地址和终止的
虚拟地址，以及映射到的真实物理地址。
        /* Step 2: map PERIPHERAL_BASE ~ PHYSMEM_END with 2MB granularity
*/
        /* BLANK BEGIN */
        for (vaddr = PERIPHERAL_BASE + KERNEL_VADDR; vaddr < PHYSMEM_END +
KERNEL_VADDR; vaddr += SIZE_2M) {
                boot_ttbr1_l2[GET_L2_INDEX(vaddr)] =
                        (vaddr - KERNEL_VADDR) /* lhigh mem, va - Kva = pa
*/
                        | UXN /* Unprivileged execute never */
                        | ACCESSED /* Set access flag */
                        | NG /* Mark as not global */
                        | DEVICE_MEMORY /* Device memory */
                        | IS_VALID;
        }
        // 修改Device Memory映射部分，注意是高地址映射，需要修改起始虚拟地址和终止的
虚拟地址，以及映射到的真实物理地址。
        /* BLANK END */
        /* LAB 1 TODO 5 END */
```

配置完成，`make build`之后进行`make qemu`我们就可以得到以下界面：

成功启动。

> 思考题 11：请思考在 `init_kernel_pt` 函数中为什么还要为低地址配置页表，并尝试验证自己的解释。

我们需要回顾一下内核启动的过程。

1. 我们逐个初始化CPU的每个核。
2. 我们需要获取CPU当前异常级别并且将异常级别下降到内核级EL1。
3. 设置启动栈准备跳转到C语言。
4. 初始化串口输出，这样我们才能显示/回显信息。
5. 配置启动页表`init_kernel_pt`并启动MMU`el1_mmu_active`.

我们看到我们在配置启动页表后就需要进入到启用MMU阶段。这样我们就需要观察一下，当缺少了低地址配置后会出现什么情况。

```
B+> 0x88138 <el1_mmu_activate+8>    bl      0x8809c <invalidate_cache_all>
    0x8813c <el1_mmu_activate+12>   tlbi    vmalle1is
    0x88140 <el1_mmu_activate+16>   isb
    0x88144 <el1_mmu_activate+20>   dsb     sy
    0x88148 <el1_mmu_activate+24>   ldr     x8, 0x881b8
    0x8814c <el1_mmu_activate+28>   msr     mair_el1, x8
    0x88150 <el1_mmu_activate+32>   ldr     x8, 0x881c0
    0x88154 <el1_mmu_activate+36>   msr     tcr_el1, x8
    0x88158 <el1_mmu_activate+40>   isb
    0x8815c <el1_mmu_activate+44>   adrp    x8, 0x86000 <boot_ttbr0_l0>
    0x88160 <el1_mmu_activate+48>   msr     ttbr0_el1, x8
    0x88164 <el1_mmu_activate+52>   adrp    x8, 0x87000 <boot_ttbr1_l0>
    0x88168 <el1_mmu_activate+56>   msr     ttbr1_el1, x8
    0x8816c <el1_mmu_activate+60>   isb
remote Thread 1.1 In: el1_mmu_activate
(gdb) break el1_mmu_activate
Breakpoint 1 at 0x88138
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, 0x0000000000088138 in el1_mmu_activate ()
(gdb) disassemble
(gdb)
```

可以发现，我们在0x8815c和0x88160处需要将ttbr0_el1进行配置，也就是低地址页表基址寄存器。此时启用的mmu在进行页表配置时会出现segmentation fault,即寻址错误。这是因为，目前运行的函数运行在低地址空间，在返回时，需要获得返回的虚拟地址的翻译。未配置低地址空间会导致对`el1_mmu_active`函数保存的返回出现错误。

错误的情况

```
问题    输出    调试控制台    终端    端口 2    GITLENS
                                                0xaaaae193663b ???
                                                                0xaaaae1ae1517 ???
                                                                                0xaaaae1f03d23 ???
                                                                                                0xaaa
ae1f040d7 ???
            0xaaaae197cf1b ???
                            0xaaaae197e973 ???
                                            0xaaaae16f5807 ???
                                                            0xffffa32573fb __libc_start_call_mai
n
        ../sysdeps/nptl/libc_start_call_main.h:58
                                            0xffffa32574cb __libc_start_main_impl
                                                                            ../csu/libc-sta
rt.c:392
        0xaaaae16fa4ef ???
                        0xffffffffffffffff ???
                                        ____________________
                                                            A fatal error internal to GDB has
been detected, further
                    debugging is not possible.  GDB will now terminate.

                                                            This is a bug, please report i
t.  For instructions, see:
                        <https://www.gnu.org/software/gdb/bugs/>.

                                                            make: *** [Makefile:41: gdb] Segment
ation fault (core dumped)
                    (base) parallels@ubuntu-linux-22-04-02-desktop:~/Desktop/labs/OS-Course-Lab$
```

完成 `init_kernel_pt` 函数后，ChCore 内核便可以在 `el1_mmu_activate` 中将 `boot_ttbr1_l0` 等物理地址写入实际寄存器(如 `ttbr1_el1`)，随后启用 MMU 后继续执行，并通过 `start_kernel` 跳转到高地址，进而跳转到内核的 `main` 函数（位于 `kernel/arch/aarch64/main.c`, 尚未发布，以 binary 提供）。

> 思考题 12：在一开始我们暂停了三个其他核心的执行，根据现有代码简要说明它们什么时候会恢复执行。思考为什么一开始只让 0 号核心执行初始化流程?
>
> 提示： `secondary_boot_flag` 将在 main 函数执行完时钟，调度器，锁的初始化后被设置。

为了保证多个核的运行，在操作系统执行过程中需要加载好时钟，调度器（调度策略），以及多线程锁的相关程序。这些程序都位于低地址中，而内核需要在高地址以内核态el1运行。因此我们需要先初始化0号核（其中一个内核），建立好其他策略程序所在的物理内存到高地址内存的映射，在确认相关策略和程序加载完毕后，才能逐个启动新的内核，防止操作系统的内核混乱。

```
0xffffff000008e1ac <main+140>   add     x2, x2, x0
0xffffff000008e1b0 <main+144>   mov     x1, #0xffffffff00000000          // #-4294967296
0xffffff000008e1b4 <main+148>   sub     x0, x5, x0
0xffffff000008e1b8 <main+152>   bl      0xffffff0000091300 <map_range_in_pgtbl_kernel>
0xffffff000008e1bc <main+156>   bl      0xffffff000008e6e0 <arch_interrupt_init>
0xffffff000008e1c0 <main+160>   bl      0xffffff0000095998 <timer_init>
0xffffff000008e1c4 <main+164>   adrp    x0, 0xffffff000008e000 <start_kernel>
0xffffff000008e1c8 <main+168>   ldr     x0, [x0, #728]
0xffffff000008e1cc <main+172>   bl      0xffffff0000096798 <printk>
0xffffff000008e1d0 <main+176>   bl      0xffffff000008fdc0 <pmu_init>
0xffffff000008e1d4 <main+180>   adrp    x0, 0xffffff000008e000 <start_kernel>
0xffffff000008e1d8 <main+184>   ldr     x0, [x0, #736]
0xffffff000008e1dc <main+188>   bl      0xffffff0000096798 <printk>
0xffffff000008e1e0 <main+192>   adrp    x0, 0xffffff000008e000 <start_kernel>
```

make grade!!!

```
-- Build files have been written to: /home/parallels/Desktop/labs/OS-Course-Lab/Lab1/bui
ld
Scanning dependencies of target clean-all
Built target clean-all
Succeeded to clean all targets
Removing config file...
Succeeded to distclean
make[1]: Leaving directory '/home/parallels/Desktop/labs/OS-Course-Lab/Lab1'
Grading lab 1 ...(may take 10 seconds)
=============================================
Jump to kernel main: 100
Score: 100/100
=============================================
```

😌