16 级计科 7 班: 操作系统原理实验 #3

Due on Monday, April 2, 2018

凌应标 周一 *9-10* 节

颜彬

16337269

Content

1 实验目的

把原来在引导扇区中实现的监控程序（内核）分离乘一个独立的执行体，存放那个在其他扇区中，为 ‘‘后来’’ 扩展内核提供发展空间。

学习汇编与 C 混合编程技术，改写实验二的监控程序，拓展其命令处理能力，增加实现实验要求 2 中的部分或全部要求。

2 实验要求

在实验二的基础上进行，保留或拓展原有功能，实现部分新增功能。

监控程序以独立的可执行程序实现，并由引导程序加载进内存适当位置，内核获得控制权后开始显示必要的操作提示信息，实现若干命令，方便使用者（测试者）操作。

制作包包含引导程序，监控程序和若干可加载并执行的用户程序组成的 1.44M 软盘映像。

3 实验方案

3.1 基础原理

3.1.1 GCC 与 NASM 的合作

计算机本质上只能运行对应架构的机器码。C 程序和汇编程序在运行时，都是先通过编译器将源码生成目标文件，再用链接器将若干个（或一个）目标文件链接成可执行文件（或纯二进制文件）。由于 C 语言生成的目标文件和汇编语言生成的目标文件本质上是一样的，这就允许我们将来自 C 和汇编的目标文件通过链接器连在一起，生成混编可执行文件。

实验一、二实质上是使用 32 位处理器的 16 位模式执行 16 位代码。NASM 产生的汇编的立即数和地址都是 16 位的。然而 GCC 只能产生 32 位的代码。如果强行将 16 位汇编和 32 位 GCC 汇编连接，会在运行时出现问题。32 位 GCC 汇编的立即数都是 32 位的，处理器会读取 32 位立即数的低 16 位作为立即数，把立即数的高 16 位看作是 ‘‘下一条指令的代码’’。

幸运的是，我们使用的处理器是 32 位的（只是模式是 16 位），其可以用某种方式处理 32 位代码：如果一个条汇编语句带有前缀 “66’’，处理器明白这条语句的立即数长度 ‘‘与当前模式不一致’’。由于当前模式是 16 位，故处理器知道这条语句的立即数是 32 位的。处理器在识别立即数时可以往后读取 32 个比特。类似地，前缀 67 表示 ‘‘地址’’ 长度与默认模式不一致。66,67 前缀可以理解成，告诉处理器 ‘‘暂时地’’ 切换到 32 位模式执行当前代码。

GCC 的编译选项 $-m16$ 可以自动地在所有（必要的）指令前加前缀 66，67。如此处理器即可正确地处理立即数长度和地址长度。注意到，$-m16$ 相当于在 C 代码的最开头内嵌一条汇编语句 $.code16gcc$。显然采用编译指令的方式比手动内嵌汇编指令方便得多。

所有的编译参数见4.2.1。

GCC 和 NASM 有一个很重要的技术细节，忽略将会产生严重的错误，见3.1.4节。

混编时运用内嵌汇编可以简化程序代码的书写。见第3.1.5小节。

3.1.2 链接器的作用

由于我们还无法实现解析 EXE 头和 ELF 头，所以连接器应增加 $-oformat\ binary$ 参数，以保证最终产生不带头的二进制文件。

在混编中，链接器代替了以前的 ORG 伪指令的作用。当然，因为链接器需要全权负责地址偏移量的计算，所以链接器不允许代码内出现任何的 ORG，否则将报错。

所有汇编程序先删除所有的 ORG 伪指令。在链接时通过 $-Ttext\ <address>$ 的方式决定代码段的偏移量。

链接主引导程序时，使用 $-Ttext\ 0x7C00$ 参数。

链接操作系统引导时，使用 $-Text\ 0x7E00$ 参数。此处假设主引导会把操作系统引导载入到内存 0x7E00 处。

链接操作系统内核时，使用 $-Text\ 0xA200$ 参数。此处假设主引导会把内核载入 0xA200 处。

所有的链接参数见 4.2.2节。

3.1.3 FAT 文件系统

本实验还实现了简单的 FAT16 文件系统，提供了若干接口，并完成了列出当前目录 ls, 运行文件 run，以及简单的文件权限检查等操作。操作系统引导借助文件系统的接口，在根目录下搜索 kernel.bin 文件，并将其载入到内存中。

在开机后，主引导程序首先将控制权交给操作系统引导程序。操作系统引导会从查找 BPB 表，找到第一个 FAT 表所在扇区和 FAT 表长度，把 FAT 表加载入内存; 还会找到数据块区中的 ‘‘根目录区’’，把它也载入到内存中。随后，操作系统引导会调用文件系统提供的接口，搜索根目录的前 5 个文件，查看其是否为 kernel.bin。

当操作系统引导找到根目录区中的 kernel.bin 项后，它会找到该项的 $cluster$ 和 $file\ size$。引导会再次读取 BPB 表，找到该 $cluster$ 所在扇区，计算内核的扇区数，把相关扇区载入到内存中。随后，操作系统引导将控制权交给内核。

至此，所有引导工作全部完成。内核接管所有的控制权。文件系统的布局图见1.

3.1.4 混编的坑点

第3.1.1节讨论了 GCC 和 NASM 混编的可行之处。但在细节上，它们的混编还有一个不易被留意的坑点。不注意的话，会导致程序完全错误。

GCC 产生的汇编本质是 32 位的，GCC 编译出的函数在 call 时，会向栈中压入 32 位地址。若 C 函数调用（16 位）汇编，汇编的 ret 指令只会出栈 16 位地址。在大部分情况下，这个错误不会被发现：因为 32 位的地址的高 16 位一般是 0, 只依靠低 16 位就可以正确跳转。但这个错误会带来极其严重的后果。栈指针在调用函数后没有返回到正确的值中。栈中被压入了多余的 16 个 0. 随后的所有访问栈操

作都会发生错误。

解决这个问题的方法很简单。如代码2所示。用 pop ecx; ret cx 的方式，手动出栈 32 个字节，并跳转 ecx 的低 16 个字即可。为了便于以后的代码书写，代码2把这几行代码定义成了宏。例如 pushl 和 calll 将执行 32 位的入栈和函数调用操作。retl 将执行 32 位的函数返回操作。

3.1.5 内嵌汇编的书写

GCC 支持 C 程序中内嵌汇编。这给编程带来了很大的帮助。例如操作系统内核 kernel.c 在开头内嵌了代码1所示的汇编。

开头的.globl _start 和 _start: 的作用是告知链接器程序的入口地址。第 3-5 行代码把 ds 和 es 设置为 0, 做了一些简单的初始化工作。最后一行的 jmpl $0, $main 直接跳转到 main 函数。这样，无论 kernel.c 在 main 函数前定义了多少数据、函数，都可以保证程序在执行时能第一时间跳转到 main 函数，避免执行错误的代码和数据。

___asm___ 语法也支持采用 intel 汇编来书写。方法是，采用 ___asm___(".intel_syntax noprefix n");。需要注意的是，在内嵌汇编执行完后，要手动把语法再改成 at&t 语法，否则其后的所有（GCC 生成的）汇编无法正确工作。方法是，___asm(".att_syntax n")___;

代码 1: kernel.c 文件头包含的内嵌汇编

```
__asm__(".globl _start\n");
__asm__("_start:\n");
__asm__("mov $0, %eax\n");
__asm__("mov %ax, %ds\n");
__asm__("mov %ax, %es\n");
__asm__("jmpl $0, $main\n");
```

3.2 实验环境

3.2.1 系统与虚拟机

- 操作系统
  本实验在 Linux 下完成。采用 Ubuntu 16.04
- 虚拟机
  bochs. 它是一款开源且跨平台的 IA-32 模拟器。

3.2.2 相关工具、指令

- 汇编器
  NASM. NASM 是一个轻量级的、模块化的 80x86 和 x86-64 汇编器。它的语法与 Intel 原语法十分相似，但更加简洁和易读。它对宏有十分强大的支持。
- 编译器
  GCC. GNU/GCC 是开源的 C 语言编译器。其产生的伪 16 位代码可以与 NASM 结合，混编生成伪 16 位程序。

代码 2: 解决函数调用时地址长度不一致的问题

```
; in file /include/bridge.inc
%macro pushl 1
    push word 0
    push word %1
%endmacro

%macro calll 1
    push word 0
    call %1
%endmacro

%macro retl 0
    pop ecx
    jmp cx

%endmacro
```

- 镜像文件产生工具
  bximage. 该命令允许生成指定大小的软件镜像。
- 二进制写入命令
  dd. dd 允许指定源文件和目标文件，将源文件的二进制比特写入目标文件中的指定位置。
- 二进制文件查看命令
  xxd. xxd 允许将二进制文件中的内容按地址顺序依次输出，可读性强
- 反汇编器
  objdump. objump 可以查看目标文件和二进制文件的反汇编代码，还能指令 intel 或 at&t 格式显示。
- 代码生成脚本
  makefile. makefile 脚本具有强大的功能，其可以识别文件依赖关系，自动构筑文件，自动执行 shell 脚本等。

3.3 程序流程

文件系统在虚拟软盘中的镜像如图1。全部引导的简要过程见图2。

4 实验过程

4.1 项目目录树介绍

项目的目录树见代码3。

filesystem/
存放的 Models 是与文件系统实现相关的文件。其中操作系统引导程序也放在这个目录下。这是由于操作系统引导与文件系统密切相关。/filesystem/API/存放的是文件系统接口。用户程序只需 include 接

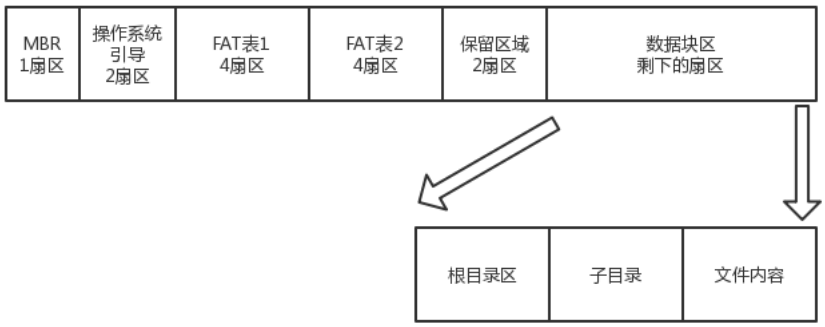

图 1: FAT16 文件系统布局

口即可使用文件系统的相关功能。

include/
存放的是内核、用户程序都可能需要的代码。其中包括打印函数、清屏函数、字符串处理函数等。

kernel/
存放的是内核程序。

loader/
存放的是主引导程序。

user/
存放的是用户程序。其中本项目还实现了简单的终端。终端程序也作为用户程序放在此处。user/stone/存放的是项目一、二的 stone 程序。

详细的整个目录树见附录B代码7。

4.2 构建 GCC+NASM 混编的全部指令

本项目采用 $makefile$ 构筑。本项目按照4.1节讲述的方式分成若干子文件夹。每个子文件夹中都有各自的 $Makefile$ 文件。整体的 $Makefile$ 会分别调用各个子文件夹中的 $Makefile$，最终构建出整个项目。

4.2.1 GCC 编译指令

所有的 C 程序都采用了表1所示的编译指令。

4.2.2 LD 链接指令

目标文件在链接时采用表2所示的指令。

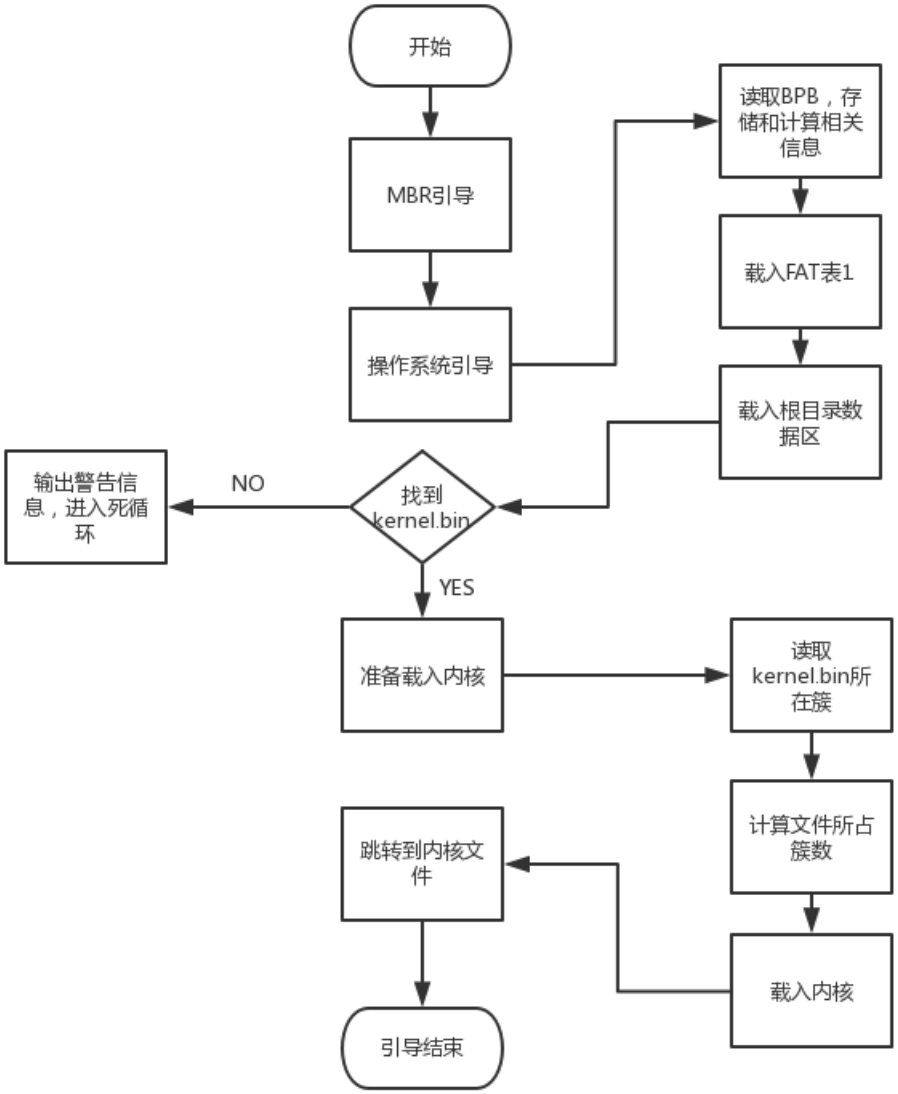
开始
MBR引导
操作系统引导
读取BPB，存储和计算相关信息
载入FAT表1
载入根目录数据区
找到kernel.bin
NO
输出警告信息，进入死循环
YES
准备载入内核
读取kernel.bin所在簇
计算文件所占簇数
载入内核
跳转到内核文件
引导结束

图 2: 引导内核的全过程

代码 3: 项目目录树介绍 (仅文件夹)

```
.
|-- filesystem/
|   |-- API/
|-- include/
|-- kernel/
|-- loader/
|-- user/
    |-- stone/

7 directories
```

表 1: GCC 编译指令

编译选项	作用
-o	D
-march=i386	X 产生原始 Itel i376 CPU 架构的代码
-m16	等价于汇编伪指令.code16gcc。 在所有必要的指令前加, 前缀 66, 67, 使得处理器能以 32 位的方式读取 GCC 产生的代码
-mpreferred-stack-boundary=2	以 2^ibytes 作为对齐量，对齐栈的界限。 该选项默认为 4(即默认 2^4 字节对齐栈界限). 为了节省空间，这里设置成 2
-ffreestanding	产生 ‘‘自立’’ 的程序文件. 即告知 GCC 不使用（几乎）所有的库头文件。 保证产生的代码不依赖于 GCC 的自带库
-c	不产生可执行文件，只执行 ‘‘编译’’ 操作产生目标文件
-Og	产生最小限度的优化. 即这个优化能化简汇编代码，但又不影响汇编程序的可读性。
-O0	不产生任何优化。 本项目文件系统 FAT 表的构建使用 C 语言的 struct 完成。 为了保证产生的二进制表项和 C 代码中的 struct 严格一致 必须采用本优化等级。

表 2: LD 链接指令

LD 链接器选项	解释
-melf_i386	链接成 intel 386 架构的代码
-N	关闭页对齐。禁止链接动态库。
-oformat binary	产生纯二进制文件，即不带文件头
-Ttext 0xA200	设 text 段（代码段）的偏移量为 0xA200。 同理还有 Tbss, Tdata 等。

4.3 文件系统

4.3.1 文件系统 API

文件系统提供了以下的 API 供内核和用户程序调用。操作系统主引导在载入内核时需要搜索 *kernel.bin*, 也是采用这套接口完成搜索。

文件系统实现了最基本的功能，例如搜索文件，进入文件夹，将文件从软盘载入内存中，执行文件等。但修改操作还未实现。所以不支持写文件和删除文件.

由于考虑到进程控制块还未实现，系统的其他部分也都很简陋，故没有实现 ‘‘文件描述符” 层次的 API 接口。只实现了很底层的接口，供内核暂时调用。

函数开头都带有双下划线，暗示这是系统的底层接口。以后实现的高层接口不会带有双下划线。

4.3.2 数据块的实现

文件系统数据块表项的定义见代码。本处采用 C 语言的方式 ‘‘自动’’ 生成表项。

本文件编译时必须采用-O0 参数。若采用-Og 参数（或等级更高的优化），表中各项的顺序可能会被 GCC 编译器重新排列，进而导致意外的事情发生。

为了告知 GCC 取消 struct 的任何对齐，这里采用了 ___attribute___((packed)) 的语法。为了让 struct 内的每个元素都有确定的长度，此处使用 uint16_t, uint8_t 的类型定义代替 unsigned int 等模糊的类型定义。

在本文件中，实例化结构并传递相应的参数，即可建立相应的表项。

4.3.3 操作系统引导的实现细节

代码5将操作系统引导中的重要代码片段列了出来。操作系统的引导需要知道分区和文件系统的细节，以确认内核的所在地址。这里代码的第 2 行将立即数 0x7E0B 强制转化为 void* 指针，得到指向 BPB 表的指针。这行代码是合法的，原因是 FAT16 文件系统的元数据在软盘中的地址是固定的，所以可以用硬编码的方式获得一个指针。void* 指针是无类型的指针，它仅表示一个内存地址。

表 3: FAT16 文件系统接口

函数声明	功能
FAT_ITEM	FAT_ITEM 是一个类型 位于 datablock 中的长 32 字节的结构（表项）
FAT_ITEM* ___get_root_dir()	返回根目录表项
int16_t ___next_item(const FAT_ITEM* p)	返回当前指针指向的下一个表项 该函数能自动跳过被删除的表项
int16_t ___has_next_item(const FAT_ITEM* p)	判断当前指针是否还有下一表项 返回 0 和 1 分别代表 false 和 true 自动跳过被删除的表项
int16_t ___FAT_item_type(const FAT_ITEM* p)	返回指针指向表项的类型 例如文件、文件夹、系统文件等 文件类型有一套宏定义，增加可读性
FAT_ITEM* ___jump_into_dir(const FAT_ITEM* p)	跳入当前表项指向的目录 调用方有义务确保指针指向的是目录 若当前表项是目录，返回正确指针
int16_t ___rm_this_file(FAT_ITEM* p)	删除指针指向的文件 若指针指向的不是文件，返回错误码 错误码有一套宏定义，增加可读性
int16_t ___run_this_file(FAT_ITEM* p)	运行指针指向的文件 若当前指向的文件不可执行，返回错误码 系统文件、文件夹都不可执行

第 3, 4 行的代码中，通过将 BPB 指针增加某个偏移量，并强制转换为 8 位、16 位的指针，得到 BPB 表项的两个数据：每簇的扇区数和保留扇区数。

示例代码的最后部分，通过简单的运计算出来数据块的起始扇区数。它等于隐藏扇区数（MBR，1 个）+ 保留扇区数（操作系统引导，2 个）+ FAT 表个数（2 个）* 每个 FAT 表扇区数（4 个）+ 1。

4.4 实现的系统库介绍

为了给以后的实验奠定基础，本项目实现了功能比较全面的字符串库和 IO 库。

其中字符串库的函数命名与 C 库命名一致。实现了 strlen, strstr, strchr, strcmp 等常用函数。

IO 库实现了 printf, puts, putch, puti, putln, putiln, newline 等库。各自用来输出字符串，整数，空行等。函数名中带 i 的表示输出整数，带 ln 的表示会换行，s 和 ch 分别表示字符串和字符。其中 printf 支持%d, %s 和%c.

代码 4: FAT 文件系统数据块项的定义

```
// in file datablock.c
// (include from FATMacro.h)
struct FAT_ITEM {
    uint8_t filename[8];
    uint8_t extendname[3];
    uint8_t mod;

    uint8_t res; //reserved
    uint8_t created_time; // ms
    uint16_t hms; // hour minute second
    uint16_t ymd; // year month day
    uint16_t recent_access_ymd;
    uint16_t bhigh_cluster;

    uint16_t modify_hms;
    uint16_t modify_ymd;
    uint16_t blow_cluster;
    uint32_t filesize; // in bytes
} __attribute__ ((packed));

// ...
FAT_ITEM kernel_bin= {
    "kernel",
    "bin",
    FAT_rw | FAT_sys,
    0, 0, 0, 0, 0, 0,
    0,
    0,
    4,
    4096 // 8 sectors
};
// ...
```

代码 5: 操作系统引导的部分源码

```
#define BPB_ADDRESS 0x7E0B;
void* bpb = (void*)(BPB_ADDRESS);
uint16_t sectorPerCluster = *(uint8_t*)(bpb+2);
uint16_t reservedSector = *(uint16_t*)(bpb+3);

// ...

uint16_t dataBlockBase = hiddenSector + reservedSector
    + numberOfFAT * sectorPerFAT
    + 1;
uint16_t rootSectorNth = dataBlockBase + 2 * sectorPerCluster;
```

代码6讲解了 printf 的简单实现。首先为了支持变长参数，必须依赖于 GCC 的内部支持，变长参数列表。这里需要包含头文件 stdarg.h。这个头文件只包含了一些宏定义，这是变长参数语法所必须的。

var_arg 用于 “取出下一个变长参数”。它是个宏函数。宏函数的第二个参数是类型，它决定了从栈中取出多长的数据。

代码 6: printf 实现代码主要部分

```
// in file utilities.h
#include <stdarg.h>
static inline int printf(const char* format, ...) {
    va_list valist;
    int narg = 0;
    int index = 0;
    // calculate narg
    va_start(valist, narg);

    // int i = var_arg(vlist, int);
    // char c = var_arg(vlist, char);
    // ...

    va_end(valist);
    return 0;
}
```

4.5 终端介绍

本项目实现了终端。终端支持基本的指令，例如 *ls* 和 *help* 和 *run* 等。

终端的实现充分依赖于文件系统。
ls 指令会搜索当前目录，列出所有的文件和文件夹。*run* 指令在运行文件前，会检查文件类型。对于不同的文件类型（尤其是不可运行的文件），终端会给出不同的错误信息。例如 “系统文件不可执行”，“文件夹不可执行”，“文件不存在” 等。见图4。

5 实验结果

图为输入 help 和 ls 命令的截图。输入 help 后，终端会输出帮助信息。输入 ls 后终端会输出根目录的所有文件和文件夹。

图为输入 run 命令的截图。当用户 run 一个文件夹时，终端会报出无法运行文件夹的错误。若 run 一个系统文件（如 kernel.bin），终端会报出无法运行系统保护文件的错误。运行不存在的文件会报出文件不存在。仅当运行的文件存在且能被执行时，程序才会被执行。

图 3: help 和 ls 命令的截图

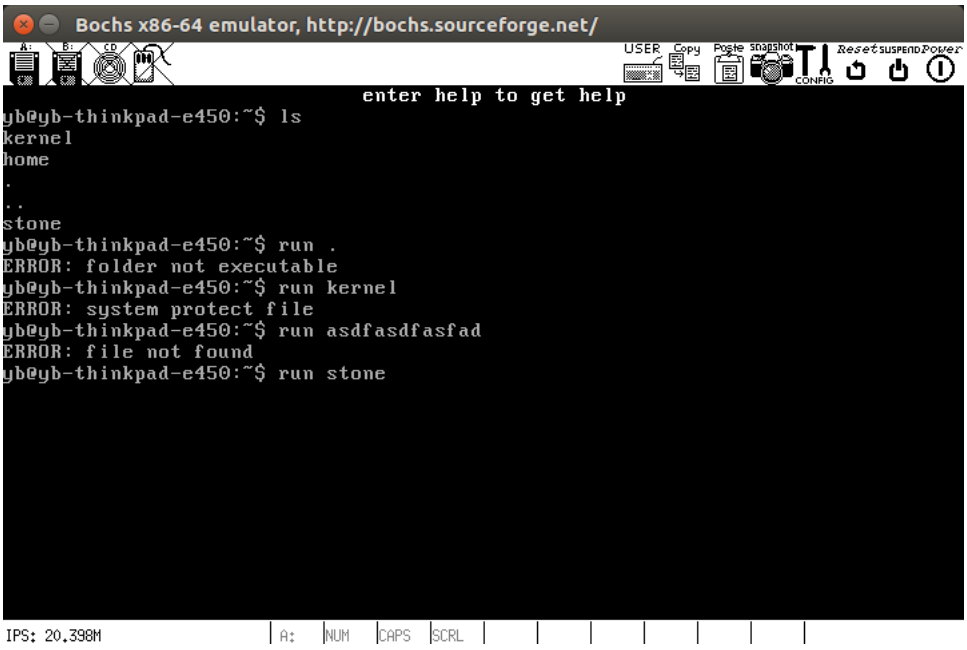

图 4: run 命令的截图

6 实验总结

6.1 亮点介绍

6.1.1 makefile

项目按照更合理的方式分为各文件夹，并改用 makefile 的方式构建项目。为 makefile 提供了 rebuild, clean, auto 等指令，分别用于重新构筑项目，清空临时文件和自动构建并运行等操作。

6.1.2 GCC+NASM 混合编程

成功地研究出了在 Linux 下使用 GCC 和 NASM 混编的方法。探索了 C 和汇编相互调用的各种困难，找到了他们的解决办法。

我将混编的方法简单地总结成了一个模板，
在网页 `https://github.com/YanB25/c-x86-hybrid-programming` 上，可直接访问下载。该模板在 Linux 下可直接正确运行。其中王永锋同学还为该模板提供了 Windows 的环境。至此，该模板在支持全平台的开发。

3.1.1节介绍了 GCC 和 NASM 混编成功的原理和本质原因。
3.1.4 节介绍了 GCC 和 NASM 混编时容易忽视的坑点难点，以及解决它们的办法。
4.2.1节详细地介绍了编译本个项目的所有编译选项，以及它们各自的作用。

6.1.3 文件系统

本项目超前地实现了 FAT16 文件系统。完成了基本的内核加载、文件读取、文件搜索、目录跳转等功能。4.3节详细地介绍了文件系统的实现。

6.1.4 终端的实现

本项目实现了一个简单的终端，它支持输入 help, ls, run 等简单命令。它会对无法识别的命令报错。也会对无法运行的文件报错。终端支持退格键修改和删除命令。

6.1.5 简单 printf 实现

4.4节介绍了 printf 的实现。

6.2 心得体会

得益于实验延后了一周，我得以用这周的空档把文件系统的基础功能实现完。如果没有这一周的时间，我可能会选择不那么繁琐的 A 线路，先把中断给实现了。

这个实验最大的困难还是在第三周的混合编程上。Linux 下的困难远比想象中的要大得多。实验三刚布置下来的几天里，我花了大量的时间研究编译指令及混编方法。熬了几次夜，但进展并不很顺利。我跟 GCC+NASM 讨论小组的同学都挺熟，那几天里我们做了很多很多的讨论。每个人都做过不同的尝试（当然也遇到了不同的失败）。当时我做的最坏打算是，这条路走不通，我就把所有的代码重构一遍，换 windows 下实验。所幸最后还是让我们找到了混编的方法，弄懂了失败的原因以及成功的原理。

混编成功后，从 ‘‘混编’’ 到 ‘‘完成 IO 函数库’’ 间还有一小段空窗期。说空窗是因为，混编后，C 语言生成的汇编代码的可读性并不太好，而此时 IO 函数又没有实现完，debug 的手段比较有限。想用 bochs 调试 C 生成的汇编，需要有一定的耐心。所幸我在这一步中没有遇到很大的困难。随后的编程就很顺利了。有了 C 程序，还能利用 IO 将变量的值打印到屏幕上，调试变得简单了很多。

在实现文件系统时也遇到了一些挫折。我最大的感想是，代码的编写，90% 以上是思考，10% 以下才是真正的代码编写。只有思考和学习后写出来的代码，才是更可靠的。

6.3 BUG 总结

写文件系统的 FAT 表时，struct 元素初始化时漏写了一个元素，导致 struct 的元素 ‘‘错位’’ 了。由于没有指示 GCC 采用更严格的警告，GCC 没有报出任何信息。FAT 表的错误导致了操作系统的内核一直无法加载进内存。

当时我困扰了比较久，因为扇区加载失败牵扯到的原因太多了。中断错误、寄存器的值错误、磁道数算错、内核没有写进虚拟镜像中……各种各样的原因我都排查后，我尝试把中间的各种变量都输出来，最后才找到了原因。

附录 A 参考文献

1. https://blog.csdn.net/longintchar/article/details/50602851
   16 位和 32 位汇编指令的不同（尤其是 push 指令）
2. https://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html#s1
   GCC 内嵌汇编的书写。
3. http://blog.51cto.com/dengqi/1349327 FAT32 文件系统讲解
4. https://blog.csdn.net/yeruby/article/details/41978199 FAT16 文件系统讲解

附录 B 其他代码

代码 7: 项目总目录树

```
.
    bochsout.log
    filesystem/
        API/
            fsapi.h
            fsErrorCode.h
        datablock.c
        DBR.asm
        DBR.c
        FAT.c
        FATMacro.h
        filesystem.h
        fsutilities.asm
        fsutilities.h
        Makefile
        README.md
    include/
        bridge.inc
        graphic.h
        Makefile
        mystring.c
        mystring.h
        utilities.asm
        utilities.h
    kernel/
        kernel.c
        kernel.h
        Makefile
    loader/
        loader.asm
        Makefile
    Makefile
    OS.img
    user/
        Makefile
        stone
        stone.c
        stone.h
        terminal.c
        terminal.h
        user.c
        user.h

7 directories, 34 files
```