实验五：实现系统调用

实验目的

1. 学习掌握PC系统的软中断指令
2. 掌握操作系统内核对用户提供服务的系统调用程序设计方法
3. 掌握C语言的库设计方法

实验要求

1. 了解PC系统的软中断指令的原理
2. 掌握x86汇编语言软中断的响应处理编程方法
3. 扩展实验四的的内核程序，增加输入输出服务的系统调用。
4. C语言的库设计，实现putch()、getch()、printf()、scanf()等基本输入输出库过程。
5. 编写实验报告，描述实验工作的过程和必要的细节，如截屏或录屏，以证实实验工作的真实性。

实验方案

实验环境

硬件：个人计算机

操作系统：Windows 10

虚拟机软件：Bochs

实验的模式：x86_64长模式

实验开发工具

开发环境：Windows Subsystem for Linux

语言工具：x86汇编语言、C语言

编译器：gcc

汇编器：as（gcc里的汇编器）

汇编调试工具：bochsdbg

链接器：ld

反汇编工具：objdump

打包文件工具：ar

磁盘映像文件浏览编辑工具：WinHex

代码编辑器：Visual Studio Code

程序设计

根据实验要求，本次实验主要的工作如下：

- 实现一个类似于PCB的数据结构
- 实现两个函数save()和restart()，用于保存进程的PCB结构和恢复PCB结构
- 利用中断实现系统调用功能
- 为系统增加一些系统调用号
- 利用系统调用，完成一些基础的C库的设计

PCB数据结构的设计

参考课本给出简化版进程控制块

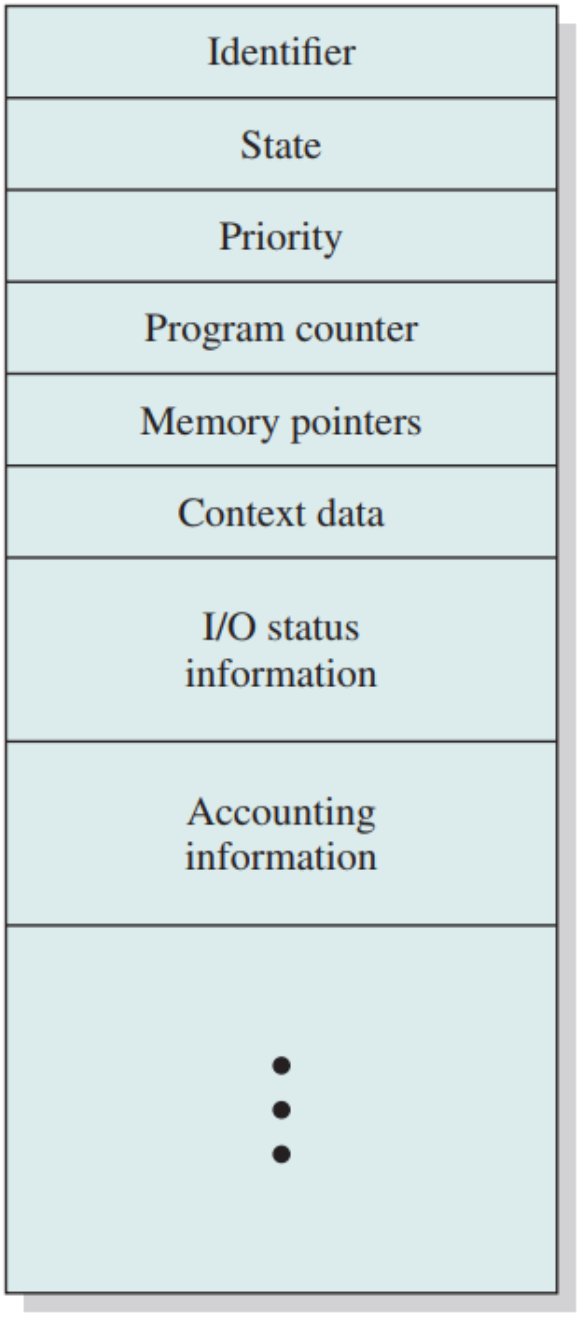


Figure 3.1 Simplified Process Control Block

由于本实验不是要实现进程模型，我只选择了其中的`Context data`部分进行了设计，整个结构体内容如下：

```
struct TrapFrame {
    uint64_t rax;
    uint64_t rbx;
    uint64_t rcx;
    uint64_t rdx;
    uint64_t rsi;
    uint64_t rdi;
    uint64_t rsp;
    uint64_t r8;
    uint64_t r9;
```

```
    uint64_t r10;
    uint64_t r11;
    uint64_t r12;
    uint64_t r13;
    uint64_t r14;
    uint64_t r15;
    uint64_t rbp;
};
```

因为我是在长模式下进行实验，所有的通用寄存器都变成了64位的，并且增加了r8到r15这8个寄存器。我的整个设计思路也很简单，因为CPU里的寄存器的值就是一个`Context data`，把它们记下来就可以了。

注：此数据结构在下一个多进程实验中需要进行扩展

save()函数和restart()函数的设计

这部分的设计，我采取了跟老师不一样的思路。整体实现方面，我采取了C语言+汇编语言混合编程。

首先，save()函数是在中断入口部分被调用的，因为要保存好所有的CPU寄存器，我首先采用的是在C语言编写的中断服务程序里调用save()函数。但是很不幸，用gcc编译后的代码，反汇编以后的结果是这样的：

```
    push %rbp
    movq %rsp, %rbp
    movq $0x0, %rax
    call save
```

前面的`push %rbp`还可以通过查找栈方式获得原始的`rbp`寄存器的值，但是在`call save`之前的一条指令，直接修改了`rax`的值，导致原有的值丢失，这是一个很大的问题。

因此，为了`save`功能的正常运行，我将所有的中断服务程序的入口都采取汇编语言进行编写，代码的框架（我选取了键盘中断作为示例，`KbHandler`是原来的键盘中断服务程序）：

```
.global KbInt
.type   KbInt, @function
KbInt:
    push %rbp
    call save
    movq %rax, %rdi
    call KbHandler
    jmp restart
```

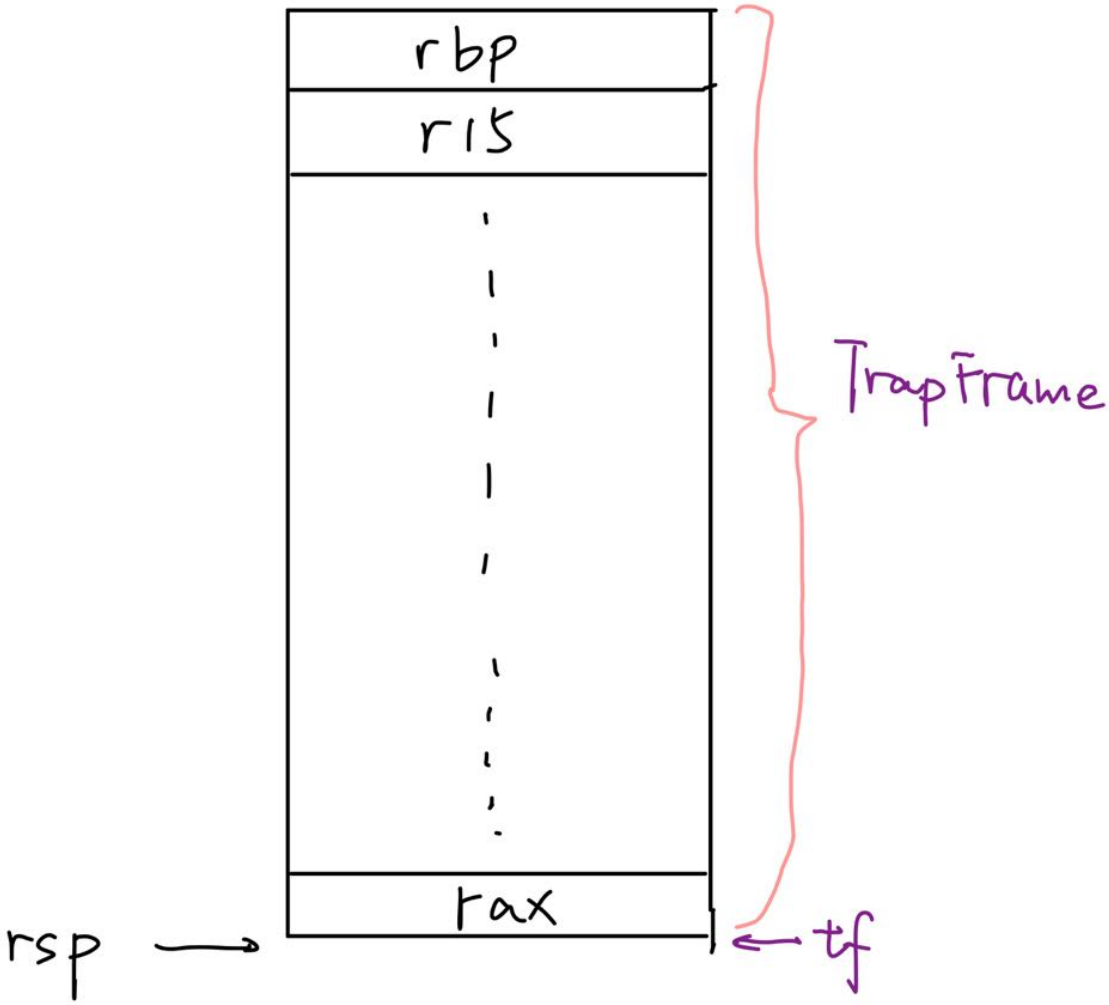
根据上面设计的中断服务程序流程，我设计了如下的save()函数：

- 记录调用者的返回地址
- 保存所有的寄存器
- 把保存后的堆栈起始地址作为返回值返回
- 恢复返回地址，回到调用方

```
.global save
```

```
# struct TrapFrame* save()
save:
.code64
    # 记录调用者的返回地址
    pop %rbp

    # 保存所有的寄存器
    push %r15
    push %r14
    push %r13
    push %r12
    push %r11
    push %r10
    push %r9
    push %r8
    push %rsp
    push %rdi
    push %rsi
    push %rdx
    push %rcx
    push %rbx
    push %rax

    movq %rsp, %rax

    # 恢复返回地址
    push %rbp
    lea 0x8(%rsp), %rbp
    ret
```

这部分实际上使用了一些技巧：在调用save函数前，预先压入了`rbp`寄存器，我们就可以在save函数里使用`rbp`暂存函数返回地址了。同时，虽然看上去只是简单的把所有寄存器的值压入栈内，实际上，我是把push完最后一个寄存器后的堆栈地址，作为返回值返回了。图示说明：



也就是说，可以把返回值当成是一个指针，而这个指针指向的内容，就是我上面设计的`struct TrapFrame`这个数据结构。而这样，我们就可以利用这个返回值作为该进程的PCB结构。

restart()函数的思路是相似的，就是把save()函数的过程反过来就好了

```
# void restart()
.global restart
restart:
    # EOI
    movb $0x20, %al
    out %al, $0x20

    # 恢复原来的寄存器
    pop %rax
    pop %rbx
    pop %rcx
    pop %rdx
    pop %rsi
    pop %rdi
    pop %rsp
    pop %r8
    pop %r9
```

```
    pop %r10
    pop %r11
    pop %r12
    pop %r13
    pop %r14
    pop %r15
    pop %rbp

    iretq
```

利用中断实现系统调用功能

整个系统调用功能的流程：

- save()保存原有寄存器的值
- 根据rax的值选择对应的功能程序
- restart()恢复原来的程序现场

伪代码如下：

```
uint64_t Syscall() {
    struct TrapFrame *tf = save();
    SyscallHandler(tf);
    restart();
}

void SyscallHandler(struct TrapFrame *tf) {
    switch (tf->rax) {
        case 0: ...
        case 1: ...
        case 2: ...
        case 3: ...
        case 4: ...
        case 5: ...
    }
}
```

值得注意的一点，因为我在save函数保存了原来用户进程的PCB数据结构，我在系统调用对应的C程序代码里，就可以直接利用结构体`struct TrapFrame`访问成员变量的方式，对功能号进行选择。

同时，由于可变参数在函数之间传递过于复杂，我这里限定了系统调用最多只有6个入口参数，分别记录在寄存器rdi、rsi、rdx、rcx、r8、r9。

为系统增加一些系统调用号

本原型操作系统现有的系统调用如下：

- sys_putc
- sys_puts
- sys_getchar
- sys_gets
- sys_exec
- sys_exit

前面四个就是简单的字符输入和字符显示的函数，在之前的实验里已经有完整的实现了，这里就不详细说明了。而后面两个系统调用是新加入的，对应功能解释如下：

- sys_exec 实现的是加载用户程序功能，即把用户程序从磁盘中读入内存，并将当前系统控制权交给该用户程序。
- sys_exit则是退出用户程序，返回内核准备接受命令的状态。

相应的代码：

```
uint64_t user_program_stack = 0x0;

/* 指明要加载的用户程序扇区号和所需扇区数 */
void sys_exec(unsigned int sec_no, unsigned int num_of_sec) {
    /* 加载用户程序到内存里 */
    void *user_code_addr = 0x400000;
    readsect(user_code_addr, sec_no, num_of_sec);

    /* 保存当前的堆栈地址，便于找到用户程序的返回地址 */
    __asm__("movq %%rsp, %0":"=r"(user_program_stack)::);

    /* 打开可屏蔽中断，将控制权交给用户程序 */
    __asm__("sti\n\tcall %0\n\t"::"r"(user_code_addr):);
}

void sys_exit() {
    /* 找到用户程序的返回地址 */
    user_program_stack -= 8;

    /* 返回内核 */
    asm("movq %0, %%rsp\n\tret\n\t"::"r"(user_program_stack):);
}
```

因为还没有实现`Scheduler`的功能，这部分实现还是比较粗糙，需要在下一个多进程实验里进行进一步的完善。

利用系统调用，完成一些基础的C库的设计

系统调用的基本格式

这部分的设计我希望全部采用C语言进行编写，但因为系统调用必须使用`int`中断指令进行调用。因此，我采取C语言＋内嵌汇编的操作：

```
void exec(unsigned int sec_no, unsigned int num_of_sec) {
    asm("int $0x80\n\t"::"a"(4), "D"(sec_no), "S"(num_of_sec):);
}
```

内嵌汇编的语法我参考的是：**https://gcc.gnu.org/onlinedocs/gcc/Using-Assembly-Language-with-C.html**，这里不多加阐述。

目前C库里的函数

现在我编写的C库函数有

- exec
- exit
- scanf
- getchar
- gets
- printf
- putc
- puts
- strlen
- strncmp

其中exec、exit、getchar、gets、putc、puts是直接调用上面的系统调用。

strlen和strncmp的实现比较简单，这里直接贴出代码了：

```
int strlen(char *str) {
    int count = 0;
    while (str[count++] != '\0') {}
    return count - 1;
}

/* Compare two string
** 0 for same, -1 for not the same
*/
int strncmp(char *str1, char *str2, unsigned int n) {
    for (int i = 0; i < n; ++i) {
        if (str1[i] != str2[i])
            return -1;
    }
    return 0;
}
```

scanf和printf的实现略微有点复杂，但整体上思路类似，此处描述printf的具体实现：

- 利用va_list实现对于可变参数的逐一读取
- 扫描输入的字符串
  - 如果是正常的字符，直接输出
  - 带有%前缀的进行判断
    - %d输出的是数字，要进行数字到字符串的转换，然后利用putc进行逐字符输出
    - %s输出的是字符串，直接调用putc进行逐字符输出

printf的源代码：

```
/* 仅支持非负整数、字符串输出
** 暂时不支持负数、浮点数输出
** 还没有错误处理！！！
*/
void printf(char *str, ...) {
    va_list ap;
    int str_len = strlen(str), arg_len = 0;

    /* 计算总共输入参数数量 */
    for (int i = 0; i < str_len; ++i) {
        if (str[i] == '%') {
            arg_len++;
        }
    }

    /* 初始化可变参数列表变量 */
    va_start(ap, arg_len);

    for (int i = 0; i < str_len; ++i) {
        if (str[i] == '%') {
            /* 处理数字输出 */
            if (str[i + 1] == 'd') {
                /* 读取参数 */
                int digit_temp = va_arg(ap, int);

                /* int2str */
                int dig_pos = 0;
                char digit_buf[9] = {0};    /* 一般int只有9位 */
                while (digit_temp > 0) {
                    digit_buf[dig_pos++] = digit_temp % 10 + '0';
                    digit_temp /= 10;
                }

                /* 输出字符串 */
                for (int j = dig_pos - 1; j >= 0; --j) {
                    putc(digit_buf[j]);
                }
            }
```

```
            /* 处理字符串输出 */
            else if (str[i + 1] == 's') {
                /* 读取参数 */
                char *print_str = va_arg(ap, char*);

                /* 输出字符串 */
                int index = 0;
                while (print_str[index] != '\0') {
                    putc(print_str[index++]);
                }
            }
            ++i;    /* 跳过%后面的字符 */
        }

        /* 正常输出 */
        else {
            putc(str[i]);
        }
    }
    va_end(ap);
}
```

scanf的源代码：

```
/* 暂时不提供异常处理 */
void scanf(char *str, ...) {
    va_list ap;
    int str_len = strlen(str), arg_len = 0;

    /* 计算总共输入参数数量 */
    for (int i = 0; i < str_len; ++i) {
        if (str[i] == '%') {
            arg_len++;
        }
    }

    va_start(ap, arg_len);
    for (int i = 0; i < str_len; ++i) {
        if (str[i] == '%') {
            char ch;
            while ((ch = getchar()) == ' ' || ch == '\n') {  /* 忽略开头的空格 */
                /* do nothing */;
            }

            /* 处理数字输入 */
            if (str[i + 1] == 'd') {
                /* 处理输入 */
                int dig_pos = 0, num_res = 0;
                char digit_buf[9] = {0};    /* 一般int只有9位 */
                for (ch; (ch >= '0' && ch <= '9'); ch = getchar()) {
```

```
                    digit_buf[dig_pos++] = ch;
                }

                /* str2int */
                for (int j = 0; j < dig_pos; ++j) {
                    num_res = num_res * 10 + (digit_buf[j] - '0');
                }

                /* 赋值语句 */
                int *scanf_num = va_arg(ap, int *);
                *scanf_num = num_res;
            }

            /* 处理字符串的输入 */
            else if (str[i + 1] == 's') {
                char *scanf_str = va_arg(ap, char*);
                int str_pos = 0;
                for (ch; ch != ' ' && ch != '\n'; ch = getchar()) {
                    scanf_str[str_pos++] = ch;
                }
                scanf_str[str_pos] = '\0';
            }
            ++i;    /* 跳过%后面的字符 */
        }
    }
}
```

静态库设计

虽然代码部分已经完成了，但我认为，这对于一个原型操作系统还是不够的。想象一下，假如上面的C库作为一个整体的libc.c，被完整地链接到一个用户程序里，整个用户程序链接后的.com文件大小是很恐怖的。而且，随着libc.c代码的增加，用户程序在没有添加额外功能情况下，整个大小也会跟着一起膨胀。这部分就涉及了链接方面的知识了。

最完美的解决办法是使用动态链接技术，将整个libc编译成libc.so文件，加载到固定的共享内存位置。这样，用户程序在使用库函数的时候，只需要访问对应的共享库就行了。很可惜的是，这部分实现涉及到另外的一些数据结构，有一点复杂。

与之相对的，就是静态链接技术，这也是我采取的办法。具体的实现是，将每个库函数分别放在一个文件里进行编译，再利用 `ar` 程序打包成 `libc.a` 这个静态库。这么做的好处是，在链接的时候，链接器只会把该用户程序使用到的库函数链接到可执行文件里，而丢弃其他的函数。

实现后的libs文件夹结构如下：

```
# ouyhlan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab5 on git:master x [11:13:44]
→  tree libs
libs
├── exec.c
├── exit.c
├── getchar.c
├── gets.c
├── libc.c
├── printf.c
├── putc.c
├── puts.c
├── scanf.c
├── strlen.c
└── strncmp.c

0 directories, 11 files
```

每个函数都有单独对应的文件，而libc.c则是记录所有的库函数。

实验文件管理

增加了C库的设计后，我重新调整了文件夹的安排，现在的安排如下：

```
→  tree
.
├── Makefile
├── bochsrc.bxrc
├── boot
│   ├── ReadSect.c
│   ├── SetPageTable.c
│   ├── boot1.S
│   ├── boot2.S
│   └── boot3.S
├── img
│   └── os1.img
├── include
│   ├── asm.h
│   ├── driver.h
│   └── libc.h
├── kern
│   ├── alltrap.S
│   ├── driver.c
│   ├── driver.s
│   ├── main.S
│   └── shell.c
├── libs
│   ├── exec.c
│   ├── exit.c
│   ├── getchar.c
│   ├── gets.c
│   ├── libc.c
│   ├── printf.c
│   ├── putc.c
│   ├── puts.c
│   ├── scanf.c
│   ├── strlen.c
│   └── strncmp.c
└── user_program
    ├── user1.c
    ├── user2.c
    ├── user3.c
    └── user4.c

6 directories, 31 files
```

可以看出来，整个文件夹的结构是非常清晰的。

实验过程和结果

（1）修改实验4的内核代码，调用save()保存中断现场，处理完后用restart()恢复中断现场。

save()函数和restart()函数

具体设计思路已经在上面的程序设计中阐述过，这里就不再重复了

```
# struct trapframe *save()
.global save
save:
.code64
    # 记录调用者的返回地址
    pop %rbp

    # 保存所有的寄存器
    push %r15
    push %r14
    push %r13
    push %r12
    push %r11
    push %r10
    push %r9
    push %r8
    push %rsp
    push %rdi
    push %rsi
    push %rdx
    push %rcx
    push %rbx
    push %rax

    movq %rsp, %rax

    # 恢复返回地址
    push %rbp
    lea 0x8(%rsp), %rbp
    ret

# void restart()
.global restart
restart:
    # EOI
    movb $0x20, %al
    out %al, $0x20
```

```
    # 恢复原来的寄存器
    pop %rax
    pop %rbx
    pop %rcx
    pop %rdx
    pop %rsi
    pop %rdi
    pop %rsp
    pop %r8
    pop %r9
    pop %r10
    pop %r11
    pop %r12
    pop %r13
    pop %r14
    pop %r15
    pop %rbp
    iretq
```

修改原有中断

因为增加了save()和restart()函数，我修改了原有的中断服务程序的结构。

键盘中断

中断程序结构

```
int 0x21 -> KbInt -> KbHandler
```

源代码

KbInt

```
.global KbInt
.type   KbInt, @function
KbInt:
    # 调用save
    push %rbp
    call save

    # 调用KbHandler
    movq %rax, %rdi
    call KbHandler

    # 调用restart
    jmp restart
```

KbHandler.c

```
void KbHandler(struct TrapFrame *tf) {
    unsigned char scan_code = inp(0x60);

    /* 暂时不支持大写输入 */
    if (scan_code < 0x80) {
        char ch = normalmap[scan_code];
        kb_manager.buf[kb_manager.wpos] = ch;
        kb_manager.wpos = (kb_manager.wpos + 1) % BUFSIZE;
    }
}
```

时钟中断

中断程序结构

时钟中断程序结构：

```
int 0x20 -> CcInt -> CcHandler
```

源代码

CcInt

```
.global CcInt
.type   CcInt, @function
CcInt:
    # 调用save
    push %rbp
    call save

    # 调用CcHandler
    movq %rax, %rdi
    call CcHandler

    # 调用restart
    jmp restart
```

CcHandler.c

```
static int cc_index = 0;
void CcHandler(struct TrapFrame *tf) {
    char *hot_wheels_shape = "|/\\";

    unsigned char *vga_addr = 0xb8000;

    /* 显示风火轮 */
    vga_addr[((24 * 80) + 79) * 2] = hot_wheels_shape[cc_index];
    vga_addr[((24 * 80) + 79) * 2 + 1] = 0x30;

    /* 更新风火轮状态 */
    cc_index = (cc_index + 1) % 3;
}
```

（2）内核增加软中断的处理程序

与老师上课讲得不一样，因为我的原型操作系统里的中断号20h和21h已经被时钟中断和键盘中断占用，所以我调整了一下：

- int 0x22 功能未定，先实现为屏幕某处显示INT22H。
- int 0x80 为系统调用
- 实验要求里int 20h实现户程序结束是返回内核准备接受命令的状态，改为系统调用里的exit

用户程序结束时返回内核

这部分功能由系统调用里的exit实现。这样设计的初衷，可以给用户进程一个统一的退出接口exit()。

值得注意的是，这部分的实现与用户程序加载的系统调用exec()是对应的关系。exec()函数会记录用户程序的返回地址，而exit()函数利用这个地址就可以直接返回内核了。

```
uint64_t user_program_stack = 0x0;

/* 指明要加载的用户程序扇区号和所需扇区数 */
void sys_exec(unsigned int sec_no, unsigned int num_of_sec) {
    /* 加载用户程序到内存里 */
    void *user_code_addr = 0x400000;
    readsect(user_code_addr, sec_no, num_of_sec);

    /* 保存当前的堆栈地址，便于找到用户程序的返回地址 */
    __asm__("movq %%rsp, %0":"=r"(user_program_stack)::);

    /* 打开可屏蔽中断，将控制权交给用户程序 */
    __asm__("sti\n\tcall %0\n\t"::"r"(user_code_addr):);
}

void sys_exit() {
    /* 找到用户程序的返回地址 */
    user_program_stack -= 8;
```

```
    /* 返回内核 */
    asm("movq %0, %%rsp\n\tret\n\t"::"r"(user_program_stack):);
}
```

这样设计的话，还可以为后面的多进程模型提供了一些便利之处——用户进程结束后，系统内核可以在exit()跳转到Scheduler，轻松地完成进程资源释放和进程的切换。

int 0x80中断处理程序（系统调用）

中断程序结构

```
int 0x80 -> Syscall -> SyscallHandler
```

源代码

Syscall

```
.global Syscall
Syscall:
    # 调用save
    push %rbp
    call save

    # 调用SyscallHandler
    movq %rax, %rdi
    call SyscallHandler

    # 调用restart
    jmp restart
```

SyscallHandler.c

```
void SyscallHandler(struct TrapFrame *tf) {
    /* 功能号选择 */
    switch (tf->rax) {
        case 0: sys_putc((char)tf->rdi) ;break;
        case 1: sys_puts((char*)tf->rdi) ;break;
        case 2: __asm__("sti"); tf->rax = sys_getchar(); __asm__("cli"); break;
        case 3: __asm__("sti"); sys_gets((char*)tf->rdi); __asm__("cli"); break;
        case 4: sys_exec((unsigned int)tf->rdi, (unsigned int)tf->rsi); break;
        case 5: sys_exit(); break;
    }
}
```

设计思路已经在上面一部分的程序设计有所体现了。

int 0x22中断处理程序（屏幕某处显示Int 0x22）

中断程序结构

```
int 0x22 -> Int22 -> Int22Handler
```

源代码

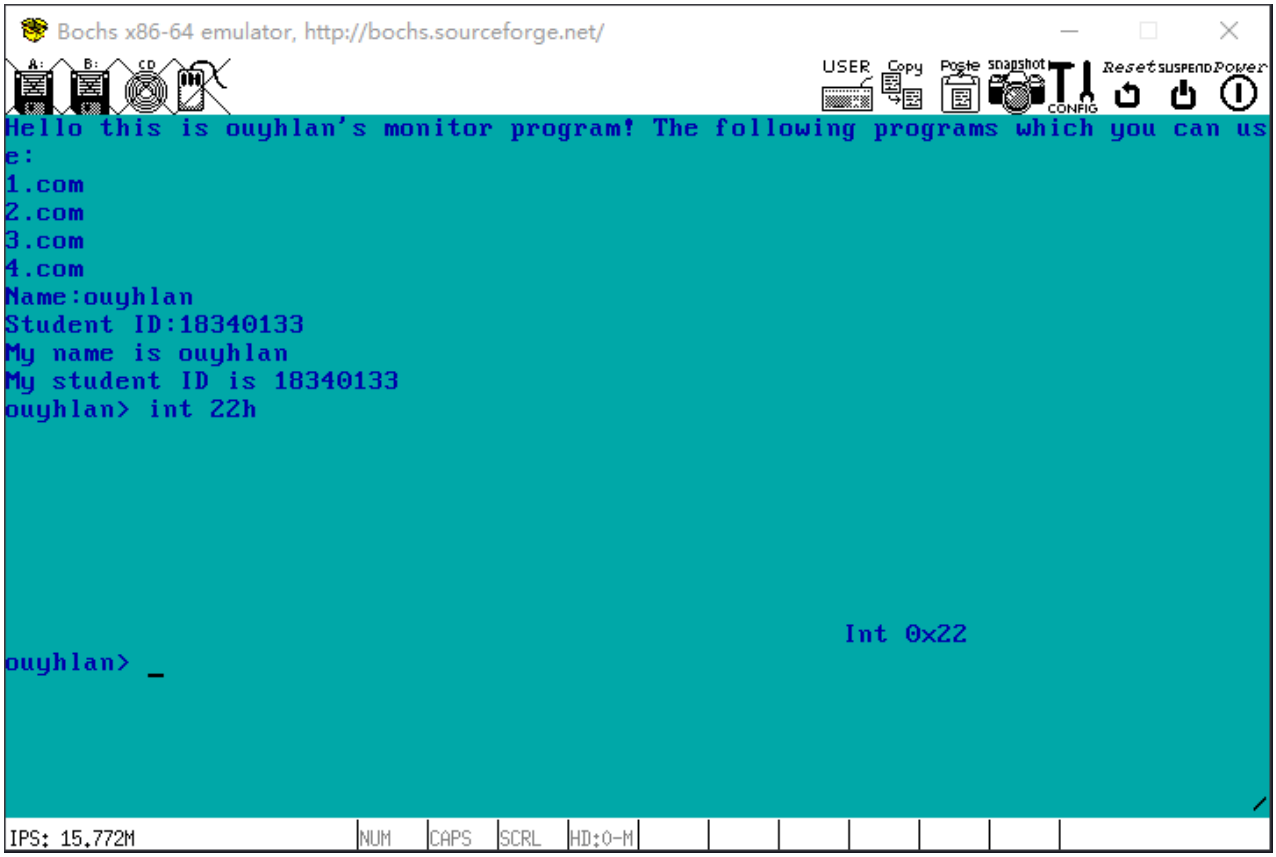
Int22

```
.global Int22
.type   Int22, @function
Int22:
    # 调用save
    push %rbp
    call save

    # 调用Int22Handler
    movq %rax, %rdi
    call Int22Handler

    # 调用restart
    jmp restart
```

Int22Handler.c

```
void Int22Handler(struct TrapFrame *tf) {
    /* 选择一个显示的位置 */
    CurPosition pos = {18, 53};
    sys_SetCurPos(pos);

    /* 显示Int 0x22 */
    sys_puts("Int 0x22");
}
```

实验效果



从上图可以看出，Int 0x22出现在了对应的位置了。

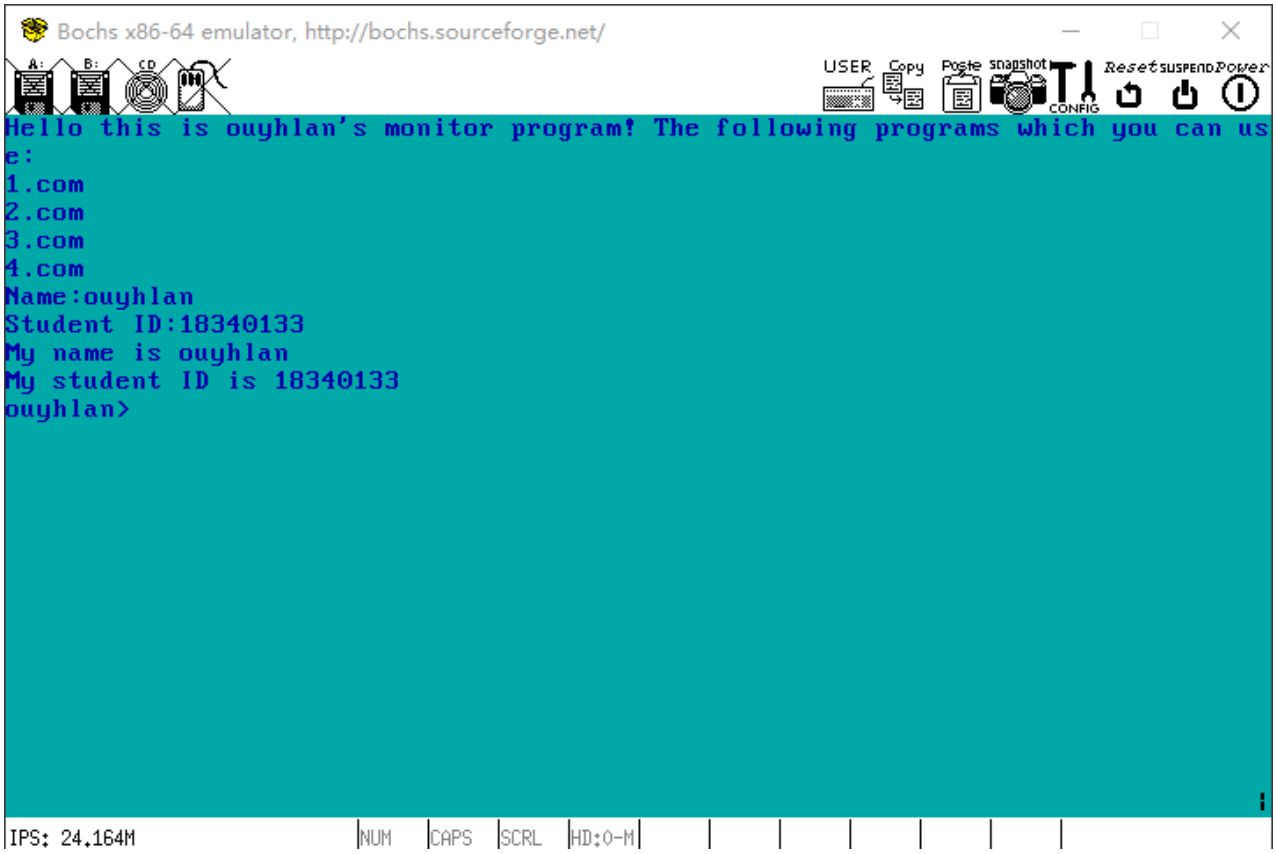
（3）保留无敌风火轮显示，取消触碰键盘显示OUCH!这样功能。

在这部分的设计里，我不仅保留了无敌风火轮的显示，我还分离了shell和内核的关系。

虽然现在我的shell代码还和系统调用代码一起链接，但在这次实验里，我的shell程序完全没有使用任何内核里的函数，而完全是使用了新添加的C库里的库函数。

可以说，我的原型操作系统里的shell程序也是一个用户进程（系统内核开机时自动创建的）。shell程序提供了一些提示信息，同时处理与用户交互。同时，shell会通过系统调用exec产生出一个新的用户进程进行用户程序的使用。

不仅如此，因为有了printf和scanf这些库函数，我的shell在交互方面还增加了一些功能，如下截图：



在开始的时候，我会要求用户输入姓名和学号作为系统登陆信息。输入完毕后，shell的输入提示符就会显示刚刚登陆的用户名。同时，右下角的风火轮仍然存在。

（4）进行C语言的库设计，实现putch()、getch()、gets()、puts()、printf()、scanf()等基本输入输出库过程

这部分的源代码已经在程序设计部分解释过，这里就展示我是如何打包这个libc.a库的。

编译源代码

因为有Makefile，我不需要手动进行编译，下面截取Makefile的编译部分代码

```
CC = gcc
CFLAGS = -ffreestanding -I ./include -mgeneral-regs-only
libc_obj = exec.o exit.o scanf.o getchar.o gets.o printf.o putc.o puts.o strlen.o
strncmp.o

# C program compile rule
%.o: %.c
    $(CC) -c $(CFLAGS) $< -o $@
```

打包静态库

这个打包实现的功能就是：

```
exec.o + exit.o + scanf.o + getchar.o + gets.o + printf.o + putc.o + puts.o + strlen.o
+ strncmp.o
->
libc.a
```

相应的Makefile命令：

```
libc.a:$(libc_obj)
    ar rcs -o libc.a $(libc_obj)
```

（5）利用自己设计的C库libs.a，编写一个使用这些库函数的C语言用户程序

因为我的shell程序基本上跟老师提供那个程序类似，并且使用到的库函数更多，所以我就直接用我的shell程序代替了。

shell程序流程：

- 打印提示符
- 等待用户输入用户名和学号
- 显示用户的用户名和学号
- 等待用户输入命令
- 判断用户命令的合法性
- 使用库函数exec加载用户程序

用户程序：

- ...
- 使用库函数 `exit` 把控制权交回给shell

源代码

```
/* shell.c */
#include "libc.h"
#define USER_1 32
#define USER_2 36
#define USER_3 40
#define USER_4 44
#define NUM_OF_PROGRAM 4

void Welcoming() {
    char *program_name[] = {"1.com", "2.com", "3.com", "4.com"};
    char name[20];
    char str[32];
    int student_id;
```

```
    puts("Hello this is ouyhlan's monitor program! The following programs which you can
use: ");

    for (int i = 0; i < NUM_OF_PROGRAM; ++i) {
        puts(program_name[i]);
    }

    printf("Name:");
    scanf("%s", name);
    printf("Student ID:");
    scanf("%d", &student_id);

    printf("My name is %s\nMy student ID is %d\n", name, student_id);
    while (1) {
        printf("%s> ", name);
        gets(str);
        int program_id = 0;
        if (strncmp(str, "1.com", 5) == 0 && (strlen(str) == 5)) {
            program_id = 1;
        }
        else if (strncmp(str, "2.com", 5) == 0 && (strlen(str) == 5)) {
            program_id = 2;
        }
        else if (strncmp(str, "3.com", 5) == 0 && (strlen(str) == 5)) {
            program_id = 3;
        }
        else if (strncmp(str, "4.com", 5) == 0 && (strlen(str) == 5)) {
            program_id = 4;
        }
        else if (strncmp(str, "int 22h", 7) == 0 && (strlen(str) == 7)) {
            asm("int $0x22\n\t");
        }
        else if (strncmp(str, "exit", 4) == 0 && (strlen(str) == 4)) {
            puts("\n[System exits normally!]\n");
            break;
        }
        else if (strlen(str) == 0) {
            continue;
        }
        else {
            puts("Wrong input! Please enter again!\n");
            continue;
        }
        switch (program_id) {
        case 1:
            exec(USER_1, 4);
            break;
        case 2:
            exec(USER_2, 4);
            break;
        case 3:
            exec(USER_3, 4);
```

```
                break;
            case 4:
                exec(USER_4, 4);
                break;
            default:
                /* Do nothing */;
            }
            puts("");
        }
    }
```

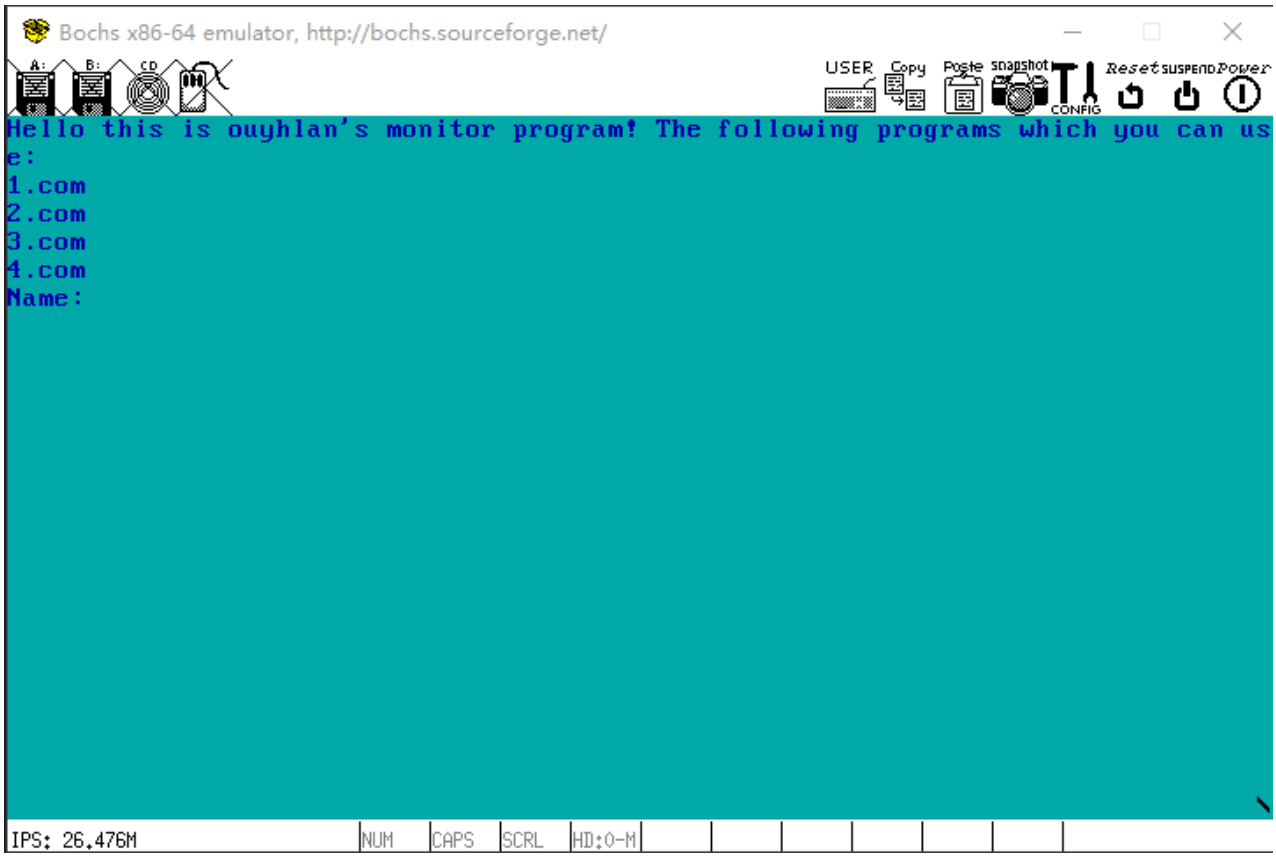
显然，我在上面的程序里使用了的库函数有：puts、printf、scanf、gets、strncmp、strlen、exec，这可以证明我的库函数的运行一切正常。

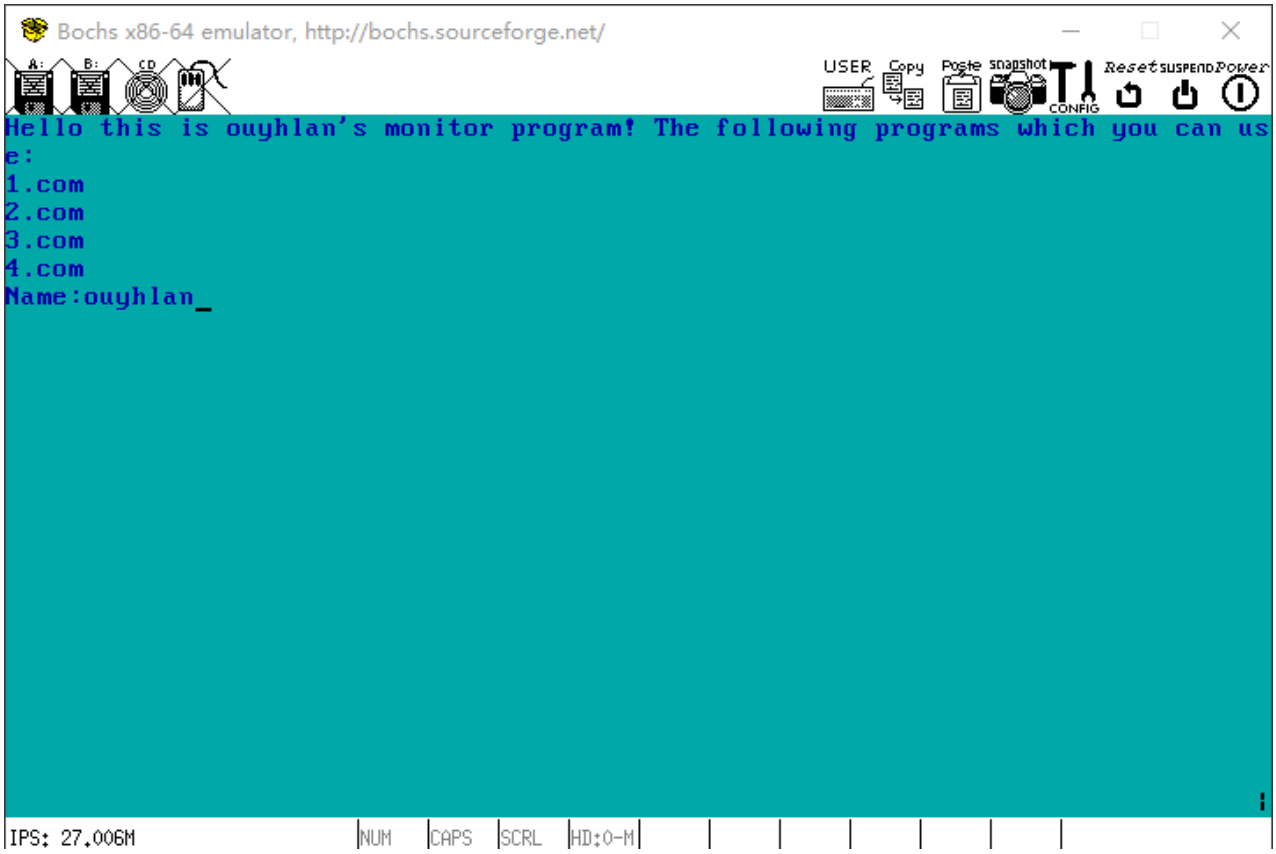
链接C库

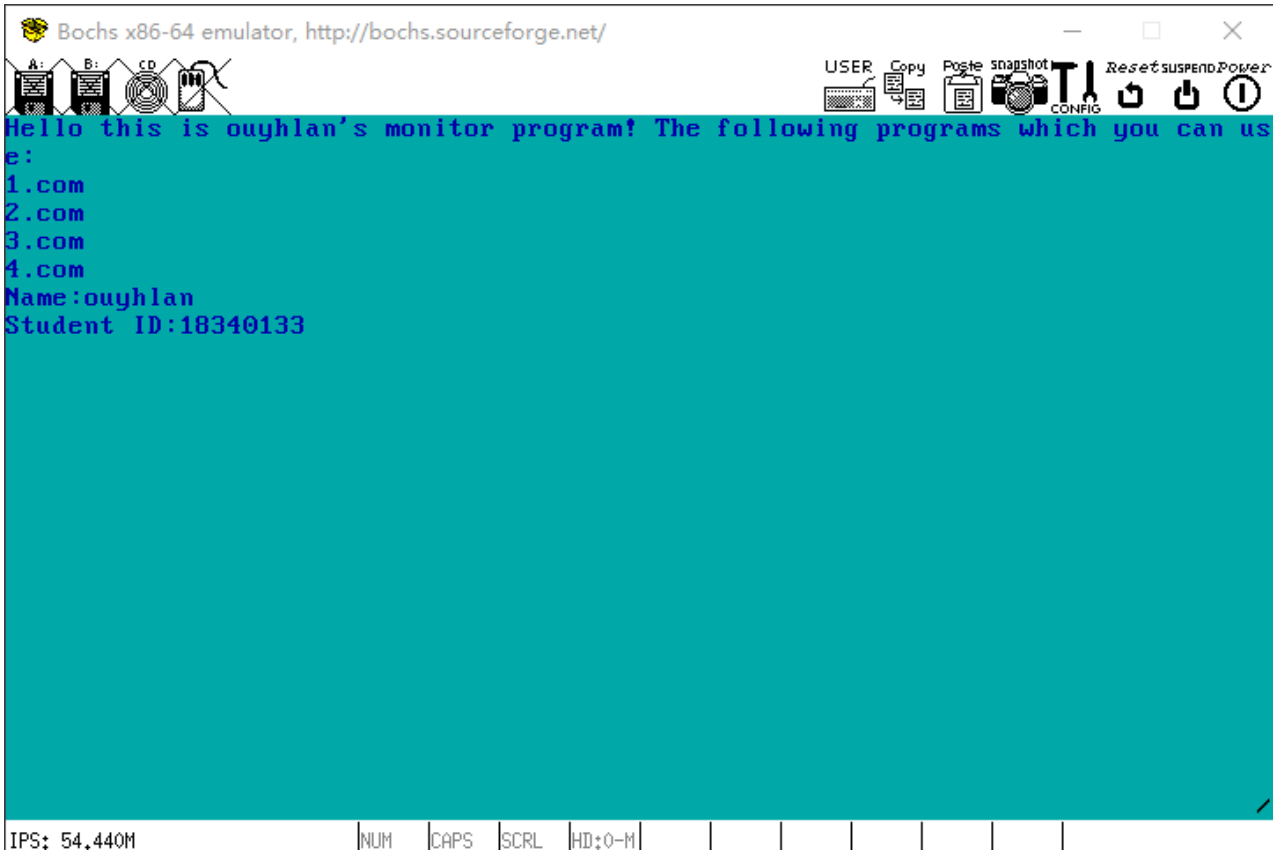
因为（4）已经产生了打包好的静态库`libc.a`，我的shell.o在链接的时候只需要加一个静态链接命令（-static）即可：

```
    ld --oformat binary -Ttext 0xffff800000000000 -Tdata 0xffff800000002000 -Tbss
0xffff800000001c00 -static shell.o libc.a -o shell.bin
```

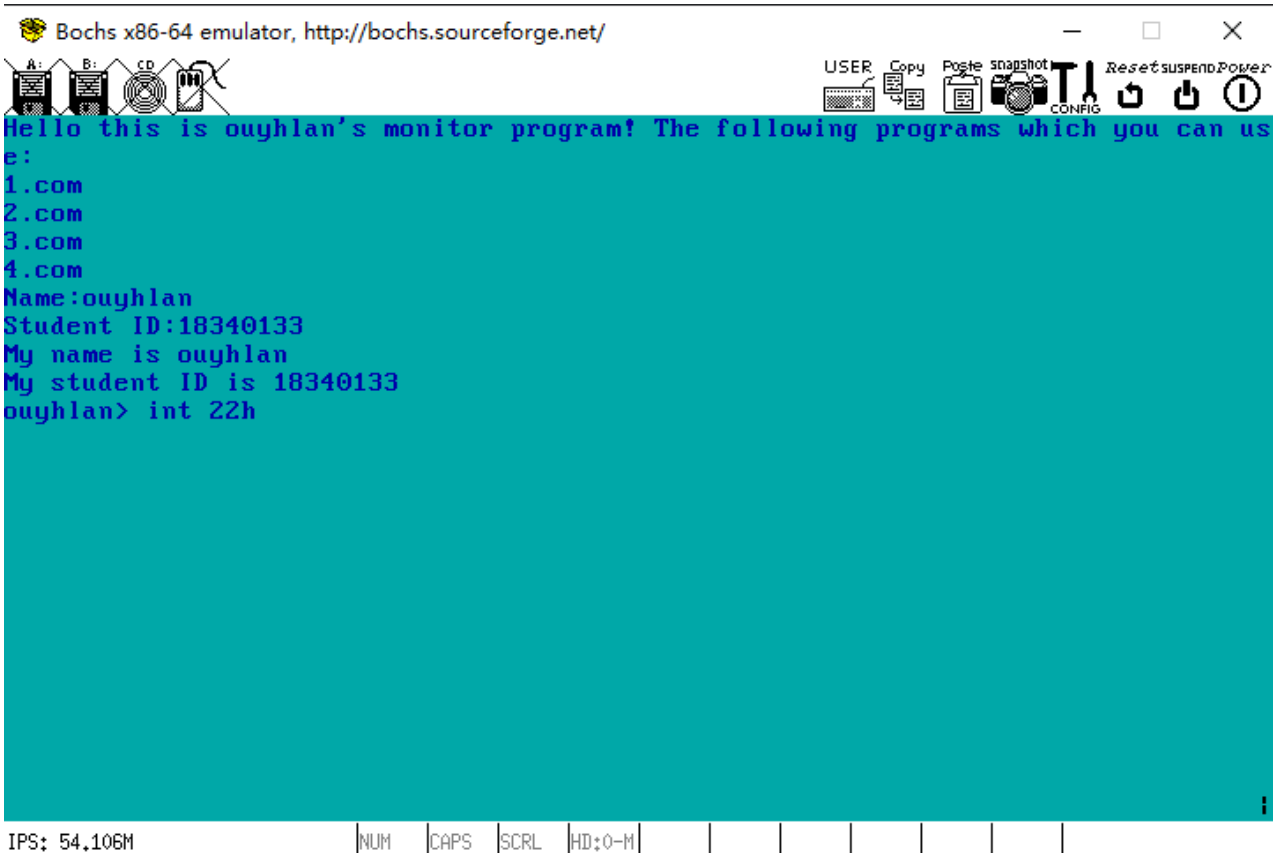
运行截图




Bochs x86-64 emulator, http://bochs.sourceforge.net/
Hello this is ouyhlan's monitor program! The following programs which you can us
e:
1.com
2.com
3.com
4.com
Name:ouyhlan
IPS: 27.006M
NUM
CAPS
SCRL


Bochs x86-64 emulator, http://bochs.sourceforge.net/
Hello this is ouyhlan's monitor program! The following programs which you can us
e:
1.com
2.com
3.com
4.com
Name:ouyhlan
Student ID:18340133
IPS: 54.440M
NUM
CAPS
SCRL

登陆后的显示：



更多的演示放在了演示视频里了。

实验中遇到的问题

movq指令不支持64位的地址赋值

如果使用了，会出现如下的错误：

```
(.text+0x9a): relocation truncated to fit: R_X86_64_32S against `Int22'
```

这个问题很奇怪，因为AMD64官方手册里有：

MOV *reg/mem64*, *reg64*	89 /r	Move the contents of a 64-bit register to a 64-bit destination register or memory operand.

这明显是支持的。

最后，我在StackOverflow里找到了解决答案：

```
movq -> movabsq
```

这个问题比较奇怪，我也解释不清原因。

引入C库以后，用户程序大小出现暴增

我分析了一下反汇编后的代码，出现这个现象的原因是链接的时候，链接器把C库里所有的函数都链接到了用户程序里，但其实有一些根本没有利用。

而解决办法就是我在上面讲得静态库。每个函数编译一份.o文件，再利用`ar`程序打包成静态库。这样，在静态链接的时候，链接器就会帮我们自动寻找需要的库函数源文件了。

实验总结

这次是我在操作系统实验课的第五个实验。在本次实验里，我修改了整个原型操作系统的框架，新增加了libs这个文件夹存放库函数文件。同时，我修改了系统引导文件的结构（在实验报告里没有提到），把它改为了三级引导结构：

- boot1进入保护模式
- boot2设置长模式需要的虚拟内存的页表
- boot3加载内核代码

修改以后，整个引导程序结构变得更加清晰了，引导代码的独立性也加强了。

本来，我是希望使用`syscall`和`sysret`这两条运行时间更快的汇编指令进行系统调用方面的功能，但因为对`STAR`和`LSTAR`寄存器的操作存在问题，我的系统未能正确实现系统调用，所以我才使用`int 0x80`作为系统调用。但这样做，我的`save`和`restart`函数就可以被系统调用复用了，这也算是因祸得福吧。

不仅如此，我在这次实验里还引入了代码版本管理软件git。本来我是想在本次实验就加入多进程模型的，但在尝试后觉得有点不妥，我就把那部分文件留在了git的另外一个分支里，准备下一次写代码的时候merge过来。同时，我看到其它同学有在反映因为操作失误而误删了自己的代码。但在git的管理下，这种失误是可以恢复的。因此，我觉得很有必要使用像git这样优秀的版本控制工具。

因为实验六就是实现多进程模型了，我希望在下一次实验里完善现有的虚拟内存管理机制，完成类似于页表切换、利用TSS寄存器完成系统栈和用户栈的切换，系统态和用户态的切换等等这些保护模式里引入的保护机制吧。