

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	2
2 实验要求	2
3 实验内容	2
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	2
3.2 重写和扩展实验三的的内核程序	2
3.3 实现部分基本输入输出库过程	2
3.4 执行调用自己设计的库过程的程序	2
4 实验原理	2
4.1 系统调用 (以 linux 为例)	2
4.2 软中断	3
4.3 DOS 程序的执行	3
5 实验过程	4
5.1 修改用户程序的执行方式	4
5.2 编写用于中断处理的现场保护和现场恢复的汇编过程	6
5.3 系统调用实现	9
5.4 实现库过程	11
5.5 编写使用库过程的测试程序	11
5.6 INT 22 实现和用户程序修改	13
5.7 软盘扇区安排	14
5.8 镜像文件生成	14
6 程序使用说明	15
6.1 实验环境	15
6.2 编译方法	15
6.3 运行与演示	15

7 总结与讨论	16
7.1 特色, 不足与改进	16
7.2 收获	16
7.3 遇到的问题	16
7.4 感想	17

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；`int 22h` 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 系统调用 (以 linux 为例)

内核提供用户空间程序与内核空间进行交互的一套标准接口，这些接口让用户态程序能受限访问硬件设备，比如申请系统资源，操作设备读写，创建新进程等。用户空间发生请求，内核

空间负责执行，这些接口便是用户空间和内核空间共同识别的桥梁，这里提到两个字“受限”，是由于为了保证内核稳定性，而不能让用户空间程序随意更改系统，必须是内核对外开放的且满足权限的程序才能调用相应接口。在用户空间和内核空间之间，有一个叫做 Syscall(系统调用, system call) 的中间层，是连接用户态和内核态的桥梁。这样即提高了内核的安全型，也便于移植，只需实现同一套接口即可。Linux 系统，用户空间通过向内核空间发出 Syscall，产生软中断，从而让程序陷入内核态，执行相应的操作。对于每个系统调用都会有一个对应的系统调用号，比很多操作系统要少很多。安全性与稳定性：内核驻留在受保护的地址空间，用户空间程序无法直接执行内核代码，也无法访问内核数据，通过系统调用性能：Linux 上下文切换时间很短，以及系统调用处理过程非常精简，内核优化得好，所以性能上往往比很多其他操作系统执行要好。[1]

4.2 软中断

当执行特定指令或满足特定条件时，处理器本身会请求软件中断。每个软件中断信号都与特定的中断处理程序相关联。软件中断可能是由于执行特殊指令而引起的，该指令在设计上会在执行时调用中断。此类指令的功能类似于子例程调用，并用于多种目的，例如请求操作系统服务和与设备驱动程序进行交互（例如，读取或写入存储介质）。程序执行错误也可能会意外触发软件中断。这些中断通常称为陷阱或异常。例如，如果处理器执行除数等于零的除法指令，则会抛出“除零”异常（请求软件中断）。通常，操作系统将捕获并处理此异常。[2]

4.3 DOS 程序的执行

4.3.1 程序的执行过程

在 DOS 提示符，如 C:> 下，键入一个可执行文件名（COM、EXE）后，在运行该程序前，DOS 完成以下工作：[3]

1. 从磁盘中找到该文件。
2. 检测用户可用内存，在可用内存的低地址段建立一个程序段前缀 (PSP, Program Segment Prefix)。
3. 把该文件从磁盘上装入内存中 PSP 的后面。
4. DOS 设置 DS、ES 的值等于 PSP 的段地址。如果该程序为 COM 文件，则把 CS、SS 的值也设置为 PSP 的段地址。
5. 开始执行该文件的第一条指令。

4.3.2 程序段前缀 PSP

DOS 运行程序时，需要该程序的一系列参数（如，程序结束地址、Ctrl_Break 处理程序的地址、出错处理地址等），另外还需要一个磁盘读、写的缓冲区，这个参数区和缓冲区，称为“程序段前缀 (PSP)”。PSP 共有 256 字节，它是运行程序时，由 DOS 自动在内存中建立的。PSP 的结构下图所示：注意：

1. PSP 起始两字节存放"INT 20H" 指令的机器码 (CDH 20H), 该指令使程序返回 DOS;
2. EXE 程序刚运行时, DS 和 ES 指向 PSP 首址, 即 INT20H 指令的机器码; COM 程序刚运行时, DS, ES, CS, SS 均指向 PSP 首址。

在 PSP 结构中，我们只关心前两个字节，它是指令"INT 20H" 的机器码 (CDH、20H)。

4.3.3 INT20H(结束程序的另一种方法)

1. 在程序执行前，DS= 程序段前缀 PSP 的段地址，把 PSP 的段地址和 0 值推入堆栈。
2. RET 指令从堆栈中取出两字，送 CS 和 IP，因此，RET 指令执行时，CS=PSP 的段地址，IP=0000H。CPU 转移到 PSP 处执行。
3. PSP 处的指令是 INT 20H，它的功能是结束程序并返回 DOS，因此，该程序能正确返回 DOS。

5 实验过程

本次实验流程如下：

1. 修改用户程序的执行方式
2. 编写用于中断处理的现场保护和现场恢复的汇编过程
3. 设计软中断程序用于系统调用
4. 实现库过程
5. 编写使用库过程的测试程序
6. 实现 INT 22
7. 去掉用户程序触碰键盘显示 OUCH! 的功能
8. 生成镜像，在虚拟机中加载

5.1 修改用户程序的执行方式

在之前的实验中，我执行程序的流程一直是如下：

1. 将用户程序所在的镜像加载进内存中
2. 将用户程序的地址信息放进一个内存块中
3. 利用上一步的内存块，用远调用 (call far) 跳入用户程序第一条指令的地址
4. 用户程序执行结束后用原返回 (retf) 回到加载程序的子过程 (loadUsrProgram)
5. 恢复内核上下文
6. 返回内核 shell

但是这么做基本等同于将用户程序当做一个内核的函数来处理，执行用户的程序和预处理和后处理并没有很好的解耦。因此这次实验中，我将其修改为类似 dos 的做法。具体的修改过程如下。

5.1.1 修改上下文

用户除了系统调用外就不应该使用内核的任何内存和资源。因此首先要修改栈寄存器，但是内核的栈也需要保存，之后需要修改各个段寄存器。

5.1.2 在程序段前缀安放返回指令

因为程序是 com 格式的，如果想要让用户在编写程序时不需要考虑如何返回，只需要在程序的开头放入 int 20h，然后将返回所需的数据预先放入栈中即可，用户程序可以通过 int 20h 的系统调用来回到内核，结束程序。有一点需要注意的是，这里可以使用 ax 存放 int 20h 这条指令，是因为 int 20h 的机器指令不超过一个字，如果超过了，会被截断。因此，如果需要加入多个指令，超过了一个字，则需要遍历对应的内存，将其放入用户栈。

5.1.3 将返回时所需的信息压栈

返回时需要 cs 和 ip 用于远跳转。然后还需要预先压入一个双字 0，用于 c 程序的双字返回。因为用户程序的 ret 执行之后就不需要管理数据区了，所以不需要修改。

5.1.4 执行用户程序

这部分和之前的相同，用 jmpf 间接远跳转，地址以及预先放入对应的数据块。注意由于 ds 已经在修改上下文时被修改，这里需要借另一个寄存器用于寻址。我使用了 gs。

5.1.5 用户程序返回后处理

将各个段地址恢复，就可以返回了。代码如下：

Code 1: loadUsrProgram

```
1  loadUsrProgram:
2  pusha
3  push ds
4  push es
5  mov bp, sp
6  mov bx, [bp+24]          ; 用户程序的段地址
7  mov word[program+2],bx
8  mov es, bx              ; 用es才能跨段读取,es:bx是读入的数据所在内存地址
9  mov bx, [bp+28]          ; 偏移地址; 存放数据的内存偏移地址
10 mov ah,2                ; 功能号
11 mov al,[bp+32]           ; 扇区数
12 mov dl,0                ; 驱动器号; 软盘为0, 硬盘和U盘为80H
13 mov dh,[bp+36]           ; 磁头号; 起始编号为0
14 mov ch,[bp+40]           ; 柱面号; 起始编号为0
15 mov cl,[bp+44]           ; 起始扇区号 ; 起始编号为1
16 int 13H                 ; 调用读磁盘 BIOS的13h功能
17 mov word[program],0x100
18 runUsrProgram:
19 mov ax, cs
20 mov gs, ax
21 mov ax,0xffff            ; 用户程序的栈顶为0xffff
22 mov cx,sp                ; 保存当前的栈顶, 用于返回后恢复
23 mov sp,ax                ; 设置用户程序的栈顶
24 mov ax,[codeOfInt20]     ; 用ax保存int 20这个语句
25 mov bx, es                ; 用户程序的段地址
```

```

26  mov ds,bx          ; 设置用户程序的数据段和es一致
27  mov ss,bx          ; 设置现在的栈段和es一致(以后操作用户程序的栈)
28  mov [es:0],ax      ; 在es:0(用户程序所在段首)放置int 20
29  push cx            ; 将内核栈顶偏移放入, 用于返回后恢复
30  push cs            ;
31  push afterRun       ; 将cs ip 先后压栈, 返回时可以远返回 retf
32  push dword 0        ; 压栈0, 用户程序如果使用ret就会回到0处执行
33  jmp far [gs:program] ; 远转移至gs:program所指的位置
34  afterRun:
35  mov ax,cs
36  mov ss,ax
37  mov es,ax
38  mov fs,ax
39  mov gs,ax
40  pop es
41  pop ds
42  popa
43  retf
44
45  codeOfInt20:
46  int 20h

```

5.2 编写用于中断处理的现场保护和现场恢复的汇编过程

5.2.1 用于保护现场的结构体设计

为了重用代码, 保护现场被设计为软中断和进程切换都需要用到的过程。其中需要设计一个专用的结构体。由于目前运行的进程只有一个, 设计好结构体后, 此次实验中只需要构建一个实例。结构体声明和寄存器映像的定义如下:

Code 2: struct RegisterImage

```

1  typedef struct RegisterImage{
2      uint16_t ax;      // 0
3      uint16_t cx;      // 2
4      uint16_t dx;      // 4
5      uint16_t bx;      // 6
6      uint16_t sp;      // 8
7      uint16_t bp;      // 10
8      uint16_t si;      // 12
9      uint16_t di;      // 14
10     uint16_t ds;      // 16
11     uint16_t es;      // 18
12     uint16_t fs;      // 20
13     uint16_t gs;      // 22
14     uint16_t ss;      // 24
15     uint16_t ip;      // 26
16     uint16_t cs;      // 28
17     uint16_t flags;    // 30

```

```

18     } RegisterImage;
19
20 RegisterImage KernalContext;
21 RegisterImage* getRegisterImage(){
22     return &KernalContext;
23 }

```

最后还声明了一个用于获取寄存器映像的函数，在之后的实验中的进程调度时可以用来获得当前需要切换到进程。

5.2.2 save 过程-保存中断现场

保存中断现场的前置条件是软中断被调用，此时已有三个对象被放入栈中，先后顺序分别是 flags, cs, ip，总共 6 个字节，为了保证结构体内的信息是调用 save 前的信息，在进入 save 之后我使用 pusha 将所有 8 个寄存器依次保存。然后调用上文的 getRegisterImage，获得指针，将各个寄存器依次写入结构体即可，由于 mov 不允许两个操作数都为内存地址，我使用了 ax 用来中转，di 用于基址寻址，其中要保存的 ip, cs, flags 是调用中断时自动压入的，无法通过其他渠道获得。代码如下：

Code 3: save 过程

```

1  save:
2  pusha
3  mov bp, sp
4  call dword getRegisterImage
5  mov di, ax
6  mov ax, [bp]          ;di
7  mov [cs:di+14], ax    ;
8  mov ax, [bp+2]        ;si
9  mov [cs:di+12], ax    ;
10 mov ax, [bp+4]        ;bp
11 mov [cs:di+10], ax    ;
12 mov ax, [bp+6]        ;sp
13 mov [cs:di+8], ax     ;
14 mov ax, [bp+8]        ;dx
15 mov [cs:di+6], ax     ;
16 mov ax, [bp+10]       ;cx
17 mov [cs:di+4], ax     ;
18 mov ax, [bp+12]       ;bx
19 mov [cs:di+2], ax     ;
20 mov ax, [bp+14]       ;ax
21 mov [cs:di], ax       ;
22 mov ax, ds            ;ds
23 mov [cs:di+16], ax    ;
24 mov ax, es            ;es
25 mov [cs:di+18], ax    ;
26 mov ax, fs            ;fs
27 mov [cs:di+20], ax    ;
28 mov ax, gs            ;gs

```



```

29  mov [cs:di+22], ax ;
30  mov ax, ss ;ss
31  mov [cs:di+24], ax ;
32  mov ax, [bp+18] ;ip
33  mov [cs:di+26], ax ;
34  mov ax, [bp+20] ;cs
35  mov [cs:di+28], ax ;
36  mov ax, [bp+22] ;flags
37  mov [cs:di+30], ax ;
38  popa
39  ret

```

要注意的是，虽然上下文保存好了，但是程序还是应该保存好当前的现场，因为系统调用可能使用参数。

5.2.3 restart 过程-恢复中断现场

恢复现场的前置条件是进行了保护现场过程，因此之前一定发生了中断（可能是时钟中断），当前的栈里只有四个对象，按地址从低到高的顺序分别是：调用 `restart` 的语句的下一条指令的地址（近返回地址），中断时被压入的 `ip,cs,flags`。用寄存器映像的内容分别修改各个通用寄存器和段寄存器，然后便是返回的重点。当前过程返回后，下一次返回时使用的是中断专用的返回，需要栈里面有 `IP,CS,flags`，然后 `iret` 调用时用 `flags` 修改标志寄存器，然后修改 `ip, cs`，最后出栈三个元素。而恢复中断现场需要修改这三个对象，才能保证 `restart` 之后 `iret` 可以正常返回，因此需要一个寄存器（`si`）用于基址寻址寄存器映像，栈基址寄存器（`bp`）用来寻址栈空间，一个通用寄存器（`dx`）用于中转数据，这三个寄存器都需要最后再恢复。代码如下：

Code 4: restart 过程

```

1  call dword getRegisterImage
2  mov si, ax
3  mov ax, [cs:si+0]
4  mov cx, [cs:si+2]
5  mov bx, [cs:si+6]
6  mov sp, [cs:si+8]
7  mov di, [cs:si+14]
8  mov ds, [cs:si+16]
9  mov es, [cs:si+18]
10 mov fs, [cs:si+20]
11 mov gs, [cs:si+22]
12 mov ss, [cs:si+24]
13 mov bp, sp
14
15 mov dx, word[cs:si+30] ; 新进程 flags
16 mov [bp+6], dx
17 mov dx, word[cs:si+28] ; 新进程 cs
18 mov [bp+4], dx
19 mov dx, word[cs:si+26] ; 新进程 ip
20 mov [bp+2], dx
21

```

```

22  mov bp, [cs:si+10]
23  mov dx, [cs:si+4]
24  mov si, [cs:si+12]
25
26  ret
27  %endmacro

```

5.3 系统调用实现

5.3.1 系统调用表定义

系统调用利用的是 21 号中断向量，在用户程序用 ah 指定了中断号后，触发 21 号中断，在内核的中断表中跳转到中断程序，执行后返回即可。这里为了保护上下文，可以使用 save 和 restart。由于 restart 会恢复上下文，如果中断处理程序有返回值在 ax，因为可能被覆盖，我安排了一个双字用于保存。而 ds 也会被恢复，为了使用内核的地址空间寻址数据，需要先保存 ds，取出 ax 后返回。代码如下：

Code 5: syscaller

```

1  syscaller: ; 不切换段
2  cli
3  call save ; 保存现场
4  mov si, cs ; si=0
5  mov ds, si ; ds=cs=0
6  mov si, ax ; 功能号
7  shr si, 8
8  add si, si
9  add si, si
10 call [sys_table+si]
11 mov [retval], ax
12 call restart
13 push ds ; 压栈 ds 用于恢复
14 mov ax, 0 ; ax 临时用于修改 ds
15 mov ds, ax ; 临时修改 ds 为 0，用于恢复 ax
16 mov ax, [retval] ; 修改 ax 为返回值
17 pop ds ; 恢复 ds
18 sti
19 iret
20 sys_table:
21 dd sys_putchar, sys_getch, sys_putchar_c

```

注意，可能是 Bochs 的 Bug 的原因，我的标号如果在数据区没有被定义成双字，在运行时并不会存在。因此在 si 做变址寻址时，需要使用 $4 \times si$ 。如果可以正常访问标号，那么使用 dw 和 $2 \times si$ 即可。

5.3.2 内核的系统调用过程编写

由于之前的实验里已经编写了比较完善的字符和 IO 库，这次除了增加一个系统调用外，其他都调用原有的函数即可。

系统调用-打印字符 保存现场，用栈基址寄存器寻址压入的参数，然后远调用（按照 c 的调用约定）putchar，再恢复栈寄存器即可。唯一需要注意的是参数所在的偏移量。总偏移量为 $4(\text{call dword})+6(\text{int 21})+2(\text{call})+16(\text{pusha})=28$ 。代码如下：

Code 6: 系统调用打印字符过程

```
1 sys_putchar: ; 系统调用：打印一个字符（不需要返回参数）
2 pusha
3 mov bp, sp
4 push word[bp+28]; 16(pusha)+2(call)+6(int 21)+4(call dword)
5 call dword putchar
6 add sp, 2
7 popa
8 ret
```

系统调用-获得键盘的一个输入（阻塞） 该过程无参数传入，所以直接远调用即可。参数放在 ax 中由中断程序处理。

系统调用-打印彩色字符 该过程是第一次实现，使用的是打印字符串的 bios 调用，传入的参数中可以包含颜色参数。其中还需要指定字符串的打印位置，因此需要使用获得光标位置的 bios 调用。参数所在的偏移与上一段原理相同，代码如下：

Code 7: 系统调用打印彩色字符过程

```
1 sys_putchar_c: ; 函数：在光标处打印一个彩色字符
2 pusha
3 mov bx, 0 ; 页号=0
4 mov ah, 03h ; 功能号：获取光标位置
5 int 10h ; dh=行，dl=列
6 mov bp, sp
7 add bp, 28 ; 参数地址，es:bp 指向要显示的字符
8 mov cx, 1 ; 显示1个字符
9 mov ax, 1301h ; AH = 13h（功能号）、AL = 01h（光标置于串尾）
10 mov bh, 0 ; 页号
11 mov bl, [bp+4] ; 颜色属性
12 int 10h ; 显示字符串（1个字符）
13 popa
14 ret
```

5.3.3 系统调用接口编写

完成了系统调用后，需要的是给用户编程的接口，这一部分代码需要与用户程序一起编译，用户程序不可以使用内核自己的过程。因为我使用的不是内联汇编，所以使用接口需要两步：

1. 在 c 的代码中调用 syscall_xx 函数，该函数由汇编定义。声明为 extern。
2. 在汇编代码中实现 syscall_xx 函数，指定功能号，并调用中断。

以打印字符的 putchar 为例，代码如下：

Code 8: 打印字符的系统调用

```

1 global syscall_putchar
2 syscall_putchar:
3     mov ah,0
4     int 21h
5     ret

```

5.4 实现库过程

大部分库过程都在之前的实验中实现，因此这次只设计一个库过程，打印彩色字符串。由于打印彩色字符的系统调用已经实现，打印彩色字符串只需要将字符串遍历打印。代码如下：

Code 9: 打印彩色字符串函数

```

1 void print_c(const char* str, uint8_t color) {
2     for(int i = 0, len = strlen(str); i < len; i++) {
3         syscall_putchar_c(str[i], color);
4     }
5 }

```

5.5 编写使用库过程的测试程序

测试程序为插入排序，程序在获得一串数字输入后，将递增序列打印出来。由于操作系统中并没有缓冲区，我也没有实现 `printf` 和 `scanf` 函数，所以获得了用户的完整输入后，还需要自己处理转换成数字。

5.5.1 atoi 函数

先去掉多余的空格，检查是否带负号，之后只需要遍历字符串来计算数字。要注意数字是否溢出。其中我使用 `isdigit` 判断开头是否为数字，该过程可以用宏来定义。代码如下：

Code 10: atoi 函数

```

1 #define isdigit(c) ((c) >= '0' && (c) <= '9')
2 int atoi(char* str) {
3     int negative = 0;
4     long long ret = 0;
5     if(0 == str)
6         return 0;
7     while(' ' == (*str))
8         str++;
9     if(0 == *str)
10        return 0;
11    negative = (*str == '-') ? 1 : 0;
12    if(!isdigit(*str))
13        return 0;
14    while(isdigit(*str))
15    {
16        ret = ret*10 + *str - '0';

```

```

17         if (ret > (negative?-(long long )INT32_MIN:INT32_MAX))
18             return negative?INT32_MIN:INT32_MAX;
19         str++;
20     }
21     return negative?-ret:ret;
22 }

```

5.5.2 main 函数

main 函数中需要打印字符串，然后获得输入，对输入的字符串的每个字符进行遍历，将数字字符串转换成数字并保存在字符数组中，调用插入排序函数，最后输出结果。返回前需要 getch()，否则执行完毕会清屏并立刻回到主界面。代码如下：

Code 11: main 函数

```

1  int main(){
2      char buff[200],numbuf[20];
3      char *greetMsg ;// 内容较长，省略
4      char *errMsg = "your_input_is_not_valid!\n";
5      char *exitMsg = "Press_any_key_to_exit..";
6      print_c(greetMsg,0x0B);
7      int cnt = 20;
8      int num[cnt];
9      int strHead=0,strTail = 0,j;
10     while (1){
11         readbuff(buff, 200);
12         while(' ' == buff[strHead]){
13             strHead++;
14         }
15         if(0 == buff[strHead])
16             continue;
17         break;
18     }
19     for (int i = 0; i < strlen(buff); i++)
20     {
21         if (!isdigit(buff[i]) && buff[i] != ' '){
22             puts(errMsg);
23             puts(exitMsg);
24             syscall_getch();
25             return 0;
26         }
27     }
28
29     for (int i = 0; i < cnt; i++,j=0)
30     {
31         j = 0;
32         strTail = strHead;
33         while('0'!=buff[strTail]&&' ' != buff[strTail]){
34             strTail++;

```

```

35     }
36     for (j = 0; j < strTail-strHead; j++)
37     {
38         numbuf[j] = buff[strHead + j];
39     }
40     numbuf[j] = '\0';
41
42     num[i] = atoi(numbuf);
43     if ('\0'==buff[strTail]){
44         cnt = i+1;
45         break;
46     }
47     while(buff[++strTail]!='\0')
48         ;
49     strHead=strTail;
50 }
51 insertionSort(num, num + cnt);
52 for (int i = 0; i < cnt; i++)
53 {
54     putnum(num[i], 10);
55     syscall_putchar('\0');
56 }
57 syscall_putchar('\n');
58 syscall_putchar('\r');
59 puts(exitMsg);
60 syscall_getch();
61 }

```

5.5.3 将程序放入镜像中测试

该程序为第五个程序，将其各项信息加入用户程序表后，就可以在内核中执行。结果请看运行与演示部分。[图1]

5.6 INT 22 实现和用户程序修改

5.6.1 INT 22 实现

按照实验要求，INT22 只输出“INT 22”，在 c 代码中实现打印函数，然后在中断过程中调用即可，代码如下：

Code 12: 22 号中断的汇编过程

```

1 sys_int22:
2     call dword sysc_int22
3     ret

```

Code 13: 22 号中断的 c 函数

```

1 void sysc_int22(){

```

```

2   char str[] = "INT22H_is_here!";
3   printPos(str, strlen(str), 22, 50);
4   }

```

5.6.2 用户程序修改-取消 Ouch 显示

在用户程序代码中删除修改中断向量的语句和有关 Ouch 代码。

以上便是实验中编写的所有代码。最后，还需要将各个中断程序的地址送入中断向量表。代码如下：

Code 14: 将中断程序地址写入中断向量表

```

1   VECTOR_IN 20h, gobackKernel
2   VECTOR_IN 21h, syscaller
3   VECTOR_IN 22h, sys_int22

```

VECTOR_IN 宏在上次实验中实现，实现可以在源代码中看到。

5.7 软盘扇区安排

这次增加了一个新的程序，在测试的时候我发现 2 个扇区是不够的，c 程序相比汇编会大很多。如果程序被截断，再运行时各种问题都可能发生。我为其安排了 8 个扇区，安排请看表1：

5.8 镜像文件生成

此次实验由于有新程序，需要单独编译成 COM 格式的文件，先使用 gcc 编译 test.c（测试源文件），mystring.c（库的实现），再使用 nasm 编译 syscall.asm（系统调用的汇编接口），最后使用 ld 指定入口为 main，代码段从 0x100 开始，生成 COM 程序。内核和其他用户程序的生成方式与之前的一样，可以参考源文件中的 Makefile。最后使用 dd 命令将引导程序，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 即可生成镜像 (.img) 文件，可以在 VMware 或者 bochs 中加载该镜像。需要注意的是，vmware 的显示和电脑一致，bochs 运行时时间流动速度较快，是正常现象。我认为是两个模拟器时钟脉冲参数不同导致的。

表 1: 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	18~19	2 (1 KB)	用户程序 1(LU.com)
1	20~21	2 (1 KB)	用户程序 2(LD.com)
1	22~23	2 (1 KB)	用户程序 3(RU.com)
1	24~25	2 (1 KB)	用户程序 4(RD.com)
1	26~33	8 (4 KB)	用户程序 5(test.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 make 即可得到镜像文件 os17338233.img。

6.3 运行与演示

6.3.1 使用库函数的测试程序

```
->ls
pid name cylinder head sector len addr seg
1 LU.com 0 1 1 1024 100 1000
2 LD.com 0 1 3 1024 100 1000
3 RU.com 0 1 5 1024 100 1000
4 RD.com 0 1 7 1024 100 1000
5 test.com 0 1 9 4096 100 1000
->run 5
This is a Insertion sort program.
please input some number(less than 2147483647)
20 18 16 12 11 222 33
11 12 16 18 20 33 222
Press any key to exit..

2020-06-12 22:46:06
```

图 1: test.com

图 1是输入一串数字后, 按下回车的结果

之前实验的所有特性在此次实验中都保留 (除了 Ouch), 和之前的实验一样。因此不再做其他的演示。具体的特性可以看录制好的 OSdemo.mkv

7 总结与讨论

7.1 特色、不足与改进

本次实验实现了系统调用后，加上执行程序的风格参照 dos，不需要约定返回方式，已经可以将用户编写程序的过程与内核完全解耦。原来的特性也完全保留，因此编写程序后，只需要修改用户程序表并将程序放入镜像即可直接使用操作系统加载。用户程序提供的库函数非常丰富，除了常见的字符串操作和 IO 外，可以打印彩色的字符串，完成数字字符串的转换等。不足之处有三点，

1. 内核的安全性不足，按照目前的方法，如果将用户程序返回地址放在内核中，下次实现多进程时会因为栈的位置不一致无法从用户程序正常返回，因此内核的许多信息都使用了用户程序的栈存放。
2. 用户的汇编程序中部分段寄存器的值与内核相关。
3. 时钟中断和软中断没有使用同一个寄存器映像保存现场，造成代码的冗余，原因在问题部分会解释。

7.2 收获

本次实验中，我对软中断和时钟中断有了更深入的了解，掌握了对 dos 执行程序的过程。在编写和调试保护现场恢复现场的代码时，汇编能力和调试能力有较大的提高。在编译自己写的 c 代码时，makefile 的编写也更加熟练 [4]。同时，在本报告的编写过程中，我也练习了利用 L^AT_EX 编写文档的能力。[5,6]

7.3 遇到的问题

这次遇到的问题数量不多，但是较上次更为复杂。主要有三点：

7.3.1 栈不平衡问题

由于使用了 c 和汇编的混合编程，而且经常出现汇编调用 c 函数的情况，在完成与 c 语言相关的工作时，必须一切都以双字为单位，比如 retf 和 call dword，而汇编则都以字为单位，一旦有不一致，就会出现栈不平衡，在 Bochs 中会出现 “WARNING: HLT instruction with IF=0!” 的问题，这个问题查资料时是不会找到和栈平衡相关的资料的。

7.3.2 标号问题

实现系统调用时，我发现在数据区定义时如果把标号声明成双字节，那么在生成的二进制文件中会找不到这个声明的数据。但是声明成双字后就正常。调试的时候用 Bochs 查看内存发现居然不存在，而且不断修改声明的方式位置，都未能解决问题，除了声明成双字。原因未知。

7.3.3 键盘中断导致的死机

使用了离开程序的中断后，我发现用户 com 程序一旦触发键盘中断 Bochs 就会出现 int13_diskette: unsupported AH=10，之后再也无法响应键盘，去掉 ouch 程序后正常，我认为这是嵌套中断导致的问题。

7.3.4 软中断嵌套时钟中断

这个问题花费了我数天时间来解决，即便询问了同学和老师，但是目前只有我遇到这个问题。因为我使用 `cli` 后认为软中断不会被打断所以我一直认为是 `restart` 和 `save` 导致的。问题发生在我将 `restart` 和 `save` 同时放入软中断过程和时钟中断过程中。用户程序一旦运行就会死机。而在内核中，即便时钟中断依然进行，但是没有任何问题。我再一开始忽视了这个现象，而一直在寻找 `restart` 和 `save` 的错误。实际上，只有同时时钟中断和软中断同时存在才会出现问题。而在我调试的过程中，我发现 `restart` 执行后，`iret` 根本无法返回，会进入 `bios` 程序的死循环，老师的解释是当前的寄存器信息与进入时的不一致，但是因为我时钟中断也执行了 `cli`，所以我以为是 `restart` 和 `save` 的问题，然而去掉软中断的 `restart` 和 `save` 之后，调试比较前后寄存器，并没有发现问题。并且在 `restart` 返回前，查看栈里的值，都是非常大的数字（比如 `fe00`），我猜测是 `bios` 的原因，认为是触发了某个中断，而且我发现 `restart` 时，本地的寄存器映像已经被修改了。并且我在调试时发现如果一直单步执行，程序时没有错误的。但是一旦使用 `next`，或者 `continue`，就会出现 `restart` 时寄存器映像被修改的问题，在同学的建议下，我在 `restart` 的执行前一句，加上了时钟中断的断点，并使用 `next`，发现果然进入了时钟中断。并且我还发现，时钟中断的 `cli` 处即使设置断点也不会触发，我猜想是因为根本没有执行。这个问题目前并没有解决，但是代码稍作修改，为时钟中断单独分配一个寄存器映像就不影响实验了。

7.4 感想

此次实验既使用了 `c` 语言，又使用了汇编，而且完成的工作都比较精细，由于需要不断的调试，我深刻体会到了自动化部署和调试工具的重要性。以及原理和实践之间的差距。这次实验即便花了这么久的时间，仍然是享受到之前实验的便利的，比如提前设计好的 `shell` 的用户程序加载，比如大量的库函数。所以每次实验还是要适当增加内容，减少以后的负担。

References

1. Gityuan, *Linux 系统调用 (syscall) 原理* <http://gityuan.com/2016/05/21/syscall/> 2016.,
2. Wikipedia, *Interrupt* — *Wikipedia, The Free Encyclopedia* <http://en.wikipedia.org/w/index.php?title=Interrupt&oldid=960304240> 2020.,
3. CSDN, 中断 `INT 20H` <https://blog.csdn.net/heavengl/article/details/6035824> 2010.,
4. how-to-write-makefile, 跟我一起写 *Makefile* <https://seisman.github.io/how-to-write-makefile/index.html> 陈皓, 2020.,
5. Leslie Lamport, *LaTeX: a document preparation system*, Addison Wesley, Massachusetts, 2nd edition, 1994.
6. Contributors to Wikibooks, *LaTeX Bibliography Management*. Wikibooks, 2019., en.wikibooks.org/wiki/LaTeX/Bibliography_Management.