实验五：实现系统调用

(10分)

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

4、掌握用户程序请求系统服务的方法

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

实验内容：

1. 修改实验4的内核代码，先编写save()和restart()两个汇编过程，分别用于中断处理的现场保护和现场恢复，内核定义一个保护现场的数据结构，以后，处理程序的开头都调用save()保存中断现场，处理完后都用restart()恢复中断现场。

保护现场的数据结构:

typedef strut {

int ax;

int bx;

int cx;

int dx;

int cs;

int ds;

int es;

int ss;

int sp;

int bp;

int di;

int si;

int ip;

int flag;

} cpuRegisters

;参考程序

;Minix的save和restart

; save

;=====================================================

save: ; save the machine state in the proc table.

push ds ; stack: psw/cs/pc/ret addr/ds

push cs ; prepare to restore ds

pop ds ; ds has now been set to cs

mov ds,ker\_ds ; word 4 in kernel text space contains ds value

pop ds\_save ; stack: psw/cs/pc/ret addr

pop ret\_save ; stack: psw/cs/pc

mov bx\_save,bx ; save bx for later ; we need a free register

mov bx,dgroup:proc\_ptr ; start save set up; make bx point to save area

add bx,OFF ; bx points to place to store cs

pop PC-OFF[bx] ; store pc in proc table

pop csreg-OFF[bx] ; store cs in proc table

pop PSW-OFF[bx] ; store psw

mov ssreg-OFF[bx],ss ; store ss

mov spreg-OFF[bx],sp ; sp as it was prior to interrupt

mov sp,bx ; now use sp to point into proc table/task save

mov bx,ds ; about to set ss

mov ss,bx ; set ss

push ds\_save ; start saving all the registers, sp first

push es ; save es between sp and bp

mov es,bx ; es now references kernel memory too

push bp ; save bp

push di ; save di

push si ; save si

push dx ; save dx

push cx ; save cx

push bx\_save ; save original bx

push ax ; all registers now saved

mov sp,offset dgroup:k\_stack ; temporary stack for interrupts

add sp,K\_STACK\_BYTES ; set sp to top of temporary stack

mov splimit,offset dgroup:k\_stack ; limit for temporary stack

add splimit,8 ; splimit checks for stack overflow

mov ax,ret\_save ; ax = address to return to

jmp ax ; return to caller; Note: sp points to saved ax

;==================================================

; restart

;==================================================

restart: ; This routine sets up and runs a proc or task.

cmp dgroup:cur\_proc,IDLE; restart user; if cur\_proc = IDLE, go idle

je \_idle ; no user is runnable, jump to idle routine

cli ; disable interrupts

mov sp,dgroup:proc\_ptr ; return to user, fetch regs from proc table

pop ax ; start restoring registers

pop bx ; restore bx

pop cx ; restore cx

pop dx ; restore dx

pop si ; restore si

pop di ; restore di

mov lds\_low,bx ; lds\_low contains bx

mov bx,sp ; bx points to saved bp register

mov bp,SPLIM-ROFF[bx] ; splimit = p\_splimit

mov splimit,bp ; ditto

mov bp,dsreg-ROFF[bx] ; bp = ds

mov lds\_low+2,bp ; lds\_low+2 contains ds

pop bp ; restore bp

pop es ; restore es

mov sp,spreg-ROFF[bx] ; restore sp

mov ss,ssreg-ROFF[bx] ; restore ss using the value of ds

push PSW-ROFF[bx] ; push psw (flags)

push csreg-ROFF[bx] ; push cs

push PC-ROFF[bx] ; push pc

lds bx,DWORD PTR lds\_low ; restore ds and bx in one fell swoop

iret ; return to user or task

1. 内核增加int 20h、int 21h和int 22h软中断的处理程序，其中，int 20h用于用户程序结束是返回内核准备接受命令的状态；int 21h用于系统调用，并实现3-5个简单系统调用功能；int22h功能未定，先实现为屏幕某处显示INT22H。

Int20h中断处理程序(用户程序结束是返回内核):

call save()

return\_kernel()

jmp restart ; 为了结构对称，不会执行此指令。

Int21h中断处理程序(系统调用):

call save()

call sys\_call()

jmp restart

Int22h中断处理程序(屏幕某处显示INT22H):

call save()

call print\_int22h()

jmp restart

1. 保留无敌风火轮显示，取消触碰键盘显示OUCH!这样功能。
2. 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，汇编产生libs.obj。

例子：

public \_printf

\_printf proc

mov ax,0500h ;系统调用5号功能，显示输出一个字符串

push ax

… ； 其他参数进栈

int 21h ；产生中断，

\_printf end

public \_getch

\_getch proc

mov ax,0600h ;系统调用6号功能，从键盘输入一个字符

push ax

… ； 其他参数进栈

int 21h ；产生中断，

\_getch end

public \_gets

\_gets proc

mov ax,0700h ;系统调用7号功能，从键盘输入一个字符串

push ax

… ； 其他参数进栈

int 21h ；产生中断，

\_gets end

public \_puts

\_puts proc

mov ax,0800h ;系统调用7号功能，从键盘输入一个字符串

push ax

… ； 其他参数进栈

int 21h ；产生中断，

\_puts end

1. 利用自己设计的C库libs.obj，编写一个使用这些库函数的C语言用户程序，再编译,在与libs.obj一起链接，产生COM程序。增加内核命令执行这个程序。

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);

}

(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

**实验环境：**

* Windows 10-64bit
* Vmware WorkStation 15 pro 15.5.1 build-15018445：虚拟机软件
* NASM version 2.13.02：汇编程序的编译器，在linux下通过sudo apt-get install nasm下载
* Ubuntu-18.04.4:安装在Vmware的虚拟机上
* 代码编辑器：Visual Studio Code 1.44.2
* GNU ld 2.30：链接器

**实验思路：**

同样地，本次实验是由上一次的实验基础上添加了一些功能，其中最为核心的功能就是系统调用部分。其中，对于本次实验的实验内容的（2）中，要求“**增加int 20h、int 21h和int 22h软中断的处理程序，其中，int 20h用于用户程序结束是返回内核准备接受命令的状态；int 21h用于系统调用，并实现3-5个简单系统调用功能；int22h功能未定，先实现为屏幕某处显示INT22H**”的这一部分，我只实现了int 21h用于系统调用，对于int 20h和int 22h的功能尚未实现。

由于增加的功能逐渐增多，故先将其列举出来方便整理。如下表所示：

|  |  |  |  |
| --- | --- | --- | --- |
| 磁头号 | 扇区号 | 扇区大小 | 含义 |
| 0 | 1 | 1个扇区 | 引导程序 |
| 0 | 2 | 1个扇区 | 用户信息表 |
| 0 | 3到18 | 16个扇区 | 内核 |
| 1 | 1到2 | 2个扇区 | 用户程序b |
| 1 | 3到4 | 2个扇区 | 用户程序a |
| 1 | 5到6 | 2个扇区 | 用户程序c |
| 1 | 7到8 | 2个扇区 | 用户程序d |
| 1 | 9 | 1个扇区 | 实验4调用int33h~36h的用户程序 |
| 1 | 10 | 3个扇区 | 展示系统调用的用户程序 |

在上表中，最后一行的内容是新增加的功能，目的是测试系统调用的成功与否，可以通过命令run 6来展示。

下面我将依次介绍上表中的细节部分。

1. **引导程序与用户程序信息表：**

引导程序（bootloader.asm）的功能还是不变的，用户程序信息表则多了一项：

UserProgInfo:

    UserProgInfoBlock 1, 'b', 1024, 0, 1, 1, offset\_userprog1

    UserProgInfoBlock 2, 'a', 1024, 0, 1, 3, offset\_userprog2

    UserProgInfoBlock 3, 'c', 1024, 0, 1, 5, offset\_userprog3

    UserProgInfoBlock 4, 'd', 1024, 0, 1, 7, offset\_userprog4

    UserProgInfoBlock 5, 'interrupt\_caller', 512, 0, 1, 9, offset\_intcaller

    UserProgInfoBlock 6, 'syscall\_test', 1536, 0, 1, 10, offset\_syscalltest

UserPrigInfoBlock 6就是新增加的测试系统调用的用户程序，它被添加到用户程序信息表中。

1. **操作系统内核**

|  |  |
| --- | --- |
| 文件名 | 功能 |
| osstarter.asm | 监控程序，接收用户命令，执行相应的用户程序 |
| liba.asm | 包含n个汇编编写的函数 |
| kernel.c | 包含n个C编写的函数 |
| stringio.h | myos\_c.c的头文件，实现了输入输出等功能 |
| systema.asm | 包含n个汇编编写的系统调用函数 |
| systemc.c | 包含n个汇编编写的系统调用函数的辅助函数 |

其中，有所变化的是内核支持的命令增多了，以及多了两个与系统调用相关的文件。

对于新增加的两个系统命令：date和reboot，其中，前者的功能是显示当前的北京时间，通过以下的函数进行实现：

汇编：

[global getDateYear]

[global getDateMonth]

[global getDateDay]

[global getDateHour]

[global getDateMinute]

[global getDateSecond]

C：

extern uint8\_t getDateYear();

extern uint8\_t getDateMonth();

extern uint8\_t getDateDay();

extern uint8\_t getDateHour();

extern uint8\_t getDateMinute();

extern uint8\_t getDateSecond();

分别取得当前的各个时间点，然后将其打印出来即可。

后者的功能是重启计算机，使用BIOS中断int 19H，重新启动系统实现。

**systema.asm:**

测试系统调用的函数主要由下面的六个函数构成。

[global sys\_showOuch]

[global sys\_toUpper]

[global sys\_toLower]

[global sys\_atoi]

[global sys\_itoa]

[global sys\_printInPos]

根据实验要求当按下键盘时不再显示OUCH！，故用来测试系统调用。同时，实现的功能还有字符串的大小写转换功能，以及字符串与整型的相互转换功能，最后通过sys\_printInPos将其打印出来。在此仅列举sys\_showOuch的相关代码：

sys\_showOuch:

    pusha             ; 保护现场

    push ds

    push es

    mov ax, cs        ; 置其他段寄存器值与CS相同

    mov ds, ax        ; 数据段

    mov bp, ouch\_str  ; BP=当前串的偏移地址

    mov ax, ds        ; ES:BP = 串地址

    mov es, ax        ; 置ES=DS

    mov cx, 4         ; CX = 串长

    mov ax, 1301h     ; AH = 13h（功能号）、AL = 01h（光标置于串尾）

    mov bx, 0038h     ; 页号为0(BH = 0) 黑底白字(BL = 07h)

    mov dh, 12        ; 行号

    mov dl, 38        ; 列号

    int 10h           ; BIOS的10h功能：显示一行字符

    pop es

    pop ds

    popa              ; 恢复现场

    ret

    ouch\_str db 'OUCH'

**systemc.c:**

最主要的实现是，将systema.asm里的字符串大小写转换，整型与字符串转换等功能用c进行实现。

1. **4个用户程序（b、a、c、d）**

与上次实验有所不同的是，这四个用户程序在执行的时候，敲下键盘的任意键不会显示OUCH！这个只需要将上次实验里有关显示OUCH！的中断处理给注释掉即可。如下所示：

    MOVE\_INT\_VECTOR 09h, 39h

    ;WRITE\_INT\_VECTOR 09h, IntOuch

上面两行代码就是实验3中新添加的中断处理部分，将第二行注释掉即可。

1. **实验4调用int33h~36h的用户程序**

和上次实验相比没有变动，故在此不再赘述。

1. **展示系统调用的用户程序（syscall\_test.asm）**

该部分程序可以通过run 6进行展示。

其中，用户程序(syscall\_test.asm)主要是对systema.asm的进行各种各样的调用，列举部分相关代码如下：

PRINT\_IN\_POS hint0, hint\_len, 2, 0

    mov ah, 00h                        ; 系统调用功能号ah=00h，显示OUCH

    int 21h

    mov ah, 0

    int 16h

    cmp al, 27                         ; 按下ESC

    je QuitUserProg                     ; 直接退出

    PRINT\_IN\_POS hint1, hint\_len, 2, 0

    mov ax, cs

    mov es, ax                         ; es=cs

    mov dx, upper\_lower                ; es:dx=串地址

    PRINT\_IN\_POS upper\_lower, 14, 3, 0

    mov ah, 01h                        ; 系统调用功能号ah=01h，大写转小写

    int 21h

    PRINT\_IN\_POS upper\_lower, 14, 4, 0

    mov ah, 0

    int 16h

    cmp al, 27                         ; 按下ESC

    je QuitUserProg                     ; 直接退出

其具体的实现代码均在systema.asm当中，下面列举出部分相关的重要代码。

**功能号：ah=00h：**

在合适的位置显示OUCH（此处是12行38列）.

sys\_showOuch:

    pusha             ; 保护现场

    push ds

    push es

    mov ax, cs        ; 置其他段寄存器值与CS相同

    mov ds, ax        ; 数据段

    mov bp, ouch\_str  ; BP=当前串的偏移地址

    mov ax, ds        ; ES:BP = 串地址

    mov es, ax        ; 置ES=DS

    mov cx, 4         ; CX = 串长

    mov ax, 1301h     ; AH = 13h（功能号）、AL = 01h（光标置于串尾）

    mov bx, 0038h     ; 页号为0(BH = 0) 黑底白字(BL = 07h)

    mov dh, 12        ; 行号

    mov dl, 38        ; 列号

    int 10h           ; BIOS的10h功能：显示一行字符

    pop es

    pop ds

    popa              ; 恢复现场

    ret

    ouch\_str db 'OUCH'

**功能号：ah=01h：**

实现字符串的大小写转换，此处的功能是大写转换成小写，其具体的实现在systemc.c里的toupper函数里。

[extern toupper]

sys\_toUpper:

    push es           ; 传递参数

    push dx           ; 传递参数

    call dword toupper

    pop dx            ; 丢弃参数

    pop es            ; 丢弃参数

    ret

其他的功能号的实现也与之类似，在此就不一一展现了。

**混合编译链接：**

老方法了，将所有需要的文件进行联合编译链接，保存为myos.sh如下：

rm -rf temp

mkdir temp

rm \*.img

nasm bootloader.asm -o ./temp/bootloader.bin

nasm userproginfo.asm -o ./temp/userproginfo.bin

cd userprog

nasm b.asm -o ../temp/b.bin

nasm a.asm -o ../temp/a.bin

nasm c.asm -o ../temp/c.bin

nasm d.asm -o ../temp/d.bin

nasm interrupt\_caller.asm -o ../temp/interrupt\_caller.bin

nasm syscall\_test.asm -o ../temp/syscall\_test.bin

cd ..

cd lib

nasm -f elf32 systema.asm -o ../temp/systema.o

gcc -fno-pie -c -m16 -march=i386 -masm=intel -nostdlib -ffreestanding -mpreferred-stack-boundary=2 -lgcc -shared systemc.c -o ../temp/systemc.o

cd ..

nasm -f elf32 hotwheel.asm -o ./temp/hotwheel.o

nasm -f elf32 osstarter.asm -o ./temp/osstarter.o

nasm -f elf32 liba.asm -o ./temp/liba.o

gcc -fno-pie -c -m16 -march=i386 -masm=intel -nostdlib -ffreestanding -mpreferred-stack-boundary=2 -lgcc -shared kernel.c -o ./temp/kernel.o

ld -m elf\_i386 -N -Ttext 0x8000 --oformat binary ./temp/osstarter.o ./temp/liba.o ./temp/kernel.o ./temp/systema.o ./temp/systemc.o ./temp/hotwheel.o -o ./temp/kernel.bin

rm ./temp/\*.o

dd if=./temp/bootloader.bin of=Condor\_OS.img bs=512 count=1 2> /dev/null

dd if=./temp/userproginfo.bin of=Condor\_OS.img bs=512 seek=1 count=1 2> /dev/null

dd if=./temp/kernel.bin of=Condor\_OS.img bs=512 seek=2 count=16 2> /dev/null

dd if=./temp/b.bin of=Condor\_OS.img bs=512 seek=18 count=2 2> /dev/null

dd if=./temp/a.bin of=Condor\_OS.img bs=512 seek=20 count=2 2> /dev/null

dd if=./temp/c.bin of=Condor\_OS.img bs=512 seek=22 count=2 2> /dev/null

dd if=./temp/d.bin of=Condor\_OS.img bs=512 seek=24 count=2 2> /dev/null

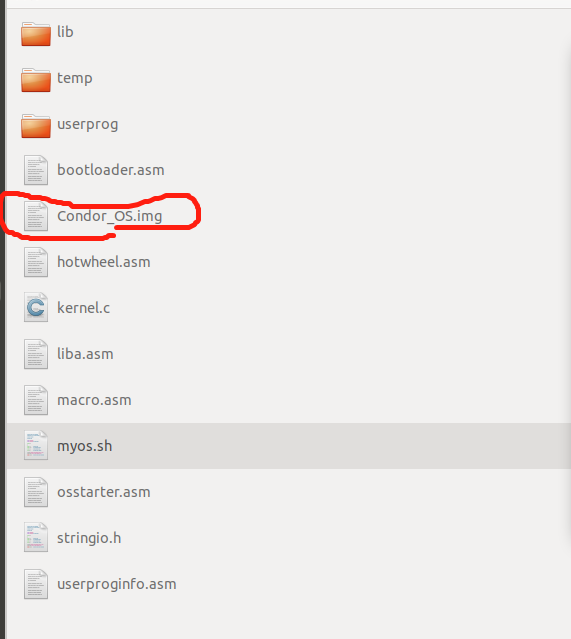
dd if=./temp/interrupt\_caller.bin of=Condor\_OS.img bs=512 seek=26 count=1 2> /dev/null

dd if=./temp/syscall\_test.bin of=Condor\_OS.img bs=512 seek=27 count=3 2> /dev/null

echo "Finished."

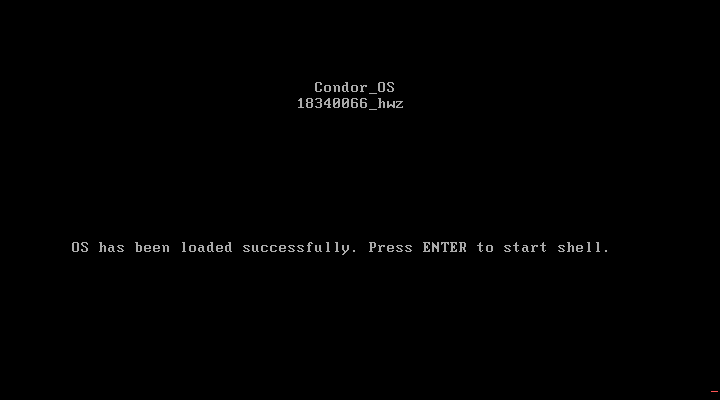
运行之，得到Condor\_OS.img如下：



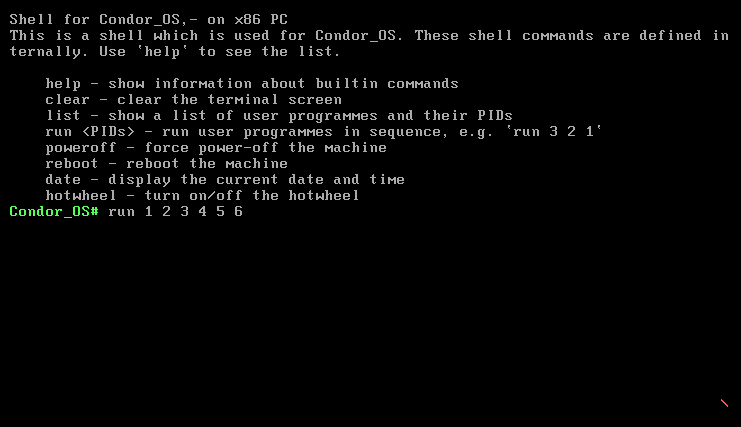


**实验结果展示：**

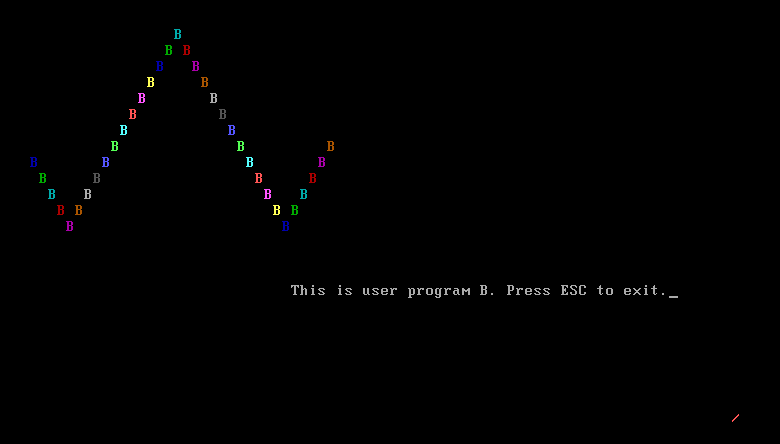
开机，风火轮在右下角转动：



进入shell界面，准备执行run命令：

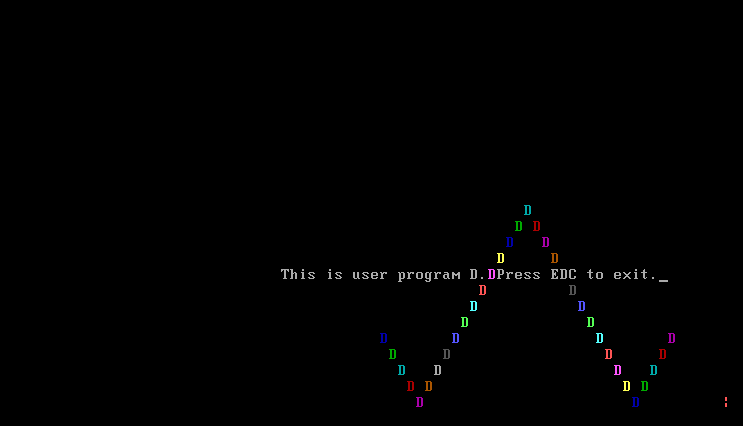


执行命令画面依次如下，可以看到，OUCH！的显示已经消失了：

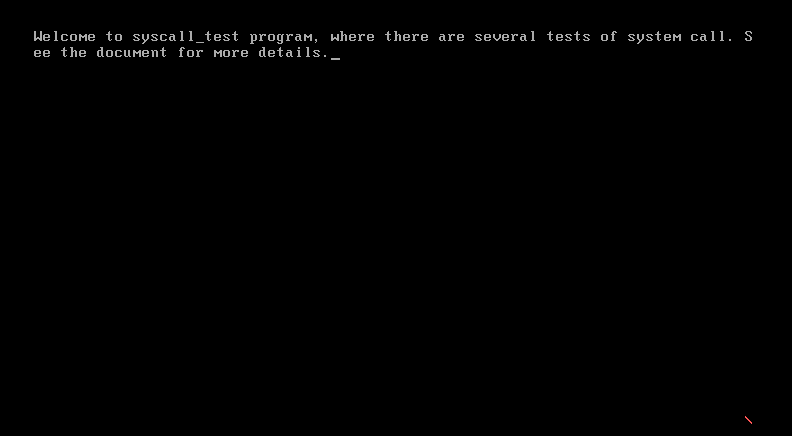


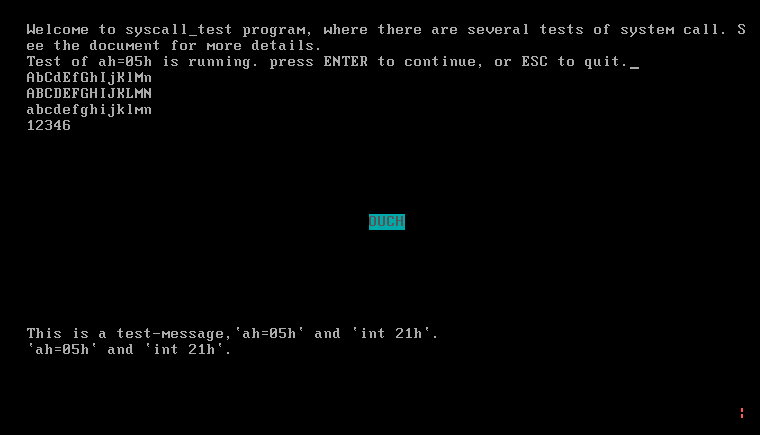




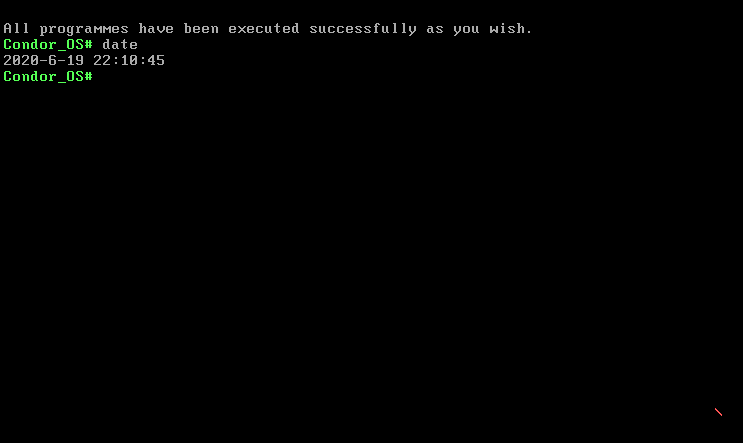








执行新增加的命令date：



**实验心得：**

最大的体会是明白了系统调用与一般调用的区别，包括运行状态、调用方法等等。如果没有系统调用，那么用户在编写自己的程序的时候就无法调用系统内核已经封装好的函数，这对于扩展我们的操作系统来说并不是一件好事。而有了系统调用，这就可以达到方便用户使用的目的。