

# 操作系统原理实验报告

## 实验项目三：具有独立内核的操作系统实现

院 (系) 名 称： 数据科学与计算机学院

专 业 名 称： 计算机科学与技术

学 生 姓 名： 王永锋

学 生 学 号： 16337237

指 导 教 师： 凌应标

二〇一八年三月三十一日

# 目 录

<b>1</b>	<b>实验目的及要求</b>	<b>1</b>
1.1	实验目的	1
1.2	实验要求	1
<b>2</b>	<b>实验方案</b>	<b>2</b>
2.1	实验工具和环境	2
2.2	相关基础原理-gcc 与 nasm 的混合编程	3
2.2.1	32 位与 16 位汇编代码的理解	3
2.2.2	实践中的验证	4
2.2.3	gcc 和 nasm 产生可执行文件的可连接性	5
2.2.4	gcc 和 nasm 下 c 与汇编混合编程的方法	6
2.3	程序功能说明及大致思路阐述	6
2.3.1	程序大致思路阐述	6
2.4	代码框架的设计	7
2.4.1	库文件说明	7
2.4.2	命令行终端的实现	7
2.4.3	命令行终端与文件系统的交互	8
<b>3</b>	<b>实验难点及亮点</b>	<b>9</b>
3.1	文件系统模块的实现	9
3.1.1	总体原理概述	9
3.1.1.1	所需文件	9
3.1.1.2	文件系统实现功能	9
3.1.2	实现细节	10
3.1.2.1	软盘的抽象	10
3.1.2.2	read_n_sector 函数的实现	11
3.1.2.3	加载指定文件所需函数的实现	11
3.2	可定制的中断系统-系统调用前奏	12
3.2.1	功能及原理说明	12

3.2.2	实现原理 .....	13
3.3	原理清晰, 结构合理的 gcc-nasm 编译工具链 .....	13
3.3.1	各文件夹的作用 .....	13
3.3.2	编译工作的进行 .....	13
<b>4</b>	<b>曾经遇到的问题 .....</b>	<b>15</b>
4.1	读端口的函数没有返回值 .....	15
4.2	无法正常的获取光标位置 .....	15
4.3	printf 函数的实现 .....	16
4.4	对段寄存器在 C 程序中起的作用的理解 .....	16
4.5	在实现文件系统前遇见的 bug .....	16
<b>5</b>	<b>实验结果 .....</b>	<b>19</b>
5.1	命令行终端的测试 .....	19
5.2	自定义系统调用的测试 .....	21
<b>6</b>	<b>实验总结 .....</b>	<b>23</b>
附录 A	文件的组织 .....	26
附录 B	int 40h 的实现 .....	28

# 1 实验目的及要求

## 1.1 实验目的

1. 把原来在引导扇区中实现的监控程序 (内核) 分离成一个独立的执行体, 存放在其它扇区中, 为“后来“扩展内核提供发展空间。
2. 学习汇编与 c 混合编程技术, 改写实验二的监控程序, 扩展其命令处理能力, 增加实现实验要求 2 中的部分或全部功能。

## 1.2 实验要求

本次实验, 要求需要完成以下目标:

- 规定时间内单独完成实验。
- 实验三必须在实验二基础上进行, 保留或扩展原有功能, 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。
- 在指定时间内, 提交所有相关源程序文件和软盘映像文件, 操作使用说明和实验报告。
- 实验报告格式不变, 实验方案、实验过程或心得体会中主要描述个人工作, 必须有展示技术性的过程细节截图和说明。

## 2 实验方案

### 2.1 实验工具和环境

本次实验平台<sup>1</sup>搭建在 win10 系统的 linux 子系统上，通过编写 makefile 文件，连接 nasm, gcc 编译工具，dd 二进制文件覆写工具，与 bochs, qemu 虚拟机加载配置文件与镜像（具体工具链详见下表 2.1）。

与前几次实验相比，这一次实验代码的规模大了很多，仅仅凭借脚本并不容易将一个项目进行高效的编译。因此，这一次我使用的是 makefile 来帮助这个项目进行自动化建构，只编译修改过的文件，而不编译不变的文件，以这种方式，加快了项目编译的速度，同时也能保证项目中各个文件之间的依赖关系不被破坏。

表 2.1 本实验所使用的工具链

软件名称	用途
bash	一个命令行终端，可提供 linux 的一些命令与执行 shell 脚本
nasm	将 x86 汇编文件编译成 .bin 二进制文件
gcc	编译工具，将 c 编译成二进制文件
ld	gcc 套件中包含的连接器，用于将多个可执行文件连接起来
make	gcc 套件中的工具，用于执行 makefile 文件
dd	将二进制文件的内容写进软盘镜像中
objdump	对可执行文件或二进制文件进行反编译
hexdump	以十六进制形式查看软盘镜像文件
bochs	虚拟机，用于加载装有自定义引导程序的软盘，使用软件模拟，速度不稳定
bochsdbg	调试工具，用于给装有自定义引导程序的软盘文件进行调试
qemu	虚拟机，用于加载装有自定义程序的软盘，使用硬件模拟，速度稳定且较快

<sup>1</sup>部分参考 [1]

## 2.2 相关基础原理-gcc 与 nasm 的混合编程

### 2.2.1 32 位与 16 位汇编代码的理解

在 gcc 和 nasm 结合实现混编的过程中，其实遇到了不少的问题。但我认为，一个最大的问题在于对 32 位汇编代码,16 位汇编代码的区别的理解，与 CPU 在 16 位实模式下读取操作数，地址的方式的理解。

首先定义 32 位汇编代码，我认为的 32 位汇编代码，是指使用了 32 位的寄存器，或者指令中的立即数是 32 位的，或操作 32 位的地址和操作数的指令。而对 16 位汇编代码，也有类似的定义。

但是，在没有任何指令前缀的情况下，其实无论是 32 位的汇编代码还是 16 位的汇编代码，在机器指令级别上其实是看不出区别的。

这里说明一下，32 位汇编指令和 16 位的汇编指令有以下共同点：

- 操作码相同，如无论是从栈中取 32 位地址做返回地址的 `retl`，还是从栈中取 16 位地址的 `ret`，机器码都是 `0xc3`。
- 寄存器索引相同。对于公有的寄存器，如 `ax`，索引该寄存器的数字与索引 `eax` 寄存器的数字在机器指令层面是一致的。

而 32 位汇编指令和 16 位汇编指令的差异在前面定义的时候已经讲得很清楚了，他们的差异主要体现在能在机器指令上看到的立即数的长度不同，还有隐含的处理器对寄存器或栈操作的行为的不同。

**Operand-size and address-size override prefix**

The default operand-size and address-size can be overridden using these prefix. See the following table:

	CS.d	REX.W	0x66 operand prefix	0x67 address prefix	Operand size	Address size
Real mode / Virtual 8086 mode	N/A	N/A	No	No	16-bit	16-bit
	N/A	N/A	No	Yes	16-bit	32-bit
	N/A	N/A	Yes	No	32-bit	16-bit
	N/A	N/A	Yes	Yes	32-bit	32-bit
Protected mode / Long compatibility mode	0	N/A	No	No	16-bit	16-bit
	0	N/A	No	Yes	16-bit	32-bit
	0	N/A	Yes	No	32-bit	16-bit
	0	N/A	Yes	Yes	32-bit	32-bit
	1	N/A	No	No	32-bit	32-bit
	1	N/A	No	Yes	32-bit	16-bit
	1	N/A	Yes	No	16-bit	32-bit
	1	N/A	Yes	Yes	16-bit	16-bit

图 2.1 处理器在不同的模式下对指令的处理方式

当时想到这里，我就有一个特别特别严重的疑惑：从操作码上 16 位汇编指令和 32 位汇编指令完全相同，那么处理器自身是如何知道以何种方式（32/16）来处

理这一些指令的呢？基于这样的疑惑，我在 wiki 上找到了这样的资料。

从这幅图中，可以看到，处理器处理指令的方式主要有两个因素来决定，一个是处理器当前所处模式，另一个是指令前缀的使用。我们当前仍然处于 16 位实模式下，因此在没有任何前缀的情况下，处理器对指令的处理方式都是以 16 位的方式来进行处理的，而执行前若有 0x66,0x67 前缀，就意味着处理器在遇到这一条指令的时候，会使用 32 位的寄存器和操作数，或者处理指令中 32 位的地址。

### 2.2.2 实践中的验证

之所以需要讲清楚 32 位汇编代码的共同点和差异点，是因为我们必须理解 gcc 生成的 32 位汇编代码与 nasm 产生的 16 位汇编代码之间的关系，同时认清 CPU 如何识别 gcc 生成的 32 位汇编代码，在理解的基础上，我们才能够有底气的使用这些工具给我们提供的各种功能，实现 C 与汇编交叉编译。

nasmm 在使用 bits 16 伪指令后，它能够生成 16 位的汇编指令，这些汇编指令有以下特点。

- 操作码前没有前缀，处理器按 16 位实模式下默认方式工作。
- 指令长度较短，指令中的立即数长 16 比特，call 指令中的偏移量长 16 比特。
- 在对栈进行操作的时候，push 指令会把 16 位的操作数 push 进栈中。
- 操作的寄存器都是 16 位的。

gcc 在使用 -m16 选项编译后，它能够生成 16 位实模式下兼容的 32 位汇编指令，这些汇编指令有以下特点。

- 操作码有前缀 0x66 或 0x67，表明处理器按照非默认方式（32 位）读取操作数和地址。
- 指令长度较长，指令中的立即数长 32 比特，call 指令中的偏移量长 32 比特。
- 对栈进行操作的时候，push 指令会把 32 位的操作数 push 进栈中。
- 操作 32 位的寄存器（如 eax 等）

这些指令在机器代码上的区别可见图图 2.2,图 2.3。

在这里，我们证实了以下两个结论：

- 无论是 gcc，还是 nasm，只要使用恰当的编译指令产生的机器指令，都能够在 16 位实模式下正确运行。
- nasm 与 gcc 产生的汇编代码，在相互调用的时候可能会产生问题，问题主要在 c 的 call 是 push32 位地址，而 nasm 的 ret 是取 16 位地址。一来一回就会产生问题。

38:	f6 e3	mul	%b1
3a:	89 c6	mov	%ax,%si
3c:	bb 40 30	mov	\$0x3040,%bx

图 2.2 nasm 生成的 16 位汇编代码

66 53	push	%ebx
67 66 8b 5c 24 08	mov	0x8(%esp),%ebx
66 53	push	%ebx
66 e8 9d ff ff ff	calll	0x70f

图 2.3 gcc 生成的 32 位汇编代码

### 2.2.3 gcc 和 nasm 产生可执行文件的可连接性

上面证实了指令的可运行性，这里在确认一下两种可执行文件的可连接性。

要保证两种可执行文件可以连接，必须保证两种文件有着相同的可执行文件的格式。所幸，通过搜寻资料，我发现 nasm 在使用 -f elf32 的情况下能够输出 elf32 格式的可执行文件，gcc 在 -m16 的默认情况下输出一种格式为 elf32 的可执行文件，同时，连接器也支持使用 -m elf\_i386 的指令，连接两个 elf32 格式的可执行文件，这就为最终连接成功奠定了基石。

同时，考虑到最终生成的可执行文件还不能直接写入软盘，我们还需要生成不包含文件头的纯二进制文件，对于这个需求，ld 工具中控制输出格式的一个参数 -oformat binary 可以用来控制输出文件的格式。

以下是编译指令样例

```

1 nasm -f elf32 -o kernel.o kernel.asm
2 gcc -c -m16 -ffreestanding -o start.o start.asm
3 ld -Ttext 0x00000 -m elf_i386 -oformat binary -o kernel.bin
   kernel.o start.o

```

在上面有两个指令没有说明的，一个是 gcc 中的 -ffreestanding，这个指令的用途是表明这个程序没有用到任何库函数，不能够对库函数做优化；另一个是 ld 工具的 -Ttext 0x00000，他的作用与 org 类似。

同时，还有一个显而易见不需要说明的是，代码中必须有 global，extern 等关键词用作将名字导出或导入，这样之后的连接器才能够正确识别并连接。不过，由于 elf 格式的特殊性，名字在编译后不会产生下划线前缀（NASM 文档中有提到）

至此，C 与汇编混合编程完成。



## 2.2.4 gcc 和 nasm 下 c 与汇编混合编程的方法

在证实指令的运行性和可连接性后，对两个不同的编译器产生的有一点点相互不兼容的地方，我们做以下妥协，保证代码的正确运行。

- 为确保统一，代码中出现的 `call` 和 `ret` 统一显式指明使用 32 位格式。具体的方式是：

```
1      ; 使用宏，在 ret 指令前显式添加 0x66 前缀
2      %macro retl 0
3      db 0x66
4      ret
5      %endmacro
6      ; 在 call 的时候添加 dword，确保 call 产生 32 位的偏移量。
7      call dword LABEL
```

- gcc 生成汇编代码的过程中，可能会生成用到段寄存器的指令（如从数据段获取数据）。这里我们需要了解 gcc 一般会默认 `cs`, `ss`, `ds`, `es` 等段寄存器都是一致的。因此在跳入 C 代码段的时候，要确保段寄存器的一致性。
- gcc 遵循 C 调用约定，因此关于参数传递以及返回值的规范需要汇编代码遵守。

## 2.3 程序功能说明及大致思路阐述

这里会对程序的大致运行流程进行一个粗略的描述，同时罗列了当前系统内核支持的功能。

### 2.3.1 程序大致思路阐述

1. 一开机，处于软盘第一个扇区的引导程序加载第 72-89 个扇区，作为系统内核，并将控制权转交给系统内核。
2. 系统内核刚开始运行，经过一系列初始化工作（如安装中断，加载 fat 表，根目录表），跳转到 `tty` 例程（即用户终端）。
3. 进入到命令行中断，系统管理员可以在这里执行一系列指令，如 `ls`, `run`, `help`, `reboot`
4. `ls` 命令：展示根目录下的文件
5. `run` 命令：加载指定的文件并执行
6. `help` 命令：显示帮助信息
7. `reboot` 命令：重启

## 2.4 代码框架的设计

### 2.4.1 库文件说明

为了能够在 C 中更有效的操作硬件，同时保证更有效率的开发，我实现了以下库函数作为内核需要调用的头文件。对于一些无法使用 C 语言实现的功能，如端口的读写，我使用汇编语言实现了一些函数，将端口读写封装成一个个 C 语言直接可用的函数，而对 bios 中断的使用也是类似，在 `basic.asm` 中实现了对一些 bios 中断的直接调用，那么在 C 中，就可以对这些硬件操作进行进一步的封装（主要在 `stdio.c` 中实现），实现一些对开发者友好的接口，从而加快开发操作系统的效率。

表 2.2 库文件说明

头文件名	功能说明
<code>basic.asm</code>	实现一些只能通过汇编实现的接口函数
<code>stdio.c</code>	存放处理 I/O 的函数（如 <code>printf</code> ）
<code>system_call.c</code>	用于存放各项系统调用（暂时只写了安装系统中断）
<code>string.c</code>	用于字符串的处理，包括比较和复制等函数
<code>global.c</code>	存放全局变量（目前有系统调用表）
<code>fsystem.c</code>	存放负责处理文件的函数

### 2.4.2 命令行终端的实现

命令行终端主要在 `tty.c` 中实现，该份代码的主要编写思路为：

- 使用终端内的全局变量，作为输入缓冲区，显示位置，光标偏移量。每输入一个字符，根据缓冲区的内容以及其他变量刷新显示内容。
- 终端建立一个事件循环，一旦有字符键入，`check_keyboard()` 函数返回的值会让终端知道有字符键入，触发终端对这个事件的发生产生响应。
- 使用 `get_keyboard()` 函数返回的值，判断键盘输入内容，从而进行对应事件的响应，如键入 `enter`，会触发命令行的执行，键入可打印字符，会触发输入缓冲区的显示刷新。
- 一旦进入解析器，解析器会读取输入缓冲区的内容，进行分词，然后根据相应的参数调用对应的过程。

### 2.4.3 命令行终端与文件系统的交互

- 运行程序的过程新建一个函数指针的指针，并且指明这个函数处在 0x2000 这个偏移量中，然后使用文件系统将相应的文件加载到对应地址，最后调用该函数实现跳转。

```
1  if(!strcmp(arguments[0],"run")){// 如果返回0，就是相等了
2  void (**my_program)();
3  *my_program = 0x2000;
4  fs_load_by_name(arguments[1],*my_program);
5  (*my_program)();
6  }
```

- 打印文件列表及信息，只需要直接调用文件系统提供的接口就行了。

```
1  if(!strcmp(arguments[0],"ls") && arguments_num == 1){
2      printf("\n\n");
3      fs_show_root_file_table();
4  }
```

## 3 实验难点及亮点

在完成本次实验的过程中, 在下面的功能中花了比较长的时间, 而且是在原有实验要求上所没有的, 有一些即使在这一次实验中没有完全用上, 也能够在今后的实验中继续用于完善操作系统。

### 3.1 文件系统模块的实现

在这一个操作系统中, 文件系统模块主要由“fsystem.h”和“fsystem.c”来实现。对于文件系统中所需要的读写格式, 我在“fat.asm”和“root.asm”中进行编码然后写进硬盘中。

#### 3.1.1 总体原理概述

##### 3.1.1.1 所需文件

表 3.1 文件系统相关文件说明

文件名	功能说明
./boot/fat.asm	硬编码 fat 表, 写入到软盘的第二个扇区
./boot/root.asm	硬编码根目录表, 写入到软盘的第 37 个扇区
./include/fsystem.h	文件系统头文件, 包含了文件系统操作的各类函数的声明
./lib/fsystem.c	文件系统代码文件, 包含了文件系统操作的函数的实现

##### 3.1.1.2 文件系统实现功能

1. 向文件系统传入文件名尝试进行加载, 加载成功的状态由返回的错误码来判断。
2. 向文件系统传入文件名读取文件信息
3. 向文件系统请求打印根目录文件信息

### 3.1.2 实现细节

#### 3.1.2.1 软盘的抽象

目前文件系统的实现中,唯一依赖的一个底层函数是定义在 `basic.c` 中的 `read_sector()` 函数,这个函数是 `int13h` 中断的再包装。文件系统其余的函数只要涉及到软盘到内存的 I/O 操作,都需要用到这一个函数。基于这一个函数,在 `int 13h` 已有的抽象下,进一步将软盘抽象,从而只通过逻辑扇区号和扇区数量就可以直接加载对应的扇区。

基于以上的封装,我实现了以下函数,从而将软盘进一步抽象,从而能够通过文件名直接访问。

表 3.2 文件系统相关函数说明

函数声明及相关功能
<code>u16 _fs_find_descriptor_number_by_name(char * file_name);</code> 内部过程: 使用文件名,找到对应文件描述符的索引。
<code>u16 _fs_find_cluster_code_by_name(char * file_name);</code> 内部过程: 使用文件名,找到该文件的簇号
<code>u16 _fs_get_file_size_by_cluster_code(u16 cluster_code);</code> 内部过程: 从簇号,找到这一整个文件占用的空间大小
<code>void _fs_show_file_by_descriptor_number(u16);</code> 内部过程: 使用文件描述符的索引,展示该文件信息
<code>u16 _fs_load_by_cluster_code(u16 first_cluster_code, void (*program)());</code> 内部过程: 使用簇号,加载 fat 表中的一系列扇区到指定的地址处。
<code>u16 fs_load_by_name(char * file_name, void (*program)());</code> 对外过程: 使用文件名,加载对应文件到指定地址处,返回错误码判断是否加载成功。
<code>u16 fs_get_file_size(char * file_name);</code> 对外过程: 使用文件名获取文件大小(为之后分配内存做准备)
<code>void fs_show_file_by_name(char * file_name);</code> 对外过程: 使用文件名,展示对应文件信息
<code>void fs_show_root_file_table();</code> 对外过程: 打印文件目录表

### 3.1.2.2 read\_n\_sector 函数的实现

这个函数作为文件系统与 bios 中断之间极其重要的抽象层，将需要磁头号，柱面号，扇区号访问的软盘，抽象成只通过从 0 开始的逻辑扇区号来访问，将软盘转变成一个像是数组的连续存储介质。同时，该函数还能够将指定数量扇区的内容加载到指定的地址中，从而完美封装了 int 13h 中断。以下是代码实现。

```
1  /* tested
2  读取指定逻辑扇区号的扇区到指定内存地址处
3  注意一次最多写一个段，也就是64k 最多读128个扇区。 */
4  void read_n_sector(u16 sector_code, u16 number, u16 segment, u16
    offset){
5      for (int i = 0; i < number; i++){
6          int sector = (sector_code+i) % 18+1;
7          int mid = (sector_code+i) / 18;
8          int cylinder = mid >> 1;
9          int head = mid & 1;
10         read_sector(head, cylinder, sector, segment, offset + i*512);
11     }
12     return ;
13 }
```

### 3.1.2.3 加载指定文件所需函数的实现

为了实现通过文件名就可以加载指定文件到指定地址，我实现了两个子函数。

#### 1. 文件名->簇号

```
1  u16 _fs_find_cluster_code_by_name(char * file_name){
2      int index = _fs_find_descriptor_number_by_name(file_name); //
    实现从文件名找到对应的文件描述符索引
3      if (index != -1) // 若索引存在
4          return root[index].cluster_code;
5      return 0;
6  }
```

#### 2. 加载从指定簇号开始的一系列扇区（由 fat 表确定簇数量）到指定函数指针处。

```
1  u16 _fs_load_by_cluster_code(u16 cluster_code, void(*program)()){
2      int size = 0;
3      while (0x0002 <= cluster_code && cluster_code <= 0xFFEF){
4          size += 512*cluster2sector;
```

```

5         int sector_number = first_cluster_by_sector + (
            cluster_code - 2) * cluster2sector;
6         read_n_sector(sector_number, cluster2sector, 0x1000,
            program);
7         program = program + cluster2sector * 512;
8         cluster_code = FAT_table[cluster_code];
9     }
10    return size;
11 }

```

## 3.2 可定制的中断系统-系统调用前奏

总是看到 dos 能够通过下面代码所示的方式调用系统调用，在这里我自己也尝试了一些这样调用中断的实现。

```

1  mov ax, 4c00h
2  int 80h

```

### 3.2.1 功能及原理说明

这里我实现了这样的—个中断，内核可以在程序中以如下代码的方式安装中断：

```

1  void test_system_call();
2  int cstart() {
3      install_system_call(2, test_system_call);
4      // 将指定的函数安装到2号系统调用中
5  }
6  void test_system_call() {
7      int origin = get_cursor();
8      set_cursor(1800);
9      printf("test custom system call!!! ");
10     set_cursor(origin);
11 }

```

然后在汇编中，我便可以用下面的方式直接调用我之前安装的系统调用测试函数！

```

1  mov ax, 0x0200h
2  int 40h

```

### 3.2.2 实现原理

在内核中，我维护着一个数组“系统调用表”，该数组存储着不同的系统调用的地址，在 C 语言中安装中断，就是将对应函数指针的内容，写到相应数字索引的系统调用表中。

而我自己的 int 40h 中断，则是先读入 ah 寄存器的内容，计算该系统调用的系统调用号的索引，然后在系统调用表中找到对应的表项并且 call 过去，运行完指定函数之后回来恢复现场，继续执行下面的代码。

同时，该中断也可通过功能号 4c，实现子程序的返回，可返回到命令行终端模式。

详情可见附录 B

## 3.3 原理清晰, 结构合理的 gcc-nasm 编译工具链

本次项目，重新组织了代码文件，对各个模块的整理，可见附录 A。

### 3.3.1 各文件夹的作用

表 3.3描述了各个文件夹的作用

表 3.3 文件夹的说明

文件夹名称	文件夹用途
boot	里面包含了引导扇区的汇编代码，以及需要硬编码到软盘的文件系统相关数据
kernel	里面包含了内核代码及命令行终端的实现
include	这里是库文件的头文件，主要用作各种函数的引用
lib	里面有各个库文件的定义，需要和用到这些函数的文件一起编译
user	里面存放了各个用户程序

### 3.3.2 编译工作的进行

在 os 文件夹下具有一个主 makefile 文件，这个文件先进入到各个子文件夹进行 building，构建各个模块的二进制文件，再回到这个主 makefile 文件进行将二进



制文件写入到软盘的工作，并且集成了各个工具的使用，直接打开虚拟机查看结果。

## 4 曾经遇到的问题

### 4.1 读端口的函数没有返回值

在 C 调用约定中，有一条这样的约定：返回值使用 `ax` 寄存器返回。在实现读端口数值的时候，我犯了这样一个愚蠢的错误，在保存寄存器的时候误把 `ax` 作为一个被调用者保存寄存器保存起来，导致返回值无法传递到调用者，代码如下所示：

```
1  ; tested 从8位端口读出8位的值
2  ; u8 read_port_byte(u16 port_number);
3  read_port_byte:
4  push ebp
5  mov ebp, esp
6  push dx
7  push ax ; !!!不能保存ax寄存器的值，要作为返回值传递给调用者
8  ; enter
9  mov dx, [ebp+8] ; 取第一个参数 端口号
10 in al, dx
11 ; leave
12 pop ax ; !!!
13 pop dx
14 mov esp, ebp
15 pop ebp
16 retl
```

相同的错误在其他地方也出现过多次，就不再赘述。

### 4.2 无法正常的获取光标位置

在获取光标位置的函数中，无法获取光标的位置。主要的原因是以下代码出现错误，导致光标索引计算错误！

```
1  cursor_index = cursor_index << 8 + low_eight;
```

事实上，`+` 号的优先级比移位高，我需要加一个括号，像下面一样才能正常运行。

```
1 cursor_index = (cursor_index << 8) + low_eight;
```

### 4.3 printf 函数的实现

在实现 printf 函数的过程中，经历了内存被神秘改写的奇特 bug，经过细细调试，定位这样的问题：栈指针初始化的不合理，导致栈段与代码段重合。

在开发的时候会有开发日志记录自己在开发过程中遇到的问题，下面从这个 bug 的解决中摘录一部分。图 4.1

### 4.4 对段寄存器在 C 程序中起的作用的理解

```
1 char * a = 0xa8000;  
2 *a = 'a';
```

这一段很简单的代码，可以直接操控显存。为什么不是 B8000 呢，因为我的 ds 被初始化为 0x1000 了。这也是段寄存器在 c 语言中起作用的典型例子，之前一直没搞清楚 c 语言中段寄存器如何起作用。在使用 bochs 调试的时候，通过反汇编，看到了这条语句在寻址过程中对指针的使用。

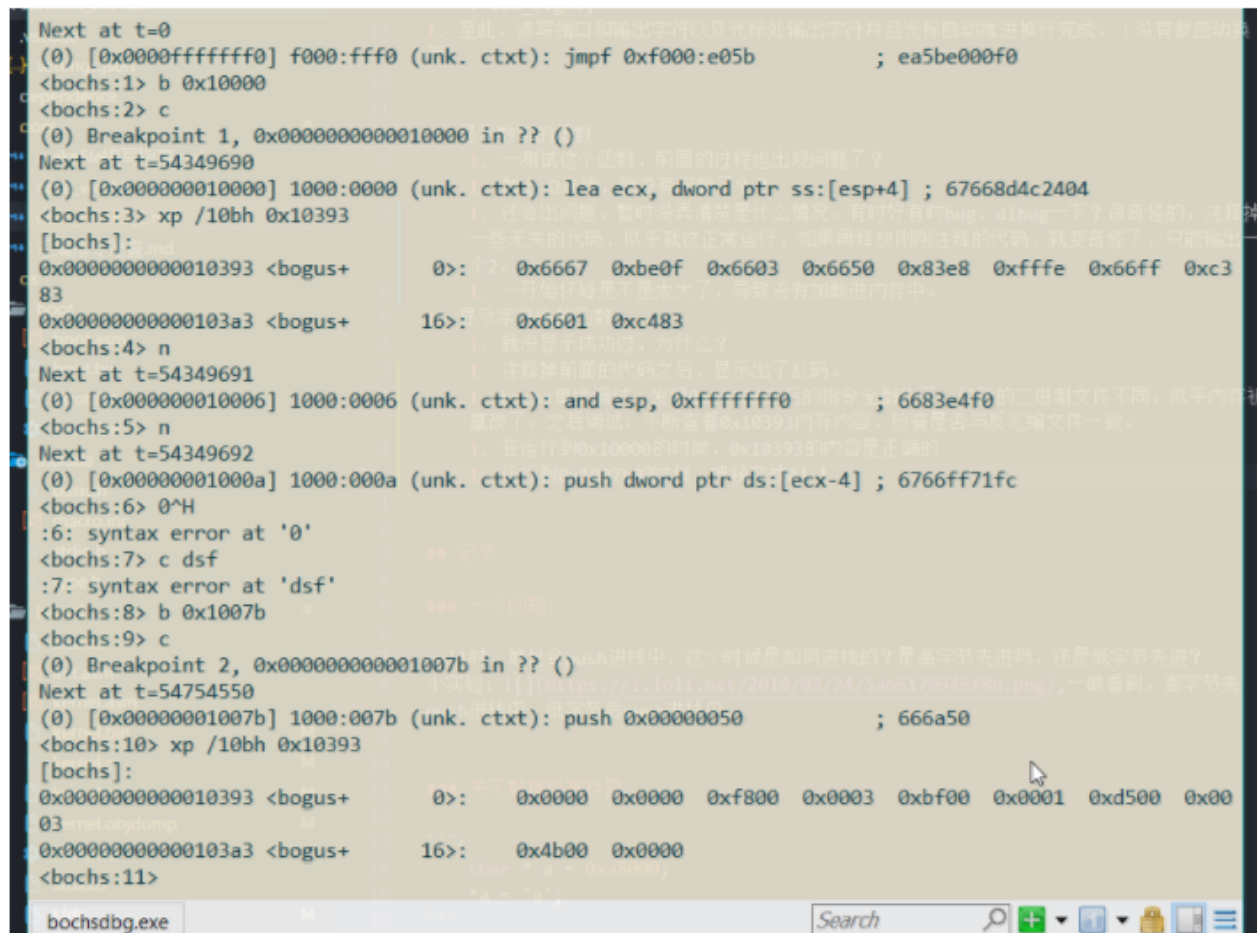
### 4.5 在实现文件系统前遇见的 bug

在运行的时候，发现如果我注释掉一段代码，就会让程序正常运行，如果我恢复那一段代码，程序就会进入一种不正常的运行状态，一些应该显示的信息没有显示出来。后来发现，是内核的大小超出了加载的扇区数量，这也是后来我想做文件系统的动机之一。

图 4.2摘录自开发日志。

## 5. 显示字符串的函数

1. 就没显示成功过，为什么？
2. 注释掉前面的代码之后，显示出了乱码。
3. bochs单步调试，发现0x1038f之后的指令全都乱了，和我的二进制文件不同，似乎内存被篡改了。之后调试：不断查看0x10393内存内容，检查是否与反汇编文件一致。
4. 在运行到0x10000的时候，0x10393的内容是正确的
5. 运行到0x1007b的时候，神秘变成0！！



```
Next at t=0
(0) [0x0000ffffffffff] f000:ffff0 (unk. ctxt): jmpf 0xf000:e05b ; ea5be000f0
<bochs:1> b 0x10000
<bochs:2> c
(0) Breakpoint 1, 0x00000000000010000 in ?? ()
Next at t=54349690
(0) [0x0000000010000] 1000:0000 (unk. ctxt): lea ecx, dword ptr ss:[esp+4] ; 67668d4c2404
<bochs:3> xp /10bh 0x10393
[bochs]:
0x00000000000010393 <bogus+ 0>: 0x6667 0xbe0f 0x6603 0x6650 0x83e8 0xfffe 0x66ff 0xc3
83
0x000000000000103a3 <bogus+ 16>: 0x6601 0xc483
<bochs:4> n
Next at t=54349691
(0) [0x0000000010006] 1000:0006 (unk. ctxt): and esp, 0xffffffff ; 6683e4f0
<bochs:5> n
Next at t=54349692
(0) [0x000000001000a] 1000:000a (unk. ctxt): push dword ptr ds:[ecx-4] ; 6766ff71fc
<bochs:6> 0^H
:6: syntax error at '0'
<bochs:7> c dsf
:7: syntax error at 'dsf'
<bochs:8> b 0x1007b
<bochs:9> c
(0) Breakpoint 2, 0x0000000000001007b in ?? ()
Next at t=54754550
(0) [0x000000001007b] 1000:007b (unk. ctxt): push 0x00000050 ; 666a50
<bochs:10> xp /10bh 0x10393
[bochs]:
0x00000000000010393 <bogus+ 0>: 0x0000 0x0000 0xf800 0x0003 0xbf00 0x0001 0xd500 0x00
03
0x000000000000103a3 <bogus+ 16>: 0x4b00 0x0000
<bochs:11>
```

6. 进一步缩小，0x1001c-0x10036
7. 0x10036处调用的函数，覆盖了0x10393所在的内存，定位来看是putc函数
8. 大概明白了，我在boot给kernel初始化的sp太小，我的代码已经超过了栈段，当我对栈进行操作时，便修改了内存。
9. 现在至少是正常显示了一些乱码，而不是死机了。
10. 问题来了
1. Next at t=49067300  
(0) [0x0000000010393] 1000:0393 (unk. ctxt): movsx eax, byte ptr ds:[ebx] ; 67660f0e03
2. ds是0x1000，而内存地址为0x103b7，一旦加起来，就不是预想的值了。
3. 这是一个很大的问题，我还没有想好怎么总结这一个bug
4. 想来，对C代码运行的环境仍没有一个很好的理解，就像是，与gcc合作却又没有遵守它的约定。
5. 两个问题，1. 连接器产生的地址加上ds，将连接器参数改掉，从0开始。2. basic.asm中的函数write\_memory没有保存ds寄存器。
11. 正常显示！

图 4.1 printf 函数-开发日志摘录

## 中断系统完成，现在实现用户程序的执行和返回

使用一个中断来实现返回到操作系统内核。

如果想保存终端的状态，需要怎么做？

刚刚遇到一个很奇怪的问题，我删掉了一些代码，命令行运行正常了??????

问题来了，如何调用用户程序呢？

1. 向文件系统传送文件名，由文件系统加载到指定地址，然后向内核返回错误码，至此，文件加载完成，加载完成后，跳转到用户程序执行。
2. 用户程序执行执行，当执行完之后，调用int40h中断返回到内核tty。

刚刚有一个很奇怪的事情，就是我注释掉一段代码之后就少了一些显示的内容，但是功能却还是正常的！

后来debug的时候发现，原本应该在内存中存在的值，在debug的时候这些值都神奇的全部变成了0，为什么！！

后来发现，在运行的时候，这些值全部都是0。

```
Next at t=58841119
(0) [0x000000010000] 1000:0000 (unk. ctxt): call .+19 (0x00010016) ; e81300
<bochs:3> xp /30bh 0x113f0
[bochs]:
0x000000000000113f0 <bogus+ 0>: 0x7020 0x7261 0x6573 0x0072 0x7572 0x006e 0x7325 0x2500
0x00000000000011400 <bogus+ 16>: 0x0000 0x0000 0x0000 0x0000 0x0000 0x0000 0x0000 0x0000
0x00000000000011410 <bogus+ 32>: 0x0000 0x0000 0x0000 0x0000 0x0000 0x0000 0x0000 0x0000
0x00000000000011420 <bogus+ 48>: 0x0000 0x0000 0x0000 0x0000 0x0000 0x0000 0x0000 0x0000
<bochs:4> _
```

大概的原因就猜到了，我只加载了10个扇区，每个扇区512字节（0x200）的话，其实在十六进制上，就只能加载到0x1400,因此后面的内容全部变成了0。

为了解决这个问题，我就在boot中加载多了8个扇区。问题就解决了。

其实要想完全解决这个问题，还得实现文件系统，由文件系统负责加载文件，并保证文件的完全加载。

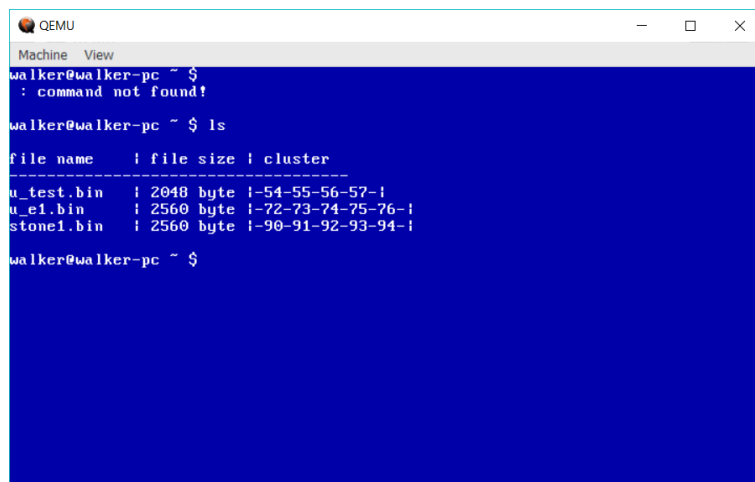
图 4.2 user 用户程序-开发日志摘录

## 5 实验结果

### 5.1 命令行终端的测试

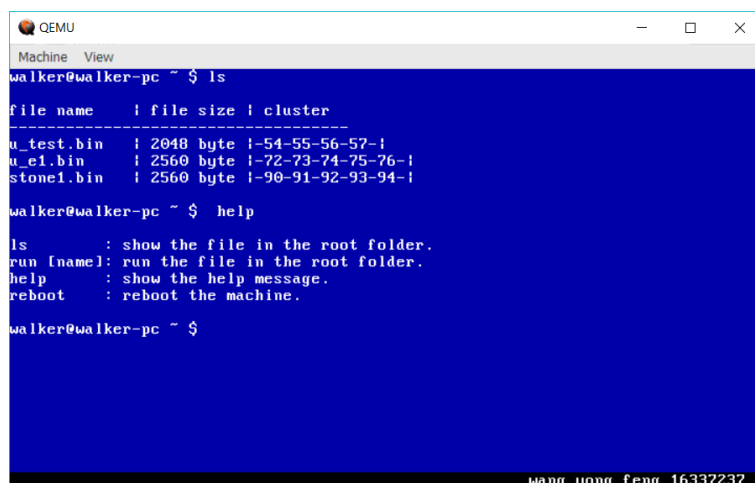
本程序经 `nasm`, `gcc` 编译后在 `qemu` 虚拟机下运行的效果可见图 5.1 与图 5.2, 其中图 5.1 中底部的名字可以自动滚动。

同时, 在 `ls` 命令中展示的文件名可以直接加载并运行。如图 5.3 与图 5.4, 通过按 `q`, 可以回到终端。



```
QEMU
Machine View
walker@walker-pc ~ $
: command not found!
walker@walker-pc ~ $ ls
file name | file size | cluster
-----
u_test.bin | 2048 byte | 1-54-55-56-57-1
u_e1.bin | 2560 byte | 1-72-73-74-75-76-1
stone1.bin | 2560 byte | 1-90-91-92-93-94-1
walker@walker-pc ~ $
```

图 5.1 命令行终端的主界面



```
QEMU
Machine View
walker@walker-pc ~ $ ls
file name | file size | cluster
-----
u_test.bin | 2048 byte | 1-54-55-56-57-1
u_e1.bin | 2560 byte | 1-72-73-74-75-76-1
stone1.bin | 2560 byte | 1-90-91-92-93-94-1
walker@walker-pc ~ $ help
ls : show the file in the root folder.
run [name]: run the file in the root folder.
help : show the help message.
reboot : reboot the machine.
walker@walker-pc ~ $
```

图 5.2 运行 `ls` 及 `help` 命令后的主界面

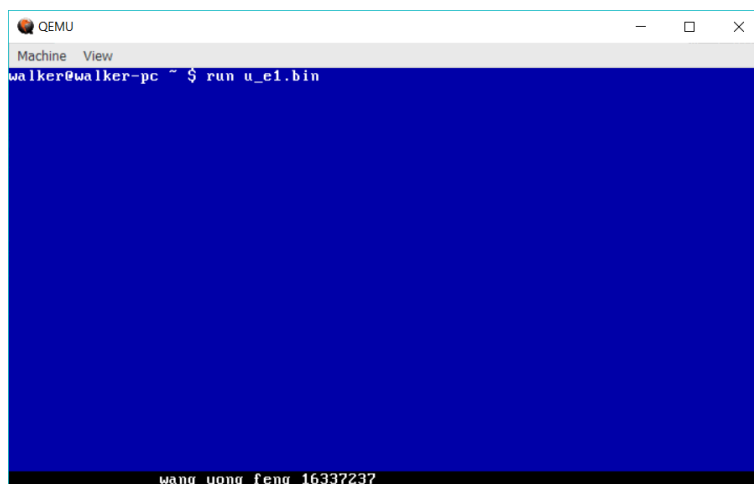


图 5.3 run 命令的调用

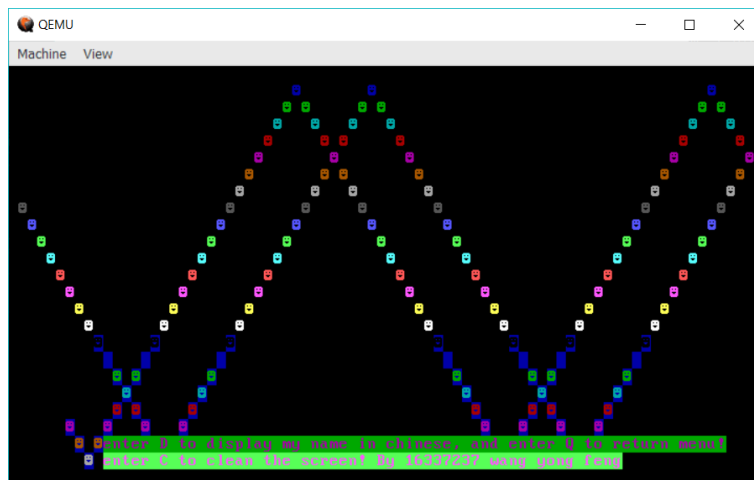


图 5.4 运行 run 命令后成功进入用户程序

## 5.2 自定义系统调用的测试

将 start.c 中的代码修改为安装 2 号系统调用。

```
1 void test_system_call();
2 int cstart() {
3     install_system_call(2, test_system_call);
4     // 将指定的函数安装到2号系统调用中
5 }
```

其中 test\_system\_call 函数定义如下所示，该函数会显示一个字符串 “test custom system call”：

```
1 void test_system_call() {
2     int origin = get_cursor();
3     set_cursor(1800);
4     printf("test custom system call!!! ");
5     set_cursor(origin);
6 }
```

然后在内核主程序中，调用 2 号系统调用。

```
1 _start:
2     call install_int40
3     call install_int8
4     call dword cstart
5     mov ah, 0x02
6     int 0x40
7     jmp $
```

在命令行中执行 make all，重新编译调用虚拟机，我们看到以下的结果，除去在初始化过程中输出的其他信息，在屏幕上还出现了 2 号中断显示的字符串，表明成功调用 2 号中断。



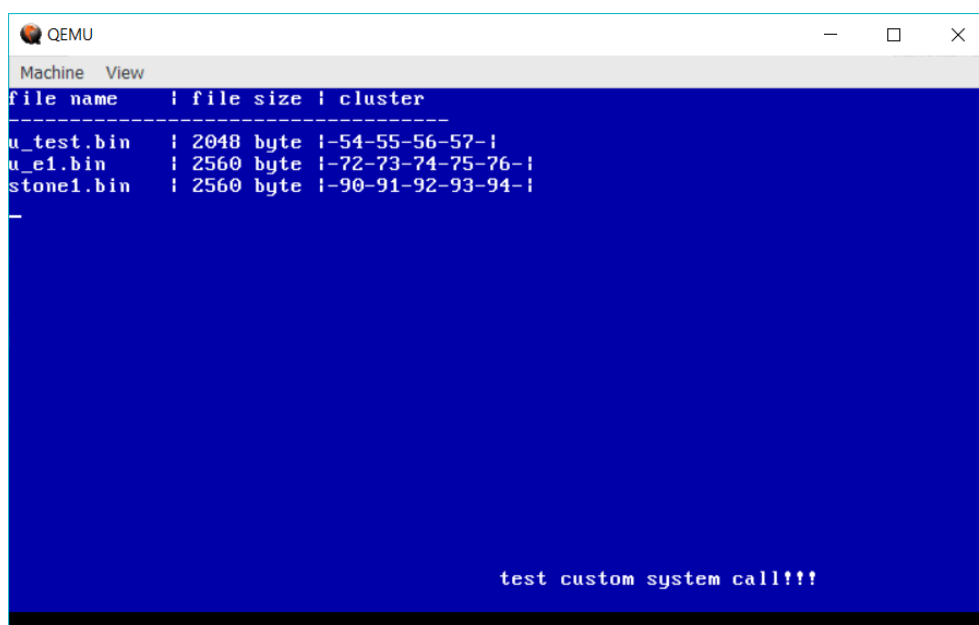


图 5.5 测试 2 号系统调用，运行截图

## 6 实验总结

这是实验三的实验总结。

在实验前期，我对 c 与汇编混合编程的概念并不了解。在一开始接触到混合编程的时候，甚至是想了半天为什么要使用 c 语言和汇编混合编程，还想了很久需要使用 C 语言做什么事情，紧接着，对 C 语言中隐藏的细节再进行了很多的思考，以想明白如何使用 c 语言替换汇编的代码。

C 语言，可以说是已经很贴近硬件底层的语言了，可是直到我将 C 语言与汇编语言结合起来使用，我才更进一步的认识到 C 语言对内存的操控能力。在 C 中，我可以对地址进行直接的操作，使用指针的方式，修改特定位置的内存的数据。同时，通过修改函数指针，我们甚至可以修改 ip，从而达到在汇编中与 call 一致的效果。

但同时，C 语言也对我们隐藏了很多硬件的细节，也正是这些细节，让我在一开始觉得 C 语言的行为有一些不可控的因素。其中一个很重要的点就是寄存器的使用，在 C 中，我们不会也不能对寄存器进行直接的操作，我们创建的所有变量要么是编译器决定使用寄存器，要么是使用用户的栈空间申请一些临时变量，这一切的细节，都由编译器来完成，而我们无需关心具体寄存器还是栈，我们只需要关心数据的处理与使用就可以了。从这个角度上来讲，使用 C 语言也更能让我们在硬件的众多细节中脱身而出，让我们有更多的精力去关注代码中的逻辑与功能。

C 语言对我们隐藏的另一个细节是段寄存器的使用。C 语言中，无论我们怎么写代码，其实都是默认在一个段中，而显然编译器也是这么认为的。编译器会直接拿 C 语言汇编出来之后的汇编地址放到代码中需要地址的地方，然后在访问的时候按照处理器的规则，加上 ds 或者 cs 去访问。

对段寄存器细节的隐藏，一方面是解脱了一些细节的纠缠，但另一方面，却让我的代码的组织变得有些困难。我当然可以整个操作系统只使用一个段，这样子就可以实现所有函数，无论是 C 还是汇编都可以自由的跳转，我甚至可以直接将绝对地址写进函数指针中，然后就可以实现类似 call 的功能去执行它。但我想让内存安排得更合理一些，我希望用户程序能和内核程序运行在不同的段中，而且用户程序还无需关心自己需要加载的内存地址。但这样的话，一方面，C 中无法实现段间的跳转，我暂时还只能通过 C 中内嵌汇编，显式调用中断来返回，另一方面，当用户程序要调用系统调用需要传递消息的时候，不在一个段中的两个程

序我无法通过一个指针直接传递消息，这样一来，分段之后，每个程序似乎都只能让自己变成孤立的一个程序，而无法和其他程序还有操作系统进行交互。直到要交报告了，这个问题现在也没有解决好。

这一次实验中，我还实现了一个类似于系统调用的中断系统，可以通过简单的在内核中通过传递函数指针的方式安装一个自定义中断处理启程，然后就可以使用 `int` 指令调用刚刚自己写的中断处理例程，但是写好之后，暂时还没怎么用到内核中来，不过提前想好系统的架构，希望能够为未来的实验做好准备吧。

同时，在这一次实验，想到要实现一个表来存储用户的程序存放安排，又想到之后的实验四需要实现文件系统，对文件系统有过简单的了解后，想到文件系统里面内置的表格已经可以让我实现读取用户程序的存放安排了，于是干脆也顺便吧文件系统给实现了。

## 参考文献

- [1] 于渊. *Orange'S*: 一个操作系统的实现. 电子工业出版社, 2009.

## 附录 A 文件的组织

```
1 |— boot
2 |   |— boot.asm
3 |   |— fat.asm
4 |   |— makefile
5 |   └─ root.asm
6 |— include
7 |   |— basic.h
8 |   |— fsystem.h
9 |   |— global.h
10 |  |— macro.inc
11 |  |— stdio.h
12 |  |— string.h
13 |  |— system_call.h
14 |  └─ type.h
15 |— kernel
16 |   |— kernel.asm
17 |   |— kernel_old.asm
18 |   |— makefile
19 |   |— start.c
20 |   |— t.lds
21 |   └─ tty.c
22 |— lib
23 |   |— basic.asm
24 |   |— fsystem.c
25 |   |— global.c
26 |   |— stdio.c
27 |   |— string.c
28 |   └─ system_call.c
29 |— user
30 |   |— makefile
31 |   |— user_e1.asm
32 |   |— user_stone_1.asm
33 |   └─ user_test.asm
34 |— a.img
35 |— bochsrc.bxrc
```

```
36 └─ makefile
37 └─ readme.md
```

## 附录 B int 40h 的实现

```
1 ; 这是新的int40，用于调用系统调用
2 ; 使用ax索引中断
3 ; 每一个项是16位+16位
4 new_int40:
5
6     cmp ah, 0x4c
7     je .return_kernel
8     mov bl, ah
9     xor ax, ax
10    mov al, 0x2
11    mul bl
12    mov si, ax
13    mov bx, system_call
14    call dword [bx + si] ; 注意这个call是32位的。
15    iret
16
17 .return_kernel:
18     pop cx
19     pop cx
20     pop cx
21
22     mov ax, 0x1000
23     mov ds, ax
24     mov es, ax
25     mov ss, ax
26     mov sp, 0x5000
27
28     push cx
29     push 0x1000
30     push start_tty
31     iret
```