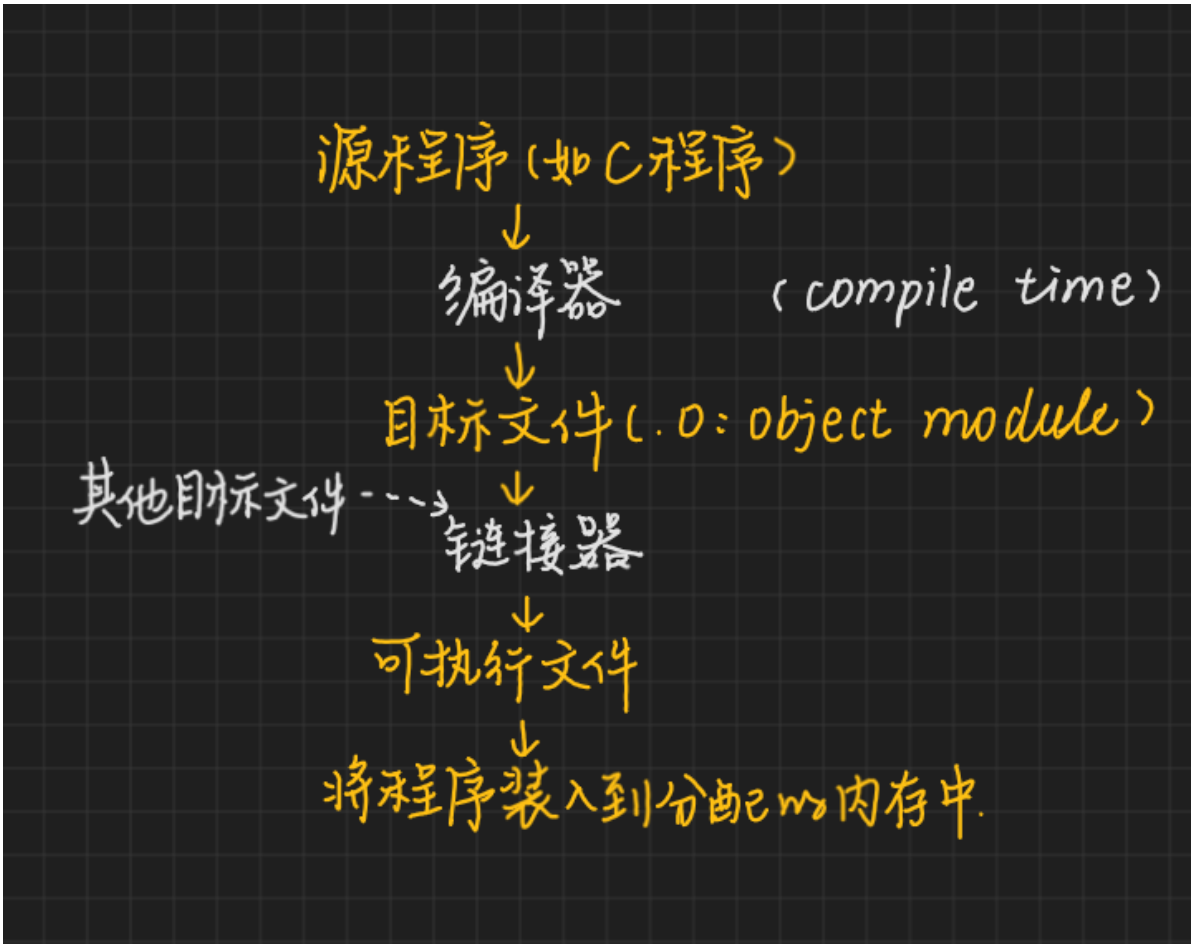
实验目的

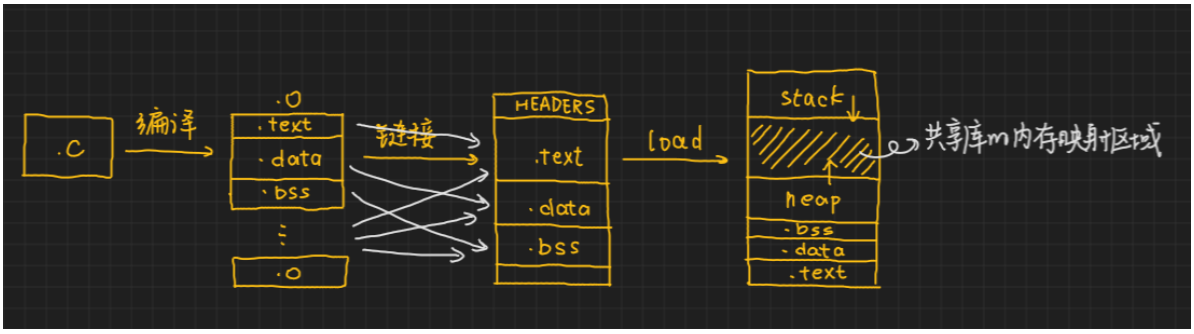
- 从操作系统角度理解 MIPS 体系结构；
- 掌握操作系统启动的基本流程；
- 掌握 ELF 文件的结构和功能；
- 具体实验内容是阅读并补全部分代码，使得MOS操作系统正常运行。

实验难点

事实上，从笔者做实验的过程出发，实验本身（即补全代码的部分）还是非常简单的，但中间对源代码的阅读、对实验原理的理解对笔者来说都有着不小的难度，有些地方理论课进度尚未跟上，只是阅读指导书的话也有些云里雾里。结合其他资料进行了理解后，笔者将在下面对初读时难以理解的点进行简单描述。

从C文件到可执行文件





之前只知道从程序到执行的过程应该要进行编译->链接->装载的过程以及分别需要使用的指令参数，但对其中的过程并不是很明晰，包括库函数在哪里被实现之类的问题也没有深入思考过。

利用objdump指令进行反汇编，可以对这其中的过程进行一个比较直观的“窥探”，这也是实验实践的一部分（即探索printf具体实现的部分）。再总结了一下理论课ppt得到上图，也可以直接看出，printf等库函数应该在“链接”的步骤实现。程序经过编译和链接后变成了可执行文件，可执行文件虽说由.data，.text和.bss这几个segment组成，但我们本质上还是可以把它的组成看成代码和数据两部分，其中代码是只读的，数据则是可读可写的。接下来，我们的操作系统把可执行文件加载到分配好的内存中，交给CPU去执行，就如最后一张图所示。但这个时候，我又陷入了“迭代学习“的困局中——什么是”共享库的内存映射区域“？事实上，这个看似冗长的名词其实就是另一个问题的答案：CPU执行的时候怎么访问代码和数据呢？回顾一下上学期计组课程的内容，这个问题就迎刃而解了——**这个区域是一些标准的系统库**，这个共享库在物理内存中只存储一份，每个进程将这个区域的虚拟地址映射到同一份共享库物理内存上。具体的访问过程，就是上学期学习过的内容。

ELF文件与实验源代码

指导书从目标文件的链接引入到ELF文件格式后，事情就变得复杂了起来。虽然需要我们补全代码的实验部分很简单，但说实话，要理解代码的其他关键部分对我来说仍是一件难事。不过好在课程组提供的ELF手册帮了我的大忙，许多阅读指导书时没能搞懂的问题在阅读了**ELF手册**后都变得晓畅了许多，这也使得我对自己通过“任务驱动型”过程写出来的实验代码有了更深的理解。

```c
/* 文件的前面是各种变量类型定义，在此省略 */
/* The ELF file header.  This appears at the start of every ELF file.  */
/* ELF 文件的文件头。所有的 ELF 文件均以此为起始 */
#define EI_NIDENT (16)

typedef struct {
    unsigned char   e_ident[EI_NIDENT];     /* Magic number and other info */
    // 存放魔数以及其他信息
    Elf32_Half      e_type;                 /* Object file type */
    // 文件类型
    Elf32_Half      e_machine;              /* Architecture */
    // 机器架构
    Elf32_Word      e_version;              /* Object file version */
    // 文件版本
    Elf32_Addr      e_entry;                /* Entry point virtual address */
    // 入口点的虚拟地址
    Elf32_Off       e_phoff;                /* Program header table file offset */
    // 程序头表所在处与此文件头的偏移
    Elf32_Off       e_shoff;                /* Section header table file offset */
    // 节头表所在处与此文件头的偏移
    Elf32_Word      e_flags;                /* Processor-specific flags */
    // 针对处理器的标记
    Elf32_Half      e_ehsize;               /* ELF header size in bytes */
    // ELF 文件头的大小（单位为字节）
    Elf32_Half      e_phentsize;            /* Program header table entry size */
    // 程序头表入口大小
    Elf32_Half      e_phnum;                /* Program header table entry count */
    // 程序头表入口数
    Elf32_Half      e_shentsize;            /* Section header table entry size */
    // 节头表入口大小
    Elf32_Half      e_shnum;                /* Section header table entry count */
    // 节头表入口数
```

思考题

Thinking 1.1

- -D意为从objfile中反汇编所有指令机器码的section
- -S意为尽可能反汇编出源代码，实际上隐含了-d参数
- 其余更详细的参数说明见：Linux objdump 命令用法详解-Linux命令大全（手册）(ipcmen.com)
- 重复编译过程：

```
extern int pclose (FILE *__stream);



extern char *ctermid (char *__s) __attribute__ ((__nothrow__ , __leaf__));
# 840 "/usr/include/stdio.h" 3 4
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));



extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;


extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
# 858 "/usr/include/stdio.h" 3 4
extern int __uflow (FILE *);
extern int __overflow (FILE *, int);
# 873 "/usr/include/stdio.h" 3 4

# 2 "testHello.c" 2

# 2 "testHello.c"
int main(){
 printf("Hello World!");
 return 0;
}
```

gcc -E testHello.c

```
testHello.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # f <main+0xf>
   f:   b8 00 00 00 00          mov    $0x0,%eax
  14:   e8 00 00 00 00          callq  19 <main+0x19>
  19:   b8 00 00 00 00          mov    $0x0,%eax
  1e:   5d                      pop    %rbp
  1f:   c3                      retq
```

gcc -c testHello.c

objdump -DS testHello.o > result.txt

```
Disassembly of section .init:

0000000000001000 <_init>:
    1000:   f3 0f 1e fa             endbr64
    1004:   48 83 ec 08             sub    $0x8,%rsp
    1008:   48 8b 05 d9 2f 00 00    mov    0x2fd9(%rip),%rax        # 3fe8 <__gmon_start__>
    100f:   48 85 c0                test   %rax,%rax
    1012:   74 02                   je     1016 <_init+0x16>
    1014:   ff d0                   callq  *%rax
    1016:   48 83 c4 08             add    $0x8,%rsp
    101a:   c3                      retq
```

```
0000000000001149 <main>:
    1149:   f3 0f 1e fa             endbr64
    114d:   55                      push   %rbp
    114e:   48 89 e5                mov    %rsp,%rbp
    1151:   48 8d 3d ac 0e 00 00    lea    0xeac(%rip),%rdi        # 2004 <_IO_stdin_used+0x
    1158:   b8 00 00 00 00          mov    $0x0,%eax
    115d:   e8 ee fe ff ff          callq  1050 <printf@plt>
    1162:   b8 00 00 00 00          mov    $0x0,%eax
    1167:   5d                      pop    %rbp
    1168:   c3                      retq
    1169:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)
```

```
Disassembly of section .plt:

0000000000001020 <.plt>:
    1020:   ff 35 9a 2f 00 00       pushq  0x2f9a(%rip)        # 3fc0 <_GLOBAL_OFFSET_TABLE_
8>
    1026:   f2 ff 25 9b 2f 00 00    bnd jmpq *0x2f9b(%rip)        # 3fc8 <_GLOBAL_OFFSET_TAB
+0x10>
    102d:   0f 1f 00                nopl   (%rax)
    1030:   f3 0f 1e fa             endbr64
    1034:   68 00 00 00 00          pushq  $0x0
    1039:   f2 e9 e1 ff ff ff       bnd jmpq 1020 <.plt>
    103f:   90                      nop
```

允许链接后编译出可执行文件，再反汇编

Thinking 1.2

使用`readelf -h`指令分别在各自的目录下查看vmlinux和testELF两个文件：

```
git@20373543:~/20373543/gxemul$ readelf -h vmlinux
ELF Header:
  Magic:   7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, big endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           MIPS R3000
  Version:                           0x1
  Entry point address:               0x80010000
  Start of program headers:          52 (bytes into file)
  Start of section headers:          37196 (bytes into file)
  Flags:                             0x1001, noreorder, o32, mips1
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         14
  Section header string table index: 11
```

```
git@20373543:~/20373543/readelf$ readelf -h testELF
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048490
  Start of program headers:          52 (bytes into file)
  Start of section headers:          4440 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         30
  Section header string table index: 27
```

我们很快就能发现Data处的相异——vmlinux为大端存储而testELF为小端存储。因此，答案是我们生成的readelf程序只能解析小端存储文件，所以无法解析vmlinux。

Thinking 1.3

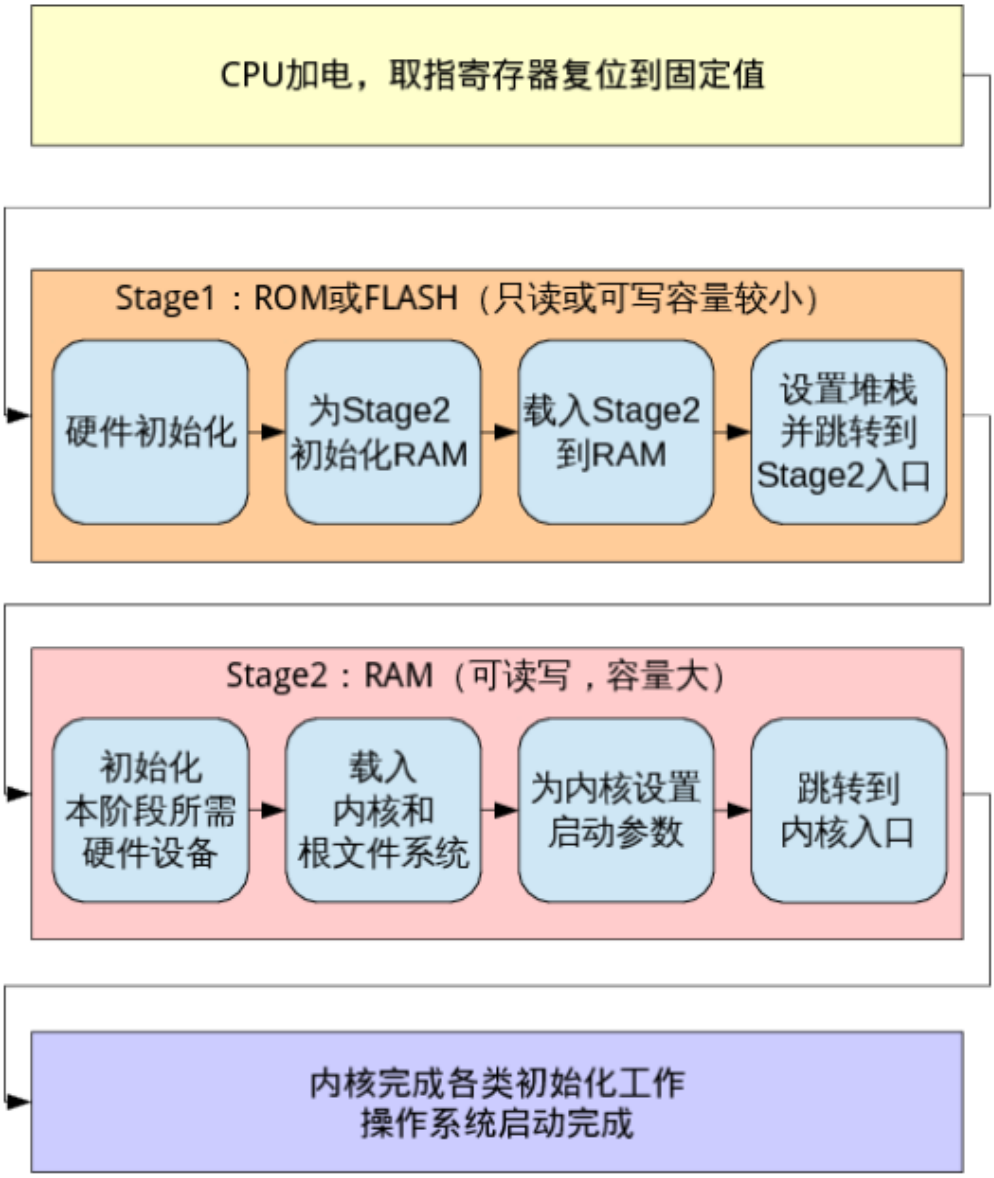


图 1.1: 启动的基本步骤

首先要明确的是，大多数 boot loader 都分为 stage1 和 stage2 两大部分。 stage 1 直接运行在存放 bootloader的存储设备上，为加载 stage 2 准备 RAM 空间，然后将 stage 2 的代码复制到 RAM 空间，并且设置堆栈，最后跳转到 stage 2 的入口函数。 stage 2 运行在 RAM 中，此时有足够的运行环境从而可以用 C 语言来实现较为复杂的功能。此时可以为内核设置启动参数，最后将 CPU 指令寄存器的内容设置为内核入口函数的地址，这样就可以保证内核入口被正确跳转到了。

Thinking 1.4

尽量避免安排重复的页面，即每个segment应与前面的segment所占的地址空间不重叠：若前一个segment地址占到页面x，此时的segment应安排到下一页开始占用。

Thinking 1.5

- 内核入口位于`0x80000000`，main函数位于`0x80010000`；
- 通过执行跳转指令`jal`，我们让内核进入main函数；
- 进行跨文件调用函数时，先把所需参数保存在`$a0-$a3`中，然后执行`jal`跳转到指定函数执行，将函数返回值存储在`$v0-$v1`中。

Thinking 1.6

```
    /* Disable interrupts */
    mtc0        zero, CP0_STATUS

    ......

    /* disable kernel mode cache */
    mfc0        t0, CP0_CONFIG
    and         t0, ~0x7
    ori         t0, 0x2
    mtc0        t0, CP0_CONFIG
```

- 首先将0写入CP0_STATUS位，以禁用全局中断；
- 然后将CP0_CONFIG位的值读到t0寄存器中，最后将计算处理过后的t0寄存器存储的值写入到CP0_CONFIG中。这一步是先禁用内核模式缓存来初始化cache，接着设置kseg0区经过cache。

实验体会

真正动手写代码的时间可能只有不到一个小时，但说实话，阅读指导书、源码和各类资料的时间远远不止于此，保守估计花了大概一整天，因此从本实验开始我真正意识到了操作系统课程的知识饱和度。

- 复杂工程代码的阅读与分析

  阅读源码是本次实验中对我来说挑战最大的部分，尤其是`start.S`与./readelf目录下的那几个文件，不过归根结底还是理论知识没有掌握扎实。以`kerelf.h`为例，其中的文件头内容在初读时让人很难从注释里解读出其释义，但回顾了理论课ppt的内存管理分配部分后就大概知道一些信息的用途，照猫画虎地完成了实验所需代码的补全后，也对一些变量的使用更加清晰（就比如说节头表的地址部分）。当然，理解这些也要感谢网络上的诸多资源（尤其是一篇mips start.S导读与初始化流程导读）。

- ELF手册

  我可以毫不犹豫地说，这本手册给我的帮助是最大的。即使是结合了理论课ppt，说实话指导书里的部分内容对我来说还是有点“谜语人”，但在助教提示查阅ELF手册后，一些文件中令人头大的部分就被解决了。不过还是得结合实验代码来进行理解。

  与其类似的还有填写printf相关代码时给出的C语言printf函数文档。不得不说，有很多细节都需要参阅各种文档/手册才能阐明，在今后的学习中也需要养成阅读、理解手册的好习惯才行。

- 一个无关紧要的细节

  指导书中对printf格式符的描述与`print.c`中给出的注释顺序好像有点不太一样（flag跟length的部分似乎反了），不过并不影响。