OS Lab 1: Booting

本学期，我们将实现一个简单的操作系统内核。第一个实验里，我们的目标是引导系统进入`main`函数，输出`Hello world!`，以及了解这套流程是怎么完成的。

实验环境

本学期实验使用AArch64架构，推荐在服务器上进行。服务器上已经为大家提供了独立的容器，其中安装了所有需要的工具，无需自行配置环境。

连接服务器可以使用`ssh root@10.176.34.211 -p 你的端口号`指令，Windows和Linux下都可运行。服务器使用密钥登录，建议大家实验课自行携带电脑，如需使用机房电脑请务必带上你的私钥。此外，推荐大家自行配置VS Code的Remote插件。

你的容器中，有两个特殊的目录（挂载点）。`/share`目录只读，供所有同学共享，我们会在其中放置一些共享文件。`~/data`目录可读写，我们为你挂载了一块4.4T的硬盘，建议你在该目录下存储实验所用到的文件。

CPU、内存和硬盘资源是所有同学共享的，我们没有为大家设置使用配额，请注意合理使用资源。

在服务器上开发，可以参考下面指令初始化代码仓库

```
cd ~/data
# clone代码仓库
git clone git://github.com/FDUCSLG/OS-2022Fall-Fudan
# 切换到本实验分支
git checkout lab1
# 新建一个分支，用于开发
git checkout -b lab1-dev
# 运行
cd OS-2022Fall-Fudan
mkdir -p build
cd build
cmake ..
make qemu
```

之后每次运行只要在`build`目录下`cmake .. && make qemu`。

我们将使用`qemu`来运行内核。**退出方法为：`Ctrl+A`，松开后按`X`。**

每次发布新的lab后，可以参考下面指令更新代码仓库

```
# 拉取远端仓库
git fetch --all
# 提交你的更改
git add .
git commit -m "your commit message"
# 切换到新lab的分支
git checkout lab2
# 新建一个分支，用于开发
git checkout -b lab2-dev
# 引入你在上个lab的更改
git merge lab1-dev
```

如果合并发生冲突，请参考错误信息自行解决。

你也可以fork一份自己的远端代码仓库。

AArch64

> AArch64等价于ARMv8的64位指令集。

本学期的教学OS将运行在AArch64架构下。AArch64属于RISC（精简指令集架构），大体上上与我们已学过的MIPS和RISCV指令集有些相似，但指令数量和复杂性远超过MIPS和RISCV。

本学期的绝大多数代码都使用C语言完成，但仍有部分代码需要使用汇编语言编写，大家需要掌握AArch64汇编的下面内容

- 各通用寄存器及其别名、函数调用约定、栈结构
- 算术、访存、跳转等基本汇编指令

指令手册：Arm Architecture Reference Manual Armv8, for A-profile architecture

我们在上课时也会对用到的一些汇编指令进行讲解。

Booting

> 操作系统如何取得先机?

我们在上学期的计组课程中已经学过，CPU上电时处于可以运行任何指令、访问任何内存的不受限制状态，并将PC设定为固定的值。在树莓派中，初始PC为0x80000。加载内核时，固件会将我们的内核代码复制到0x80000处，这样操作系统就占据了先机。

我们使用了链接器脚本控制内核入口（`_start`函数）刚好被链接在生成的内核文件的开头，所用的脚本可参见`linker.ld`。介绍一些简单的语法：

- `.` 被称为 location counter，代表了当前位置代码的实际上会运行在内存中的哪个地址。如 `. = 0xFFFF000000080000;` 代表我们告诉链接器内核的第一条指令运行在 0xFFFF000000080000。但事实上，我们一开始是运行在物理地址 0x80000（为什么仍能正常运行？），开启页表后，我们才会跳到高地址（虚拟地址），大部分代码会在那里完成。

- text 段中的第一行为 `KEEP(*(.text.boot))` 是指把 `start.S` 中我们自定义的 `.text.boot` 段放在 text 段的开头。`KEEP` 则是告诉链接器始终保留此段，否则编译器可能会将未用到的段优化掉。

- `PROVIDE(etext = .)` 声明了一个符号 `etext`（但并不会为其分配具体的内存空间，即在 C 代码中我们不能对其进行赋值），并将其地址设为当前的 location counter。这使得我们可以在汇编或 C 代码中得到内核各段（text/data/bss）的地址范围，我们在初始化 BSS 的时候需要用到这些符号。

链接器脚本参考：Linker Scripts。我们不要求大家自行编写链接器脚本。

AArch64中有EL3、EL2、EL1、EL0四个特权级，我们的lab只使用其中的EL1、EL0特权级。`_start` 主要完成一些架构相关的初始化工作：唤醒所有CPU核，在EL3、EL2特权级下进行简单的配置，进入EL1特权级，使用 `kernel_pt` 开启虚拟地址，设置内核栈，最后跳转到 `main` 函数。

`kernel_pt` 是 `aarch64/kernel_pt.c` 中预定义的页表，它基本上就是直接映射，映射关系为虚拟地址等于物理地址加上 `0xffff0000_00000000`。`kernel_pt` 还标记了一些用于MMIO的地址为 `PTE_DEVICE`，提示cache需要对该地址特殊处理。关于页表的细节我们会在后续lab中介绍。

> 真实系统的启动更为复杂。大家应该听说过BIOS，这是固件制造商内置的一段引导程序，会初始化设备，并通过ACPI等确定的格式将硬件信息传递给操作系统。这样的设计使得同一份操作系统内核可以在不同固件上使用，对于商用系统十分便利。
>
> 在我们的实验中，只使用了一个简单的bootloader，可参阅 `boot` 目录下的代码。该bootloader仅对设备做了基础的初始化工作，没有构建ACPI表，也没有提供SBI。因此，我们的系统仍需通过硬编码MMIO地址的方式访问设备，生成的内核镜像仅能在rpi-3上运行。

.init section

在内核初始化时，需要调用很多模块的初始化函数。借助链接器脚本，我们可以将需要在初始化时调用的函数指针都放置到 `.init` section中，内核初始化时遍历调用 `.init` section中放置的所有函数指针，即可完成所有模块的初始化。

> 这个设计不适用于需要精确控制初始化顺序的情况。

putchar

与之前在计组实验中做的类似，我们使用MMIO方式访问UART设备，向固件设定的地址写一字节数据，即可输出一个字符。

为避免该操作被编译器优化影响，我们需要为地址加上`volatile`标记，以免编译器将其优化掉，并添加编译器屏障，以免编译器将其乱序处理，可参考`aarch64/intrinsic.h`：`device_put_u32()`。

> 之前提及的`PTE_DEVICE`防止了CPU cache和乱序执行的影响。

作业与提交

> `aarch64/intrinsic.h`提供了一些架构相关的有用函数。

在`main.c`的`main`函数中添加代码，实现下面功能：

- 仅CPU0进入初始化流程，其余CPU直接退出。（我们没有实现电源管理功能，CPU退出可以参考提供的`arch_stop_cpu()`函数）
- 初始化`.bss`段，将其填充为`0`。（为什么？）
- 先后调用`do_early_init()`和`do_init()`函数。（可以结合链接器脚本自行阅读一下它们的实现）
- OS的`main`函数不能返回，请在完成前面步骤后让CPU0退出。

在`main.c`中添加函数1，该函数将`hello[]`填充为`Hello world!`。通过宏`early_init_func`将函数1的指针加入`.init.early` section。

在`main.c`中添加函数2，该函数调用`uart_put_char`输出`hello[]`的内容。通过宏`init_func`将函数2的指针加入`.init` section。

> 我们后续会直接为大家提供用法类似于`printf`的`printk`函数，无需大家自行对`uart_put_char`进行进一步封装。

如果一切顺利，你将看到`Hello world!`

提交：将实验报告提交到 elearning 上，格式为`学号-lab1.pdf`。本次实验中，报告不计分。