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. 学习多道程序与 CPU 分时技术
2. 掌握操作系统内核的二态进程模型设计与实现方法
3. 掌握进程表示方法
4. 掌握时间片轮转调度的实现

2 实验要求

1. 了解操作系统内核的二态进程模型
2. 扩展实验五的的内核程序，增加一条命令可同时创建多个进程分时运行，增加进程控制块和进程表数据结构。
3. 修改时钟中断处理程序，调用时间片轮转调度算法。
4. 设计实现时间片轮转调度算法，每次时钟中断，就切换进程，实现进程轮流运行。
5. 修改 save() 和 restart() 两个汇编过程，利用进程控制块保存当前被中断进程的现场，并从进程控制块恢复下一个进程的现场。

3 实验内容

3.1 设计 PCB 数据结构

修改实验 5 的内核代码，定义进程控制块 PCB 类型，包括进程号、程序名、进程内存地址信息、CPU 寄存器保存区、进程状态等必要数据项，再定义一个 PCB 数组，最大进程数为 10 个。

3.2 设计多进程命令

扩展实验五的的内核程序，增加一条命令可同时执行多个用户程序，内核加载这些程序，创建多个进程，再实现分时运行

3.3 修改时钟中断处理程序

修改时钟中断处理程序，保留无敌风火轮显示，而且增加调用进程调度过程

4 实验原理

4.1 两进程模型

在任何时刻，一个进程要么正在执行，要么未执行，因而可以构建最简单的模型。进程可处于以下两种状态之一：运行态或未运行态，如图 3. 5（a）所示。操作系统创建一个新进程时，它将该进程以未运行态加入系统，操作系统知道这个进程的存在，并正在等待执行机会。时而不时地，当前正在运行的进程会被中断，此时操作系统中的分派器部分将选择一个新进程运行。前一个进程从运行态转换为未运行态，后一个进程则转换为运行态。[1]

表 1: 导致进程终止的原因

事件	说明
正常完成	进程自行执行一个操作系统服务调用，表示它已经结束运行
超过时限	进程运行时间超过规定的时限。可以测量多种类型的时间，包括总运行时间（“挂钟时间”）、花费在执行上的时间，以及对于交互进程从上一次用户输入到当前时刻的时间总量
无可用内存	系统无法满足进程需要的内存空间
超出范围	进程试图访问不允许访问的内存单元
保护错误	进程试图使用不允许使用的资源或文件，或试图以一种不正确的方式使用，如往只读文件中写
算术错误	进程试图进行被禁止的计算，如除以零或存储大于硬件可以接纳的数字
时间超出	进程等待某一事件发生的时间超过了规定的最大值
I/O 失败	在输入或输出期间发生错误，如找不到文件、在超过规定的最多努力次数后仍然读/写失败（如遇到磁带上的一个坏区时）或无效操作（如从行式打印机中读）
无效指令	进程试图执行一个不存在的指令（通常是由于转移到了数据区并企图执行数据）
特权指令	进程试图使用为操作系统保留的指令
数据误用	错误类型或未初始化的一块数据
操作员或操作系统干涉	由于某些原因，操作员或操作系统终止进程（如出现死锁时）
父进程终止	当一个父进程终止时，操作系统可能会自动终止该进程的所有子进程
父进程请求	父进程通常具有终止其任何子进程的权力

4.1.1 进程的创建

进程的创建将一个新进程添加到正被管理的进程集时，操作系统需要建立用于管理该进程的数据结构（见 33 节），并在内存中给它分配地址空间，这些行为构成了一个新进程的创建过程。触发进程创建的事件通常有 4 个，如表 3. 1 所示。在批处理环境中，响应作业提交时会创建进程；在交互环境中，当新用户试图登录时会创建进程。不论哪种情况，操作系统都负责新进程的创建工作。操作系统也可能会代表应用程序创建进程。例如，如果用户请求打印一个文件，则操作系统可以创建一个管理打印的进程，进而使请求进程可以继续执行，与完成打印任务的时间无关。

4.1.2 进程终止

表**??**概括了进程终止的典型原因。任何一个计算机系统都必须为进程提供表示其完成的方法，批处理作业中应包含一个 Halt 指令或其他操作系统显式服务调用来终止。在前一种情况下，Halt 指令将产生一个中断，警告操作系统一个进程已经完成。对交互式应用程序，用户的行为将指出何时进程完成。例如，在分时系统中，当用户退出系统或关闭自己的终端时，该用户的进程将被终止。在个人计算机或工作站中，用户可以结束一个应用程序（如字处理或电子表格）。所有这些行为最终将导致给操作系统发出一个服务请求，以终止发出请求的进程。此外，很多错误和故障条件会导致进程终止。表 32 列出了一些最常见的识别条件最后，在有些操作系统中，进程可被创建它的进程终止，或在父进程终止时而终止。

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. 设计 PCB 数据结构
3. 修改时钟中断程序
4. 修改 shell 的运行逻辑（增加多进程处理）

5.1 修改用户程序的执行方式

在上次的实验中，我已经修改了执行程序为如下方式：

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. 预先埋好用户程序返回所需的信息
4. 利用上一步的内存块，用远跳转（jmp far）跳入用户程序第一条指令的地址
5. 用户程序执行结束后用 20 号中断程序转去执行后处理程序（恢复内核段寄存器等）
6. 返回内核 shell

这次实验中，由于需要增加一种运行程序的方式，原来的执行方式需要分开，即分为加载程序和运行程序两个过程，什么时候执行由内核处理。

5.1.1 加载模块

由于加载完成后，不应该限制用何种方式运行程序。因此加载过程需要将所有运行程序前的准备都完成。具体有三点：

1. 将用户程序所在的镜像通过 13 号中断加载进内存中
2. 在用户程序头放入 int 20h
3. 在栈中预先埋好地址用于几次返回

在上次的代码的基础上，首先将 int 13h 之前的语句分出。由于使用用户栈需要修改 ss，在 ss 修改为用户程序所在段后，就不能使用内核栈了，因此我使用内核的内存来保存需要的信息（内核的 sp 等）其中还需要使用 es 等段寄存器用于修改用户程序信息，返回前需要恢复。代码如下：

Code 1: loadUsrProgram

```
1  loadUsrProgram:
2  pusha
3  push es
4  mov bp, sp
5  mov bx, [bp+22]          ; 程序信息结构体指针
6  mov ax, [bx+20]          ; 存放数据的段地址
7  mov es, ax               ; 用es才能跨段读取,es:bx是读入的数据所在内存地址
```

```
8   mov ah,2                    ; 功能号
9   mov al,[bx+12]              ; 扇区数
10  mov dl,0                    ; 驱动器号；软盘为0，硬盘和U盘为80H
11  mov dh, [bx+4]              ; 磁头号；起始编号为0
12  mov ch, [bx]                ; 柱面号；起始编号为0
13  mov cl, [bx+8]              ; 起始扇区号 ；起始编号为1
14  mov bx, [bx+16]             ; 存放数据的内存偏移地址
15  int 13H                     ; 调用读磁盘BIOS的13h功能
16  mov word[cs:savesp],sp      ; 保存当前的栈顶，用于返回后恢复
17  mov sp,0xff00               ; 设置用户程序的栈顶
18  mov bx, es
19  mov ss, bx                  ; 设置现在的栈段和es一致(以后push和pop就会操作用户程序的栈)
20  mov ax,[cs:codeOfInt20]     ; 用ax保存int 20这个语句
21  mov [es:0],ax               ; 在es:0(用户程序所在段首)放置int 20
22  push cs                     ;
23  push afterRun               ; 将cs ip先后压栈，返回时可以远返回retf
24  push dword 0                ; 压栈0，用户程序如果使用ret就会触发ip=0的操作，开始执行从es:0
25  mov bx, cs
26  mov ss, bx
27  mov sp,[cs:savesp]
28  pop es
29  popa
30  retf
```

5.1.2 运行模块

运行模块是用于批处理或者执行单个程序的，将各个段寄存器和栈寄存器设置好，压入地址并远返回即可转到用户程序的 100h 处开始执行。代码如下：

Code 2: runUsrProgram

```
1   runUsrProgram:
2   pusha
3   mov bp, sp
4   mov bx, [bp+20]             ; 程序信息结构体指针
5   mov ax, cs
6   mov gs, ax
7   mov word[gs:savesp],sp      ; 保存当前的栈顶，用于返回后恢复
8   mov ax,0xff00               ; 用户程序的栈顶为0xff00
9   mov sp, ax                  ; 设置用户程序的栈顶
10  sub sp, 8
11  mov bx, [bx+20]
12  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
13  mov ds, bx                  ; 设置用户程序的数据段和es一致
14  mov ss, bx                  ; 设置现在的栈段和es一致(以后push和pop就会操作用户程序的栈)
15  push bx
16  push 0x100
17  retf                        ;远返回至bx:100
```

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 3: struct RegisterImage

```
typedef struct RegisterImage{
    uint16_t ax;      // 0
    uint16_t cx;      // 2
    uint16_t dx;      // 4
    uint16_t bx;      // 6
    uint16_t sp;      // 8
    uint16_t bp;      // 10
    uint16_t si;      // 12
    uint16_t di;      // 14
    uint16_t ds;      // 16
    uint16_t es;      // 18
    uint16_t fs;      // 20
    uint16_t gs;      // 22
    uint16_t ss;      // 24
    uint16_t ip;      // 26
    uint16_t cs;      // 28
    uint16_t flags;   // 30
} RegisterImage;

RegisterImage KernalContext;
RegisterImage* getRegisterImage(){
    return &KernalContext;
}
```

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的进程。

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 4: save 过程

```
save:
pusha
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 5: restart 过程

```
    call dword getRegisterImage
    mov si, ax
    mov ax, [cs:si+0]
    mov cx, [cs:si+2]
    mov bx, [cs:si+6]
    mov sp, [cs:si+8]
    mov di, [cs:si+14]
    mov ds, [cs:si+16]
    mov es, [cs:si+18]
    mov fs, [cs:si+20]
    mov gs, [cs:si+22]
    mov ss, [cs:si+24]
    mov bp,sp

    mov dx, word[cs:si+30]              ; 新进程flags
    mov [bp+6],dx
    mov dx, word[cs:si+28]              ; 新进程cs
    mov [bp+4],dx
    mov dx, word[cs:si+26]              ; 新进程ip
    mov [bp+2],dx

    mov bp, [cs:si+10]
    mov dx, [cs:si+4]
    mov si, [cs:si+12]

    ret
%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 6: syscaller

```
    syscaller:;不切换段
    cli
    call save ;保存现场
    mov si,cs ;si=0
    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(call dword)+6(int 21)+2(call)+16(pusha)=28。代码如下：

Code 7: 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 8: 系统调用打印彩色字符过程

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

```
global syscall_putchar
syscall_putchar:
    mov ah,0
    int 21h
    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 10: 打印彩色字符串函数

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

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 11: atoi 函数

```
#define isdigit(c) ((c) >= '0' && (c) <= '9')
int atoi(char* str) {
    int negative = 0;
    long long ret = 0;
    if(0 == str)
        return 0;
    while(' ' == (*str))
        str++;
    if(0 == *str)
        return 0;
    negative = (*str == '-') ? 1 : 0;
    if(!isdigit(*str))
        return 0;
    while(isdigit(*str))
    {
        ret = ret*10 + *str -'0';
        if(ret > (negative?-(long long )INT32_MIN:INT32_MAX))
            return negative?INT32_MIN:INT32_MAX;
        str++;
    }
    return negative?-ret:ret;
}
```

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 12: main 函数

```
int main(){
    char buff[200],numbuf[20];
    char *greetMsg ;//内容较长，省略
    char *errMsg = "your input is not valid!\n";
    char *exitMsg = "Press any key to exit..";
    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]=='␣')
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(' ');
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 13: 22 号中断的汇编过程

```
1  sys_int22:
2    call dword sysc_int22
3    ret
```

Code 14: 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 15: 将中断程序地址写入中断向量表

```
1    VECTOR_IN 20h,int20
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2：

表 2: 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)

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。

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

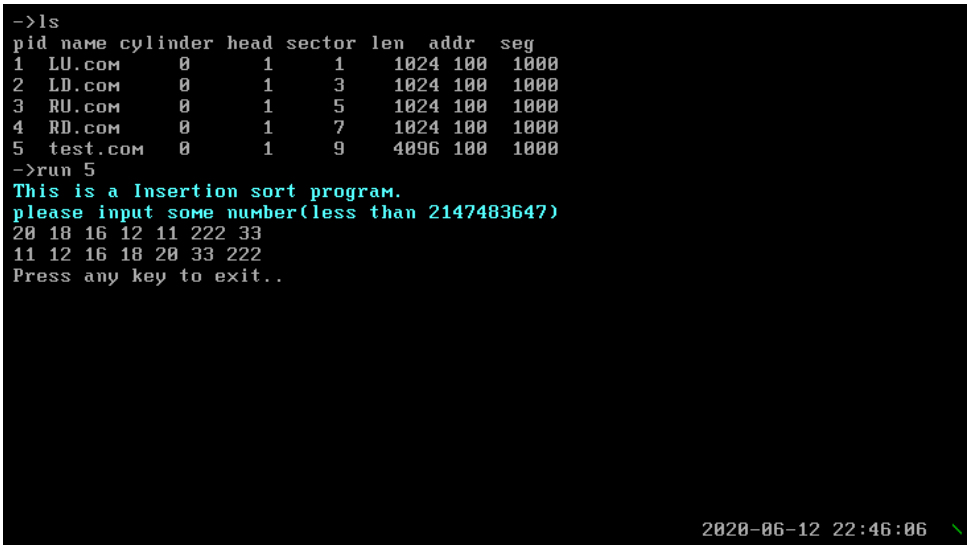

图 1: test.com

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 LaTeX 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.