国科大操作系统研讨课任务书

RISC-V 版本

版本 2025 Fall

目录

第一章 引导、镜像文件和 ELF 文件 **1**

1 引言 1

2 实验要点解读 1

3 操作系统的引导 2

3.1 什么是引导 2

3.2 RISC V 开发板上的引导过程 2

3.3 跳转表 3

3.4 任务 1：第一个引导块的制作 4

4 镜像文件 7

4.1 镜像文件组成 7

4.2 ELF 文件 9

4.3 任务 2：加载和初始化内存 11

4.4 任务 3：加载并选择启动多个用户程序之一 13

4.5 任务 4：镜像文件的紧密排列 15

4.6 任务 5：列出用户程序和批处理 16

Project 1
引导、镜像文件和 ELF 文件

1 引言

我们的课程是要同学们自己从头写一个操作系统，一个包含了操作系统主要功能的小小的操作系统，“麻雀虽小，五脏俱全”。

开发操作系统可以说是非常考验一个人计算机综合素养的工作，除了要对操作系统、组成原理、数据结构的相关知识有一定的了解，还需要在编程方面掌握 C 语言、汇编语言，在工程方面掌握 Makefile、ld 等编译工具链的使用，了解相关的硬件知识；在后期，还会涉及到网络、存储的内容，可以说是需要 “上知天文，下知地理”。也正因为这样的特点，我们在正式课开始之前加入了预备课的介绍，这些内容都是后续课程中一定会用到的，请大家一定要掌握里面涉及到的知识和工具，至少确保完成预备课的作业。

操作系统的复杂性也正体现了其把握计算机系统全局的中枢软件的本质。希望同学们学完这门课之后，能够体会这种对各种系统资源的管理和调度的思想，不仅能说清楚自己的小操作系统中的每一行代码的功能，也能讲出设计上的思考过程。

在实验一中，我们将从操作系统的引导（boot）开始，掌握和实现操作系统的启动过程，并在实现的过程中，熟悉 Linux 下相关工具的使用、C 语言和汇编语言的编写、镜像文件的制作等内容。俗话说得好，“万事开头难”，希望大家能够认真完成实验！接下来，我们将从理论到实验，迈出开发属于我们自己的操作系统的第一步。

2 实验要点解读

Project 1 的要点是：

1. 掌握操作系统启动的完整过程，体会软件和硬件、内核和应用的分工与约定

2. 掌握 ELF 文件的结构和功能，以及 ELF 文件的装载

3. 设计加载映像的索引结构

简单点说，就是要学会在裸金属机器 (bare-metal) 上面跑程序。远古时期的程序其实都是直接在裸金属机器上跑的，没有什么操作系统。后来，大家觉得有些功能每次都自己写太麻烦了，就把他们抽取成了固定的模块，用什么功能就调用什么模块就好了。操作系统可以被认为是跑在裸金属机器上面的一段程序，负责为其他的用户程序提供一些通用的服务。所以，实验一的最重要的两个需要搞懂的知识点：

1. 程序是从哪里，如何开始运行的？

2. 程序是如何通过调用终端服务实现简单的输入输出功能？

3 操作系统的引导

3.1 什么是引导

我们需要实现的操作系统实际上是一个特殊的程序，用来控制其他用户程序在计算机上的执行。我们在组成原理课上学过，程序是随着 `PC` 寄存器的指示逐步执行下去的，既然操作系统也是一个程序，那么我们就需要让它从运行起来，这就是引导程序（Boot Loader）需要做的事情。引导程序主要的任务就是将操作系统代码从外部存储设备（本实验中为 SD 卡），搬运并展开到内存中，然后将 `PC` 跳转至此处执行，整个过程就称为“引导”（Booting）。

所以，这一节讲的是主要是上面提到的第一个问题：**程序是从哪里，如何开始运行的**。最简单的一个答案是：CPU 上电以后，`PC`（Program Counter）寄存器会被设置到一个固定的地址上。CPU 会从这个地址读取指令并开始执行。

当然喽，细心的你估计很快就想到了一个问题：我们怎么才能把程序放在这个地址上？答案是把这个地址映射到 ROM 上。ROM 和内存（RAM）很像，只不过是只读的，内容一旦烧好了就没法改（但是现在的技术也可以修改一些 ROM，这是组成原理课程所学的内容）。所以，ROM 里面的程序一般是厂家提前烧好的。CPU 一启动，先执行 ROM 里面的程序。在我们常用的 PC 上面这个东西叫做 BIOS（Basic Input-Output System），而其他系统上有时又称为固件（Firmware）。

通常 ROM 上的程序会做必要的初始化工作，然后读取磁盘（或者其他指定设备）头 512 字节的数据到内存的指定地址，最后跳转到该地址开始执行。这 512 字节的数据就是操作系统的 bootloader，它负责把操作系统的主体部分载入到内存，然后将控制权交由操作系统。

3.2 RISC V 开发板上的引导过程

上面介绍的这一过程是常识性的 boot 过程。针对我们手中的开发板，我们需要更详细的了解其中的启动过程。关于我们使用的 PYNQ 板卡的信息在 Project 0 中已经有所介绍；在没有拿到板卡之前，大家在 QEMU 上完成 Project，其模拟了板卡上的启动流程。QEMU 的相关信息已经在预备课上跟大家介绍过。

图P1-1中，蓝色的框表示的是 PYNQ 上的 ARM 硬核上完成的动作，棕色的框表示的是 RISC-V 核加电后进行的动作。而黄色的部分是我们的实验需要完成的。前面的流程是硬件或者 BIOS 自动完成的动作。从 `bootblock.S` 的代码被自动加载后的部分是我们需要自己完成的部分。可以认为，从这里开始，我们自己编写的操作系统开始启动。

对于我们的实验来说，引导分为三个过程：

1. **BIOS 阶段:** 在 CPU 上电后，`PC` 会自动跳转到一个位置开始执行。这个位置上的代码主要的任务就是将存储设备上的第一个扇区（一个扇区的大小为 512B）的

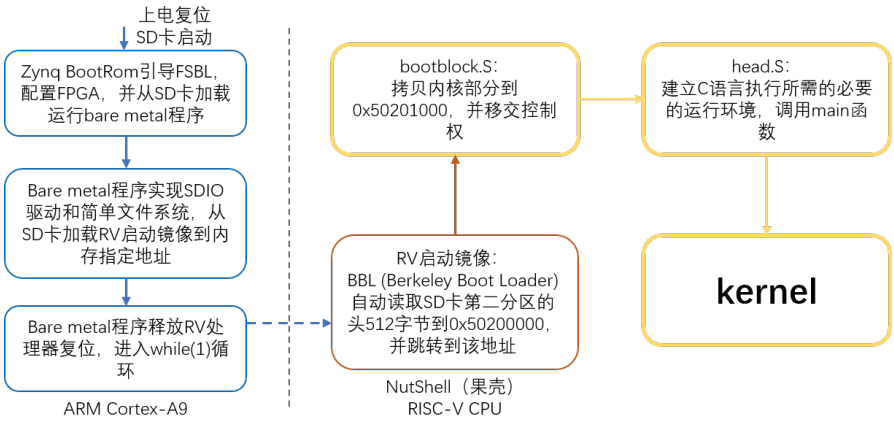

图 P1-1: 实验整体启动流程

内容拷贝到一个固定的位置（在我们的开发板中，这个位置是 `0x50200000`），这 512B 的数据就是我们的 Boot Loader。拷贝完成后，`PC` 跳转到 Boot Loader 代码的开头部分，至此，控制权被移交给 Boot Loader。

2. **Boot Loader 阶段:** 由于 Boot Loader 的代码只有 512 字节，因此只完成 1 个重要的工作：将操作系统代码搬运并展开到内存。Boot Loader 通过 BIOS（对应于我们这里的 BBL）提供的调用读取 SD 卡上的操作系统内核，并放置到内存的指定位置。读盘结束后，Boot Loader 将跳转到操作系统的入口代码开始执行，至此，操作系统的引导过程结束，真正的操作系统已经运行起来啦!

3. **OS 阶段:** 这个阶段运行的就是我们真正的操作系统代码了，在这个阶段的初期，我们会进行各种初始化，这部分也将在以后的实验中详细讲解。

在本实验中,我们将通过循序渐进的几个任务,逐步实现一个操作系统的 Boot Loader。

3.3 跳转表

对于操作系统来说，一个非常重要的特性就是提供给应用程序一套标准的服务，又称系统调用 (syscall)，我们后面的实验会涉及到这一点。同样的，在操作系统刚加载时，BIOS 也会给操作系统引导代码提供一套服务，如最基本的显示和磁盘读写操作。一般来说，BIOS 的服务只提供给操作系统，应用程序不可见，因此 BIOS 服务的接口设计是多种多样的。在我们的实验课中，我们采用了相对简单的跳转表的接口。

跳转表，顾名思义，是一个用来跳转的表。我们需要实现的跳转表，是一个由内核进行初始化、记录硬件提供给操作系统的 ABI（例如串口输入输出、SD 卡读写等操作）的入口地址的表格。（ABI 的介绍见 P0）在标准的 RISC-V 架构中，这些 ABI 是通过 SBI (Supervisor Binary Interface) 调用来实现的，涉及到保护态的切换，而其他架构中没有 SBI 调用。还没学过《操作系统》的同学们现在可能暂时不知道特权级的概念，不太理解 Supervisor 的意思，这会在之后的理论课中讲解。我们把这些涉及不同架构的硬件 ABI 调用放入跳转表进行封装，操作系统使用跳转表的 API (Application Programming Interface, 应用程序编程接口) 来执行原本的功能。这样，进行操作系统移植时，我们只

需要修改跳转表表项内容，而不需要直接面对各种不同的 BIOS 功能调用接口。API 是一种计算接口，它定义多个软件中介之间的交互，以及可以进行的调用等操作的种类 [1]。这里我们指的是跳转表这个软件提供给调用者的编程接口。

实现跳转表的另一个目的在于，在我们进行 Project 2 的实验并实现系统调用之前，用户程序实际上都是以内核进程的方式运行，需要用到内核提供的 API。而在本学期的实验课中，内核代码与用户/测试程序代码分开编译，用户程序需要内核填写的跳转表来逐步过渡到系统调用方式，并不能直接在编译过程中找到内核中的函数。

跳转表的初始化代码与相应的 API 已提供给大家，同学们只需了解其过程即可。

3.4 任务 1：第一个引导块的制作

实验要求

了解掌握操作系统引导块的加载过程，编写 Boot Block，调用 BIOS 中的输入输出函数，在终端成功输出`"It's [who]'s bootloader..."`。将 `who` 改为作者自己的名字。需要特别说明的是，在任务 1 和任务 2 中，大家还暂时没有实现自己的镜像制作工具的 `createimage`，因此需要使用我们提供给大家的可执行文件。当然，为了让大家能够自己完成 `createimage` 的设计，这份可执行文件只能满足任务 1 和任务 2 的需求，在任务 3 中会直接报错退出。

注意事项

1. 注意启动 QEMU 模拟器之后，还需要在命令行中输入 loadboot 命令才能进一步启动操作系统。

文件说明

实验一的初始代码只包含一些基本的目录和文件。具体的说明见表P1-1。以后，我们会在实验一的基础上逐步增加代码和功能。请同学们一定要学会 Git 使用，做好代码管理。需要大家补全的部分，我们基本上都在代码框架中有所注释。

实验步骤

注：带星号标记的步骤需要等到 PYNQ 板卡发给大家之后，才能连接 SD 卡与 PYNQ 板卡上板运行。

1. 填写 `bootblock.S` 代码,要求添加的内容为打印字符串 `"It's [who]'s bootloader..."`。
2. 运行 `make dirs` 命令创建 `build` 目录。
3. 运行 `make elf` 命令进行交叉编译，生成二进制文件。
4. 将提供的可执行文件 `createimage`，复制一份到 `build` 目录中，连续执行三个命令以生成镜像文件 `image`：

编号	文件/文件夹	说明
1	arch/riscv 文件夹	RISC-V 架构相关内容，主要为汇编代码、相关宏定义 bios/common.c：BIOS 所提供的 API 函数，包括：输入输出、读取 SD 卡，不需要修改 boot/bootblock.S：引导程序，接下来将在任务 1 中填写打印代码，在任务 2 中添加移动内核代码，在任务 3 中读取用户程序信息 crt0/crt0.S：测试程序入口代码，负责准备测试程序 C 语言执行环境，需要在任务 3 中补全 include：头文件，包含一些宏定义，不需要修改 kernel/head.S：内核入口代码，负责准备内核 C 语言执行环境，需要在任务 2 中补全
2	include 文件夹	内核使用的头文件，其中 os/task.h 需要在任务 3 中补全
3	init 文件夹	初始化相关 main.c：内核的入口，操作系统的起点，在任务 3、任务 5 中需要补充装载用户程序的逻辑
4	kernel 文件夹	内核相关文件 loader/loader.c：装载器的实现，需要在任务 3 中补全以加载用户程序
5	libs 文件夹	为内核提供的库函数 string.c：字符串操作函数库
6	tiny_libc 文件夹	为用户程序提供的超小型 libc 库 include：头文件，本次实验为用户程序提供了跳转表头文件 bios.h，不需要修改
7	tools 文件夹	工具 createimage.c：引导块工具代码，任务 3 中需要实现
8	Makefile	Makefile 文件，不需要修改
9	riscv.lds	链接器脚本文件，不需要修改
10	createimage	将 bootblock 和 kernel 制作成镜像文件的工具，无法在任务 3-5 中使用，需要同学们用自己写好的 createimage 来替代

表 P1-1: P1 文件说明

```
cd build
../createimage --extended bootblock main
cd ..
```

5. 执行 `make run` 命令，在 QEMU 上能看到屏幕输出字符串：

   `"It's [who]'s bootloader..."`

6*. 使用 `make floppy` 命令将 `image` 文件写入到 SD 卡。

7*. 将 SD 卡插入到板子上，使用 `make minicom` 命令监视串口输出，然后 restart 开发板。

8*. 板子加电后，系统启动，当屏幕可以打印出字符串 `"It's [who]'s bootloader..."` 说明实验完成。

注意事项

1. 注意 `createimage` 的执行命令里面，`extended` 前面有两个减号，不要遗漏。在文件前添加一个或两个减号，是一种为执行的命令（程序）添加参数选项的一种常用表示方法。

2. `createimage` 执行的时候可能会提示没有运行权限：`Permission denied`。这是因为把程序从外面拷贝进虚拟机的时候没有自动设置可执行权限，需要执行下面的命令让它可以执行：`sudo chmod +x createimage`，其中 `sudo` 是使后续的命令执行环境临时获取管理员权限的命令，`chmod` 是更改文件夹、文件权限的命令，`+x` 的意思是添加可执行 (execute) 权限。后续实验遇到类似的报错时都可以使用手动赋予执行权限的方式解决。

要点讲解

任务 1 中需要补全的位置都用与 `"//TODO:[p1-task1]"` 类似的样式做了标记，大家可以搜索该标记，从而能够更快定位到需要修改/补全的地方。对于后续的任务，我们也定义了类似的样式。

同时，实验中的 BIOS（即 BBL）提供了若干与底层硬件相关的 API，例如输入输出函数、读取 SD 卡函数。这些函数的具体实现较为复杂，且与操作系统本身无关，因此同学们只需要知道如何调用 BIOS API 即可。为了简化同学们的使用，BIOS 内的各函数使用统一的入口地址 `0x50150000`，即 `bios_func_entry`。同学们在汇编语言层面调用的时候，需要把函数编号放在 a7 寄存器中，函数参数依次放入 `a0`、`a1` ... 寄存器中，然后 `jal bios_func_entry` 即可，可供调用的接口编号定义在 `biosdef.h` 中。这里给大家一个小小的提示：各位同学可以参考 `common.c` 里面的各个 BIOS C 语言函数原型，看看它们是如何填写参数、调用 `call_bios` 函数的。如有对汇编代码的更多了解需求，请查询 Project 0 的 RISC-V 伪指令表。

现将本次实验中用到的函数原型说明如下：

- `void bios_putstr(char *str)` 在终端打印字符串 `str`。
- `uintptr_t bios_sd_read(unsigned mem_address, unsigned num_of_blocks, unsigned block_id)` 从 SD 卡的第 `block_id` 个扇区开始（扇区从 0 开始编号）读取 `num_of_blocks` 个扇区，放入内存 `mem_address` 处。
- `uintptr_t bios_sd_write(unsigned mem_address, unsigned num_of_blocks, unsigned block_id)` 将内存 `mem_address` 处的内容，写入到 SD 卡的第 `block_id` 个扇区开始（扇区从 0 开始编号）的 `num_of_blocks` 个扇区中。
- `void bios_putchar(int ch)` 在终端打印字符 `ch`。
- `int bios_getchar()` 读取终端输入字符 `ch`，如果未读取到任何字符则返回`-1`。

> **Note 3.1** 扩展名为`.S` 和`.s` 的汇编文件是不一样的。`.S` 的汇编文件会被预处理，可以像 C 语言一样加预处理指令，比如 `include` 一类的。但`.s` 文件不会进行预处理。

实验总结

该实验仅仅是在引导程序中让大家实现简单的打印任务，还没有涉及调用 BIOS 将操作系统代码搬运并展开到内存的部分，实际上，我们现在还没有写操作系统部分代码，也没有进行镜像文件 (image) 的制作，甚至同学们可能都不知道镜像文件是什么。

所谓 “好的开始是成功的一半”，我们已经跨出了最重要的一步，在我们的裸机器上已经可以运行起来我们的引导程序了！在接下来的实验中，大家将**编写操作系统内核代码，完善 Boot Loader 代码，制作镜像文件**。最终，一个精简而又完整的操作系统将真正地运行在我们的开发板上。

4 镜像文件

接下来，我们要做一个完整的操作系统镜像文件了。这一节最核心的要点是**把握住镜像文件的格式**，以及在了解了它的格式的基础上，**清楚地理解代码是如何正确地执行和跳转的**。

4.1 镜像文件组成

这里说的镜像文件指的是操作系统镜像文件。镜像文件是将特定的一系列文件按照一定的格式制作而成的单一的文件，用以方便用户下载和使用。我们制作的镜像文件应该包含三个部分，第一个部分是 Boot Loader，它位于我们最终制作完成的镜像开头；第二个部分是 Kernel，也就是操作系统内核部分，它放在 Boot Loader 的后面；第三个部分是若干用户程序 (App1, App2, ...)。它们在 SD 卡的位置如图P1-2所示。

> **Note 4.1** Boot Block 通常位于存储介质的开头（第一条轨道上的第一个块），用于保存用于启动系统的特殊数据，Boot Loader 即存储在 Boot Block 中的代码。

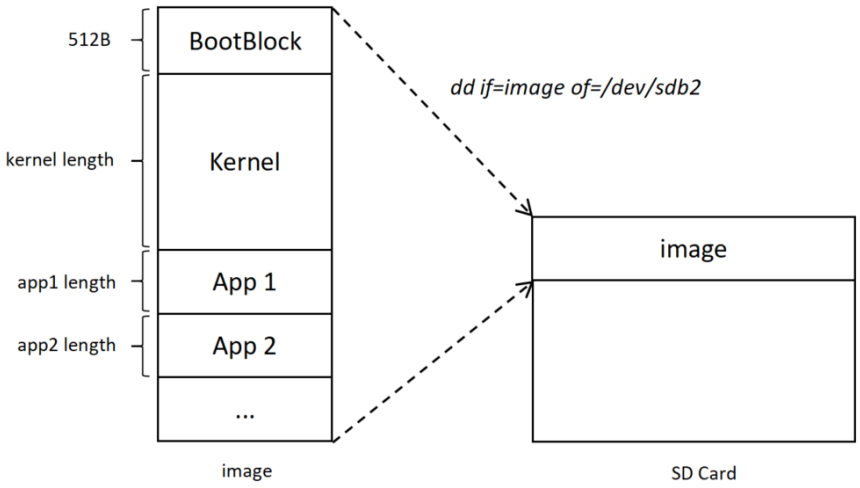

图 P1-2: Boot Loader、Kernel 的位置

关于镜像文件的制作，我们采用以下步骤完成（见图P1-3）：

1) 编译 Boot Loader

2) 编译 Kernel

3) 编译用户程序，生成 ELF 文件

4) 使用镜像制作工具 `createimage` 合并上面的三个部分，生成最终的镜像文件

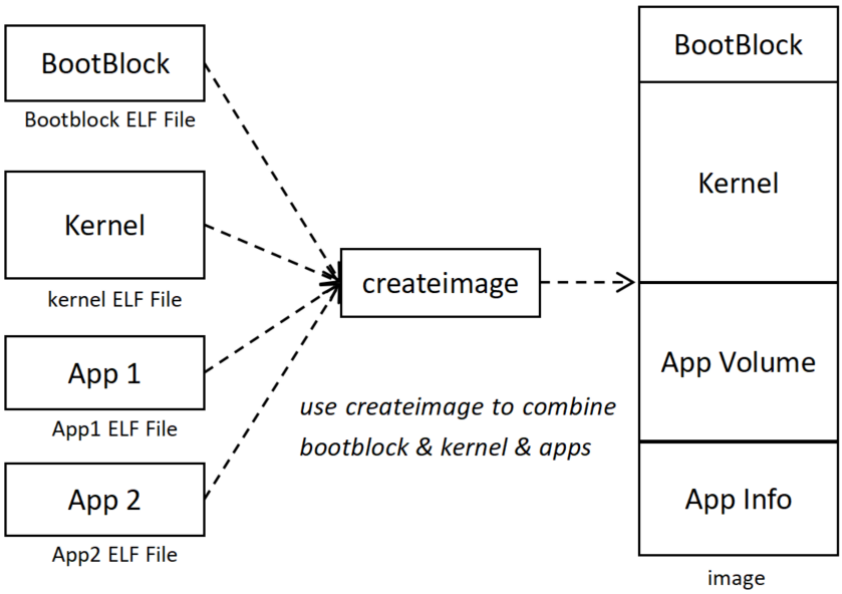

图 P1-3: 镜像生成过程

到这里大家可能会感到非常困惑，无从下手。不用担心，接下来的章节将从最基本的编译环节出发，详细阐述镜像文件的制作流程以及相关知识。经过这一部分的学习，你将学习并掌握镜像文件的制作方法，并最终成功制作出一份属于自己的镜像文件！

4.2 ELF 文件

刚才已经说过，经过链接这一步骤后，编译器会将分离的二进制代码合并成一个可执行文件。那么这个可执行文件是什么呢？它的内部结构又是如何呢？这一节我们将介绍相关的知识。事实上，我们生成的可执行文件的格式是 ELF，在计算机系统中，是一种用于二进制文件、可执行文件、目标代码、共享库和核心转储的文件格式。ELF 是 UNIX 系统实验室（USL）为应用程序二进制接口（Application Binary Interface，ABI）而开发和发布的，也是 Linux 的主要可执行文件格式。

ELF 文件由 4 部分组成，分别是 ELF 头（ELF header）、程序头表（Program header table）、节（Section）和节头表（Section header table）。实际上，一个文件中不一定包含全部内容，而且他们的位置也未必如同所示这样安排，只有 ELF 头的位置是固定的，其余各部分的位置、大小等信息由 ELF 头中的各项值来决定。整个结构如图P1-4所示。请同学们自行查询 ELF 文件相关信息，以辅助完成本次实验。

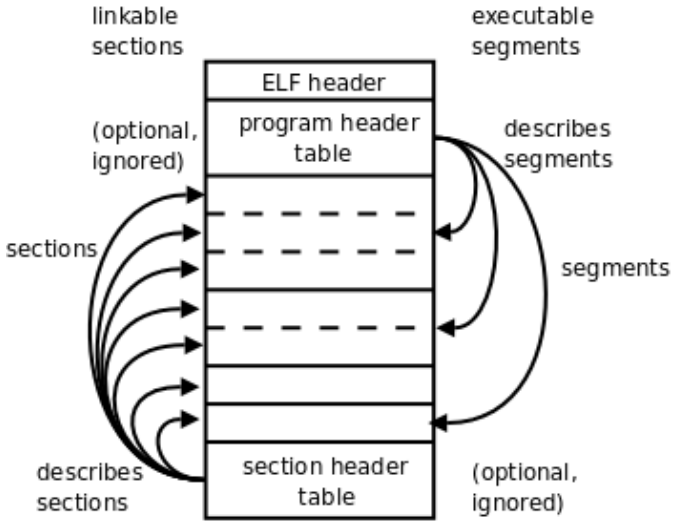

图 P1-4: ELF 文件结构

在这里我们举个例子，对一个可执行文件 `main`，在终端输入命令 `objdump -h main` 之后，会出现如下的内容：

```
main:     文件格式 elf64-x86-64

节：
Idx Name          Size      VMA               LMA               File off  Algn
  0 .interp       0000001c  00000000000002a8  00000000000002a8  000002a8  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag 00000020  00000000000002c4  00000000000002c4  000002c4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .gnu.hash     00000024  00000000000002e8  00000000000002e8  000002e8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .dynsym       000000c0  0000000000000310  0000000000000310  00000310  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .dynstr       0000009d  00000000000003d0  00000000000003d0  000003d0  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
```

```
 5 .gnu.version  00000010  000000000000046e  000000000000046e  0000046e  2**1
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
 6 .gnu.version_r 00000030  0000000000000480  0000000000000480  00000480  2**3
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
 7 .rela.dyn     000000c0  00000000000004b0  00000000000004b0  000004b0  2**3
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
 8 .rela.plt     00000030  0000000000000570  0000000000000570  00000570  2**3
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
 9 .init         00000017  0000000000001000  0000000000001000  00001000  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
10 .plt          00000030  0000000000001020  0000000000001020  00001020  2**4
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
11 .plt.got      00000008  0000000000001050  0000000000001050  00001050  2**3
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
12 .text         000001ce  0000000000001060  0000000000001060  00001060  2**4
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
13 .fini         00000009  0000000000001230  0000000000001230  00001230  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
14 .rodata       00000011  0000000000002000  0000000000002000  00002000  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
15 .eh_frame_hdr 00000044  0000000000002014  0000000000002014  00002014  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
16 .eh_frame     00000134  0000000000002058  0000000000002058  00002058  2**3
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
17 .init_array   00000008  0000000000003de8  0000000000003de8  00002de8  2**3
                 CONTENTS, ALLOC, LOAD, DATA
18 .fini_array   00000008  0000000000003df0  0000000000003df0  00002df0  2**3
                 CONTENTS, ALLOC, LOAD, DATA
19 .dynamic      000001e0  0000000000003df8  0000000000003df8  00002df8  2**3
                 CONTENTS, ALLOC, LOAD, DATA
20 .got          00000028  0000000000003fd8  0000000000003fd8  00002fd8  2**3
                 CONTENTS, ALLOC, LOAD, DATA
21 .got.plt      00000028  0000000000004000  0000000000004000  00003000  2**3
                 CONTENTS, ALLOC, LOAD, DATA
22 .data         00000010  0000000000004028  0000000000004028  00003028  2**3
                 CONTENTS, ALLOC, LOAD, DATA
23 .bss          00000008  0000000000004038  0000000000004038  00003038  2**0
                 ALLOC
24 .comment      00000022  0000000000000000  0000000000000000  00003038  2**0
                 CONTENTS, READONLY
```

可以看到，`main` 里有如此多的段，每一个段都有着自己的用处，比如：`.bss` 段中存放的都是没有初始化或者初始化为 0 的数据；`.data` 段里存放的都是初始化了不为 0 的数据；`.rodata` 是 read only data 的缩写，因此这一段里面存的都是类似于 C 语言中 `const` 修饰的变量，都是这种不可修改的数据；`.text` 段里存放着代码数据。

目前，我们对工具进行了设置，使编译出的 ELF 文件并不会产生很多段，以简化我们实验的内容。ELF 文件关于占用空间大小有 `size` 与 `mem` 两种不同的参数，请自行了解两者的区别并加以运用。

4.3 任务 2：加载和初始化内存

实验要求

调用 BIOS 的 API 从 SD 卡中加载内核，完成内存空间的初始化，并跳转至内核执行，在进入内核之后成功打印出 `"Hello OS"`。

文件说明

继续使用任务 1 的项目代码。

实验步骤

注：带星号标记的步骤需要等到 PYNQ 板卡发给大家之后，才能连接 SD 卡与 PYNQ 板卡上板运行。

1. 补全 `bootblock.S` 文件中的代码，添加的内容为调用 BIOS API 将起始于 SD 卡第二个扇区的内核代码段移动至内存。
2. 补全 `head.S` 文件中的代码，添加的内容为清空 BSS 段，设置栈指针，跳转到内核 main 函数。
3. 补全 `main.c` 文件中的代码，添加的内容为在打印 `"bss check: t version: X"` 之后，调用跳转表 API 读取键盘输入，并回显到屏幕上。
4. 运行 `make dirs` 命令创建 `build` 目录。
5. 运行 `make elf` 命令进行交叉编译，生成二进制文件。
6. 将我们提供的可执行文件 `createimage` 复制一份到 `build` 目录下，执行命令 `cd build && ./createimage --extended bootblock main && cd ..` 以生成镜像文件 `image`。
7. 运行 `make run` 命令启动 QEMU，当屏幕上可以打印字符串 `"Hello OS!"` 与 `"bss check: t version: X"`，并且最后能够持续接收屏幕输入并回显时，则说明 QEMU 测试通过。

8*. 运行 `make floppy` 命令，将 `image` 写到 SD 卡中。

9*. 将 SD 卡插入到板子上，使用 `make minicom` 监视串口输出，然后 restart 开发板。

10*. 开发板上电后，系统启动，当屏幕可以打印出字符串 `"Hello OS!"`，接下来输出 `"bss check: t version: X"`，最后可以持续地接收输入并输出在屏幕上，说明实验完成。

注意事项

1. 将内核从 SD 卡拷贝到内存中需要使用汇编 API，因为 C 语言的栈在进入内核之后才得到初始化。函数的用法请见任务 1 的注意事项。

2. 对于内核的放置位置，由于 boot loader 被放置的内存地址为 `0x50200000`，因此我们将内核拷贝到它的后面，也就是 `0x50201000`。从现在起，同学们可以自己总结内存空间的利用情况，并自行记录空间分配。

3. 读取完内核后，boot loader 最后需要完成的一个工作就是跳转到内核代码的入口，这个入口地址是哪里呢？其实我们在进行链接的时候已经将入口函数放到了内核文件的最前面，放到内存后，这个位置就是 `0x50201000`。至于我们是怎么放的，大家可以参考预备课的内容。

4. 此外，读取用户输入的函数是 `bios_getchar(void)`，该函数会立即检测键盘输入。如果此时键盘没有任何键被按下会返回 -1；如果有某个键被按下则返回对应的 ASCII 码。因此，在做键盘输入相关的动作时需要自己想办法处理掉 -1 的情况，避免将 -1 当作真正的输入直接使用，否则屏幕上会看到很奇怪的输出。

要点讲解

加载内核这一个任务相对比较简单。只要把内核拷贝到 `0x50201000`，再跳转过去即可。特别需要提示一下的是，如果常量的值较大，建议用如下形式载入常量：

```
lui     a0,     %hi(const_value) # 载入常量的高位
addi    a0, a0, %lo(const_value) # 载入常量的低位
```

若是编写汇编代码，建议使用伪指令 `li rd, immediate`，将立即数 `immediate` 加载到寄存器 `rd` 当中。这条伪指令在最终的程序中可能会占 1 或 2 条指令空间，这一点需要注意。

另外，一个需要注意的问题是内核占了几个扇区。这个在 `createimage` 的时候会显示。我们提供的 `createimage` 文件会把扇区的数目写在了头一个扇区的倒数第 4 个字节的位置 (`0x502001fc`)，长度为 2 字节。用 `lh` 指令可以载入两字节到寄存器。为了与之兼容，任务 3 中这部分代码需要同学们由自己设计和实现，我们推荐大家也这样实现。

最后大家需要填写一个 `head.S`。这个文件的设计目标是为 C 语言的执行创建环境，包括创建栈空间和初始化 bss 段。具体过程是从 Boot Loader 先跳转到 `head.S` 的 `_start`，完成初始化，再从 `_start` 跳到 `main.c` 中的 `main` 函数。其中最主要的是，要设置好 C 函数运行所需的栈空间。在本次实验中，我们把栈地址设置到 `0x50500000`，也就是内核所在位置后面的一段空闲内存中。另外，还需要清空 bss 段所在的内存。C 语言的未初始化全局变量之所以一般默认是 0，是因为有操作系统为我们将 bss 段清零。为了保持 C 语言运行的一些相关约定，我们的操作系统也需要用汇编程序把 bss 段清零，然后才能跳到 C 语言函数中。在 `head.S` 为 C 语言准备好一系列环境以后，最后跳转到 `main.c` 的 `main` 函数。这样，我们从 `main.c` 的角度看到的就是和我们平时常用的

C 语言一样的环境了：函数在栈上运行，未初始化的全局变量默认为 0, 从 `main` 函数开始执行。自然，同学们将会遇到一个问题：bss 段的内存地址范围是多少呢? 这里给大家一点提示——可以到链接器脚本 `riscv.lds` 中寻找。关于链接器脚本的功能在 Project 0 中有所介绍。

实验总结

通过完成本次的实验，想必你已经理解了操作系统是如何启动起来，以及内核的加载过程了。当屏幕上输出 `"Hello OS"` 的时候，可以说我们已经完成了一个最简单的操作系统了。虽然它只能输出几个字符串，但在接下来的几个实验中，我们会逐步完善它的功能。

可能你在这里还有一些疑问，比如 `createimage` 工具是如何合并多个 ELF 文件的。下面的任务由大家自己填写一个 `createimage` 镜像制作工具，虽然大部分代码已经实现好了，但是同学们可以通过阅读 `createimage` 工具对 ELF 文件的处理，更好地理解 ELF 的结构。

4.4 任务 3：加载并选择启动多个用户程序之一

实验要求

填写完成 `createimage.c` 文件，实现将 bootblock，kernel 以及用户程序结合为一个操作系统镜像，要求在 kernel 中可以交互式地输入数字（task id）选择运行哪一个用户程序。这里的 task id 即为 `image` 镜像文件中的第几个用户程序。其中 boot block 存放在镜像的第一个扇区，kernel 存放在从镜像的第二个扇区开始的地方，各个用户程序接续存储。同时，在任务 3 中 Kernel 和各个用户程序在镜像文件里面所占的扇区数是固定的，例如设置都占 15 个扇区。

`createimage` 文件中大部分函数已经完成，需要同学们实现以下函数的功能：

`write_img_info()` 将 kernel 所占扇区数和用户程序的数目写入 `image` 文件的特定位置供系统启动时读取。

完成了 `createimage` 之后，本任务还需要在 `main.c` 中添加对应的代码，在屏幕上打印出 `"Hello OS"` 之后，通过交互式输入 task id 的方式，加载并执行 task id 对应的程序（task id 必须小于 task 总数，这一点也会检查）。还需要实现加载所需的函数 `load_task_img`，其位于 `loader.c` 中，主要功能是从 SD 卡中拷贝指定的用户程序到内存中。任务三可以使用 task id 作为传入参数。

用户程序拷贝进内存后，同样需要进行 C 语言运行环境的初始化，包括堆栈和 bss 段的初始化，这部分代码在 `crt0.S` 中，需要同学们实现。

细心的同学可能已经发现了，各个用户程序的入口点都已经在 `Makefile` 中自动计算了，即 App1 入口点为 `0x52000000`、App2 入口点为 `0x52010000` ... 这样做的原因有如下两方面：一方面是我们现在的操作系统暂未实现虚拟内存机制，只能在物理地址上将各个用户程序错开；另一方面，在编译时候计算入口点的话，大家就可以直接把用户程序的扇区拷贝到对应的位置上去，这样也简化了大家的实现。

需要注意的是，在本学期的实验中，内核的代码和数据都存放在 0x50200000-0x51ffffff 内存范围内，因此同学们需要把用户程序加载到 0x52000000 开始的内存区域中，以初步实现内核与用户的地址分离。这一点在 Project 2 中会作为一个检查点，请同学们在加载时额外注意一下。

文件说明

请继续使用之前的代码进行实现。

实验步骤

注：带星号标记的步骤需要等到 PYNQ 板卡发给大家之后，才能连接 SD 卡与 PYNQ 板卡上板运行。

1. 完善 createimage.c、loader.c、crt0.S、bootblock.S。createimage.c 的功能是把各个已经编译好的 ELF 文件制作成镜像文件，loader.c 中实现了操作系统内核加载一个用户程序的功能，crt0.s 是每个用户程序都需要的初始化自身 C 语言环境功能。

2. 在 main.c 中根据键盘输入的 task id，使用 load_task_img 加载相应的用户程序，并执行之。

3. 执行 make all 命令进行交叉编译,并使用同学们自己写的 createimage 在 build 目录内生成 image 镜像文件。

4. 使用 make run 命令启动 QEMU，观察到测试程序按照定义好的加载顺序依次执行。

5*. 使用 make floppy 命令将 image 写入 SD 卡。

6*. 将 SD 卡插入到板子上，使用 make minicom 命令监视串口输出，然后 restart 开发板。

7*. 进入 BBL 界面后，输入 loadboot 命令，当屏幕可以正常按任务 3 中的任务说明执行用户程序时，表明实验完成。

注意事项

不要使用之前提供的 createimage 可执行文件，它并不支持用户程序的加载。本次任务需要使用自己编写的 createimage 完成实验。

用户程序在开始执行前需要对 C 语言执行环境做一些必要的初始化，然后再调用用户程序的 main 函数执行用户程序本身，执行后退出程序。像这样每个程序都需要重复的动作，完全可以放在一个单独的 crt0.S 中，常见的操作系统中也是这么实现的，程序都会链接一个 crt0.o。在我们的操作系统中，编译的时候用户程序都和 crt0.S 编译到一起。我们在 start-code 中的 arch/riscv/crt0 中也提供了这一文件，需要大家

去补完。在 Project 1 中，该部分代码需要先为用户程序准备好 C 语言运行时环境，例如创建栈帧、清空 bss 段等，在用户程序执行完之后需要负责回收栈帧并跳回内核中。当然，在真实系统中，用户程序的启动会更复杂。在后续的 Project 3 中，同学们将会通过系统调用实现用户程序的退出机制，从而进一步完善 `crt0.S`。这一点就且听下回分解了。

实验总结

在本实验中大家都自己完成了一个 `createimage` 镜像制作工具，并实现了在 kernel 中多个程序的选择加载执行。到此为止，S-core 的任务已经完成，还想继续挑战自己的同学请继续看后续任务。

后续的 Project 里面，我们将进入操作系统实验课的重头戏，一步一步，从中断处理、内存管理、进程管理、文件系统等各个方面将这个只能打印字符串的内核完善成一个完整的内核。

4.5 任务 4：镜像文件的紧密排列

本任务是做 A-core 和 C-core 的同学必须完成的。

实验要求

在前面的任务 3 中，大家设计的 `createimage` 在制作镜像文件的时候，kernel 和各个用户程序所占的扇区数都是固定的。这样做虽然实现上较为简单，但在 `image` 文件中留下了大量的“空泡”，从而大幅降低了镜像文件内的空间利用率，这在实际场景中显然是不好的。

因此，在本任务中，除了 boot block 仍然占用第一个扇区以外，同学们需要压缩 kernel 与用户程序、用户程序与用户程序之间产生的“空泡”，即让 kernel、app 1、app2 ... 在镜像文件中紧密排列。

同时，之前任务中使用 task id 来装载并启动用户程序。但是用 task id 对应用户程序地址很不直观。因此，在本任务中，我们希望以用户程序的 name，即编译出的用户程序 ELF 文件名来交互式装载并启动用户程序。

另外，作为 A-core 的额外要求，我们应当尽量确保自己编写的代码更可能没有错误。一个错误可能在编译器的优化中暴露出来，所以请自行学习并修改 `Makefile` 文件，对镜像文件中的程序（内核、用户程序）的编译启用 `-O2` 优化选项，以帮助我们在未来的实验中尽早发现可能存在的一些 bug。

要点解读

当然，细心的同学已经发现，用户程序的 ELF 文件名在运行 `createimage` 的时候已经通过 argv 参数的形式传入其中，所以，在 `create_image` 函数中，`*files` 即为我们想要的 task name。在本次实验中，kernel 和各个用户程序之间紧密排列，因此同学们需要设计 `task_info_t` 结构体（`createimage.c` 和 `task.h`），来方便 loader 进行加载

和定位。例如, loader 需要根据 task name 来找到目标用户程序对应哪一个 `task_info_t` 结构体，也需要用户程序在 `image` 文件中的偏移量和大小来提取并放到指定的装载入口。`createimage` 中获取偏移量和大小的方式，请同学们看一下 `createimage.c` 中的 `write_segment` 函数。当然，各个用户程序的 `task_info_t` 结构体也需要写入 `image` 中作为 App Info，等待内核的读取。具体的格式，以及应该怎么排列 `image` 文件中各个结构的顺序，请同学们自行思考实现。

此外，字符串处理的一些常用函数我们提供在了 `libs/string.c` 文件里面，大家可以根据自己的需要进行调用。接收键盘输入需要调用 `bios_getchar` 函数，具体定义在 `include/common.h` 里面，大家可以参考函数定义进行调用。

4.6 任务 5：列出用户程序和批处理

本任务是做 C-core 的同学必须完成的。

实验要求

纵观计算机系统的发展历史，批处理技术在其上留下了可谓浓墨重彩的一笔。批处理是指用户将一批作业提交给操作系统后就不再干预，由操作系统控制它们自动运行。这种采用批量处理作业技术的操作系统称为批处理操作系统。本次任务中我们需要实现简单的批处理系统。

此外, 即便我们在任务 4 中已经可以根据用户程序的 name 来启动对应的用户程序，但是我们并不知道都有哪些用户程序可以执行，因此操作系统应该支持让用户查看可以执行的用户程序。

在本任务中，选做 C-core 的同学需要先实现一个简单的命令，其作用是列出所有的用户程序的名字。之后，需要实现一个简单的批处理系统来顺序依次执行多个用户程序，多个程序之间可以传递输入输出。具体要求为：第一个程序中包含数字，并把它输出；第二个程序接收第一个程序的输出作为输入，并输出这个数字加上 10 的结果；再把第二个程序的输出结果输入给第三个程序，对数字乘以 3 并输出；最后再把第三个程序的输出结果输入给第四个程序，对这个数字进行平方运算，输出平方结果。四个程序批处理完成，且打印各自的输出结果。

该批处理任务需要用单独的名字去启动。注意是批处理，所以在 kernel 中输入一次启动命令，kernel 应可以自动依次启动四个测试程序。批处理执行的“顺序”需要以某种方式来指定，如果把“顺序”定义在 kernel 中，一旦我们需要更改批处理作业的执行顺序，那么就必须重新编译内核，这不是一个批处理系统应该有的表现。因此，大家需要引入一个额外的“批处理文件”，并且可以使用一些命令来读取、写入、执行这个批处理文件。需要大家实现两个批处理相关的命令，一个是把顺序写入“批处理文件”，另一个是按“批处理文件”规定的顺序执行。

注意，“批处理文件”作为一种“文件”，就不应该像内存中的内容一样易失，它应该存在于“外存”中，在我们实验中也就是写入到我们的 image 中。这个 image 上的空间应该在 createimage 的时候预留好。这样如果我们在其中写入过内容，重启操作系统后也依旧可以读到相同的内容。

要点解读

在任务 4 中，每个用户程序的信息都被存入 task_info_t 结构体中，其中包含了名字信息，可以借此来列出所有的用户程序的名字。

对于批处理，首先要完成多个程序的加载，并依次执行。让一个用户程序运行结束后直接跳转到另一个用户程序是不合理的，也是不符合操作系统的安全规则的。因此，同学们需要考虑在代码上实现一种机制，使得用户程序运行结束后会跳转回 kernel 中，而 kernel 会再直接跳转到下一个用户程序运行。

其次，每个程序都会有输出，且后面的程序需要用前面的输出作为输入，这就需要约定一种存放该数据的方式，建议约定固定的内存地址。而且，数字的长度也需要约定。

此外，“批处理文件” 存在于 image 中，在我们启动操作系统后可以读取和写入，因此我们需要在 createimage 时为其预留好一定的空间。此外，前面的任务中我们都没有用到 sd 卡的写接口，这里需要用到，具体接口定义请同学们参见前面的任务书，使用与 bios_sd_read 类似的 bios_sd_write 函数。

在上述任务中，批处理文件的大小、位置、内容格式均不做要求，命令也不做格式要求，请同学们自行思考实现。

对于 C-core 的任务，我们可能不会给予太多的提示和参考，请有余力的同学们更多的自己查阅相关的资料，通过思考完成相关的任务要求，包括自行开发测试程序和完成 OS 功能。

参考文献

[1] “应用程序接口.” https://zh.wikipedia.org/zh-cn/%E5%BA%94%E7%94%A8%E7%A8%8B%E5%BA%8F%E6%8E%A5%E5%8F%A3?oldformat=true, 2024. [Online; accessed 21-August-2024].