实验报告 实验六

姓名：王钦　学号：13349112　班级：计科二班

实验目的

1. 理解系统调用的实现方法。
2. 实现原型操作系统中一些基本的系统调用。
3. 设计并实现一测试系统调用的用户程序，利用系统调用实现用户界面和内部功能。
4. 在原型操作系统上建立一个初步C语言开发环境，理解操作系统与高级语言之间的关系。

实验内容

在实验四的基础上，进化你的原型操作系统，增加下列操作系统功能：
(1)参考下面的系统调用功能表，增加一些其他功能
系统调用表（局部设计）
(2)扩展MYOS内核，实现上表中的所有(包括你增加的)系统调用，并开发一个用户程序，展示这些系统调用的使用效果。
(3)设计一个C程序库，封闭getch(),gets(),putch(),puts(),scanf()和printf()等利用系统调用实现的细节，并参考下面程序，开发一个用户程序，测试这些函数功能。

```
include stdio.h
main(){
        char ch,str[80];
        int a;
        getch(&ch);
        gets(str);
        scanf “(a=%” d,&a);
        putch(ch);
        puts(str);
        printint “(ch=%c, a=%d, str=%” s, ch, a, str);
}
```

实验平台

gcc+ld+nasm+Linux+vim

算法流程图

功能一览

1. 系统内置功能:
terminal,装载内核shell，为用户提供一个与操作系统交互的工具,开机后自动进入,以下所有功能都在terminal中交互
date, 显示当前日期
time, 显示当前时间
asc, 显示一个字符的asc码
clear, 清除当前屏幕所有字符，刷新屏幕
help, 显示系统帮助信息
man ，显示内置函数的帮助信息,比如 man date ,显示date的相关帮助
start, 开始创建并执行四个进程并且每秒18.2次的调度，分别在屏幕1/4处打印一些个性化信息(不同配置的虚拟机动画速度不一样，建议使用vmware测试)
2. 用户程序:
run ,软盘中含有三个用户程序,输入run 123,可分别执行三个用户程序，当然也可以通过改变执行序列来改变执行的顺序
3. 自定义中断:
时钟中断：通过PTR每秒发出18.2次的信号来从8592芯片的RT0引脚发出终端号int 08h来触发的用户时钟软中断 int 1ch,实现在terminal的右下角 一个横杠在转动.
另外有自定义中断int 33h,int 34h,int 35h,int 36h分别在屏幕四分之一的位置打印个性化信息
4. 进程调度: 软盘中共存放了用于展示进程调度的四个应用程序，每个应用程序分别代表一个进程。开启装载进程并进行进程调度由1中系统内置功能的 start指令激活。

实验步骤及效果

1. 编辑修改ASM 文件,和C文件

2. 使用make命令配合makefile文件进行编译

3. 运行bochs or vmware虚拟机进行测试,进入后所看到的欢迎界面,同时进入terminal界面等待输入指令(测试多进程调度功能时请配合我提交文件中的bochsrc文件使用bochs测试，因为不同配置的虚拟机下延时动画时间不一样，我这里以我的bochs的配置为准)

4. 我们可以执行man python
查看python的作用，主要功能是输入一个数学表达式然后返回表达式的结果，类似python命令行的作用。其他类似的系统内置功能也可以使用man命令来查看帮助信息。 但目前python只支持加法和减法且只能有两个操作数（其实主要为了展示ah=3,4将字符串转为数值和将数值转为字符串的系统调用效果），我这里仿照linux系统的系统调用，设置 int 80h 为所有系统调用的入口。可以看到图中分别输入加法减法，返回计算结果。如果输入的不符合格式就会返回错误提示。最后输入exit退出python命令行工具。

5. 在terminal中输入start命令测试多进程调度功能 6. 接下来测

试ah等于0,1,2,5的系统调用(显示OUCH，字母大小写变化等)，输入 run 2 ,运行第二个用户程序，

下面是第二个用户程序的代码：

```
org 0x1000
;org 0x100

;#0
mov ah,0
int 80h

;LISTEN_EXIT——
listen0:
        mov ah,0
        int 16h


;#2
mov ax,0xb800
mov es,ax
mov dx,1994D
mov ah,2
int 80h

;LISTEN_EXIT——
        mov ah,0
        int 16h

;#1
mov ax,0xb800
mov es,ax
mov dx,1994D
mov ah,1
int 80h

;LISTEN_EXIT——
        mov ah,0
        int 16h



;#5
mov ax,cs
mov ds,ax
mov es,ax

mov cx,0317h  ;position
mov ah,5
mov dx,msg
int 80h


;LISTEN_EXIT——
listen:
        mov ah,0
        int 16h

ret



msg:
        db "hello world!"

times 512-($-$$) db 0    填充剩余扇区;0
```

下面我们将上面代码分解一下，详细介绍。

5.1 首先执行执行ah=0的系统调用将 OUCH 打印在屏幕中间

```
org 0x1000
;org 0x100

;#0
mov ah,0
```

```
int 80h
;LISTEN_EXIT——
listen0:
        mov ah,0
        int 16h
```

5.2 现在按下任意键，将执行ah=2的系统调用把 OUCH的第一个字母大写O变成小写o

```
;#2
mov ax,0xb800
mov es,ax
mov dx,1994D
mov ah,2
int 80h

;LISTEN_EXIT——
        mov ah,0
        int 16h
```

5.3 按下任意键，执行ah=1的系统调用把 oUCH的第一个字母小写o变回大写O

```
;#1
mov ax,0xb800
mov es,ax
mov dx,1994D
mov ah,1
int 80h

;LISTEN_EXIT——
        mov ah,0
        int 16h
```

5.4 按下任意键，执行ah=5的系统调用在屏幕3行17列的位置打印一个helloworld

```
;#5
mov ax,cs
mov ds,ax
mov es,ax

mov cx,0317h  ;position
mov ah,5
mov dx,msg
int 80h


;LISTEN_EXIT——
listen:
        mov ah,0
        int 16h

ret

msg:
        db "hello world!"
```

5.5 返回操作系统

6. 接下来我们测试封装的osclib_share.c库里面实验所要求的getch,gets,scanf,putch,putch,printint ,输入run 1执行第一个用户程序
代码如下:

```
void main(){
        screen_init();
        getch( &ch);
        gets( key);
        putch('\r');
        putch('\n');
        scanf("a=%d",&a);

        putch( ch);
        puts( key);
        printint( "ch=%c,a=%d,str=%s",ch,a,key);

        wait_key();
}
```

6.1 首先程序等待输入一个字符

6.2 输入字符 C,回车再输入一个字符串I am a string回车

6.3 再输入一个int类型的数字16,回车后执行下面代码

```
        putch( ch);
        puts( key);
        printint( "ch=%c,a=%d,str=%s",ch,a,key);
```

可以看到相继打印出了之前输入的内容

7. 和之前的实验一样，内核也有设置中断,系统内置程序和重复运行多个用户程序，这里就不再赘述了。

内存和软盘存储管理

1. 引导程序加载到内存0x7c00处运行
2. 引导程序将操作系统加载到0x7e00处运行
3. 操作系统讲用户程序加载到0x1000处运行
4. 软盘第1个柱面的第一个扇区存储操作系统引导程序
5. 软盘第1个柱面剩下所有扇区2~36扇区存储操作系统内核
6. 软盘第2,3,4柱面分别存储三个用户程序

更多细节信息请阅读我的Makefile文件

主要函数模块解释

内核架构解释:
os.c为内核主要控制模块
osclib.c os.asm主要为os.c提供函数实现.
oslib.asm 为osclib.c提供更底层的函数封装
osclib_share.c,oslib_share.asm 为用户程序中所需要用到的函数，从osclib.c oslib.asm中取出一部分作为内核和用户的共享库(使用户程序体积减少)。
os_syscall.asm 初始化系统调用和设置系统调用相关模块
更多细节信息请阅读我的Makefile文件

1. os.c: main 函数模块，这个在之前报告中已经解释这里就不在赘述

```
void main(){
        //----------------------init
        init_ss();
        screen_init();
        interrupt_init();
        syscall_init();
        print_welcome_msg();
        print_message();
        print_flag(); //root@wangqin4377@:    position
        //----------------------init_end

        while(1){
                char length = listen_key();
                if_screen_scroll(); //bottom of screen
                flag_scroll();//move flag to next line
                print_flag();
        }
}
```

2. 本次试验主要为了实现系统调用的工作，故在 os.asm中实现了下列函数供设置系统调用使用

```
        ;---PARAM: ah is syscall num   ebx is address of syscall  bx:temp cx:function ax:sysnum
        setting_up_syscall:
        mov bx,0
        mov es,bx
        mov al,ah
        mov ah,0
        shl al,2
        mov bx,0xfe00
        add bx,ax
        mov [es:bx],ecx
        ret
```

3. 初始化设置系统调用,设置全部6个系统调用功能，调用上面的函数实现

```
syscall_init:

;----#0 syscall
mov ah,0
mov ecx,0
mov cx,display_center_ouch
call setting_up_syscall

;----#1 syscall
mov ah,1
mov ecx,0
mov cx,letter_upper
call setting_up_syscall

;----#2 syscall
mov ah,2
mov ecx,0
mov cx,letter_lower
call setting_up_syscall

;----#3 syscall
mov ah,3
mov ecx,0
```

```
mov cx,atoi_syscall
call setting_up_syscall

;----#4 syscall
mov ah,4
mov ecx,0
mov cx,itoa_syscall
call setting_up_syscall

;----#5 syscall
mov ah,5
mov ecx,0
mov cx,display_str
call setting_up_syscall
ret
```

4. 对scanf,printint,gets..等函数的封装: 其实这些函数在之前的实验中已经实现，只是名字不一样罢了。本次实验单独抽出来相关代码放在osclib_share.c中。

5. python命令行工具：存放在python_extension.c中,主要功能的就是使用系统调用实现字符串和数值之间的转换。

```
        unsigned short int itoa_temp;
        char * itoa_ans;
        char * itoa( short int x){
                itoa_temp = x;
                __asm__("mov $4,%ah");   // syscall num
                __asm__("push %bp");
                __asm__("int $0x80");    // syscall
                __asm__("pop %bp");
                return itoa_ans;
        }

        char *atoi_temp;
        unsigned short int atoi_ans;
        unsigned short int atoi( char * str){
                atoi_temp = str;
                __asm__("mov $3,%ah");   // syscall num
                __asm__("int $0x80");    //syscall
                return atoi_ans;
        }
```

实验心得及仍需改进之处

实验心得:

本次实验过程中遇到了很多挫折，最大的一个问题就是编写进程调度的算法的时候由于自己疏忽在创建进程启动进程调度的时候没有把时钟中断清零导致，无法发生下一次时钟中断,原本以为0x1c用户时钟中断可以不需要像0x08那样每次对8259芯片进行重新设置。时钟中断正常工作后又发现调度出现一些问题，比如第一个进程的时间片用完之后不能换到第二个进程执行，当然这个问题还是经过调试很容易解决的。后来发现第二个进程执行用完时间片后切换到第一个进程时第一个进程无法按照原来停止执行的cs:ip处进行执行，这个问题经过调试也是比较容易解决的。最难调试的问题是栈的切换，我这里为每个进程都设置了一个应用程序栈，进程时间片用完之后切换进程的时候进程的上下文数据RSS中 也会包

含这个进程所对应的用户程序栈的栈指针。也就是切换进程的时候也要对应用程序栈 ss:sp进行切换,但是c语言调用汇编的时候是向栈里压两个字，而汇编返回c调用处的时候是从栈里弹出一个字到 IP指令指针寄存器。这种不对称的出栈入栈很容易造成使用栈的过程中导致栈数据混乱。而且当时钟中断的处理程序即进程调度程序执行完毕后要从栈里弹出三个字的数据到 IP,CS,Flags这些寄存器，调度的时候就要手动的先把原来的 IP,cs,flags弹出来保存然后压入新进程的 ip,cs,flags,所以保持栈清晰，有规律是非常重要的，否则进程调度将会失败。栈切换这里这里调了很长时间，得到的教训就是一定要注意细节，c调用汇编或汇编调用c都要特别注意有没有冗余的数据被压栈，注意一定要先保存 ip,cs,flags,sp和一些段寄存器的数据,合适的时机再切换栈指针寄存器。当换一个新进程开始执行的时候一定要保证这个进程上次被换下来的时候,栈指针寄存器 sp一定是上次的位置减去三个字，因为要压入 ip,cs,flags。总之注意写代码的时候注意细节总会可以大大降低debug的时间从而减少写代码的总时间，提高编程效率。同时也是 作为一个专业学生好的习惯。这次实验中我优化了操作系统和内核的结构，将 shell和内核分开，使主内核文件 os.c代码量大大减少，向Linux提倡的微内核走进了一步。 实验仍需改进之处:

仍需完善细节，比如说python的命令行工具加入乘法，除法完全是几行代码的问题
可以考虑将用户程序做成elf格式，动态链接系统的代码库。目前的情况是用户程序自己带着一份和操作系统一样的代码库，分别联合编译
考虑精简操作系统代码，减少冗余代码
继续优化调整内核架构和内存磁盘管理