# 操作系统原理实验课程
# 实验报告
# 实验六 进程模型

学院：数据科学与计算机学院
专业：计算机科学与技术
年级：**2016** 级
实验人姓名（学号）：王锡淮（**16337236**）

2018 年 4 月 24 日

# 目录

**1 实验目的** **2**

**2 实验要求** **2**

**3 实验内容** **2**

**4 实验方案** **3**

4.1 关于虚拟机的问题 . . . . . 3

4.2 实验工具以及环境 . . . . . 3

4.3 方案及其思想 . . . . . 3

4.3.1 五状态进程模型 . . . . . 3

4.3.2 内存管理模块 . . . . . 11

4.3.3 调度管理模块 . . . . . 13

4.4 代码文档组成说明 . . . . . 15

**5 实验过程** **16**

5.1 主要流程 . . . . . 16

5.2 输入输出说明 . . . . . 16

5.3 结果展示 . . . . . 16

5.4 遇到的问题以及解决情况 . . . . . 18

**6 实验总结** **21**

**A 参考文献** **22**

# 1 实验目的

1. 学会实现进程控制块。
2. 学会实现进程表。
3. 学会实现进程切换以及模式切换，以及利用时钟中断。
4. 学会实现基本的调度。
5. 学会实现进程模型，在内核实现多进程的二状态模型，理解简单进程的构造方法和时间片轮转调度过程。
6. 学会实现解释多进程的控制台命令，建立相应进程并能启动执行。
7. 至少一个进程可用于测试前一版本的系统调用，搭建完整的操作系统框架，为后续实验项目打下扎实基础。

# 2 实验要求

保留原型原有特征的基础上，设计满足下列要求的新原型操作系统：

1. 在 c 程序中定义进程表，进程数量至少 4 个。
2. 内核一次性加载多个用户程序运行时，采用时间片轮转调度进程运行，用户程序的输出各占 1/4 屏幕区域，信息输出有动感，以便观察程序是否在执行。
3. 在原型中保证原有的系统调用服务可用。再编写 1 个用户程序，展示系统调用服务还能工作。

# 3 实验内容

1. 建立了未运行态、就绪态、运行态、阻塞态、推出态五状态进程模型。
2. 实现了“run”指令，可以加载不定数目的程序文件创建进程并运行。
3. 继承了以往实验的系统调用、单级文件系统、中断等等，在原型中保证原有的系统调用服务可用，利用了以往实验使用的一个用户程序，展示系统调用服务还能工作。
4. 实现了基本的内存管理模块，包括动态内存分配和回收。
5. 完成了基本的调度管理，包括长程调度、中程调度、短程调度。

# 4 实验方案

## 4.1 关于虚拟机的问题

首先是 Bochs 效率太低，运行速度较低，每次加载一个用户程序都要等一段时间，而且不利于观察；但是我的镜像文件在 VMWare 上运行不了；所以我在本次实验中使用 qemu 来进行最后的展示，但是也发生了问题，也就是说在两个虚拟机中运行的结果并不相同。

## 4.2 实验工具以及环境

实验工具：sublime text 编写程序-nasm 和 gcc 混合编译-ld 连接器进行连接-objdump 进行反汇编得到和 C 语言一行一行匹配的汇编代码-dd for windows 将二进制文件复制到软盘上-bochs 调试-bochs 和 qemu 进行展示。

环境：windows 10 操作系统。

## 4.3 方案及其思想

在本实验中新增或完善的模块主要包括五状态进程模型，内存管理模块以及调度管理模块。

### 4.3.1 五状态进程模型

**实验原理**

本实验中实现了一个包含未运行态、就绪态、运行态、阻塞态、退出态（Origin，ready，running，blocked，exit）的五状态进程模型（较以往实验的实现更加完善），该模型的状态转移关系如图1所示。

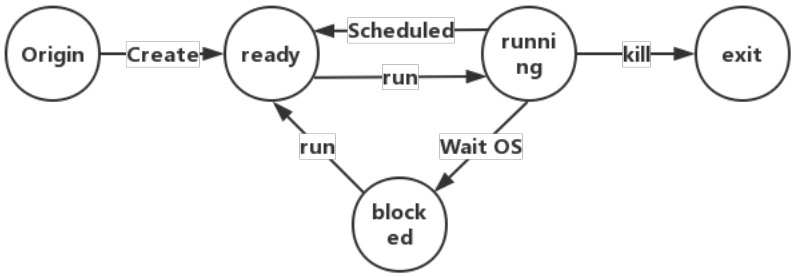



图 1: 五状态进程模型

其中，Create 指的是在内核中完成进程的创建过程，其详细内容如下：

- 分配唯一的进程标识符，并在主进程表中加入一个新表项，在本实验中按照顺序赋予进程标识符。
- 分配空间，包括进程映像中的所有元素，由于本实验中尚未实现动态分配数据空间（比如 malloc() 和 new()）的功能，所以只需要考虑代码段、数据段和栈空间的分配，本实验的栈空间同意大小为 1K，而其余空间大小读取自文件信息。
- 初始化进程控制块，除了程序计数器、代码段地址、用户栈段地址、用户栈指针由内存管理模块确定内存位置后指定之外，其余信息一般初始化为 0。
- 连接，放入相应状态调度队列中，本实验中只有一个就绪队列。

run 过程在本实验中是指通过 run 指令或者. 指令将就绪态的进程通过调度策略选择运行，而 run 过程则会根据调度算法将 CPU 控制权从正在运行的进程中转移到选择的下一个进程。

kill 过程在本实验中是 kill 指令，该指令将进程状态置为退出态，并释放该进程所占用的内存。

在本实验中用户进程只需要等待 OS 的执行这一事件，即只有当请求系统调用，控制权转换回内核之后（模式切换），用户进程才会被阻塞，响应地用户进程会在内核执行完毕后（只有 run 和. 指令被执行的时候），重新进入就绪态。

**程序流程和算法思想**

该模块的流程如图2所示。

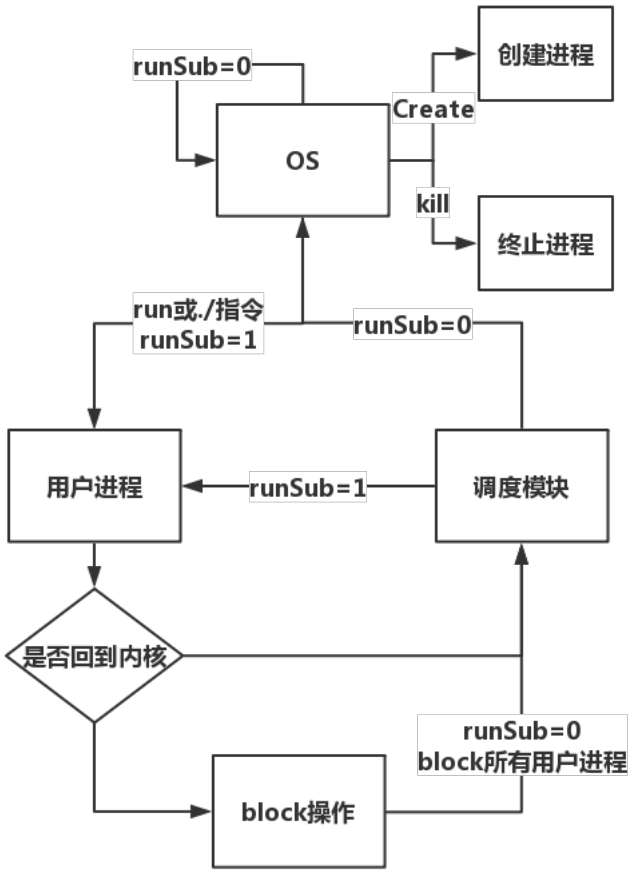



图 2: 进程模块流程

在该模块内的算法思想是用统一的接口实现调度，即无论是 block 操作，普通的进程调度切换和批量进程的执行还是单独程序的执行中的切换都是使用 int 8h 进行处理。

**数据结构和程序模块功能**

在实现上述原理的过程中主要涉及的是进程控制块的设计，进程的保存和切换，代码展示如下：

```
struct PCB
{
        short gs;
```

```
        short fs;
        short es;
        short ds;
        short di;
        short si;
        short bp;
        short sp;
        short bx;
        short dx;
        short cx;
        short ax;
        // short retaddr;

        short ip;
        short cs;
        short flags;
        short sp_now;
        short ss_now;
};

enum Processstatus
{
        Origin=0, ready, running, blocked, suspend, exit
};

struct Process
{
        struct PCB pcb;
        int id;
        int blockNum;
        char name[30];
        enum Processstatus status;
};
```

这是进程表的设计，其中的 blockNum 是内存单元结构（见内存管理模块）的标号，用于将来实现挂起态时寻找内存位置。

```
cli
mov al, 20h
out 20h, al
```

```
out 0A0h, al
pop es
pop ds
popa

push bx
push ax
push ds
mov ax, cs
mov ds, ax
mov bx, [cs:_curProcessId]
mov ax, [cs:_SizeOfProcessStruct]
mul bl
mov bx, ax
add bx, _processTable
mov word [cs:old_process], bx
pop ds
pop ax
pop bx

push ax
; flags/cs/ip
push ds
push cs
pop ds

pusha
xor eax, eax
mov ax, _schedule
call dword eax
popa
pop ds
pop ax
; _curProcessId 已经改好

; 用户栈里面有flags(user)\cs(user)\ip(user)
; flags/cs/ip
pop word [cs:in2b_store_ip] ;ip
pop word [cs:in2b_store_cs] ;cs
```

```
popf
; 恢复标志变量，存储ip和cs

push word [cs:in2b_store_ip]
push word [cs:in2b_store_cs]

push bx
push ax
push ds
mov ax, cs
mov ds, ax
mov bx, [cs:_curProcessId]
mov ax, [cs:_SizeOfProcessStruct]
mul bl
mov bx, ax
add bx, _processTable
mov word [cs:new_process], bx
pop ds
pop ax
pop bx


call save
call restart
```

这是 8h 号中断中切换进程的主要代码，其功能为根据当前进程号保存当前进程的 pcb 地址，再运行调度算法，得到下一个进程的进程号，再得到其 pcb 地址，进行切换。

```
save:
        mov [cs:store_sp_dispatch], sp;
        pop word [cs:store_retaddr]
        ;此时栈中为进程ip
        push ds
        ;;;; 切换到PCB表所在 ;;;;;
        push cs
        pop ds
        push si
        push di
        ; mov si, [cs:new_process] ;new_process
        mov di, [cs:old_process] ;old_process
```

```

        pop word [cs:store_di] ;di
        pop word [cs:store_si] ;si
        pop word [cs:store_ds] ;ds
        pop word [cs:store_cs] ;cs
        pop word [cs:store_ip] ;ip
        ;;;sp恢复到进入函数之前的值;;;;
        mov word [di+16*2], ss
        mov word [di+15*2], sp
        push cs
        pop ss
        add di, 15*2
        mov sp, di
        pushf
        push word [cs:store_cs]
        push word [cs:store_ip]
        pusha
        ;;; si,di have been changed;;;;
        pop di ;si
        pop di

        push word [cs:store_si]
        push word [cs:store_di]
        push word [cs:store_ds]

        push es
        push fs
        push gs

        mov [cs:store_ax_dispatch], ax
        mov ax, [cs:store_sp_dispatch]
        mov sp, ax
        mov ax, [cs:store_ax_dispatch]
        jmp word [cs:store_retaddr]


restart:
        ; 中断已经关闭
        ;;注意调用时的压栈等等
```

```
        push cs
        pop ss
        pop word [cs:store_retaddr]
        ;;; 切换到PCB表 ;;;;
        mov sp, [cs:new_process]
        mov ax, sp
        add ax, 12*2
        mov word [cs:store_sp], ax
        ;;restore
        pop gs
        pop fs
        pop es
        pop ds
        popa
        mov sp, word [cs:store_sp]
        pop word [cs:store_ip]
        pop word [cs:store_cs]
        popf
        pop word [cs:store_sp]
        pop ss
        mov word [cs:store_ax], ax
        mov ax, word [cs:store_sp]
        mov sp, ax
        mov ax, word [cs:store_ax]
        sti
        pushf
        push word [cs:store_cs]
        push word [cs:store_ip]
        iret
```

本实验中实现的 save 和 restart 过程主要利用的是将栈指针指向 pcb 地址的方法，直接利用 push 和 pop，实现较为紧凑。

```
int2Bh:
    ; 注意进来这里的时候，其实cs是PCB的cs（内核），
    其余的ds,es,ss都是用户态
    ; 用户栈里面有
    \ip(user)\flags(user)\cs(checkinput/user)\ip(checkinput)
    ; Process的大小是 _SizeOfProcessStruct，pcb在其首地址。

```

```
    pop word [cs:in2b_store_ip] ;ip
    pop word [cs:in2b_store_cs] ;cs
    popf
    pop word [cs:in2b_store_ip] ;ip
    ; 恢复标志变量，存储ip和cs

    pusha
    push ds
    mov ax, cs
    mov ds, ax
    mov eax, 0
    mov ax, _block
    call eax
    pop ds
    popa

    pushf
    push word [cs:in2b_store_cs]
    push word [cs:in2b_store_ip]
    ; flags\cs\ip
    jmp int8h
```

以上是返回内核态的代码（模式切换），其主要流程是调用 block 操作的函数，将所有用户程序阻塞，将 runSub（相当于控制内核态的优先级的变量）置为 1（内核态优先级提升），于是下一次调用时钟中断时会调用内核进程。

### 4.3.2 内存管理模块

**实验原理**

本实验中实现了一个具有动态分配进程的内存空间功能的内存管理模块。该模块使用的是普通的动态分区方法，使用的放置算法是首次匹配放置算法。将内存中每一块内存的信息都记录再一个链表中，根据信息分配空间，根据分配的空间更改信息。

关于重定位，这里使用了替代方案，即将 cs 改成能够使用户程序的 org 偏移量为 0 的恰当值，这里使用的方案是将物理地址（20 位）右移 4 位作为 cs 的值。**程序流程和算法思想**

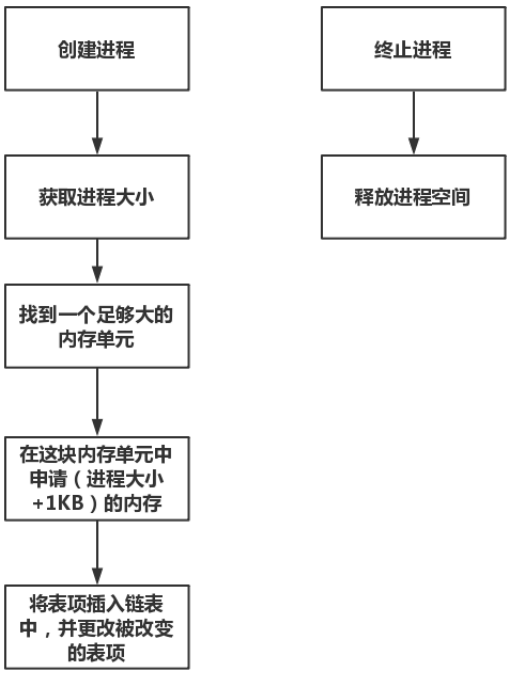



图 3: 内存管理模块

该模块使用的是普通的动态分区方法，使用的放置算法是首次匹配放置算法。

**数据结构和程序模块功能**

内存单元结构定义如下：

```
enum memoryStatus
{
        unused = 0, used
};


struct MemoryBlock
{
        int beginAddr;
        int endAddr;
        int pre;
        int next;
```

```
        int used;
        enum memoryStatus status;
};
```

其主要难点在于用数组实现一个链表，以及终注意申请内存块时处理剩下的内存块（保留或合并），和释放内存块时将其合并入内存块状态未使用的内存单元。

重定位的替代方案如下：

```
void createProcess(int id, char * name, int size, int fileId)
{
        processTable[id].id = id;
        processTable[id].status = ready;
        int blockNum = findEnoughBlock(size);
        blockNum = require(size, blockNum);
        processTable[id].blockNum = blockNum;
        strncpy(name, processTable[id].name, strlen(name));
        int offsetOfPrg, segOfPrg;
        int offsetMask = 0x0000ffff;
        int segMask = 0xf0000;
        offsetOfPrg = memoryTable[blockNum].beginAddr & offsetMask;
        segOfPrg = (memoryTable[blockNum].beginAddr & segMask) >> 4;
        //以下部分代码取巧，方法是直接改变cs和ss段
        （加上偏移量右移4位的值）以使得程序（org 0h）不需要org偏移量。
        processTable[id].pcb.cs = segOfPrg + (offsetOfPrg >> 4);
        processTable[id].pcb.ip = 0;
        processTable[id].pcb.ss_now = segOfPrg + (offsetOfPrg >> 4);
        processTable[id].pcb.sp_now = size-1;
        loadReal(information[fileId].lmaddress,
        information[fileId].size, offsetOfPrg, segOfPrg);
}
```

其中，memoryTable[blockNum] 记录了分配得到的内存块信息，需要注意的是申请的内存块大小要 4 字节对齐。

### 4.3.3 调度管理模块

#### 实验原理

本实验中，调度管理采用的方案如下：

1. 长程调度：所有程序都将进入系统中进行处理。

2. 中程调度：由于目前没有实现挂起状态（因为文件系统还未完善，无法实现进程块的换入与换出），不存在中程调度。

3. 短程调度：本实验中的短程调度策略为 Robin Round（时间片轮转），其中变量 runSub 用以提高内核的优先级，runSub 为 0 时内核优先级最高，反之最低。

**程序流程和算法思想**

程序流程为顺序查看进程状态，将就绪态进程调入运行。

**数据结构和程序模块功能**

```
void schedule() {
if(runSub == 0)
{
        curProcessId = 0;
        return ;
}
else if(curProcessId != 0 ||(runSub == 1))
{
        for(int i = (curProcessId+1)%ProcessSize;
        i !=curProcessId; i = (i+1)%ProcessSize)
        {
                if(i != 0 && processTable[i].status == ready)
                {
                        processTable[i].status = running;
                        processTable[curProcessId].status
                         = ready;
                        curProcessId = i;
                        return ;
                }
        }
}//else if(curProcessId != 0 ||(runSub == 1))

}//void schedule
```

由于该算法保证能够不断遍历进程表，故不会出现进程饥饿问题。

## 4.4 代码文档组成说明

```
 dump.txt
 tree.txt

├boot
      bochsout.txt
      bochsrc
      boot.img
      linkscript
      loader.asm
      log.txt
      run.bat

├doc
      boot.img

├kernel
   │  FAT.asm
   │  os.c
   │
   └lib
          fileRecords.txt
          manual.txt
          utils.asm
          utilsAsm.h
          utilsC.c
          utilsC.h

└tools
   └user
           ball_A.asm
           ball_B.asm
           ball_C.asm
           ball_D.asm
           printBigName.asm
           testInt34_37.asm
           testSysCall.asm
```

图 4: 代码文档组成说明

说明：Dump.txt 是整个操作系统的反汇编文件；boot 子文件夹下面的 loader.asm 是引导程序，run.bat 是批处理文件，其余的是使用 bochs 和 ld 的配置文件；doc 子文件夹下的是操作系统的镜像，其中 boot.tmp 是带有调试信息的文件；kernel 子文件夹下面的是 FAT 文件和内核 C 语言代码，其中的 lib 文件夹下面的是内核汇编语言的代码和所需要用到的文件；tools 文件夹下面的 testInt34_37.asm 是测试 int 34h 到 int 37h 的用户程序，testSysCall.asm

是测试系统调用的用户程序，user 文件夹下面的是 5 个用户程序。

# 5 实验过程

## 5.1 主要流程

创建.img 虚拟软盘，编写汇编以及 C 语言的代码，其中 loader 和用户程序的代码使用 nasm 直接汇编出二进制文件，其余的和系统内核相关的代码使用混合编译技术得到二进制文件，再使用 bochs 调试，以及最后展示结果。

## 5.2 输入输出说明

本操作系统中的输出分为三种：提示信息（白色）、指令应答（紫色）、输入的回显（黄色）；输入为设定好的指令，在输入错误时不会有反应，在本实验中有一个特殊的指令“run”，在输入该指令并按下回车键后可以输入不定数目的程序文件名字，并且在最后按下两次回车后，输入和所有程序和本就存在的在等待内核执行完毕的进程会一起运行。

## 5.3 结果展示

首先查看用户手册获取使用方法。

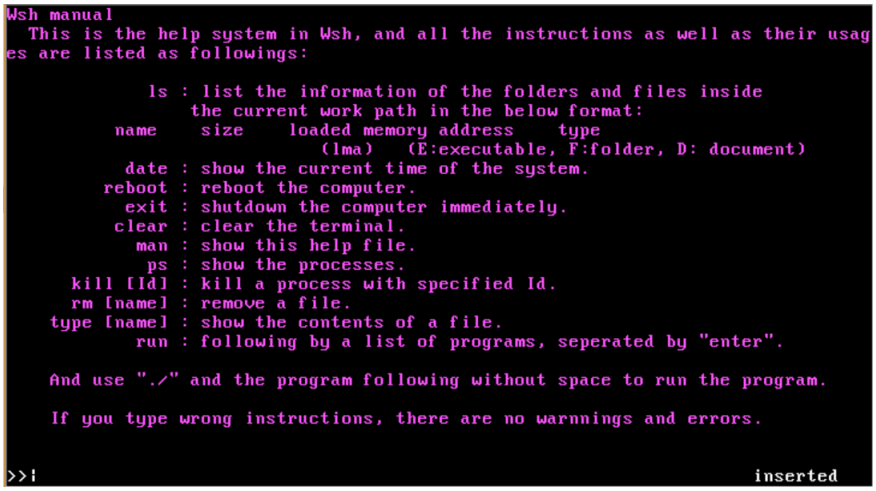



图 5: 用户手册

观察到和本次实验相关的指令有 kill，run，./，ps，于是进行测试。首先使用 ps 指令查看当前进程，ls 指令查看有什么用户程序可以调用。

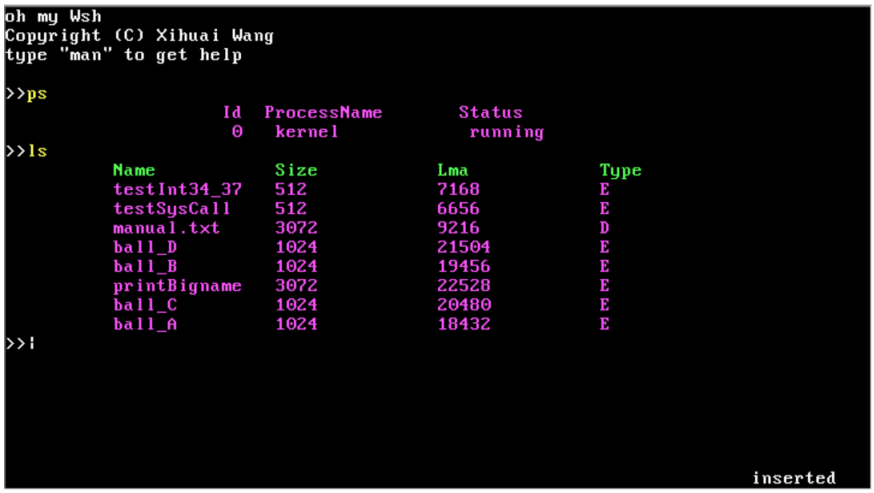
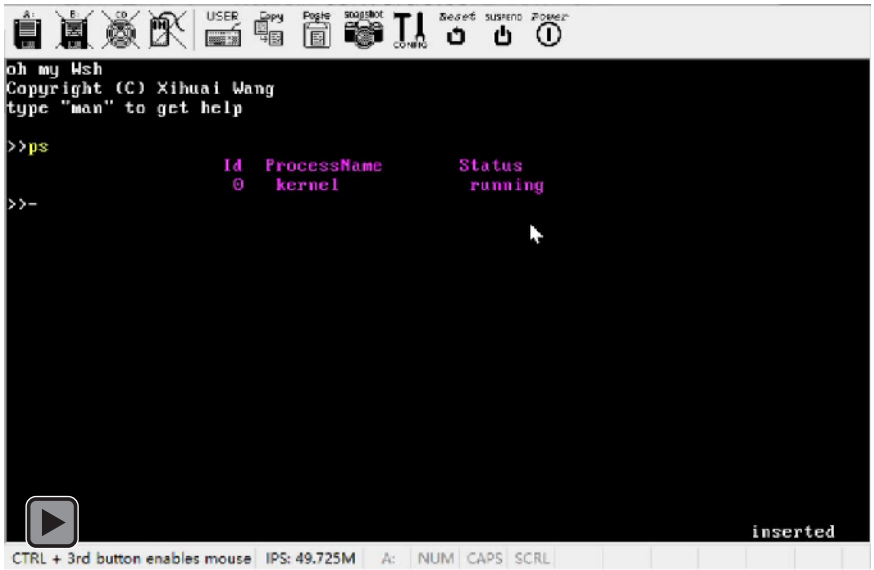
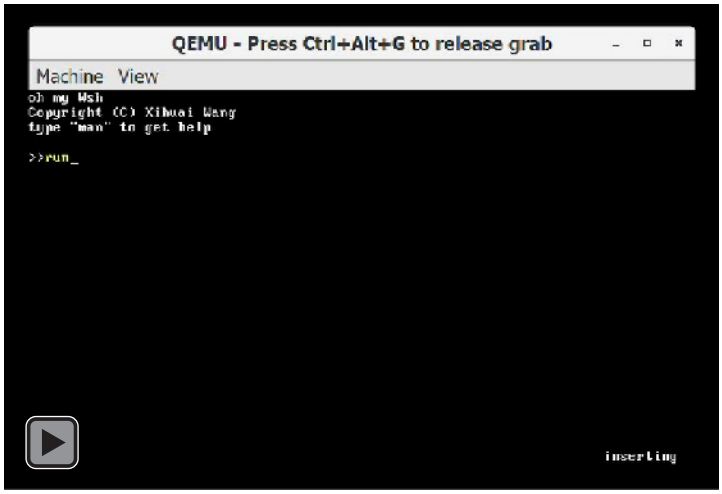



图 6: demo1

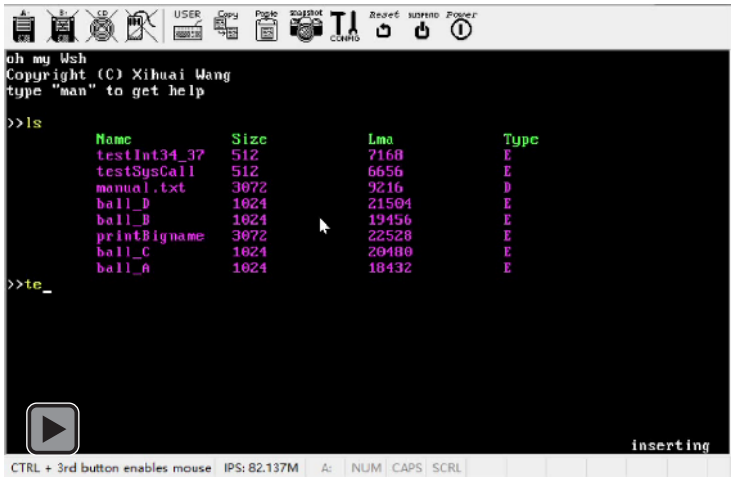
随后，测试./和 kill 指令，在执行 printBigname 后按下 q 键返回内核，并在之后将其删除（exit 态）。



接着测试 run 指令，同时载入 5 个用户程序进行测试，并返回内核。




最后是系统调用的测试。




## 5.4 遇到的问题以及解决情况

1. 进程表项的大小由于对齐的要求从 4Ch 变成了 50h，而我一开始使用的方法时将进程表项的大小硬编码在汇编程序中，导致 pcb 表的地址寻址错误，这是通过在 bochs 中查看内存内容发现的。解决方法是使用一个 constant 全局变量记录进程表项的大小，其值为 sizeof(Process)，注意 sizeof 是关键字，而不是函数，可以正常调用。

2. 改动 ss 或者 sp 时，一定要直接赋值或通过内存单元进行保存和赋值，而不能通过栈，否则

会出错。

3. 使用统一的进程切换方式的重要性（本实验中为 int 8h），一开始我有 3 个不相同的进程切换函数（从内核到进程，从进程到内核，批处理进程运行），由于在编程时无法考虑时钟中断何时发生，如果当前进程号（curProcessId）在 8 号中断到来之前改变，则当前进程和下一个进程的进程号在此时相同，无法完成进程切换（举个例子，此时 curProcessId 为 1，要切换到 2 号进程，curProcessId 在时钟中断到来前改变为 2，则 int 8h 完成的是从 2 号进程切换到 2 号进程，进程切换失败）。这是通过不断查看 curProcessId 对应的内存单元的值发现的。解决方法是只用 int 8h 来完成实质的进程切换，只在其中改变 curProcessId 的值（调用调度函数），其余函数的功能是通过改变条件改变调度函数的选择。

4. 在 save 过程中保存下来的值不一定和期望的值（cs 和 ss 等值和给定的不同），但这是正常情况。我在 debug 某个 bug 时，逐步观测 save 过程的操作，发现保留下来的值很奇怪，如图7所示，在 cs 和 ip 对应的位置是 0xf000 和 0xe869, 以为是 save 过程出错，可是改正了其他 bug 后程序正常运行了，才知道原来这是正常值，是时钟中断在不屏蔽中断字的中断中引发保留下来的值，这也说明 int 8h 一定不能改变中断上下文。

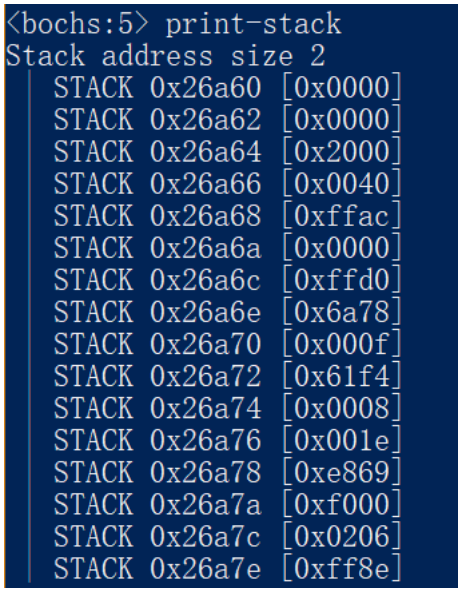



图 7: save 过程截图

5. gcc 的优化会将一部分在 c 语言中声明和赋值但是只在汇编语言部分使用的全局变量的赋值语句忽视掉，不能开优化。

比如下面这个例子：

```
runSub = 1;
wakeUp();
__asm__("int $0x8\n");
runSub = 0;
clear();
```

在不开优化的情况下，得到的汇编代码是

```
os.c:56
        runSub = 1;
 213: 66 c7 06 20 61 01 00 mov  DWORD PTR ds:0x6120,0x1
 21a: 00 00
os.c:57
        wakeUp();
 21c: 66 e8 0d 39 00 00     call 3b2f <_wakeUp>
os.c:58
        __asm__("int $0x8\n");
 222: cd 08                 int  0x8
os.c:59
        runSub = 0;
 224: 66 c7 06 20 61 00 00 mov  DWORD PTR ds:0x6120,0x0
 22b: 00 00
os.c:60
        clear();
 22d: 66 e8 02 15 00 00     call 1735 <_clear>
 233: e9 79 04              jmp  6af <.debug_info+0x186>
```

而如果开了优化选项，得到的则是没有第 0x213 和 0x21a 行的结果。而且这只是情况中的一种，开优化会得到许多奇怪的代码。

6. 在汇编中调用 C 语言函数时，使用的是用户（调用者）的栈，但是如果使用全局变量，要使用内核的 ds（在本实验中也即内核的 cs）。遇到的情况是一致加载不对变量，发现是 ds 不对。

7. 关于 int 8h 作为唯一的切换函数的进一步想法，遇到了一个情况是在没有关中断的时候，在 8 号中断内部时钟中断再次被调用，造成 pcb 的 save 过程出错。解决方法是在进入清空中断屏蔽字（cli）。这个错误十分难以发现，因为从 bochs 观察的结果是 ip 突然变化，进入时钟中断，而且难复现。

8. 有一个很隐秘的错误，我实现 save 过程时采用的是将 ss 和 sp 指向特定 pcb 的地址压栈的方法。这样就要注意最后要将 sp 和 ss 改回来，不然在之后的过程中可能会修改到其它进程的 pcb。

# 6 实验总结

对于本次实验，一开始我是很疑惑的，我想着的总是现在自己在用的操作系统，于是以为要实现的是能在后台运行的进程模型，终于找了我的疑惑在哪：

原来支持多终端和后台的操作系统才能完成这一个目标，对于现在学的操作系统概念（单道、多道、分时）都是已经提供好了作业序列，即使是分时系统（其有多个作业序列），所以我的操作系统不能实现的功能是正常的。所以在这次实验中使用的方法应当是在命令行里面加入一批作业名称，然后一起执行。

对于 C 语言和汇编的某些约定真是马虎不得，在这次实验中我就吃了亏，要我帮忙 Debug 的同学也有不少是吃了这些亏。比如进入中断一般要禁用中断、无论是函数还是中断都要保证除了出口参数，其余寄存器都不能改变。

还有对于挂起态、动态分配堆空间、多级文件、重定位的思考。关于挂起态，我的想法是将整个内存块移入到磁盘的特定位置，这就需要一个完善的文件系统，所以和文件系统的完善相关联；关于动态分配堆空间，我还想不到什么好的实现方法，虽然我已经能够动态分配内存，直接给进程分配一个堆空间然后记录使用情况就可以了吗？关于重定位，我的感觉是我实现不了，因为不知道怎么读取和处理符号表。

在本次实验中我才真正学会了用 C 语言处理逻辑，汇编处理底层操作，感觉方便了不少。

下一个实验争取实现挂起态和动态分配堆空间，这样就可以真的使用“链表”而不用使用数组来模拟了。

# Appendices

## A 参考文献

1. 操作系统：精髓与设计原理（第七版）

2. Orange'S：一个操作系统的实现