国科大操作系统研讨课任务书

RISC-V 版本

版本 *1.0*

目录

Project 1
引导、内核镜像和 ELF 文件

1 引言

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 等工具链的使用，了解相关的硬件知识，甚至在后期，还会涉及到网络的内容，可以说是需要“上知天文，下知地理”。这也体现了操作系统在整个计算机系统中的地位。

当然，大家不需要过于担心自己的“积累”能否胜任这项工作，实际上，我们开设这门实验课的初衷也是希望大家能够针对这些内容进行学习和融会贯通，掌握不好的内容，借此机会了解它的原理；已经掌握的知识，借由实际开发的机会更好的去理解其中的细节。

在实验一，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，学习 Linux 下相关工具、C 语言和汇编、内核镜像的制作等内容。另外，由于我们给出的任务书内容有限，不可能罗列所有的知识和内容，更多的起到的是指导作用，因此希望大家对于任务书中讲解不详细的地方，自己去网上查找相关资料或者互相讨论。

俗话说的好“万事开头难”，最后希望大家能够认真完成实验！接下来，我们将从理论到实验，迈出我们制作属于我们操作系统的第一步。

2 实验要点解读

这一部分的要点是：

1. 掌握操作系统启动的完整过程，体会软件和硬件的分工与约定
2. 掌握 ELF 文件的结构和功能

简单点说，就是要学会，如何在裸金属机器 (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，哪有什么操作系统。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。

学 C 语言的时候，大家都写过 hello world。本章的内容实际上就是裸金属机器上面的 hello world。所以在学习这一章的内容，逻辑上和学习 hello world 是一样的，只需要

搞懂：1. **程序从哪里开始运行？** 2. **程序如何输出？**

3 操作系统的引导

3.1 什么是引导

操作系统，实际上也是一个特殊的程序，既然是程序，那么我们就需要把它运行起来，那么怎么样把操作系统运行起来呢？这就是引导（Boot Loader）需要做的事情啦。引导主要的任务就是将操作系统代码从存储设备（SD 卡），搬运到内存中。

所以，这一节讲的是主要是上面提到的第一个问题：**程序从哪里运行**。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？这个地址是内存吗？是磁盘吗？是 SD 卡吗？还是网络上的某个地址？下面我们用常识推断一下。首先网络、磁盘、SD 卡恐怕都是不可能的。记不记得自己在用电脑的时候，插个 U 盘电脑都说正在安装驱动程序。这种设备没个驱动程序怎么可能能访问得了？显然不可能。那就只能是内存这种相对简单而且比较核心的设备喽？也不可能。我们都学过，内存重启后东西就没了。程序怎么可能放在内存里面？

看到这里，你肯定会问：那怎么办呀？聪明的工程师们，把这个地址映射到了 ROM 上。你可以简单的理解，ROM 和内存很像，只不过是只读的，内容一旦烧好了就没法改。所以，ROM 里面的程序一般是厂家提前烧好的。CPU 以启动，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)。或者你也许听说过 UEFI(Unified Extensible Firmware Interface)。它们都是这类烧在 ROM 里面的程序。

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 512B 的数据到内存的指定地址，最后跳转到该地址。这 512B 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。

> **Note 3.1** 操作系统的启动英文称作“boot”。这个词是 bootstrap 的缩写，意思是鞋带（靴子上的那种）。之所以将操作系统启动称为 boot，源自于一个英文的成语“pull oneself up by one's bootstraps”，直译过来就是用自己的鞋带把自己提起来。看了上面这个麻烦的过程，你也许体会到了为什么以前的工程师们给启动取这么个名字。另外，其实现在很多地方用的 ROM 是可以反复烧写的，并不是“只读”的。

其实仔细想一想，这个过程很好理解。整个启动的过程就好像从幼儿园读到大学一样。最开始处理器刚加电，就是幼儿园阶段，什么都干不了，只能等老师来喂自己（处理器 PC 复位到固定地址，开始执行)。之后，升入小学（ROM 上的程序把自己拷贝入内存)，开始学到一些基本的知识（ROM 程序进行一系列初始化后可以读磁盘第一个扇区了)。再升入到中学，以之前学到的知识为基础，学到各学科的全部基本知识（第一个扇区的程序加载比自己大得多的系统内核进内存)。最后进入到大学，可以开始自己学习了（操作系统内核接管处理器，执行起各种程序)。

整个过程是一个逐步搭建新的环境，解除旧环境的限制的过程，由简单而到复杂，一步一台阶。

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。我们的实验环境采用 XILINX 的 PYNQ 开发板。整个实验的启动流程如图P1-1所示。PYNQ 上面有一个 ARM 处理器和 FPGA。开发板启动后会先启动 ARM 核，ARM 核会读取 SD 卡上的程序。该程序会把一个 RISC V 核烧到 FPGA 上，之后触发 RISC V 核的初始化过程。RISC V 同样会执行一小段预先放置好的程序，经过一系列的动作，最后加载 SD 卡第二个分区的头 512B 的数据（加载时假定了 SD 卡第一个分区为 34MB，起始地址为 716800）到 0x50200000。

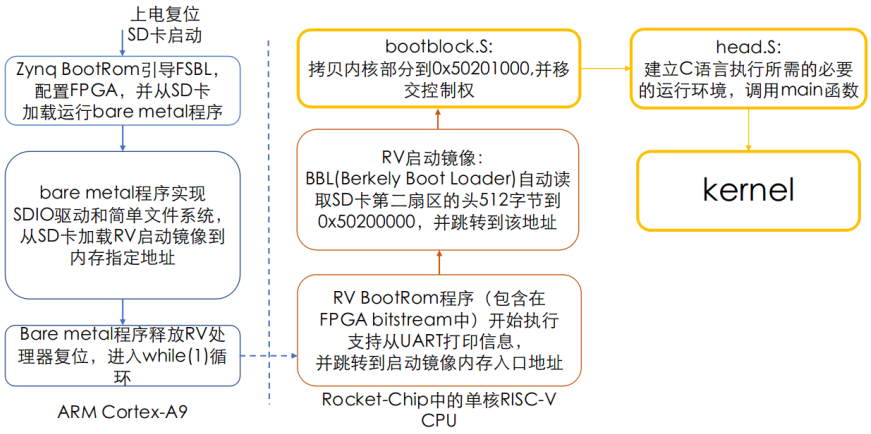

图 P1-1: 实验整体启动流程

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。而黄色加粗的部分是我们的实验需要完成的。前面的流程是硬件或者 BBL 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，执行地址自动会跳转到一个位置开始执行。这段代码主要的任务就是将存储设备上的第一个扇区（512B）的内容，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，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:** Boot Loader 的代码由于只有 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。

接下来，我们将通过循序渐进的三个任务，逐步实现一个操作系统的 Boot Loader。

3.3 地址空间情况

这里可以介绍一下我们 RISC-V 板卡的地址空间情况，如表P1-1所示。其中，最需要注意的是，BBL 所需的内存空间放置了 BBL 运行所需的数据和代码。请一定不要修改这段内存。BBL 为我们提供了读写 SD 卡和输出字符串的相关服务。如果不小心修改了它的数据或代码，可能导致相关功能异常。我们自己将要编写的内核可以使用的空间

地址范围	权限	作用
0x0-0x1000	ARWX	debug-controller
0x2000-0x3000	RW	hcpf
0x3000-0x4000	ARWX	error-device
0x10000-0x20000	RX	rom
0x2000000-0x2010000	ARW	clint
0xc000000-0x10000000	ARW	interrupt-controller
0x50000000-0x80000000	RWXC	memory
0xe0000000-0xe0300000	RW	zynqmmio-axi4
0xe0000000-0xe0001000	RW	serial
0xe0002000-0xe0003000	RW	usb
0xe000b000-0xe000c000	RW	ethernet
0xe0100000-0xe0101000	RW	mmc
0xe0300000-0xe0301000	RW	gpio
0xf8000000-0xf8000c00	RW	slcr
0xf8000000-0xf8000c00	RW	zynqmmio-axi4

表 P1-1: 地址空间

为 0x50200000-0x80000000 这一段地址。

另外一点需要注意的是，如果错误地读写了 0x0 或者其他非 memory 的地址，那么很有可能触发中断。由于前面的实验我们没有设置中断处理机制，所以一旦访问错误的地址，在开发板上看到的现象就是程序卡死，不再继续执行。建议在调试的时候，多使用 QEMU+gdb。或者分成小段一点一点调试，在出现内存相关的错误的情况下，试图直接找到大段代码中的错误很困难。一般应该一小段一小段逐步缩小范围，从而正确地找到错误的发生位置。

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 BBL 中的输入输出函数，在终端成功输出“It’s Boot Loader!”。

文件说明

见表P1-2所示。

编号	文件名称	文件说明
1	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
1	head.s	内核入口代码，负责准备 C 语言执行环境
2	createimage	将 bootblock 制作成 512B 引导块的工具，不需要修改
3	Makefile	Makefile 文件，不需要修改
4	ld.script	链接器脚本文件，不需要修改
5	kernel.c	内核的主函数所在，任务 2 中完成
6	createimage.c	引导块工具代码，任务 3 中需要实现

表 P1-2: P1 文件说明

实验步骤

1. 填写 bootblock.s 代码，要求添加的内容为打印字符串“It’s Boot Loader!”。
2. 运行 make all 命令进行交叉编译，生成二进制文件生成镜像。
3. 使用 make floppy 命令将 bootblock 写到 SD 卡的第一个扇区。
4. 将 SD 卡插入到板子上，使用 minicom 连接，然后 restart 开发板。
5. 板子加电后，系统启动，当屏幕可以打印出字符串“It’s Boot Loader!”说明实验完成。

要点讲解

为了简化大家的实现，这里给大家封装好了几个函数。定义在了 sbi.h 中。任务 1 中需要从汇编语言调用，为了简单，这里还额外提供了一个 sbiasm.h。里面定义了一个宏 SBI_CALL。这个宏是 sbi.h 中的 C 函数的一个极度简化的版本。需要自己把参数放入到 a0,a1,a2 中。可供调用的接口类型定义在 sbidef.h 中。对照 C 语言版本，就可以理解 sbiasm.h 具体怎么使用。

现将本次实验中用到的函数原型说明如下：

- void sbi_console_putstr(char *str) 在终端打印字符串 str。(汇编语言中使用 SBI_CALL SBI_CONSOLE_PUTSTR 进行调用，下面类似)
- uintptr_t sbi_sd_read(unsigned int mem_address, unsigned int num_of_blocks, unsigned int 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。
- void sbi_console_putchar(int ch) 在终端打印字符 ch。
- int sbi_console_getchar() 读取终端输入字符 ch, 如果未读取到任何字符则返回-1。

这里有一个比较奇怪的点：为了让后面的打印字符串的函数能正常工作，需要先用 sbi_console_putchar 输出一个单字符，原因未知。

> **Note 3.2** 这次作业有汇编文件。请大家一定注意，汇编文件扩展名为.S 和.s（大写和小写）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到内存的部分，实际上，我们现在还没有写操作系统代码，也没有进行内核镜像的制作。

所谓好的开始是成功的一半，我们已经跨出了最终要的一步，在我们的开发板上已经可以运行起来我们的程序了！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，制作内核镜像**。最终，一个精简而又完整的操作系统真正的运行在我们的开发板上。

4 内核镜像

接下来，我们要做一个完整的内核镜像了。这一节最核心的要点是把握住：可执行文件是有格式的。就好像平时我们播放的视频文件有 avi/mp4/wmv 之类的格式一样，可执行文件也不是简单的就是一大串机器指令，它也是按照一定的格式组织起来的。

ELF 文件简单的理解就是描述了，文件中的哪段代码需要被拷贝到内存的什么位置。比如你编译了一个 hello world 程序，这个程序要被装载到内存中执行。那么程序中的每个段落，需要被放在内存中的哪个位置才能正确执行？这就是 ELF 中记录的内容。比如我们都学过，汇编中有可以跳转到某个绝对地址的跳转指令。那么就要求我的程序被加载进内存的时候必须放到我指定的位置，不能乱放。否则跳转不就跳错了吗？还有我的数据，如果给我随便放一个位置，我不就找不到了吗？所以必须告诉系统，程序中的每个数据/代码该被放在哪里。ELF 本质上就是记录了这些信息的一种文件格式。

4.1 内核镜像组成

对于我们制作的内核镜像应该包含两个部分，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，也就是操作系统部分，它放在 Boot Loader 的后面，它们在 SD 卡的位置如图P1-2所示。

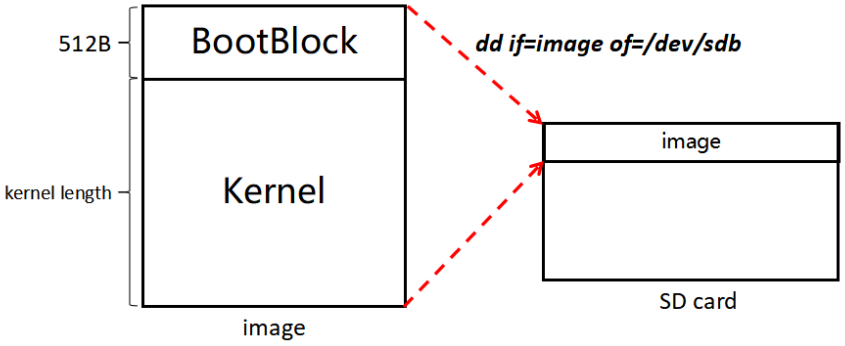

图 P1-2: Boot Loader、Kernel 的位置

关于内核镜像的制作，我们是采用以下的步骤完成的（见图P1-3）：1）编译 Boot Loader，2）编译 Kernel，3）使用镜像制作工具 creatimage 合并 Boot Loader 和 Kernel 代码，生成镜像文件。

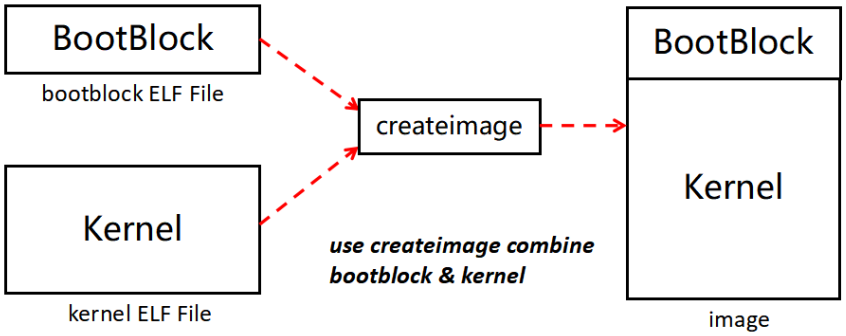

图 P1-3: 镜像生成过程

猛地这么说大家可能会非常困惑，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，详细阐述内核镜像的制作流程以及相关知识。在经过这一部分的学习，你将学习并掌握项目的编译流程，内核镜像的制作方法，并最终成功制作出一份属于自己的内核镜像！

4.2 编译流程

刚才也说了，我们需要把内核编译成机器可以执行的代码，那么就涉及到了 4 个步骤：**预编译、编译、汇编、链接**（如图P1-4所示）。

为了更加清楚阐述各个阶段的关系，我们通过一个小项目来具体阐述。项目代码有 hello.h(Listing 1)、hello.c(Listing 2)、main.c(Listing 3) 三个文件。这 3 个文件主要完成输出”hello world”的简单工作。

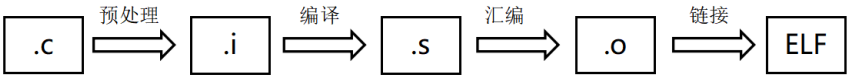

图 P1-4: 编译流程

Listing 1: hello.h

```h
#ifndef HELLO_H
#define HELLO_H

void hello_world();

#endif /* HELLO_H */
```

Listing 2: hello.c

```c
#include "hello.h"
#include "stdio.h"

void hello_world()
{
    printf("Hello World!\n");
}
```

Listing 3: main.c

```c
#include "hello.h"

int main()
{
    hello_world();
    return 0;
}
```

预编译

编译器在预编译这一步骤不进行语言间的转化，只进行宏扩展。GCC 编译器可以分步执行编译链接的步骤，我们只需要在后面添加参数-E 就可以只进行预编译这一步骤，现在我们在终端输入下列命令：

```
$ gcc -E hello.c -o hello.i
$ gcc -E main.c -o main.i
```

我们打开我们生成的文件，hello.i 和 main.i，可以发现，里面的内容如代码 4所示。这里只展示 main.i, 因为 hello.i 引用了标准输入输出，导致预编译完的文件很长，限于篇幅无法展示。

Listing 4: 预编译结果

```
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "main.c"
# 1 "hello.h" 1



void hello_world();
# 2 "main.c" 2

int main()
{
    hello_world();
    return 0;
}
```

可以发现，相对于预编译之前，生成的新文件只是简单的做了一下宏替换。

编译

在编译这一步骤，编译器主要的工作是将高级语言（C 语言等编程语言）转化成汇编语言。同理，我们将刚才生成的.i 文件继续编译，添加-S 参数，在终端输入以下命令：

```
$ gcc -S hello.i -o hello.s
$ gcc -S main.i -o main.s
```

我们打开生成的.s 文件，发现里面内容如下（同样只展示 main.s, 如代码5所示）：

Listing 5: 编译结果

```
        .file   "main.c"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $0, %eax
        call    hello_world@PLT
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
```

```
        ret
        .cfi_endproc
.LFE0:
        .size        main, .-main
        .ident       "GCC: (Gentoo 9.1.0-r1 p1.1) 9.1.0"
        .section        .note.GNU-stack,"",@progbits
```

可以发现，原来的 **C 语言代码已经被转化成为了汇编代码**，这也是编译这一步所进行的工作。

汇编

大家都知道，真正跑在电脑里的代码并不是 C 语言，也不是汇编语言，而是只有机器才能识别的二进制机器语言。而汇编这一步骤，所做的就是将编译所生成的汇编代码转化成只有机器才能识别的二进制机器代码。我们在终端里输入以下命令：

```
$ gcc -c hello.s -o hello.o
$ gcc -c main.s -o main.o
```

打开生成的.o 文件，里面的内容如图P1-5所示。是的！我们生成的文件已经是一个二进制文件，里面存放的数据都是只有机器才能识别的机器代码啦

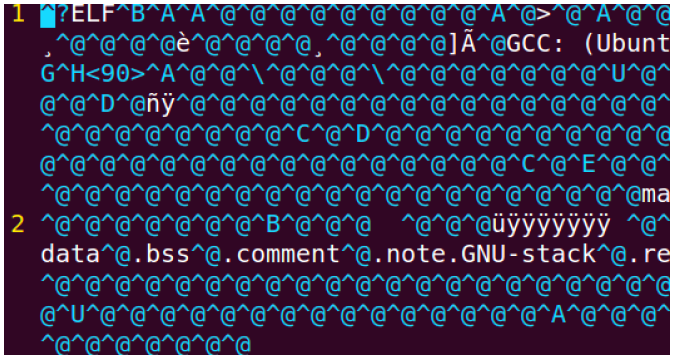

图 P1-5: 编译出的 main.o

链接

到目前，我们生成的文件有 main.o 和 hello.o 这 2 个二进制文件，但是它们现在还不能直接运行，因为它们是彼此分离的，所以我们需要通过链接这一重要的步骤去将彼此分离的二进制代码合并成最终的可执行文件。我们在终端输入以下命令：

```
$ gcc hello.o main.o -o main
$ ./main
Hello World!
```

最终，我们的“Hello World”项目从 3 个文件，合并成为了一个名为 main 的可执行文件，并可以打印出“Hello World!”。是不是很神奇？

经过了上述的流程，你应该已经对一个项目从编译到执行已经有了清楚的认识，但是仍然会有一些问题在困扰你：难道每编译一次项目都要手打这么多复杂命令吗？链接

是通过什么规则将这些可执行文件合并在一起的呢？链接生成的可执行文件又是一个什么文件呢？

当然，这些问题我们都将在接下来的部分具体阐述，给介绍项目编译利器——Makefile，介绍如何通过链接器脚本控制我们的链接过程，以及跟我们内核制作密不可分的一种可执行文件——ELF。

4.3 Makefile

经过了上述的小例子，大家可能发现在编译多个文件的时候，如果手动去一条一条命令的去编译那将是一个非常繁琐的过程，特别是很多大型项目文件多达成百上千个的时候，一个一个手动编译无疑是十分蠢的举动。

所幸在 Linux 下，有着 Makefile 这一利器，关于 Makefile 大家可以通俗的理解成一个脚本文件，它所做的事情就是项目的整体编译，只需要一个 make 命令，我们便可以将包含大量文件的项目编译成功。

在本次操作系统实验课中，大部分的项目代码我们都已经写好了 Makefile 供大家使用，但是如果你仍希望更加深入的理解 Makefile，熟悉掌握相关知识的话，可以查阅网上给多的资料 [1][2][3]，了解如何编写 Makefile。

4.4 ELF 文件

刚才已经说过，经过链接这一步骤后，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它的格式是 ELF，在计算机科学中，是一种用于二进制文件、可执行文件、目标代码、共享库和核心转储格式文件。是 UNIX 系统实验室（USL）作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P1-6所示。

在这里我们举个例子，就用我们刚刚生成的 main 可执行文件，在终端输入以下的指令：

```
  $ objdump -h main

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
```

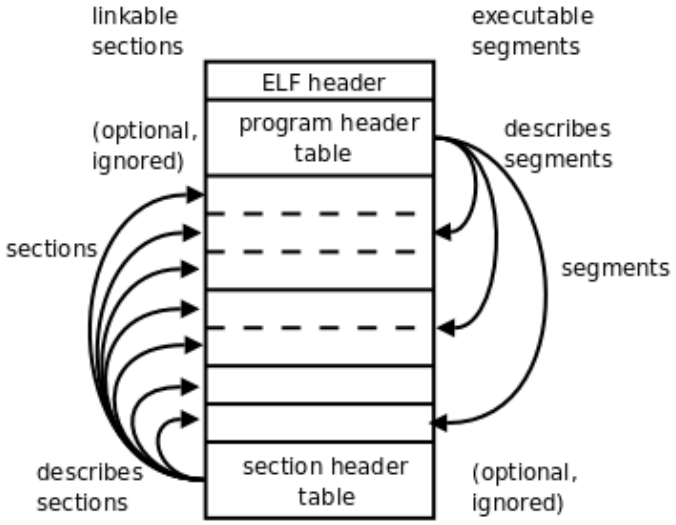

图 P1-6: ELF 文件结构

```
  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 里存放的都是初始化了不为 0 的数据，.rodata 段的缩写实际上是 read only data，因此它里面存的都是类似于 const 变量修饰的不可写的数据，.text 段里存放着代码数据。

4.5 链接器脚本

刚才已经说过，在链接阶段，会将多个.o 文件合并成为一个 ELF 格式的可执行文件。ELF 里包含各种段，每个段包含的内容也不一样，有的包含数据，有的包含代码。在刚才的 demo 里我们直接使用了 gcc 命令进行链接，这个链接是使用了默认的规则。因此生成的 ELF 文件的布局我们都不是清楚的，比如代码段的位置，数据段的位置我们都不知道。但在内核编译的过程中，很多内容我们都需要将它放到固定的位置，比如内核的入口函数地址（清楚了入口地址我们才能跳到这里去运行内核代码），栈堆地址等等。因此，我们需要自己制定规则，去布置各个段在 ELF 文件中的位置，这也是链接器脚本的功能。

链接器脚本的书写是一个繁琐的过程，但是我们已经在大家以后的代码框架中都准备好了链接器脚本，配合 Makefile 一起使用，大家直接使用就可以了。

4.6 任务 2：开始制作内核镜像

实验要求

了解和掌握内核镜像制作步骤，补全 Boot Loader 的加载内核部分代码，完成操作系统的完整引导过程，并在进入到 OS 阶段时打印出“Hello OS”。

文件说明

继续使用任务 1 的项目代码。

实验步骤

1. 补全 bootblock.s 文件中的代码，添加的内容为调用 SBI 将位于 SD 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。
3. 补全 kernel.c 文件中的代码，添加的内容为调用 SBI，输出字符串“Hello OS”。
4. 继续补全 kernel.c 文件，添加的内容在输出“Hello OS”之后，调用 SBI 连续接收输入，并同时把输入的字符打印出来。

5. 运行 make all 命令进行交叉编译，生成二进制文件生成镜像。
6. 运行 make floppy 命令，将 image 写到 SD 卡的第一个扇区。
7. 将 SD 卡插入到板子上，使用 minicom 连接，然后 restart 开发板。
8. 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“Version: 1”；最后可以持续的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 SBI，函数的用法请见任务 2 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。
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，这个入口地址使哪里呢？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，大家可以阅读链接器脚本文件 riscv.lds 的内容以及 kernel.c 的内容自己思考。

要点讲解

制作内核镜像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。可以参考 [4] 和 [5]。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
```

另外，一个需要注意的问题是内核占了几个 sector。这个在 createimage 的时候会显示。我们建议把 sector 的数目写在了头一个 sector 的倒数第 4 个字节的位置 (0x502001fc)，长度为 2 字节。用 lh 可以载入两字节到寄存器。

最后大家需要填写一个 head.S。这里的设计目标是先跳到 head.S 中的 _start，再从 _start 跳到 kernel.c 中的 main 函数。这样设计的原因是希望大家理解，bootblock.S 把内核拷入内存后，一般还需要做一些动作来为 C 语言的执行创建环境。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。这里没有操作系统了，为了保持 C 语言运行的一些相关假设，我们需要用汇编程序手工把 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 kernel.c 的 main 函数。这样，我们从 kernel.c 的角度看到的就是和我们平时管用的 C 语言很像的一个环境了：函数在栈上运行，全局变量默认为 0，从 main 函数开始执行。

做 A-core 和 C-core 的同学必须完成：

在任务中，我们将内核移动到了 0x50201000，正好放到了 bootloader 的后面。对于做 A-core 和 C-core 的同学，我们希望将内核拷贝到 0x50200000 中，并从 0x50200000 处开始执行内核代码。注意：因为内核需要拷贝到 0x50200000 处，因此会覆盖 bootloader，如果在 bootloader 执行到一半时被覆盖会导致错误的发生，请大家思考如何规避这个问题。另外，一般的处理器都有指令 cache，直接将代码拷贝覆盖掉 bootloader 时，可能由于指令 cache 的原因读到的代码和实际内存里面的代码不一致。此时，需要用 fence.i 指令把指令 cache 重新刷一下。

实验总结

通过完成本次的实验，想必你已经深刻理解了操作系统是如何启动起来，以及镜像文件的制作流程了！当屏幕上输出“Hello OS”的时候，可以说我们已经完成了一个操作系统了，虽然它只能输出一个字符串，但接下来，我们将一步一步，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。

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，下面的一节将向大家介绍如何自己手写一个 crerateimage 镜像制作工具，虽然这和我们的操作系统本身没有太大关系，但是这对你理解操作系统镜像的制作会有很大帮助。

5 ELF 文件

5.1 什么是 ELF 文件

ELF 文件是一种目标文件格式，用于定义不同类型目标文件是什么样的格式存储的，都存放了些什么东西。主要用于 linux 平台。可执行文件、可重定位文件 (.o)、共享目标文件 (.so)、核心转储文件都是以 elf 文件格式存储的。ELF 文件组成部分：文件头、段表 (section) 头、程序头。

5.2 文件头

ELF 文件头定义了文件的整体属性信息，比较重要的几个属性是：魔术字，入口地址，程序头位置、长度和数量，文件头大小(52 字节)，段表位置、长度和个数。在/usr/include/elf.h 中可以找到文件头结构定义，文件头的结构体如下：

```
1  // elf.h
2
3  #define EI_NIDENT (16)
4
5  typedef struct
6  {
7    unsigned char e_ident[EI_NIDENT];     /* Magic number and other info */
8    Elf32_Half    e_type;                 /* Object file type */
9    Elf32_Half    e_machine;              /* Architecture */
```

```
  Elf32_Word    e_version;              /* Object file version */
  Elf32_Addr    e_entry;                /* Entry point virtual address */
  Elf32_Off     e_phoff;                /* Program header table file offset */
  Elf32_Off     e_shoff;                /* Section header table file offset */
  Elf32_Word    e_flags;                /* Processor-specific flags */
  Elf32_Half    e_ehsize;               /* ELF header size in bytes */
  Elf32_Half    e_phentsize;            /* Program header table entry size */
  Elf32_Half    e_phnum;                /* Program header table entry count */
  Elf32_Half    e_shentsize;            /* Section header table entry size */
  Elf32_Half    e_shnum;                /* Section header table entry count */
  Elf32_Half    e_shstrndx;             /* Section header string table index */
} Elf32_Ehdr;

typedef struct
{
  unsigned char e_ident[EI_NIDENT];     /* Magic number and other info */
  Elf64_Half    e_type;                 /* Object file type */
  Elf64_Half    e_machine;              /* Architecture */
  Elf64_Word    e_version;              /* Object file version */
  Elf64_Addr    e_entry;                /* Entry point virtual address */
  Elf64_Off     e_phoff;                /* Program header table file offset */
  Elf64_Off     e_shoff;                /* Section header table file offset */
  Elf64_Word    e_flags;                /* Processor-specific flags */
  Elf64_Half    e_ehsize;               /* ELF header size in bytes */
  Elf64_Half    e_phentsize;            /* Program header table entry size */
  Elf64_Half    e_phnum;                /* Program header table entry count */
  Elf64_Half    e_shentsize;            /* Section header table entry size */
  Elf64_Half    e_shnum;                /* Section header table entry count */
  Elf64_Half    e_shstrndx;             /* Section header string table index */
} Elf64_Ehdr;
```

名称	大小	对齐	用途
Elf64_Addr	8	8	无符号程序地址
Elf64_Half	4	4	无符号中等大小整数
Elf64_Off	8	8	无符号文件偏移
Elf64_Sword	8	8	有符号大整数

表 P1-3: 类型说明

使用文件头结构体需包含 #include <elf.h> 头文件。结构体 Elf64_Ehdr 最开头是 16 个字节的 e_ident, 其中包含用以表示 ELF 文件的字符，以及其他一些与机器无关的信息。开头的 4 个字节值固定不变，为 0x7f 和 ELF 三个字符。

5.3 程序头

在 ELF 中把权限相同、又连在一起的段 (section) 叫做 segment，操作系统正是按照“segment”来映射可执行文件的。描述这些“segment”的结构叫做程序头，它描述了 elf 文件该如何被操作系统映射到内存空间中。在/usr/include/elf.h 中可以找到文件头结构定义，程序头的结构体如下:

```c
/* Program segment header.  */

typedef struct
{
  Elf32_Word    p_type;                 /* Segment type */
  Elf32_Off     p_offset;               /* Segment file offset */
  Elf32_Addr    p_vaddr;                /* Segment virtual address */
  Elf32_Addr    p_paddr;                /* Segment physical address */
  Elf32_Word    p_filesz;               /* Segment size in file */
  Elf32_Word    p_memsz;                /* Segment size in memory */
  Elf32_Word    p_flags;                /* Segment flags */
  Elf32_Word    p_align;                /* Segment alignment */
} Elf32_Phdr;

typedef struct
{
  Elf64_Word    p_type;                 /* Segment type */
  Elf64_Word    p_flags;                /* Segment flags */
  Elf64_Off     p_offset;               /* Segment file offset */
  Elf64_Addr    p_vaddr;                /* Segment virtual address */
  Elf64_Addr    p_paddr;                /* Segment physical address */
  Elf64_Xword   p_filesz;               /* Segment size in file */
  Elf64_Xword   p_memsz;                /* Segment size in memory */
  Elf64_Xword   p_align;                /* Segment alignment */
} Elf64_Phdr;
```

5.4 段表头

包含了描述文件节区的信息，每个节区在表中都有一项，每一项给出诸如节区名称、节区大小这类信息。用于链接的目标文件必须包含节区头部表，其他目标文件可以有，也可以没有这个表。

```c
/* Section header.  */

typedef struct
{
  Elf32_Word    sh_name;                /* Section name (string tbl index) */
  Elf32_Word    sh_type;                /* Section type */
  Elf32_Word    sh_flags;               /* Section flags */
  Elf32_Addr    sh_addr;                /* Section virtual addr at execution */
  Elf32_Off     sh_offset;              /* Section file offset */
  Elf32_Word    sh_size;                /* Section size in bytes */
  Elf32_Word    sh_link;                /* Link to another section */
  Elf32_Word    sh_info;                /* Additional section information */
  Elf32_Word    sh_addralign;           /* Section alignment */
  Elf32_Word    sh_entsize;             /* Entry size if section holds table */
} Elf32_Shdr;

typedef struct
{
  Elf64_Word    sh_name;                /* Section name (string tbl index) */
  Elf64_Word    sh_type;                /* Section type */
  Elf64_Xword   sh_flags;               /* Section flags */
  Elf64_Addr    sh_addr;                /* Section virtual addr at execution */
  Elf64_Off     sh_offset;              /* Section file offset */
  Elf64_Xword   sh_size;                /* Section size in bytes */
```

```
25    Elf64_Word    sh_link;              /* Link to another section */
26    Elf64_Word    sh_info;              /* Additional section information */
27    Elf64_Xword   sh_addralign;         /* Section alignment */
28    Elf64_Xword   sh_entsize;           /* Entry size if section holds table */
29  } Elf64_Shdr;
```

5.5 任务 3：制作 createimage

实验要求

编写 createimage.c 文件实现将 bootblock 和 kernel 结合为一个操作系统镜像，要求可以传入参数进行内核镜像的制作。

并提供操作系统镜像的一些信息。其中 bootblock 存放在镜像的第一个扇区，kernel 存放在镜像的第二个扇区。一共需要实现以下函数：

read_exec_file() 读取 ELF 格式的一个文件。

write_bootblock() 将可执行文件 bootblock 写入内核镜像”image”文件中。

write_kernel() 将可执行文件 kernel 写入镜像文件”image”文件中。

count_kernel_sectors() 计算 kernel 有多少个扇区。

record_kernel_sectors() 将 kernel 的扇区个数写入 bootblock 的 os_size 位

extend_opt() 打印出—extend 选项要打印出来的信息。包括内核的大小，可执行文件在磁盘上存放的扇区以及在磁盘上写的大小等信息。

文件说明

请继续使用之前的代码进行实现。

实验步骤

实验步骤同任务 2 相同，唯一不同的就是我们需要使用自己制作的 createimage 工具进行镜像的工作。以下为步骤：

1. 编写 createimage.c 代码。
2. 进行交叉编译，并使用 creatimage 工具，生成符合实验要求的二进制文件 image。
3. 将 image 写入 SD 卡。
4. 将 SD 卡插入到板子上，使用 minicom 连接，然后 restart 开发板。
5. 进入到达 BBL 界面后，输入 loadboot 命令，当屏幕可以照常输出和输入任务 2 中的内容说明实验完成。

注意事项

不要使用之前提供的 createimage 可执行文件，而是使用自己制作的镜像制作工具完成实验。

要点解读

这一节要求实现 createimage.c。这里需要注意的是，我们采用的是 64 位 RISC-V 版本。所以，在实现的时候请注意，所有和 ELF 相关的类型都要用 Elf64 开头的版本。

需要实现下面几个函数：(你也可以自己设计程序的结构)

- void create_image(int nfiles, char *files[]) 程序的主体部分，制作内核镜像 image。nfiles 为需要制作进内核的 ELF 文件数，files 为文件名的数组。这两个参数均由 main 函数处理命令行参数后得到。
- void read_ehdr(Elf64_Ehdr * ehdr, FILE * fp) 读取 fp 所指向文件的 ELF 头。
- void read_phdr(Elf64_Phdr * phdr, FILE * fp, int ph, Elf64_Ehdr ehdr) 根据 ELF 头的信息，读取该文件的一个程序头。ph 指示程序头的编号。
- void write_segment(Elf64_Ehdr ehdr, Elf64_Phdr phdr, FILE * fp, FILE * img, int *nbytes, int *first) 将程序头 phdr 指示的段从 fp 中读出，并写在内核 img 中正确的位置。nbytes 所指变量记录 img 中已经写入的字节数;first 所指变量记录本次写入是否为首次写入（对应 bootblock），该信息在计算段在镜像中的偏移地址时用到。注意程序头中 p_filesz 域和 p_memsz 域的关系：p_filesz 为该段在 ELF 文件中的大小，而 p_memsz 为 ELF 程序被加载器加载后在内存中所占大小，它可能比 p_filesz 大（多出的空间供程序运行时使用)。我们的操作系统在开发板上运行时是直接由引导块搬运到内存里的，因此写入镜像的大小应为 p_memsz，其中 p_filesz 个字节来自 ELF 文件。写入一个段后，将该扇区（512 字节）填 0 补齐。(理论上，按扇区补齐的动作本应在每个文件的所有段写入完成后进行，但在制作镜像时使用的 bootblock 与 kernel 正常情况下均只有一个非空的段，因此可以进行这样的简化。)
- void write_os_size(int nbytes, FILE * img) 将 kernel 段的大小（双字节整数，按扇区计）写入镜像中 OS_SIZE_LOC 处，供引导块执行时取用。

在–extended 选项启用时需要打印出相关信息（可参考 start_code 中 createimage 程序运行的输出)。

做 A-core 和 C-core 的同学必须完成：

在实验检查时，我们将使用一个额外的大 kernel 来检查大家的 createimage 是否完备，这个 kernel 会比大家自己编译出来的 kernel 大几倍，超出一个扇区的大小。在这种情况下，同学们需要考虑你的 createimage 程序是否支持各种各样的 kernel 大小。这个功能在后续的实验中是非常必要的，因为我们后面的 kernel 也会变得越来越大。

实验总结

在本实验中大家都自己手写了一个 crerateimage 镜像制作工具，想必大家都知道什么是 ELF 文件，了解 ELF 文件的文件头、程序头等结构体，并且知道 createimage 工具是如何合并多个 ELF 文件的。

接下来，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。

5.6 任务 4：C-Core 设计——双系统的启动引导

在实际中，一台计算机可能安装了不止一个操作系统，如有些同学可能尝试过安装 Windows+Linux/Mac 双系统。那么多系统计算机上电启动的时候，往往会出现一个选择进入哪个操作系统的界面。如 GNU 的 Grub 或 Windows 启动管理器就是这样的实现，它允许用户可以在计算机内同时拥有多个操作系统，并在计算机启动时选择希望运行的操作系统。本任务只需要做 C-Core 的同学完成。

图 P1-7: Windows 启动管理器

在本次的 C-Core 设计中，我们需要实现一个具有上述双系统引导能力的 bootblock，效果是当板卡上电启动时，询问用户输入从而决定拷贝并进入哪一个内核。

实验要求

在完成 A-Core 代码的基础上，自行修改 bootblock.S 和 createimage.c，并准备两个内核（记为 kernel0 和 kernel1），启动后实现双系统选择引导的功能：当板卡上电执行到 bootblock 时，不直接拷贝内核，而是询问用户输入。若输入'0' 字符（或其它指定的字符），则拷贝 kernel0 到内核入口，若输入其它字符，则拷贝 kernel1 到内核入口。然后再跳转到内核入口。

要点讲解

我们可以把本节的任务分为三个子任务实现：

1. 首先，如果你的 createimage 不支持合并两个以上的二进制文件为 image，那么就需要修改 createimage.c。因为两个系统的二进制文件再加上 bootblock，是 3 个二进制文件的合并。

2. 准备两个测试的内核。为了测试，你可以让两个内核打印不同的字符串以区分它们。至于如何编译，以及用它们制作 image，同学们可以参考 start_code 中的 Makefile。

3. 在 bootblock 加入读取串口输入的功能：当没有接收到字符时自旋等待。当接收到'0' 字符（或其它指定的字符），则拷贝 kernel0 并跳转到内核入口。若接收到其它字符，则拷贝 kernel1 并跳转到内核入口。

还有一点需要同学们思考，我们在 bootblock 里拷贝内核时，需要知道内核在 SD 卡的**起始扇区**和**所占扇区长度**。我们是如何获取这些信息的呢？

当时我们只有一个内核，因此我们当然知道内核起始扇区是 1（bootblock 一定只占用第一个扇区）。对于扇区长度，我们规定了一个地址 os_size_loc，在 createimage 里向这里写入内核占用了几个扇区。所以我们在 bootblock 拷贝内核时，从 os_size_loc 地址处读取扇区长度，就可以拷贝内核了。

那么当 image 里含有两个内核的时候，我们怎么知道这两个内核的起始扇区和占用扇区长度呢？显然之前用的办法是不够的，至于现在每个内核的起始扇区和占用扇区长度存放哪里，需要由同学们自己设计。

参考文献

[1] 阮一峰, "Make 命令教程." http://www.ruanyifeng.com/blog/2015/02/make.html, 2004. [Online; accessed 31-August-2021].

[2] 陈皓, "如何调试 makefile 变量." https://coolshell.cn/articles/3790.html, 2004. [Online; accessed 31-August-2021].

[3] 陈皓, "跟我一起写 makefile." https://blog.csdn.net/haoel/article/details/2886, 2004. [Online; 网友整理版: https://seisman.github.io/how-to-write-makefile/index.html].

[4] A. B. Palmer Dabbelt, Michael Clark, "Risc-v assembly programmer's manual." https://github.com/riscv/riscv-asm-manual/blob/master/riscv-asm.md, 2019. [Online; accessed 27-August-2021].

[5] R.-V. Foundation, "Risc-v instruction set reference." https://rv8.io/isa.html, 2019. [Online; accessed 31-August-2021].