# 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、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，但是“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”。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操作系统开发涉及的相关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。这个实验课，可能是同学们第一次面对一个如此复杂的系统设计。

操作系统是把握计算机系统全局的中枢，希望同学们学完这门课，能够讲清楚自己设计的小操作系统里的每一行代码的功能。

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、镜像文件的制作等内容。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、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，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），搬运并展开到内存中。

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 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)。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 [1]，板上有 ARM 和 FPGA，RISC-V 核是烧写在 FPGA 里的。在开发板上电时由 ARM 核启动相关程序 (BOOT.bin)，根据板卡上开关的状态，将 RISC-V 处理器核烧入 FPGA。之后，RISC-V 核自动加载相关程序。课程中所使用的 RISC-V 核为 NutShell[2]。NutShell 为国科大第一届“一生一芯”计划的产出，已在 Github 等网站上开源。在 PYNQ 板卡上，时钟主频为 40MHz。由于资源限制，我们提供的 RISC-V 核心均没有浮点模块。详细的开发板介绍会在同学们拿到板卡之后向大家说明。在没有拿到板卡之前大家是在 QEMU[3] 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。

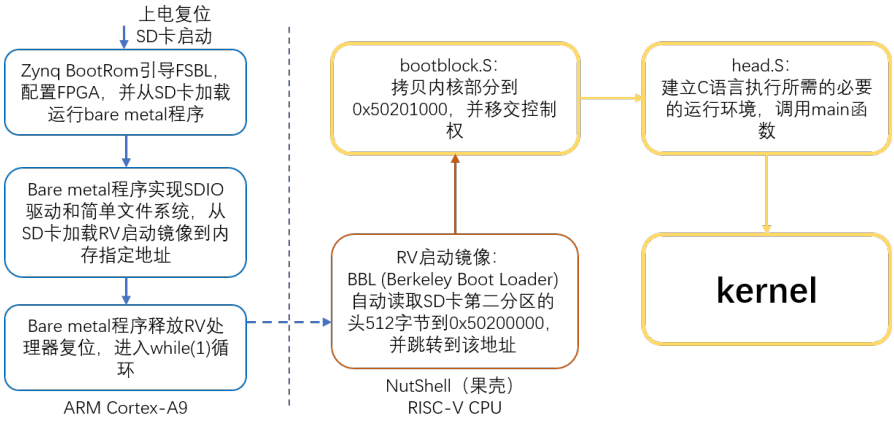


图 P1-1: 实验整体启动流程

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 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，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。

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，内存空间 0x50000000-0x501FFFFF 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

| 地址范围 | 权限 | 作用 |
|---|---|---|
| 0x38000000-0x3800FFFF | ARW | clint |
| 0x3C000000-0x3FFFFFFF | ARW | interrupt-controller |
| 0x50000000-0x5FFFFFFF | RWXC | memory |
| 0xE0000000-0xE0000FFF | RW | serial |
| 0xE000B000-0xE000BFFF | RW | ethernet |
| 0xE0100000-0xE0100FFF | RW | sdio |
| 0xF8000000-0xF800FFFF | RW | slcr |

表 P1-1: 地址空间

以使用的空间为 0x50200000-0x5FFFFFFF 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!”。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。

#### 文件说明

见表P1-2所示。

#### 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 Boot Loader!”。
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，执行命令 cd build && ./createimage –extended bootblock main && cd ..，以生成镜像文件 image。
5. 执行 make run 命令，在 qemu 上能看到屏幕输出字符串”It’s Boot Loader!”。

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 Boot Loader!”说明实验完成。

#### 要点讲解

任务 1 中需要补全的位置都用// TODO:[p1-task1] 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/项目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

| 编号 | 文件/文件夹 | 说明 |
| --- | --- | --- |
| 1 | arch/riscv 文件夹 | RISC-V 架构相关内容，主要为汇编代码以及相关宏定义<br>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<br>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<br>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<br>include：头文件，包含一些宏定义，不需要修改<br>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<br>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<br>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<br>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<br>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<br>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 P1-2: P1 文件说明

用的时候，只需要将参数依次放入 a0、a1、a2 ... 寄存器中，然后 call bios_func_entry 即可，可供调用的接口类型定义在 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。

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_sdread(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)
- 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 3.2** 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并展开到内存的部分，实际上，我们现在还没有写操作系统代码，也没有进行镜像文件的制作。

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，制作镜像文件**。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 镜像文件

接下来，我们要做一个完整的镜像文件了。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 镜像文件组成

对于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。

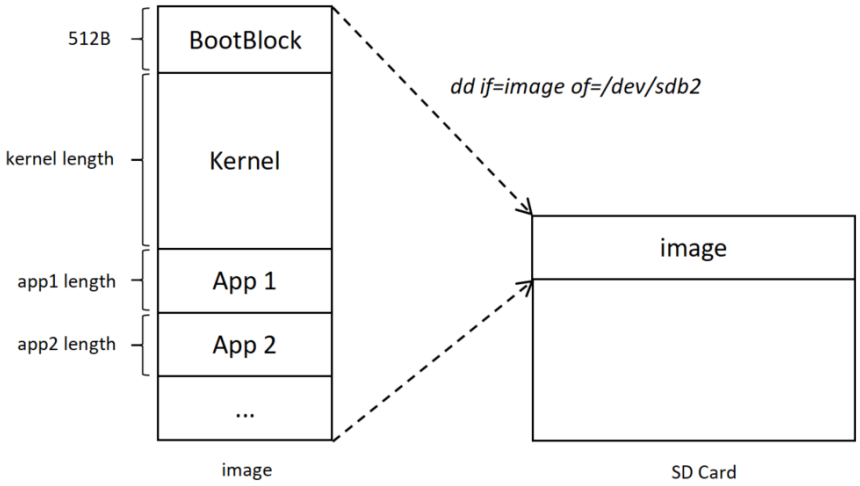


图 P1-2: Boot Loader、Kernel 的位置

关于镜像文件的制作，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）编译用户程序，4）使用镜像制作工具 creatimage 合并 Boot Loader、Kernel 和用户 ELF 文件，生成最终的镜像文件。

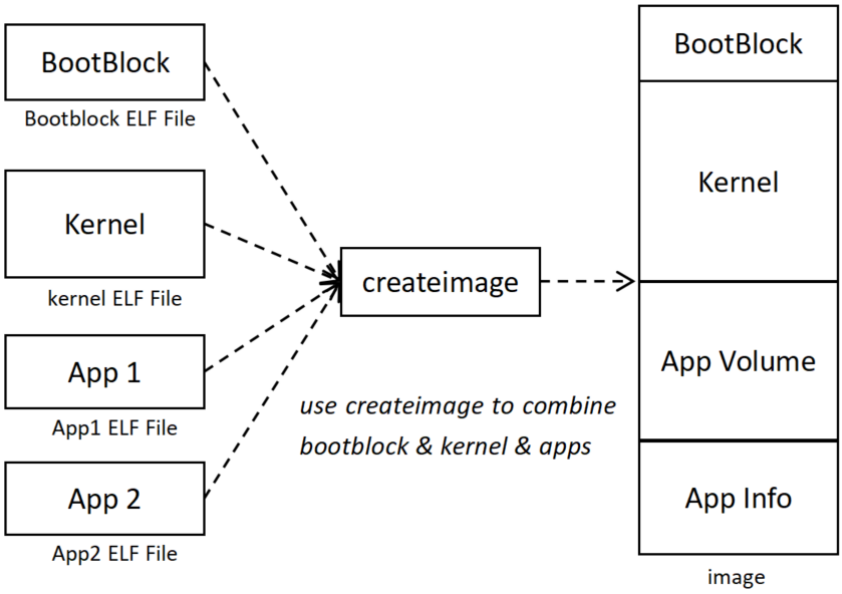


图 P1-3: 镜像生成过程

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，详细阐述镜像文件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，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 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）。

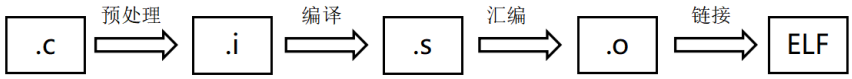


图 P1-4: 编译流程

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。

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

### 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!
```

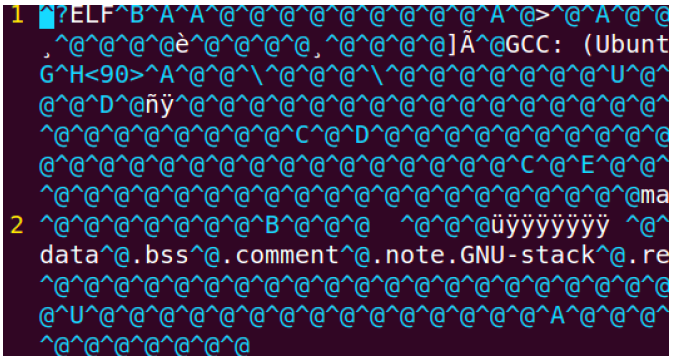


图 P1-5: 编译出的 main.o

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愚蠢的举动。

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 [4][5][6]，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。

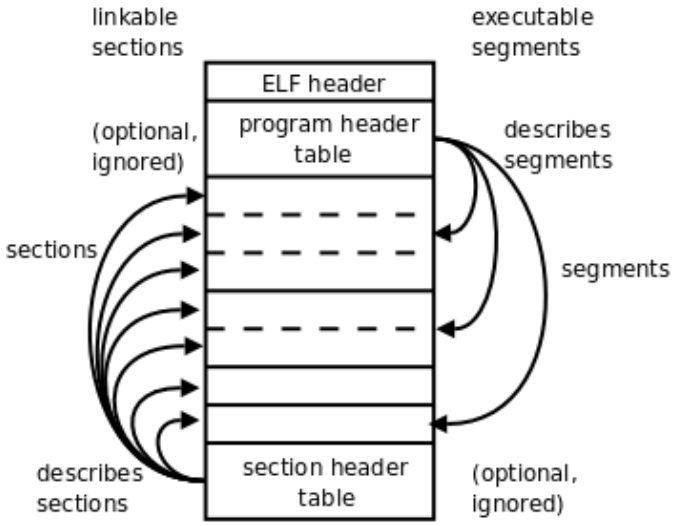


图 P1-6: ELF 文件结构

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
  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。

此外，在汇编或者 C 代码中，链接器脚本里面定义的符号是可以直接引用的，例如表示数据段起始位置的符号 ___DATA_BEGIN___。但是请注意，只有这个符号的地址是有意义的，因为这个地址里面并没有存放什么有意义的值。所以在汇编中，我们可以使用 la 指令来加载其地址；在 C 语言中，可以先声明 extern 之后再使用 & 获得其地址。

### 4.6 跳转表

对于操作系统来说，一个非常重要的特性就是可移植性。在本学期的实验课中，同学们的 UCAS-OS 是基于 RISC-V 指令集架构而实现的，但大家肯定不希望自己辛辛苦苦设计的操作系统只能运行在 RISC-V 架构的处理器上。如果底层处理器更换了一种新的指令集架构，比如 MIPS 架构，那么我们希望付出尽量少的改动来移植我们的操作系统，而实现跳转表的目的之一正在于此。

跳转表由内核进行初始化，将硬件提供给操作系统的 ABI 入口地址放入各个表项内，例如串口输入输出、SD 卡读写等操作。在 RISC-V 架构中，这些 ABI 是通过 SBI 调用来实现的，而 MIPS 架构中没有 SBI 调用，因此要移植操作系统的话，就势必要对这些函数进行修改。但如果操作系统里面大量使用了这些 SBI 调用来操作串口、SD 卡的话，移植的代价就会显著增大。因此，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 API 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，不需要面对繁杂的操作系统源代码，从而降低了移植操作系统的开销。

实现跳转表的另一个目的在于: 在 Project 2 实现系统调用之前，用户程序实际上都是内核进程，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，从而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。

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。

### 4.7 任务 2：开始制作镜像文件

#### 实验要求

掌握镜像文件的制作步骤，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

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: 2”之后，调用跳转表 API 读取屏幕输入，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: 2”，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: 2”；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 BIOS API，因为跳转表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。
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 head.S 的内容自己思考。

#### 要点讲解

制作镜像文件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 [7] 和 [8]。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。我们提供的 createimage 文件会把 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。为了与之兼容，我们希望同学们在任务 3 里面也这样做。

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 main.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 0x51000000，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 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。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。

#### 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，虽然它只能输出几个字符串，但在接下来的数个 Project 中，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 createimage 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 文件主要由文件头、段表头、程序头组成。

### 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、又连在一起的节 (section) 叫做段 (segment)，操作系统正是按照“segment”来装载可执行文件的。描述这些“segment”的结构叫做程序头 (program header)，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;
```

值得一提的是，程序头表内的 p_filesz 表示当前 segment 在 ELF 文件中所占的大小，p_memsz 表示当前 segment 被搬运并展开到内存中所占用的大小。同学们可能会有疑问，为什么要定义两个表示大小的变量呢？难道 p_memsz 还会大于 p_filesz 不成？

事实上，确实存在这样的情况。因为一个 segment 是由若干个 section 组成的，而对于其内数据都为 0 的 bss 段而言，没必要在浪费宝贵的文件空间来保存一段全为 0 的数据，因此 ELF 文件的设计者想出了一个聪明的办法：只要让 p_memsz 大于等于 p_filesz，多余的部分由操作系统在装载 ELF 文件的时候自动填 0 就行了。这样做既节省了文件存储空间，也满足了 bss 段内起始数据为 0 的假设。

那么，会有 p_filesz 大于 p_memsz 的情况吗？这种情况是不存在的，如果 p_filesz 大于 p_memsz 的话，那么就说明 segment 中存在着冗余数据，对于惜“存储”如金的 ELF 设计者来说，这比要了他们的命还难受！

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

## 5.5 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 bootblock 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，用户程序的相关信息和各个用户程序接续存储。同时，在任务三中 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）。加载所需的函数 load_task_img 位于 loader.c 中，任务三可以使用 task id 作为传入参数。

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 Project2 中会作为一个检查点，请同学们在装载时额外注意一下。

### 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. 编写 createimage.c、loader.c、crt0.S 代码，完善 bootblock.S。
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，我们也可以每次写完程序，编译的时候和 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 Project1 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（也可以由 loader 代劳）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，用户程序肯定不是只用一条跳转指令就能返回内核的，并且也不用在启动时创建栈帧、退出时回收栈帧。在后续的 Project3 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。

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。

#### 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 中的程序加载执行。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，并且知道用户程序与内核之间的基本关系。

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。

### 5.6 任务 4：镜像文件压缩

#### 实验要求

本任务是做 A-core 和 C-core 的同学必须完成的。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，这在实际生活中显然是不被允许的。

因此，在本任务中，除了 bootblock 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，任务三中使用 task id 来装载并启动用户程序。但当用户程序多起来之后，内核编写者面对着一个陌生的 task id 也会茫然失措，不知道这个 task id 究竟对应着哪一个用户程序。因此，在任务四中，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。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。

#### 要点解读

由于在本次实验中，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。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。

同时，由于 kernel 有可能和用户程序共享一个扇区，因此原来在任务三中根据内核扇区数读取内核的逻辑也需要修改。这一点也交给大家进行实现了。

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。

## 5.7 任务 5：批处理运行多个用户程序

### 实验要求

本任务是做 C-core 的同学必须完成的。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。

根据内存中允许存放的作业数，批处理操作系统又分为单道批处理系统和多道批处理系统两种类型。早期的批处理系统属于单道批处理系统，其目的在于减少作业间转换的人工操作，即当前正在运行的作业才能驻留内存，作业的执行顺序是先进先出。而对于多道批处理系统而言，其内存中可同时存在若干道作业，作业执行的次序与进入内存的次序无严格的对应关系，因为这些作业是通过一定的作业调度算法来使用 CPU 的，一个作业在等待 I/O 处理时，CPU 调度另外一个作业运行，因此 CPU 的利用率显著地提高了。

在本任务中，选做 C-Core 的同学需要实现一个简单的单道批处理系统来顺序依次执行多个用户程序。为了定义程序执行的“顺序”，大家需要引入一个额外的批处理文件，例如.txt 文件，而不是将“顺序”定义在内核中。否则，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。同时，批处理操作系统不具有交互性，它是为了提高 CPU 的利用率而提出的一种操作系统，因此在任务 5 中大家不能通过交互式界面输入作业的执行顺序，最多只能通过交互式界面选择需要使用哪一个批处理文件。

此外，对于 C-core 而言，同学们需要自己编写能够体现批处理执行顺序的用户程序。

### 要点解读

要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。另外，各个程序的执行顺序是在内核中通过读取 image 镜像文件的某块区域来决定的，这个区域的内容会在 createimage 中做好，比如内容为 1432，则按照这个编号顺序来执行程序；区域内容也可以是程序名字的组合，由同学们自己决定。本任务要求一个程序可以多次出现，也可以不出现在序列中。序列的长度也是可变的。

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。

# 参考文献


[1] Xilinx, "Pynq-z2 - tul embedded." https://www.tulembedded.com/FPGA/ProductsPYNQ-Z2.html, 2021. [Online; accessed 28-August-2022].

[2] 王华强等, "Nutshell 文档." https://oscpu.github.io/NutShell-doc/, 2020. [Online; accessed 28-August-2022].

[3] F. Bellard, "Qemu." https://www.qemu.org/, 2022. [Online; accessed 28-August-2022].

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

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

[6] 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].

[7] 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].

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