操作系统实验项目

实现系统调用

郑戈涵　17338233　931252924@qq.com

摘要

本次实验共完成三个任务: 实现中断保护，完成软中断程序编写，生成自己的 COM 程序

目录

1 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

2 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。

3 实验内容

3.1 编写用于中断处理的现场保护和现场恢复的汇编过程

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。

3.2 重写和扩展实验三的的内核程序

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。

3.3 实现部分基本输入输出库过程

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。

3.4 执行调用自己设计的库过程的程序

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。

4 实验原理

4.1 中断

所有计算机都提供允许其他模块 (I/O、存储器）中断处理器正常处理过程的机制。

表 1: 中断的分类

程序中断	在某些条件下由指令执行的结果产生，如算术溢出、除数为 0、试图执行一条非法机器指令及访问用户不允许的存储器位置
时钟中断	由处理器内部的计时器产生，允许操作系统以一定的规律执行函数
I/0 中断	由 I/O 控制器产生，用于发信号通知一个操作的正常完成或各种错误条件
硬件失效中断	由诸如掉电或存储器奇偶校验错之类的故障产生

中断的分类 [4]

4.2 中断术语

[3]

- 按中断源进行分类：发出中断请求的设备称为中断源。按中断源的不同，中断可分为:
  - 内中断：即程序运行错误引起的中断
  - 外中断：即由外部设备、接口卡引起的中断
  - 软件中断：由写在程序中的语句引起的中断程序的执行，称为软件中断
- 允许/禁止 (开/关) 中断：CPU 通过指令限制某些设备发出中断请求，称为屏蔽中断。
- 从 CPU 要不要接收中断即能不能限制某些中断发生的角度，中断可分为
  - 可屏蔽中断：可被 CPU 通过指令限制某些设备发出中断请求的中断
  - 不可屏蔽中断：不允许屏蔽的中断如电源掉电
- 中断允许触发器：在 CPU 内部设置一个中断允许触发器，只有该触发器置“1”，才允许中断；置“0”，不允许中断。
- 指令系统中，开中断指令，使中断触发器置“1”
- 关中断指令，使中断触发器置“0”
- 中断优先级：为了管理众多的中断请求，需要按每个（类）中断处理的急迫程度，对中断进行分级管理，称其为中断优先级。在有多个中断请求时，总是响应与处理优先级高的设备的中断请求。
- 中断嵌套：当 CPU 正在处理优先级较低的一个中断，又来了优先级更高的一个中断请求，则 CPU 先停止低优先级的中断处理过程，去响应优先级更高的中断请求，在优先级更高的中断处理完成之后，再继续处理低优先级的中断，这种情况称为中断嵌套。

4.3 中断处理的过程

[3] 下面是保护模式中断处理的过程，实模式类似。

1. 系统将所有的中断信号统一进行了编号（一共 256 个：0 ~ 255），这个号称为中断向量，具体哪个中断向量表示哪种中断有的是规定好的，也有的是在给定范围内自行设定的。中断向量和中断服务程序的对应关系主要是由 IDT（中断向量表）负责。操作系统在 IDT 中设置好各种中断向量对应的中断描述符（一共有三类中断门描述符：任务门、中断门和陷阱

门），留待 CPU 查询使用。而 IDT 本身的位置是由 idtr 保存的，当然这个地址也是由 OS 填充的。中断服务程序具体负责处理中断（异常）的代码是由软件，也就是操作系统实现的，这部分代码属于操作系统内核代码。也就是说从 CPU 检测中断信号到加载中断服务程序以及从中断服务程序中恢复执行被暂停的程序，这个流程基本上是硬件确定下来的，而具体的中断向量和服务程序的对应关系设置和中断服务程序的内容是由操作系统确定的。

2. CPU 在执行完当前程序的每一条指令后，都会去确认在执行刚才的指令过程中中断控制器（如：8259A）是否发送中断请求过来，如果有那么 CPU 就会在相应的时钟脉冲到来时从总线上读取中断请求对应的中断向量 [2]。对于异常和系统调用那样的软中断，因为中断向量是直接给出的，所以和通过 IRQ（中断请求）线发送的硬件中断请求不同，不会再专门去取其对应的中断向量。
3. 根据中断向量到 IDT 表中取得处理这个向量的中断程序的段选择符 CPU 根据得到的中断向量到 IDT 表里找到该向量对应的中断描述符，中断描述符里保存着中断服务程序的段选择符。
4. 根据取得的段选择符到 GDT 中找相应的段描述符 CPU 使用 IDT 查到的中断服务程序的段选择符从 GDT 中取得相应的段描述符，段描述符里保存了中断服务程序的段基址和属性信息，此时 CPU 就得到了中断服务程序的起始地址。这里，CPU 会根据当前 cs 寄存器里的 CPL 和 GDT 的段描述符的 DPL，以确保中断服务程序是高于当前程序的，如果这次中断是编程异常（如：int 80h 系统调用），那么还要检查 CPL 和 IDT 表中中断描述符的 DPL，以保证当前程序有权限使用中断服务程序，这可以避免用户应用程序访问特殊的陷阱门和中断门。
5. CPU 根据特权级的判断设定即将运行的中断服务程序要使用的栈的地址 CPU 会根据 CPL 和中断服务程序段描述符的 DPL 信息确认是否发生了特权级的转换，比如当前程序正运行在用户态，而中断程序是运行在内核态的，则意味着发生了特权级的转换，这时 CPU 会从当前程序的 TSS 信息（该信息在内存中的首地址存在 TR 寄存器中）里取得该程序的内核栈地址，即包括 ss 和 esp 的值，并立即将系统当前使用的栈切换成新的栈。这个栈就是即将运行的中断服务程序要使用的栈。紧接着就将当前程序使用的 ss,esp 压到新栈中保存起来。也就说比如当前在某个函数中，使用的栈，在中断发生时，需要切换新的栈。
6. 保护当前程序的现场，CPU 开始利用栈保护被暂停执行的程序的现场：依次压入当前程序使用的 eflags，cs，eip，errorCode（如果是有错误码的异常）信息。
7. 跳转到中断服务程序的第一条指令开始执行 CPU 利用中断服务程序的段描述符将其第一条指令的地址加载到 cs 和 eip 寄存器中，开始执行中断服务程序。这意味着先前的程序被暂停执行，中断服务程序正式开始工作。
8. 中断服务程序处理完毕，恢复执行先前中断的程序在每个中断服务程序的最后，必须有中断完成返回先前程序的指令，这就是 iret（或 iretd）。程序执行这条返回指令时，会从栈里弹出先前保存的被暂停程序的现场信息，即 eflags,cs,eip 重新开始执行。

5 实验过程

本次实验流程如下

1. 编写中断向量操作程序
2. 编写 x86 汇编语言对时钟中断的响应处理程序
3. 编写对键盘中断的响应处理程序
4. 修改内核和用户程序
5. 生成镜像，在虚拟机中加载

5.1 编写中断向量操作程序

5.1.1 移动中断向量宏

原则上虽然我们将要利用中断向量表来执行自己写的程序，但是被覆盖的中断的原本的功能应该还是有，因此需要将中断向量转移到其他的向量号上。为了多次使用这段代码，我将其设为宏，代码如下：

Code 1: MOVE_VECTOR

```
%macro MOVE_VECTOR 2
  pusha                ; 保护现场
  push es
  mov ax, 0
  mov   es, ax         ; 数据段
  mov ax, word[es:%1*4]
  mov word[es:%2*4],ax
  mov ax, word[es:%1*4+2]
  mov word[es:%2*4+2],ax
  pop es
  popa                 ; 恢复现场
%endmacro
```

将 0 + 中断号 × 4 的位置的数据取出来放到目标中断号对应的位置即可

5.1.2 中断向量写入宏

写入中断向量也会多次用到，因此我将其设为宏，代码如下

Code 2: VECTOR_IN

```
%macro VECTOR_IN 2
  pusha                ; 保护现场
  push es
  mov ax, 0
  mov   es, ax         ; 数据段
  mov word[es:%1*4],%2
  mov ax, cs
  mov word[es:%1*4+2],ax
  pop es
  popa                 ; 恢复现场
%endmacro
```

将自己送入的中断程序的地址放入 0 开始中断号 *4 的偏移量的位置即可，注意每次放入的单位为一个字 (word)，第一个字为 IP，第二个为 CS。

5.2 时钟中断的响应处理程序

单个程序是作为引导程序的，因此尽量不能超过 512 字节，并且末尾两个字节为 0x55aa。代码由老师提供的修改得到。代码如下：

Code 3: Timer(test)

```
org 7c00h
%include "header.inc"
VECTOR_IN 08h,Timer
sti
jmp $                       ; 死循环
; 时钟中断处理程序

Timer:
    pusha
    push gs
    push ds
    mov ax,cs
    mov ds,ax
    mov ax,0B800h                   ; 文本窗口显存起始地址
    mov gs,ax               ; GS = B800h

    dec byte [count]                ; 递减计数变量
    jnz Exit                        ; >0: 跳转
    mov byte[count],delay                   ; 重置计数变量=初值delay
    mov ah,[color]
    dec byte[color]
    jz restoreColor
draw:
    mov si, wheel
    add si, [wheelOffset]
    mov al,[si]
  mov [gs:((80*24+79)*2)],ax        ; =0: 递增显示字符的ASCII码值
    inc byte[wheelOffset]
    cmp byte[wheelOffset],4
    jne Exit
    mov byte[wheelOffset],0
Exit:
    mov al,20h                      ; AL = EOI
    out 20h,al                      ; 发送EOI到主8259A
    out 0A0h,al                     ; 发送EOI到从8259A
    pop ds
    pop gs
    popa
    iret                            ; 从中断返回
```

```
40 %endmacro
```

主模块 首先在第一行用 org 0x7c00，这样可以不修改段基址而能正常读取内存中的数据，然后使用 include 将上面两节完成的代码引用进来，其中包括写入中断向量和移动中断向量的宏，由于测试时我同时使用了 bochs，而 bochs 执行引导程序时是会显示一段文字的，因此我选择在程序开始时清屏。设置好各个段的基址 (gs=0B800h) 然后将中断响应处理程序 Timer 所在的地址写入向量表，然后最重要的一点是要开中断，否则时钟不会触发。

风火轮模块 要实现风火轮，只要在显存里的某个位置轮流放入那四个字符即可，因此把他们按顺序放在内存中，然后用循环遍历他们即可，为了让风火轮显示不那么快，我在每次显示之前都执行一段延迟的指令，然后进入绘制部分。绘制部分只需要用内存中的一个变量记住当前风火轮的位置，然后用寄存器将这个偏移量加到风火轮字符串对应的地址上，得到需要的字符的地址，然后把字符放入显存里。我的程序每次还会改变风火轮的颜色，只要维护一个颜色的变量，每次写入显存前赋值到 ah 寄存器里即可。每一次中断处理结束，需要发送一个 EOI 给 8259A，以便继续接收中断。最后 iret 返回。

修改一下上次的 makefile，直接用 make 命令生成镜像即可。在 vmware 或 bochs 中打开。效果如下：

图 1: 风火轮程序运行过程截图

5.3 键盘中断的响应处理程序

键盘中断号为 09h, 处理程序只需要打印信息，然后返回即可。代码如下：

Code 4: ouch.asm

```
1 %include "utility.inc"
2
3 Ouch:
4     pusha
5     push ds
```

```
6       mov ax,cs
7       mov ds,ax
8     PRINT msg, msg_len, 20, 40
9       call Delay
10    PRINT clearMsg, msg_len, 20, 40
11
12      int 40h
13
14      mov al,20h              ; AL = EOI
15      out 20h,al              ; 发送EOI到主8529A
16      out 0A0h,al             ; 发送EOI到从8529A
17
18      pop ds
19      popa
20      iret
21  msg db 'OUCH! OUCH!'
22  msg_len equ $-msg
23  clearMsg db '            '
24
25  Delay:
26      pusha
27      mov ax,500
28  dd:
29      mov cx,50000
30  ddd:
31      loop ddd
32      dec ax
33      cmp ax,0
34      jne dd
35      popa
36      ret
```

utility 库中的函数是上一节中定义的写入向量等宏。由于该程序需要在用户程序中被执行，因此不能调用清屏，需要用空字符串来覆盖。在内存中定义好对应的字符串，然后调用 PRINT 打印 Ouch 字符串，延迟一段时间，再打印空白字符串覆盖，然后发送 EOI 到 8259A 再 iret 返回即可。Delay 方法和第一次实验一样。

5.4 修改内核和用户程序

时钟中断在内核运行时就需要，因此在内核初始化阶段应该将其地址送入中断向量表。键盘响应只需要在用户程序中运行，因此修改用户程序即可。

5.4.1 修改内核

为了让原来的时钟中断向量还能被使用，我将其放入 0x39 中，在内核开始时移动中断向量并将自己编写的程序的地址写入向量表即可。代码只有两句：

Code 5: 内核程序更改

```
MOVE_VECTOR 08h,39H
VECTOR_IN 08h,Timer
```

5.4.2 修改用户程序

和内核一样，键盘中断程序也需要移位，这次我放入 0x40 中，在用户程序开始时需要移动中断向量并将自己编写的程序的地址写入向量表，离开时需要恢复。中断程序需要放在用户程序结尾。

Code 6: 用户程序更改

```
MOVE_VECTOR 09h,40h
VECTOR_IN 09h,Ouch
MOVE_VECTOR 40h,09h
%include "ouch.asm"
```

5.5 扩展内核程序-实时显示时间

由于上一次实验中内核已经比较完善，这次实验使用时钟中断为 shell 提供新的特性-显示时间，和实现风火轮一样，只要将对应的程序放入中断向量表中即可。所以完成扩展总共有三个步骤：

1. 从 BIOS 获得日期时间的信息
2. 用 c 代码完成信息的加工和显示
3. 修改中断程序
4. 在合适的位置加载中断向量

5.5.1 从 BIOS 获得日期时间的信息

从 BIOS 获得日期时间的信息是通过读端口实现的，首先设置好对应的寄存器，用寄存器向 70h 端口写入要访问的单元地址，然后从 71h 端口读入即可获得数据。该函数是用来给 c 代码调用的，因此首先需要确定他的声明。为了使函数能够复用，可以把读端口需要设置的寄存器值作为参数传给汇编过程，，为了代码的可读性，我为参数设计了枚举类型 Date，总共有 6 项，请看表2

函数的返回值是 BIOS 返回的数字，这些数字是 BCD 码，需要转换为十进制整数。

表 2: 日期时间枚举表

名称	枚举变量名	数值
秒	**Second**	**0**
分	**Minute**	**2**
小时	**Hour**	**4**
日	**Day**	**7**
月	Month	8
年	Year	9

5.5.2 加工和显示获得的信息

要调用汇编的函数，首先需要用 extern 声明该函数。获得日期时间对应的数字信息后，利用上次实验实现的数字转字符串函数就能得到对应的字符串，然后加工需要用到 strcat 函数，合成成一个完整的字符串就可以显示了. 为了美观，显示小于 10 的数字前我都手动补了 0，保证不会总是跳动。工具函数的代码请看附录的9，显示的 c 代码如下：

Code 7: split

```
void printDate(){
    int year=getDateInfo(Year),
    month=getDateInfo(Month),
    day=getDateInfo(Day),
    hour=getDateInfo(Hour),
    minute=getDateInfo(Minute),
    second=getDateInfo(Second);

    char TimeBuffer[22]={0};
    char *yearPrefix = "20";
    strcat(TimeBuffer, yearPrefix);
    strcat(TimeBuffer, itoa(bcd2dec(year), 10));
    strAppend(TimeBuffer, '-');
    if(month<10){
        strAppend(TimeBuffer, '0');
    }
    strcat(TimeBuffer, itoa(bcd2dec(month), 10));
    strAppend(TimeBuffer, '-');
    if(day<10){
        strAppend(TimeBuffer, '0');
    }
    strcat(TimeBuffer, itoa(bcd2dec(day), 10));
    strAppend(TimeBuffer, ' ');
    if(hour<10){
        strAppend(TimeBuffer, '0');
    }
    strcat(TimeBuffer, itoa(bcd2dec(hour), 10));
    strAppend(TimeBuffer, ':');
    if(minute<10){
        strAppend(TimeBuffer, '0');
    }
    strcat(TimeBuffer, itoa(bcd2dec(minute), 10));
    strAppend(TimeBuffer, ':');
    if(second<10){
        strAppend(TimeBuffer, '0');
    }
    strcat(TimeBuffer, itoa(bcd2dec(second), 10));

    printPos(TimeBuffer,20,24,58);
}
```

其中使用的 printpos 和 itoa 都是上次实验实现的。限于篇幅，代码不在此放出。注意，之前使用的 printpos 函数是参考的老师的代码，老师的版本会移动光标，使用这个版本会导致光标永远集中在显示的时间字符串后面，无法输入 shell 命令，因此在调用 10 号中断前需要设置 al=0 而非 1(将光标置于串尾)。

5.5.3 修改中断程序

由于显示时间会挡住用户程序的显示，而且由于两者可能会同时修改显存，有时候会产生问题，所以我设置了两份中断程序，一份用于内核，带时间和风火轮，一份用于用户程序，只有风火轮。只用风火轮的不需要修改，用户程序的只需要在返回前调用显示日期时间的 c 函数即可。

5.5.4 在合适的位置加载中断向量

修改内核程序 由于要安排两个版本，内核程序和加载用户程序的函数都需要修改，在 5.4.1 小节的基础上，内核程序只需要修改放入的地址为新中断程序的地址即可。

修改加载用户程序的函数 加载用户程序前需要将只有风火轮的中断程序地址放入中断向量表，从用户程序返回后再将新的中断程序地址放入向量表。

5.6 软盘扇区安排

由于修改的内容都是在原有代码的基础上修改的，没有增加新的程序，所以扇区安排和上次一样，安排请看表3：

5.7 镜像文件生成

使用 gcc 生成 c 源文件对应的汇编，nasm 生成内核汇编源文件和库过程汇编源文件的 elf 格式的汇编，使用 ld 链接可以生成二进制文件，将引导程序，内核程序和用户程序放入镜像中的合适位置即可生成镜像。

代码相关信息请看 readme.md。

在 wsl 或 linux 的 shell 中执行 *make all/make img* 即可生成镜像 (.img) 文件，在 VMware 中加载该镜像，查看结果。到此，实验结束。

表 3: 软盘扇区安排

磁头号	扇区号	扇区数（大小）	内容
0	1	1（512 B）	引导程序
0	2∼17	16（8 KB）	操作系统内核
1	1∼2	2（1 KB）	用户程序 1(LU.com)
1	3∼4	2（1 KB）	用户程序 2(LD.com)
1	5∼6	2（1 KB）	用户程序 3(RU.com)
1	7∼8	2（1 KB）	用户程序 4(RD.com)

6 程序使用说明

6.1 实验环境

1. 调试，运行工具：bochs
2. 汇编器：nasm 2.13.02
3. 编译器：gcc 7.5.0
4. 链接器：ld 2.30
5. 编译环境：wsl Ubuntu
6. VSCode 1.44.2

6.2 编译方法

6.2.1 系统要求

生成镜像文件时，可以使用 linux 操作系统或者带 wsl 的 windows 系统。

6.2.2 编译过程与参数

在源代码目录下使用 wsl，执行下列代码即可得到镜像文件 os17338233.img。

代码 8: 生成镜像

```
1 make
```

6.3 运行与演示

运行镜像时请使用 bochs，如果使用 vmware，在运行用户程序时会遇到问题，其他正常，bochs 上是完全正常的。

6.3.1 风火轮

图 2是 5.4 节完成后的 shell 的界面

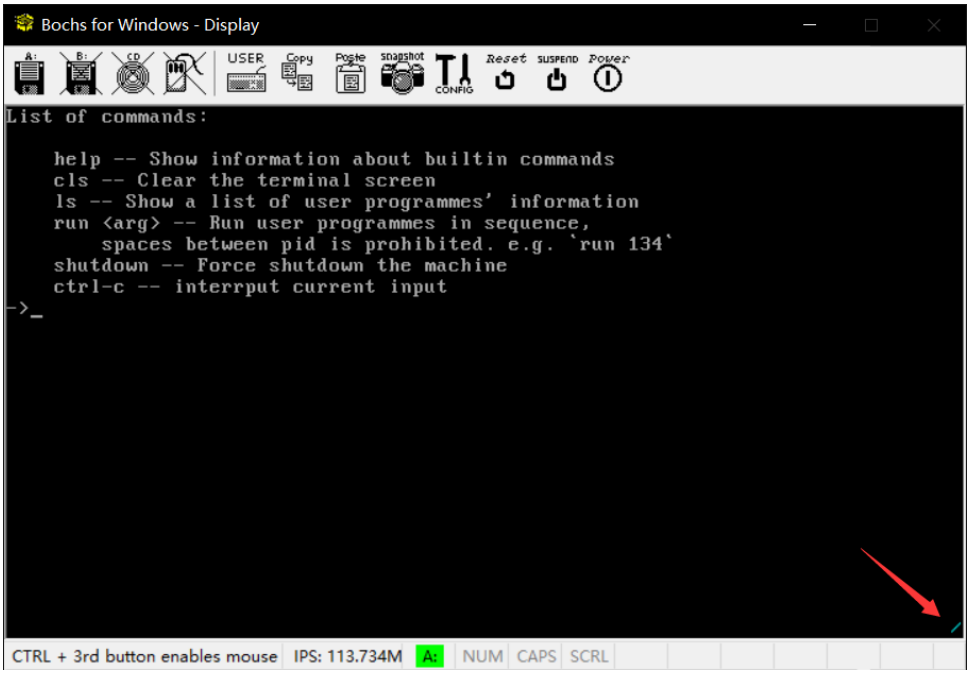

图 2: 带风火轮的 shell

6.3.2 带时间显示和风火轮

图3是 5.5 节完成后进入 shell 前的界面

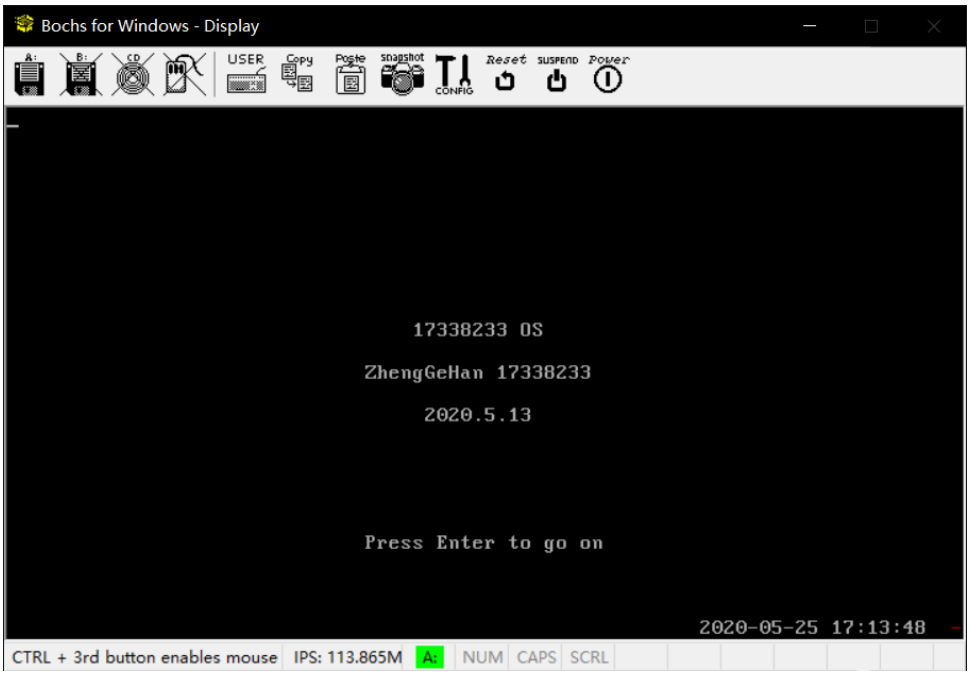

图 3: 带时间显示和风火轮的 shell 进入界面

图4是 5.5 节完成后的 shell 的界面

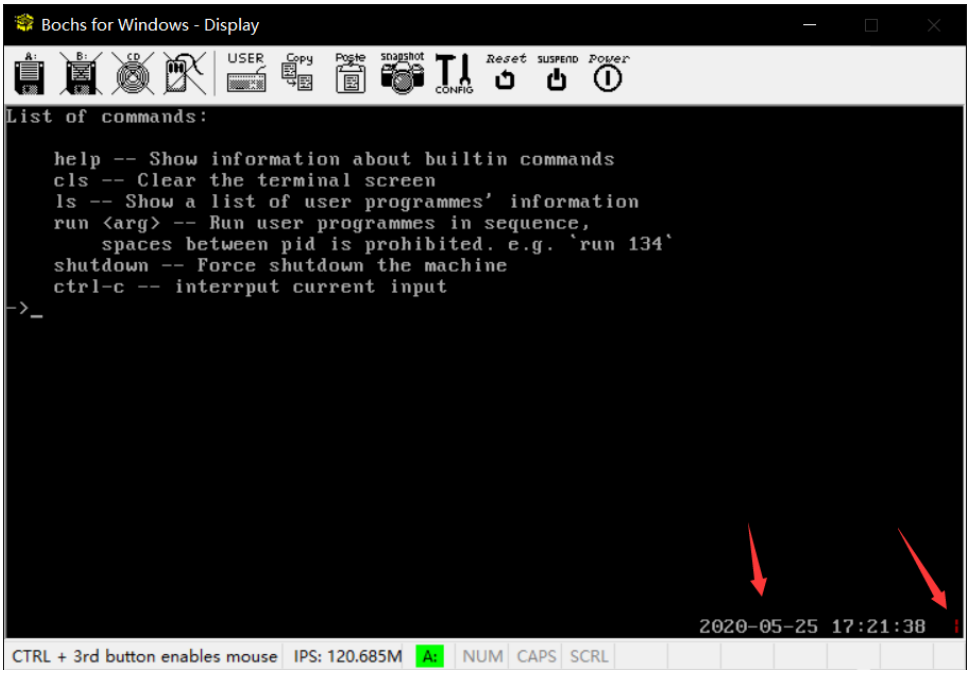

图 4: 带时间显示和风火轮的 shell

6.3.3 带时间显示和风火轮

进入用户程序后的效果如图 5 所示。

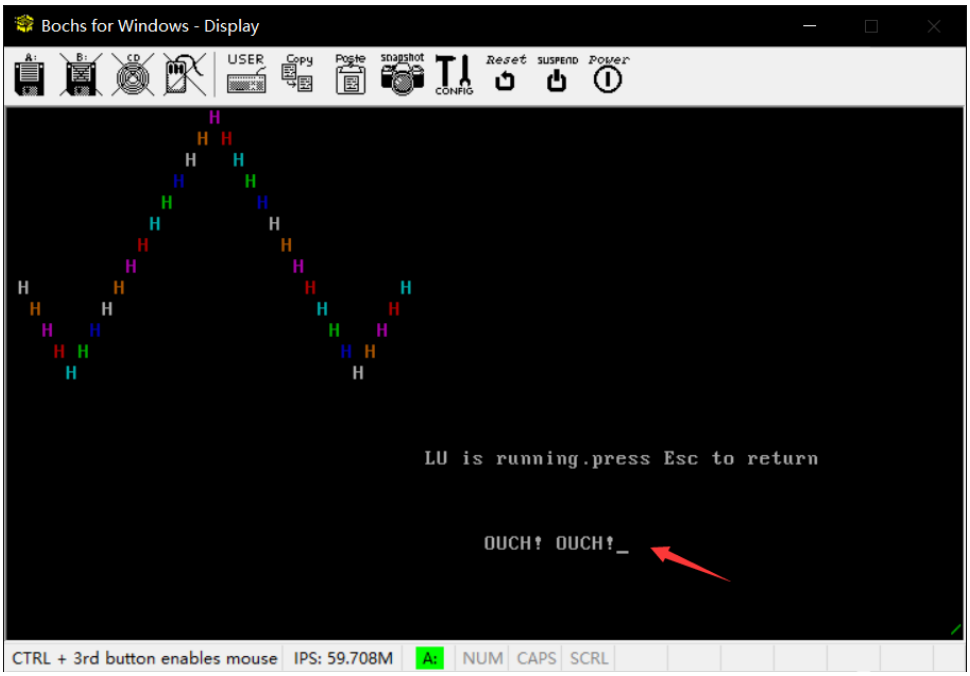

图 5: 按下任意键后的"Ouch!Ouch!"

由于时间中断的效果在虚拟机中观察比比较图片的效果更好，并且虚拟机的时间流逝速度和实际时间不同，可能会看到时间变换较快，因此具体的演示过程可以查看已录制好的 OSdemo.mkv

7 总结与讨论

7.1 特色, 不足与改进

本程序在上次提供的简洁的 shell 的基础上增加了实时时间显示。使用 gcc+nasm+ld+makefile 完成镜像生成的全部步骤。不足之处是由于键盘按下和放开共触发两次中断，用户会观察到闪烁两次的现象，我没有解决这个问题。

7.2 收获

本次实验中，我更加了解了如何利用中断向量表完成自己需要的功能，在丰富 shell 功能的过程中，我也练习了字符串基本操作函数的编写，在中断程序中使用 c 函数，在 c 函数中调用汇编的读端口过程使得我对混合编程的编程约定也更加熟悉。由于是第二次使用 makefile，在编写上更加熟悉了 [5]。同时，在本报告的编写过程中，我也练习了利用 LaTeX 编写文档的能力。[1, 2]

7.3 遇到的问题和感想

本次实验代码量较小，但是调试的难度与以前相比更大，因为 bochs 并不会在中断时停下，而我没有很方便的方法能在内存中找到中断程序的代码并设置断点，因此要么只能靠打印数据来调试，要么把程序移出中断程序来单独调试，比较复杂。在使用 BIOS 中断获得日期信息时，由于使用了混合编程，按照 _cdecl 约定，无论什么类型只要在 4 个字节内，都是压栈 4 个字节。因此取出数据时要非常小心。在调试这段程序时，由于无法找到中断程序的位置，我不得不重新编写单独的程序来调试。

在实验开始时，我是直接将中断程序移到内核里去的，当我写了独立的引导程序来执行中断程序后，我发现中断不触发了，是因为没有用 sti 开中断。并且查阅资料后我才知道中断程序中需要先开中断，在离开前关中断，否则可能在重要的位置被中断打断出现异常。这说明即使程序运行正常，未必就没有问题。

times 指令也在我的编程过程中导致了很多问题，由于我习惯将扇区剩余的位置清零，会在代码的最后一段放 times 指令，结果在放入中断程序时因为失误放在了 times 的后面，导致中断程序超出了用户程序的 2 个扇区的大小，被截断后放入了内存，导致中断处理时的莫名其妙的错误。

这次实验中我还找到了之前设计用户程序的 bug，如下图，我将控制恢复颜色的语句放在了回到循环开始的语句块之前，导致回到循环之前都会修改字符的颜色值，由于这个 bug 被触发是在键盘中断触发时才遇到，因此我一直认为是这次实验中处理中断向量表后导致的错误，花费大量时间在调试上，可见每次实验都必须完全搞清楚才能更轻松的进行下一次实验。

在大量的调试过程中，我发现中断程序返回 (iret) 后，并非直接回到原来的指令处，而是转去执行 0xfe830 处的代码，这里应该属于 bios 的，问了其他同学后他们也看到了这个现象。目前并不知道原因，我也没有找到任何资料说明 iret 后不会返回，我猜测是 bios 为中断设计了中断后处理程序。与写保护模式的同学交流后得知，保护模式在 iret 后是直接返回的。因此大概是 bios 设计好的工作。

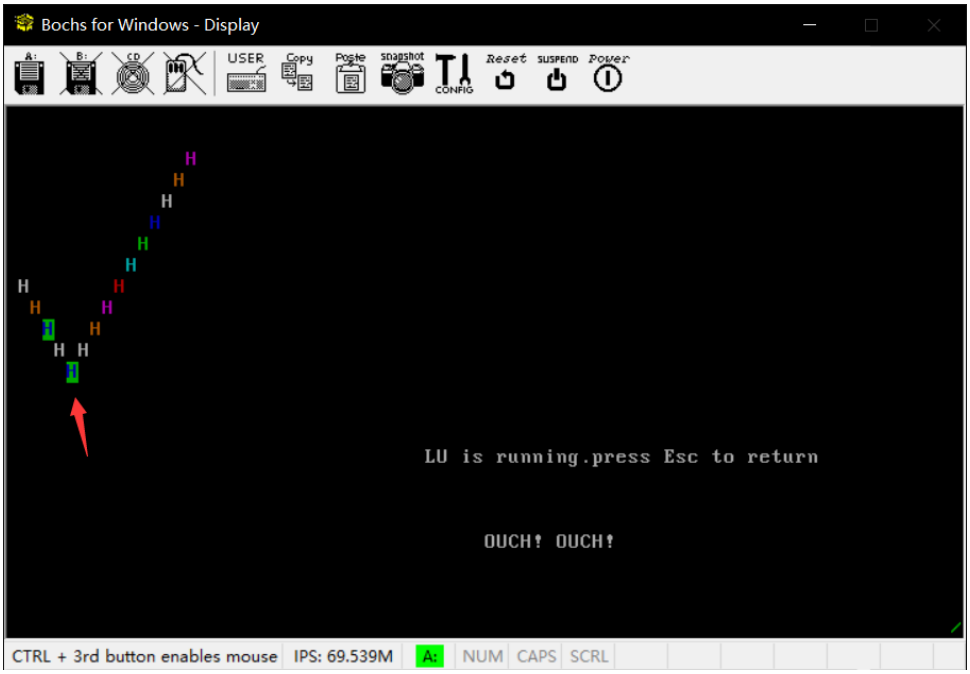

图 6: 旧用户程序的 bug：按下键盘后显示的字符会变色

References

1. Leslie Lamport, *LaTeX: a document preparation system*, Addison Wesley, Massachusetts, 2nd edition, 1994.
2. Contributors to Wikibooks, *LaTeX Bibliography Management*. Wikibooks, 2019., en.wikibooks.org/wiki/LaTeX/Bibliography_Management.
3. 中断及中断处理过程 jdksummer, 2012., https://www.cnblogs.com/jdksummer/articles/2687265.html
4. Operating Systems: Internals and Design Principles (8th Edition) William Stallings, Pearson, 2014.,
5. how-to-write-makefile, 跟我一起写 *Makefile* 陈皓, 2020.,

8 附录

Code 9: 加工和显示日期时间

```
char *strcat(char *dest, const char *src)
{
    char *tmp = dest;

    while (*dest)
        dest++;
```

```
7       while ((*dest++ = *src++) != '\0');
8       return tmp;
9   }
10  void strAppend(char*str, char c){
11    int len = strlen(str);
12    str[len] = c;
13    str[len + 1] = '\0';
14  }
15  uint8_t bcd2dec(uint8_t bcd)
16  {
17      return ((bcd & 0xF0) >> 4) * 10 + (bcd & 0x0F);
18  }
```