一、引言

制作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，还会涉及到网络等内容，可以说是需要“上知天文，下知地理”。

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，不会的，那就借此机会学会，如果会的，那就更加熟悉的去运用和掌握。

在本此实验，我们将从操作系统的引导开始，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。

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，我们将从理论到实践，迈出我们制作属于我们自己的操作系统的第一步。

二、操作系统的引导

2.1 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。

那么问题来了，Boot Loader 又是由谁搬运到内存里的呢？其实，对于龙芯处理器而言，这部分工作是由 PMON 完成的，关于 PMON 的介绍在这里大家可以不深究，否则可能会糊涂，通俗而言，它就是开发板自带的 BIOS，大家可以简单的认为 PMON 就是上电后直接存在于内存的一些代码，我们可以直接使用它们完成一些必要的操作。

2.2 引导的过程

下面我们将一起了解一下引导的过程，这个过程分为 3 个阶段：**BIOS 阶段、Boot Loader 阶段、操作系统阶段**。这三个阶段串行执行，依次递进，密不可分。

1）**BIOS 阶段**：在 CPU 上电后，执行地址自动会跳转到一个位置，这个位置就是 PMON 的一处可执行代码，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 0xa0800000**）。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，控制权从 PMON 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 调用读取 SD 上的操作系统内核，并放置到内存的制定位置，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。

2.3 任务 1：第一个引导块的制作

2.3.1 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 PMON 中的 BIOS，在终端成功输出“It’s Boot Loader!”。

3.3.2 文件说明

编号	文件名称	文件说明
1	bootblock.s	引导程序，接下来将在任务 2 中填写打印代码， 在任务 3 中添加移动内核代码
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	内核入口代码，任务 3 中完成
6	createimage.c	引导块工具代码，任务 4 中需要实现

2.3.3 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。

```
parallels@ubuntu:/media/psf/Home/buffer/mips_os/project_1/finished_code$ make all
gcc createimage.c -o createimage
mipsel-linux-gcc -G 0 -O2  -fno-pic -mno-abicalls -fno-builtin -nostdinc -mips3 -
 -nostdlib -e main -Wl,-m -Wl,elf32ltsmip -T ld.script
mipsel-linux-gcc -G 0 -O2  -fno-pic -mno-abicalls -fno-builtin -nostdinc -mips3 -
dlib -Wl,-m -Wl,elf32ltsmip -T ld.script
./createimage --extended bootblock kernel
bootblock size is 272 byte!
Bootblock messager :
Bootblock image memory size is 0x110
Bootblock image memory offset is 0x60
kernel messager :
kernel image memory size is 0x90
kernel image memory offset is 0x60
kernel sector size is 0x1
```

（3）使用 make floppy 命令将 bootblock 写到 SD 卡的第一个扇区。

```
parallels@ubuntu:/media/psf/Home/buffer/mips_os/project_1/finished_code$ make floppy
sudo fdisk -l /dev/sdb
[sudo] parallels 的密码：
Disk /dev/sdb: 3.7 GiB, 3904897024 bytes, 7626752 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x00000000

设备        启动 Start    末尾    扇区 Size Id 类型
/dev/sdb4            0 33554431 33554432  16G  0 空
sudo dd if=image of=/dev/sdb conv=notrunc
记录了2+0 的读入
记录了2+0 的写出
1024 bytes (1.0 kB, 1.0 KiB) copied, 0.0219983 s, 46.5 kB/s
```

（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 PMON 界面后，输入 loadboot 命令，当屏幕可以打印出字符串“It’s Boot Loader!”说明实验完成。

```
PMON> loadboot
new_sdcard_open is 0x8007a700, new_sdcard_read is 0x8007b1a8
sdcard_open
clk = 63000000 div_tmp = 1 bit = 1
SDcard Init Success
```

```
49 74 27 73 20 62 6f 6f 74 62 6c 6f 63 6b 0d 0a
10 00 00 00 00 80 e4 bf a8 b1 07 80 00 02 80 a0
00 02 80 a0 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 aa
It's bootblock
```

2.3.4 注意事项

（1）以下为本次实验需要使用了 BIOS 函数地址

函数	参数	地址	功能
read_sd_card(addr, offset, size)	addr：要移动数据到内存的位置 offset：要移动数据在 SD 卡的偏移量 size：要读取数据的大小	0x8007b1cc	从 SD 卡读数据到内存
printstr(str)	str：要打印的字符串首地址	0x8007b980	打印字符串
printch(ch)	ch：要打印的字符	0x8007ba00	打印字符

（2）关于如何在汇编中调用函数，请掌握 MIPS 汇编中函数的调用，思考传入的参数放到哪个寄存器，返回参数返回到哪个寄存器。

2.3.5 实验总结

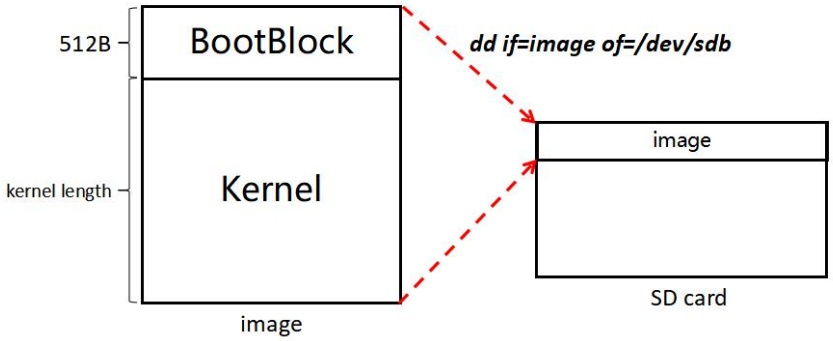
该实验仅仅是在引导程序中让大家实现简单的打印任务，还没有涉及调用 BIOS 将操作系统代码搬运到内存的部分，实际上，我们现在还没有写操作系统代码，也没有进行内核镜像的制作。

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。

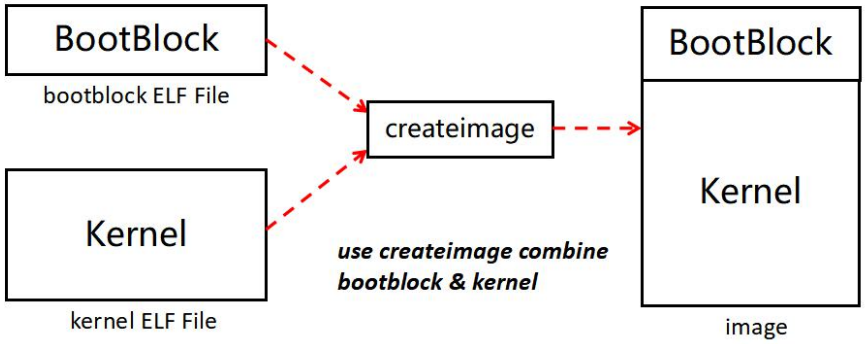
三、内核镜像

3.1 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 卡的位置如下：



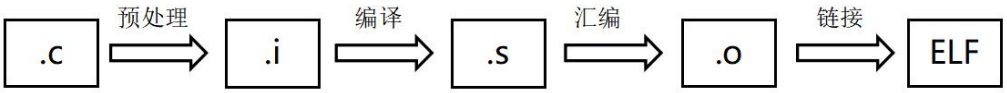
关于内核镜像的制作，我们是采用以下的步骤完成的：1）编译 Boot Loader，2）编译 Kernel，3）使用镜像制作工具 creatimage 合并 Boot Loader 和 Kernel 代码，生成镜像文件。



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！

3.2 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**。



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，具体请见附件，里面有 hello.h、hello.c、main.c 三个文件。这 3 个文件主要完成输出”hello world”的简单工作。

3.2.1 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：

```
parallels@ubuntu:~/hello_world$ gcc -E hello.c -o hello.i
parallels@ubuntu:~/hello_world$ 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，里面的内容如下：

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

3.2.2 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：

```
parallels@ubuntu:~/hello_world$ gcc -S hello.i -o hello.s
parallels@ubuntu:~/hello_world$ 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：

```
    .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
    movl    $0, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.10) 5.4.0 20160609"
    .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。

3.2.3 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：

```
parallels@ubuntu:~/hello_world$ gcc -c hello.i -o hello.o
parallels@ubuntu:~/hello_world$ gcc -c main.i -o main.o
```

打开生成的.o 文件，我们发现里面内容如下：

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~

3.2.4 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：

```
parallels@ubuntu:~/hello_world$ gcc  hello.o main.o -o main
parallels@ubuntu:~/hello_world$ ./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。

3.3 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。

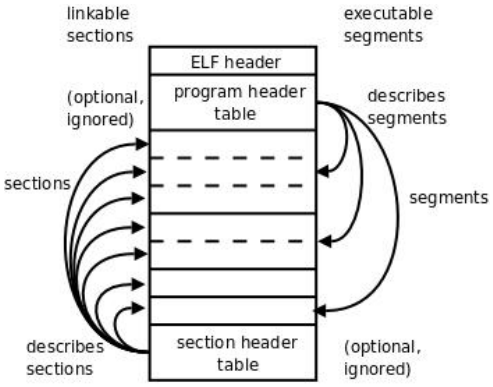
在本次操作系统实验课中，大部分的项目代码我们都已经写好了 Makefile 供大家使用，但是如果你仍希望更加深入的理解 Makefile，熟悉掌握相关知识的话，可以查阅网上给多的资料，了解如何编写 Makefile。

3.4 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。



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：

```
parallels@ubuntu:~/test$ objdump -h main

main:     文件格式 elf64-x86-64

节：
Idx Name          Size      VMA               LMA               File off  Algn
  0 .interp       0000001c  0000000000400238  0000000000400238  00000238  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag 00000020  0000000000400254  0000000000400254  00000254  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .note.gnu.build-id 00000024  0000000000400274  0000000000400274  00000274  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .gnu.hash     0000001c  0000000000400298  0000000000400298  00000298  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .dynsym       000000c0  00000000004002b8  00000000004002b8  000002b8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .dynstr       00000056  0000000000400378  0000000000400378  00000378  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .gnu.version  00000010  00000000004003ce  00000000004003ce  000003ce  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .gnu.version_r 00000020  00000000004003e0  00000000004003e0  000003e0  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .rela.dyn     00000018  0000000000400400  0000000000400400  00000400  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  9 .rela.plt     00000090  0000000000400418  0000000000400418  00000418  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 10 .init         0000001a  00000000004004a8  00000000004004a8  000004a8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 11 .plt          00000070  00000000004004d0  00000000004004d0  000004d0  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 12 .plt.got      00000008  0000000000400540  0000000000400540  00000540  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .text         00000342  0000000000400550  0000000000400550  00000550  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 14 .fini         00000009  0000000000400894  0000000000400894  00000894  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 15 .rodata       0000001d  00000000004008a0  00000000004008a0  000008a0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 16 .eh_frame_hdr 00000034  00000000004008c0  00000000004008c0  000008c0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 17 .eh_frame     000000f4  00000000004008f8  00000000004008f8  000008f8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 18 .init_array   00000008  0000000000600e10  0000000000600e10  00000e10  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 19 .fini_array   00000008  0000000000600e18  0000000000600e18  00000e18  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 20 .jcr          00000008  0000000000600e20  0000000000600e20  00000e20  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 21 .dynamic      000001d0  0000000000600e28  0000000000600e28  00000e28  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 22 .got          00000008  0000000000600ff8  0000000000600ff8  00000ff8  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 23 .got.plt      00000048  0000000000601000  0000000000601000  00001000  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 24 .data         00000010  0000000000601048  0000000000601048  00001048  2**3
                  CONTENTS, ALLOC, LOAD, DATA
```

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。

3.5 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。

3.6 任务 2：开始制作内核镜像

3.6.1 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”。

3.6.2 文件说明

继续使用任务 1 的项目代码。

3.6.3 实验步骤

（1）补全 bootblock.s 文件中的代码，添加的内容为调用 BIOS 将位于 SD 第二个扇区的内核代码段移动至内存。

（2）补全 kernel.c 文件中的代码，添加的内容为调用 BIOS，输出字符串“Hello OS”。

（3）运行 make all 命令进行交叉编译，生成二进制文件生成镜像。

```
parallels@ubuntu:~/mips_os/project_1/task_3$ make
mipsel-linux-gcc -G 0 -O2  -fno-pic -mno-abicalls -fno-builtin -nostdinc
Wl,-m -Wl,elf32ltsmip -T ld.script
mipsel-linux-gcc -G 0 -O2  -fno-pic -mno-abicalls -fno-builtin -nostdinc
2ltsmip -T ld.script
```

（4）运行 make floppy 命令，将 image 写到 SD 卡的第一个扇区。

```
parallels@ubuntu:/media/psf/Home/buffer/mips_os/project_1/finished_code$ make floppy
sudo fdisk -l /dev/sdb
[sudo] parallels 的密码：
Disk /dev/sdb: 3.7 GiB, 3904897024 bytes, 7626752 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x00000000

设备       启动 Start    末尾    扇区 Size Id 类型
/dev/sdb4           0 33554431 33554432  16G  0 空
sudo dd if=image of=/dev/sdb conv=notrunc
记录了2+0 的读入
记录了2+0 的写出
1024 bytes (1.0 kB, 1.0 KiB) copied, 0.0219983 s, 46.5 kB/s
```

（5）将 SD 卡插入到板子上，使用 minicom 连接，然后 restart 开发板。

（6）进入到 PMON 界面后，输入 loadboot 命令，当屏幕可以打印出字符串“Hello OS”说明实验完成。

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 aa
It's bootblock
Hello OS!
```

3.6.4 注意事项

（1）将内核从 SD 卡拷贝到内存中需要使用 PMON 提供的 BIOS，函数的入口地址请见任务 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 0xa0800000，因此我们将内核拷贝到它的后面，也就是 0xa0800200。

（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 0xa0800200。至于我们是怎么放的，大家可以阅读链接器脚本文件 ld.script 的内容以及 kernel.c 的内容自己思考。

3.6.5 附加题

在上述任务中，我们将内核移动到了 0xa0800200，正好放到了 bootloader 的后面。在本次附加题中，我们希望将内核拷贝到 0xa0800000 中，并从 0xa0800000 处开始执行内核代码。

注意：因为内核需要拷贝到 0xa0800000 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。

3.6.6 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。

四、ELF 文件

4.1 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。

4.2 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：

```
65 #define EI_NIDENT (16)
66
67 typedef struct
68 {
69   unsigned char e_ident[EI_NIDENT]; /* Magic number and other info */
70   Elf32_Half    e_type;         /* Object file type */
71   Elf32_Half    e_machine;      /* Architecture */
72   Elf32_Word    e_version;      /* Object file version */
73   Elf32_Addr    e_entry;        /* Entry point virtual address */
74   Elf32_Off e_phoff;        /* Program header table file offset */
75   Elf32_Off e_shoff;        /* Section header table file offset */
76   Elf32_Word    e_flags;        /* Processor-specific flags */
77   Elf32_Half    e_ehsize;       /* ELF header size in bytes */
78   Elf32_Half    e_phentsize;        /* Program header table entry size */
79   Elf32_Half    e_phnum;        /* Program header table entry count */
80   Elf32_Half    e_shentsize;        /* Section header table entry size */
81   Elf32_Half    e_shnum;        /* Section header table entry count */
82   Elf32_Half    e_shstrndx;     /* Section header string table index */
83 } Elf32_Ehdr;
84
85 typedef struct
86 {
87   unsigned char e_ident[EI_NIDENT]; /* Magic number and other info */
88   Elf64_Half    e_type;         /* Object file type */
89   Elf64_Half    e_machine;      /* Architecture */
90   Elf64_Word    e_version;      /* Object file version */
91   Elf64_Addr    e_entry;        /* Entry point virtual address */
92   Elf64_Off e_phoff;        /* Program header table file offset */
93   Elf64_Off e_shoff;        /* Section header table file offset */
94   Elf64_Word    e_flags;        /* Processor-specific flags */
95   Elf64_Half    e_ehsize;       /* ELF header size in bytes */
96   Elf64_Half    e_phentsize;        /* Program header table entry size */
97   Elf64_Half    e_phnum;        /* Program header table entry count */
98   Elf64_Half    e_shentsize;        /* Section header table entry size */
99   Elf64_Half    e_shnum;        /* Section header table entry count */
100  Elf64_Half    e_shstrndx;     /* Section header string table index */
101 } Elf64_Ehdr;
```

数据类型说明：

名称	大小	对齐	用途
Elf32_Addr	4	4	无符号程序地址
Elf32_Half	2	2	无符号中等大小整数
Elf32_Off	4	4	无符号文件偏移
Elf32_Sword	4	4	有符号大整数

使用文件头结构体需包含#include <elf.h>头文件。结构体 Elf32_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。

4.3 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：

```
565 /* Program segment header.  */
566
567 typedef struct
568 {
569   Elf32_Word    p_type;         /* Segment type */
570   Elf32_Off p_offset;       /* Segment file offset */
571   Elf32_Addr    p_vaddr;        /* Segment virtual address */
572   Elf32_Addr    p_paddr;        /* Segment physical address */
573   Elf32_Word    p_filesz;       /* Segment size in file */
574   Elf32_Word    p_memsz;        /* Segment size in memory */
575   Elf32_Word    p_flags;        /* Segment flags */
576   Elf32_Word    p_align;        /* Segment alignment */
577 } Elf32_Phdr;
578
579 typedef struct
580 {
581   Elf64_Word    p_type;         /* Segment type */
582   Elf64_Word    p_flags;        /* Segment flags */
583   Elf64_Off p_offset;       /* Segment file offset */
584   Elf64_Addr    p_vaddr;        /* Segment virtual address */
585   Elf64_Addr    p_paddr;        /* Segment physical address */
586   Elf64_Xword   p_filesz;       /* Segment size in file */
587   Elf64_Xword   p_memsz;        /* Segment size in memory */
588   Elf64_Xword   p_align;        /* Segment alignment */
589 } Elf64_Phdr;
```

4.4 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。

```
271 /* Section header.  */
272
273 typedef struct
274 {
275   Elf32_Word    sh_name;        /* Section name (string tbl index) */
276   Elf32_Word    sh_type;        /* Section type */
277   Elf32_Word    sh_flags;       /* Section flags */
278   Elf32_Addr    sh_addr;        /* Section virtual addr at execution */
279   Elf32_Off sh_offset;      /* Section file offset */
280   Elf32_Word    sh_size;        /* Section size in bytes */
281   Elf32_Word    sh_link;        /* Link to another section */
282   Elf32_Word    sh_info;        /* Additional section information */
283   Elf32_Word    sh_addralign;       /* Section alignment */
284   Elf32_Word    sh_entsize;     /* Entry size if section holds table */
285 } Elf32_Shdr;
286
287 typedef struct
288 {
289   Elf64_Word    sh_name;        /* Section name (string tbl index) */
290   Elf64_Word    sh_type;        /* Section type */
291   Elf64_Xword   sh_flags;       /* Section flags */
292   Elf64_Addr    sh_addr;        /* Section virtual addr at execution */
293   Elf64_Off sh_offset;      /* Section file offset */
294   Elf64_Xword   sh_size;        /* Section size in bytes */
295   Elf64_Word    sh_link;        /* Link to another section */
296   Elf64_Word    sh_info;        /* Additional section information */
297   Elf64_Xword   sh_addralign;       /* Section alignment */
298   Elf64_Xword   sh_entsize;     /* Entry size if section holds table */
299 } Elf64_Shdr;
```

4.5 任务 3：制作 createimage

4.5.1 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 位置处。
- 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。

4.5.2 文件说明

请继续使用之前的代码进行实现。

4.5.3 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 PMON 界面后，输入 loadboot 命令，当屏幕可以打印出字符串“It’s Boot Loader!”和“Hello OS”说明实验完成。

4.5.4 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。

4.5.5 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。