《操作系统原理实验》

实验报告

（实验七）

学 院 名 称 ： 数据科学与计算机学院

专业（班级）： 16 计科 2 班

学 生 姓 名 ： 朱志儒

学 号 ： 16337341

时 间 ： 2018 年 5 月 29 日

实 验 七 ： 进程控制与通信

一． 实验目的

1、 完善实验6中的二状态进程模型，实现五状态进程模型，从而使进程可以分工合作，并发运行。

2、 了解派生进程、结束进程、阻塞进程等过程中父、子进程之间的关系和分别进行的操作。

3、 理解原语的概念并实现进程控制原语do_fork(), do_exit(), do_wait(), wakeup, blocked。

二． 实验要求

1、 在实验五或更后的原型基础上，进化你的原型操作系统，原型保留原有特征的基础上，设计满足下列要求的新原型操作系统:

(1)实现控制的基本原语do_fork()、do_wait()、do_exit()、blocked()和wakeup()。

(2)内核实现三系统调用fork()、wait()和exit()，并在c库中封装相关的系统调用。

(3)编写一个c语言程序，实现多进程合作的应用程序。

2、 进程合作的应用程序可以在下面的基础上完成:由父进程生成一个字符串，交给子进程统计其中字母的个数，然后在父进程中输出这一统计结果。

3、 编译连接你编写的用户程序，产生一个com文件，放进程原型操作系统映像盘中。

三． 实验方案

1、 虚拟机配置

使用Vmware Workstation配置虚拟机，虚拟机的配置：核心数为1的处理器、4MB的内存、10MB的磁盘、1.44MB的软盘。

2、 软件工具与作用

Notepad++：编写程序时使用的编辑器；

16位编辑器WinHex：可以以16进制的方式打开并编辑任意文件；

TAMS汇编工具：可以将汇编代码编译成对应的二进制代码；

NAMS汇编工具：可以将汇编代码编译成对应的二进制代码；

TCC编译器：可以将c代码编译成对应的二进制代码；

TLINK链接器：将多个.obj文件链接成.com文件；

Bochs：可调试操作系统

WinImage：可以创建虚拟软盘。

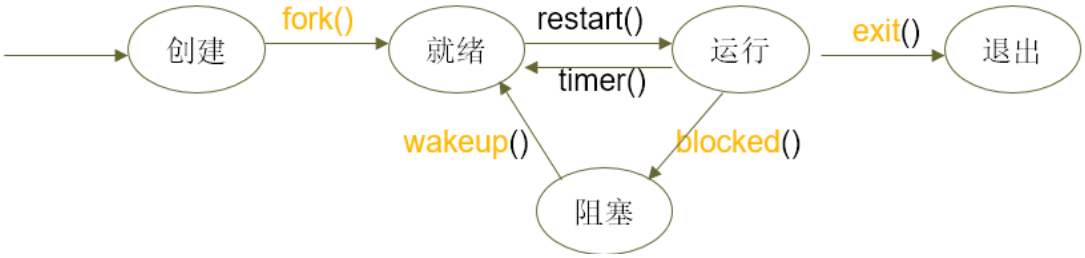
3、 基础原理

（1） 五状态进程模型

前一个原型中，操作系统可以用固定数量的进程解决多道程序技术。但是，这些进程模型还比较简单，进程之间互不往来，没有直接的合作。在实际的应用开发中,如果可以建立一组合作进程,并发运行,那么软件工程意义上更有优势。

在这个项目中，我们完善进程模型，进程能够按需要产生子进程，一组进程分工合作，并发运行，各自完成一定的工作。合作进程在并发时，需要协调一些事件的时序，实现简单的同步，确保进程并发的情况正确完成使命。

五状态进程模型如图所示：



（2） 进程控制的基本操作

在这个项目中，我们完善进程模型

a) 扩展PCB结构，增加必要的数据项

b) 进程创建do_fork()原语，在c语言中用fork()调用

c) 进程终止do_exit()原语，在c语言中用exit(int exit_value)调用

d) 进程等待子进程结束do_wait()原语，在c语言中用wait(&exit_value)调用

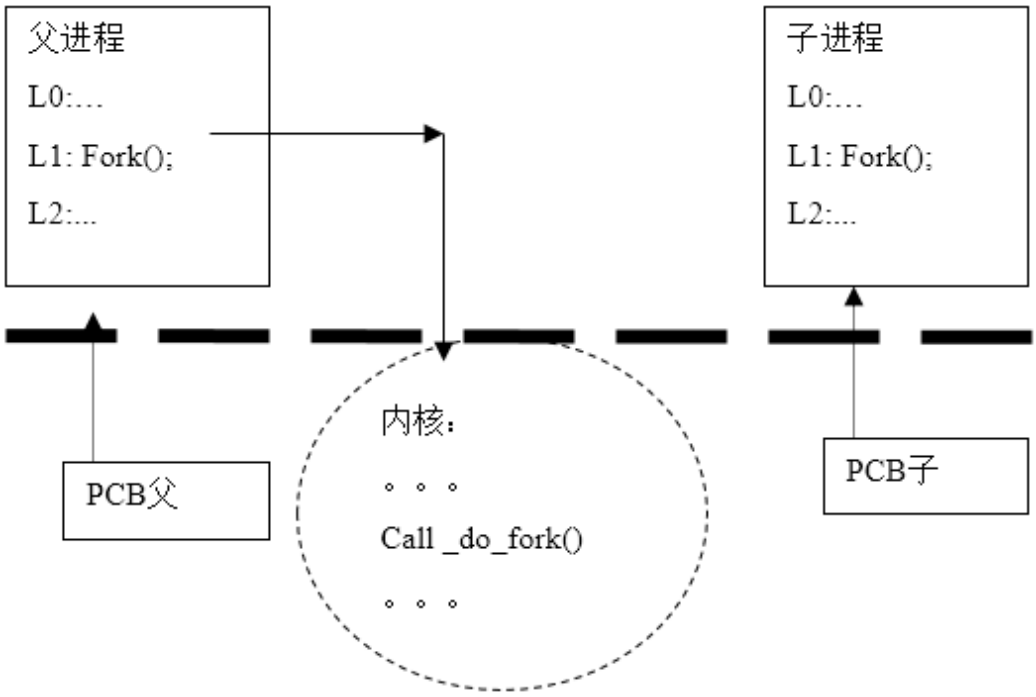
e) 进程唤醒wakeup原语（内核过程）

f) 进程唤醒blocked原语（内核过程）

（3） 进程创建

进程的派生过程do_fork()函数所执行的操作：先在PCB表查找空闲的表项，若没有找到，则派生失败。若找到空闲的PCB表项，则将父进程的PCB表复制给子进程，并将父进程的堆栈内容复制给子进程的堆栈，然后为子进程分配一个PCBID，共享代码段和数据段，并且父子进程中对do_fork()函数的返值分别设置。内核完成进程创建后，系统中增加了一个进程。

do_fork()功能的示意图：



（4） do_fork()的功能描述

原理中讲到，进程创建主要工作是完成一个进程映像的构造。

参考unix早期的fork()做法，我们实现的进程创建功能中，父子进程共享代码段和全局数据段。子进程的执行点(CS:IP)从父进程中继承过来，复制而得。创建成功后，父子进程以后执行轨迹取决于各自的条件和机遇，即程序代码、内核的调度过程和同步要求。

系统调用的返回值获得方式：C语言用ax传递，所以放在进程PCB的ax寄存器中。

fork()调用功能如下：

a) 寻找一个自由的PCB块，如果没有，创建失败，调用返回－1。

b) 以调用fork()的当前进程为父进程，复制父进程的PCB内容到自由PCB中。

c) 产生一个唯一的ID作为子进程的ID，存入至PCB的相应项中。

d) 为子进程分配新栈区，从父进程的栈区中复制整个栈的内容到子进程的

栈区中。

e) 调整子进程的栈段和栈指针，子进程的父亲指针指向父进程。

即pcb_list[s].fPCB= pcb_list[CurrentPCBno];

f) 在父进程的调用返回ax中送子进程的ID，子进程调用返回ax送0。

（5） do_fork()

创建进程功能要指定一个系统调用号，在内核的系统调用总控程序中，增加一个分支，调用do_fork() 完成子进程创建。

（6） wait()

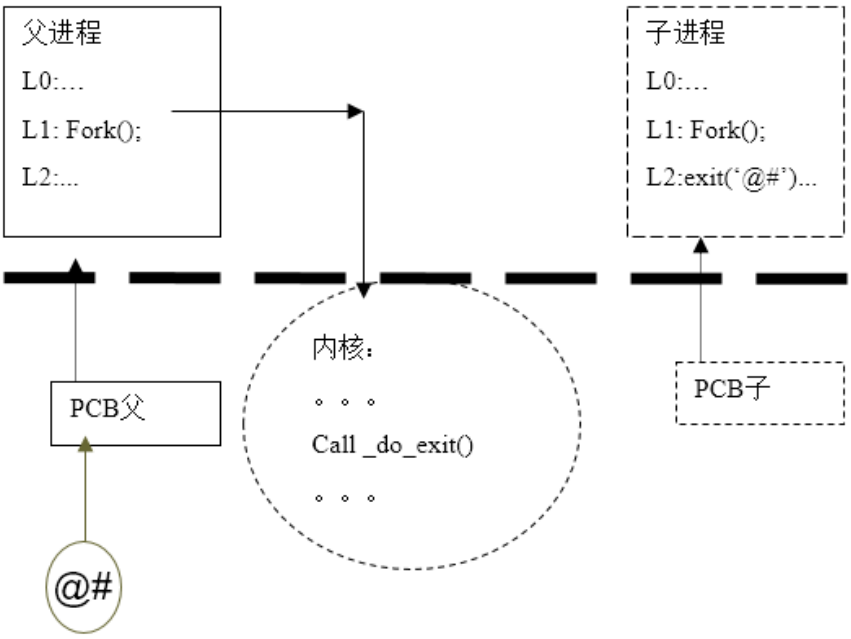
父进程如果想等待子进程结束后再处理子进程的后事，需要一个系统调用实现同步。我们模仿UNIX的做法，设置wait()实现这一功能。相应地，内核的进程应该增加一种阻塞状态。当进程调用wait()系统调用时，内核将当前进程阻塞，并调用进程调度过程挑选另一个就绪进程接权。相应调度模块也要修改，禁止将CPU交权给阻塞状态的进程。

（7） exit()

父进程如果想等待子进程结束后再处理子进程的后事，需要一个系统调用wait()，进程被阻塞。而子进程终止时，调用exit()，向父进程报告这一事件，可以传递一个字节的信息给父进程，并解除父进程的阻塞，并调用进程调度过程挑选另一个就绪进程接权。

（8） do_exit()

进程结束，内核清除发出调用的进程，收回进程占有的所有资源，系统中减少了一个进程，并将遗言传递给其父进程，如图所示。



4、 方案思想

在实验六中有一个问题没有解决,那就是多进程并发执行后有一定概率不能成功返回内核，也就是说内核不能响应用户输入的指令。而在本次实验中成功解决该问题，通过将内核作为一个独立的进程加入进程表，使用时钟中断和调度程序成功返回内核。

首先，在PCB.h文件中，在PCB结构中加入FID用于指明父进程。由于这次实验实现五状态进程模型，所以调度函数schedule()需要修改。在PCB_initial(PCB *ptr, int process_ID, int seg)函数中将FID默认初始化为0,然后分别实现控制原语do_fork()、do_wait()、do_exit(int ss)、wakeup()和bloked()，接着实现创建子进程函数createSubPCB()。

接着，在kliba.asm文件中，修改run_process()函数以加载多个扇区的测试程序，修改时钟中断程序Timer，实现超时返回内核。编写PCB_Restore()函数以实现进程间的切换，然后编写stackCopy()函数，实现将父进程的堆栈复制到子进程。修改21h中断程序，添加4号forking、5号waiting、6号exiting三

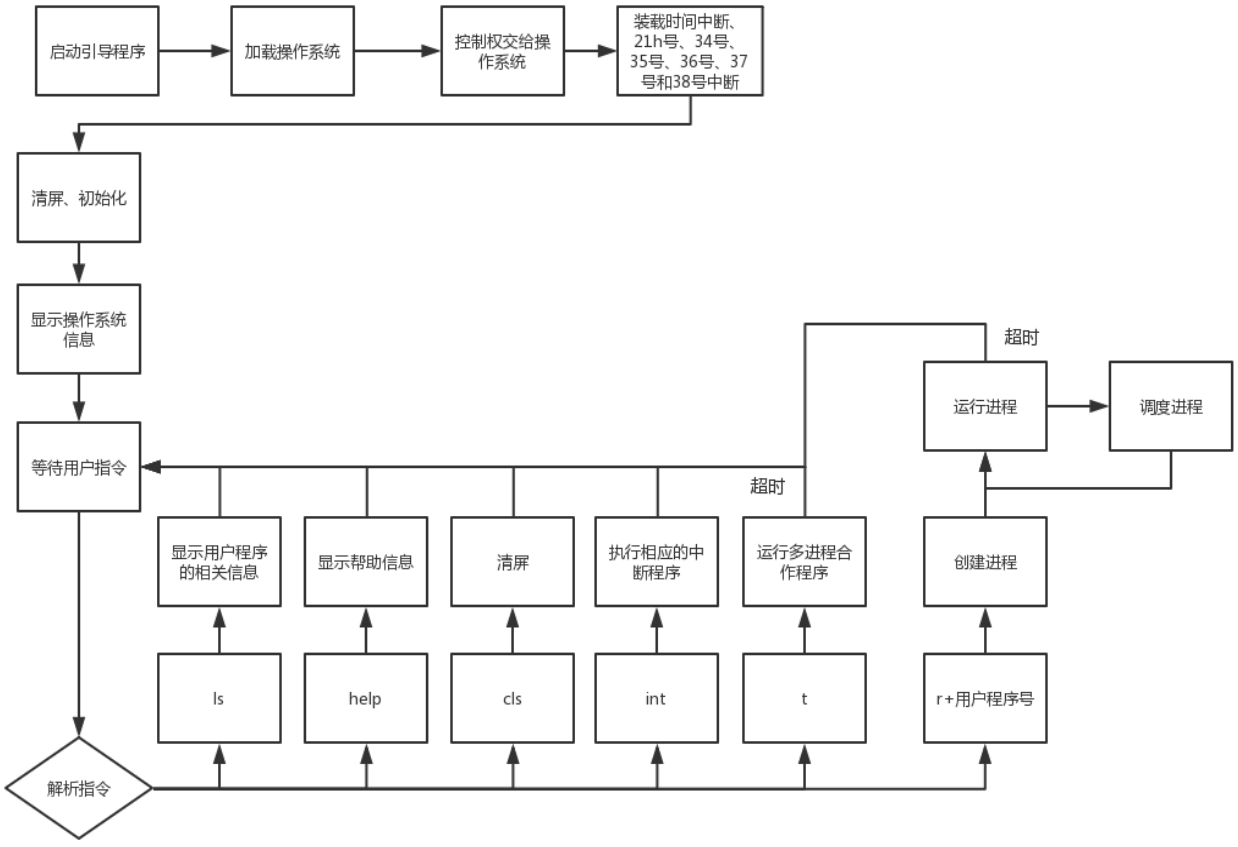
个系统调用服务，然后分别实现forking、waiting、exiting三个服务。

然后，在kernal.c文件中，由于测试程序是多进程合作程序，所以需要修改run_test()函数，以成功加载多进程合作程序。修改create_process(char *comm)函数，将内核作为独立的进程加入进程表。

在实验四&五实现的C库的基础上,添加fork()、wait()、exit()三个函数，这三个函数编写于use_lib.asm文件中，fork()和wait()函数的实现较为简单，只需封装系统调用服务即可，exit()稍微复杂一点，传一个参数后再调用系统服务即可。

最后,在user.asm和userc.c文件中编写多进程合作程序,我在老师给的案例的基础上增加了几个输出以便更好的展示父进程和子进程的工作过程。

5、 程序流程



6、 算法与数据结构

（1）算法：

调度算法：

首先进行判断，如果当前进程为运行态，就将其转为就绪态。接着遍历整个进程表，如果所有的进程均为阻塞态或是退出态，则将控制权转交给内核。如何存在一个进程为新建态或就绪态，则执行该进程。

fork：

执行 fork 时，将调用 21h 的 4 号功能。在中断里面，将首先对当前进程的进程控制信息进行保存，即执行一个 save_PCB 的操作，以获得最新的进程控制块。然后，调用 do_fork 函数，首先，创建一个新的子进程控制块，该子进程的进程控制块复制了父进程的内容，但是 ss 为新的栈底，ax 为 0，如果创建不成功，将返回 -1，然后，修改父进程 ax 为子进程 ID ，接着将父进程的堆栈完全拷贝给子进程，最后重新启动进程，对寄存器进程更新，即进行 restart 操作。

wait：

执行 wait 时，将调用 21h 的 5 号功能。在中断里面，将首先对当前进程的进程控制信息进行保存，即执行一个 save_PCB 的操作，以获得最新的进程控制块。然后，调用 do_wait 函数，首先,将当前进程的运行状态改为 PCB_BLOCKED，然后，调用 schedule 函数，寻找一个处于就绪态的进程，最后进行 restart，执行新的进程。

exit：

执行 exit 时，将调用 21h 的 6 号功能。在中断里面，将首先对当前进

程的进程控制信息进行保存，即执行一个 save_PCB 的操作，以获得最新的进程控制块。然后，调用 do_exit 函数，首先，将当前进程的运行状态改为 PCB_EXIT，AX 改为 exit中传递过来的参数，父进程的运行状态改为 PCB_READY，然后，调用 schedule 函数，寻找当前进程退出后下一个可以执行的进程，最后进行 restart 操作，执行新的进程。

7、 程序关键模块

（1）在PCB.h文件中：

a）修改PCB结构，代码如下：

```
typedef struct PCB{
    Register regs;
    PCB_STATUS status;
    int ID;
    int FID;
} PCB;
```

b）修改调度函数，代码如下：

```
void schedule() {
    int i, flag = 1;
    if (current_process_PCB_ptr->status == PCB_RUNNING)
        current_process_PCB_ptr->status = PCB_READY;
    for (i = 1; i < process_number; ++i) {
        if (PCB_LIST[i].status != PCB_BLOCKED && PCB_LIST[i].status != PCB_EXIT) {    flag = 0;
            break;} }
    if (flag) {
        current_process_number = 0;
        current_process_PCB_ptr = get_current_process_PCB();
        current_process_PCB_ptr->status = PCB_RUNNING;
        kernal_mode = 1;
        return; }
    do {
        current_process_number++;
        if (current_process_number >= process_number)
            current_process_number = 1;
    } while (PCB_LIST[current_process_number].status == PCB_BLOCKED || PCB_LIST[current_process_number].status == PCB_EXIT);
```

```
    current_process_PCB_ptr = get_current_process_PCB();
    if (current_process_PCB_ptr->status == PCB_NEW)
        first_time = 1;
    current_process_PCB_ptr->status = PCB_RUNNING;
    return;}
```

c) 实现do_fork()函数，代码如下：

```
int do_fork() {
    int sub_ID;
    print("kernal: forking\r\n");
    sub_ID = createSubPCB();
    if (sub_ID == -1) {
        current_process_PCB_ptr->regs.ax = -1;
        return -1;}
    sub_PCB = &PCB_LIST[sub_ID];
    current_process_PCB_ptr->regs.ax = sub_ID;
    sub_ss = sub_PCB->regs.ss;
    f_ss = current_process_PCB_ptr->regs.ss;
    stack_size = 0x100;
    stackCopy();
    PCB_Restore();}
```

d) 实现createSubPCB()函数，代码如下：

```
int createSubPCB() {
    if (process_number > MAX_PCB_NUMBER) return -1;
    t_PCB = &PCB_LIST[process_number];
    t_PCB->ID = process_number;
    t_PCB->status = PCB_READY;
    t_PCB->FID = current_process_number;
    t_PCB->regs.gs = 0xb800;
    t_PCB->regs.es = current_process_PCB_ptr->regs.es;
    t_PCB->regs.ds = current_process_PCB_ptr->regs.ds;
    t_PCB->regs.fs = current_process_PCB_ptr->regs.fs;
    t_PCB->regs.ss = current_seg;
    t_PCB->regs.di = current_process_PCB_ptr->regs.di;
    t_PCB->regs.si = current_process_PCB_ptr->regs.si;
    t_PCB->regs.bp = current_process_PCB_ptr->regs.bp;
    t_PCB->regs.sp = current_process_PCB_ptr->regs.sp;
    t_PCB->regs.ax = 0;
    t_PCB->regs.bx = current_process_PCB_ptr->regs.bx;
    t_PCB->regs.cx = current_process_PCB_ptr->regs.cx;
    t_PCB->regs.dx = current_process_PCB_ptr->regs.dx;
    t_PCB->regs.ip = current_process_PCB_ptr->regs.ip;
    t_PCB->regs.cs = current_process_PCB_ptr->regs.cs;
```

```
    t_PCB->regs.flags = current_process_PCB_ptr->regs.flags;
    process_number++;
    current_seg += 0x1000;
    print("kernal: sub process created!\r\n");
    return process_number - 1;}
```

e) 实现do_wait()函数，代码如下：

```
void do_wait() {
    print("kernal: waiting...\r\n");
    blocked();}
```

f) 实现do_exit()函数，代码如下：

```
void do_exit(int ss) {
    print("kernal: exiting\r\n");
    PCB_LIST[current_process_number].status = PCB_EXIT;
    PCB_LIST[current_process_PCB_ptr->FID].status = PCB_READY;
    PCB_LIST[current_process_PCB_ptr->FID].regs.ax = ss;
    current_seg -= 0x1000;
    process_number--;
    wakeup();}
```

g) 实现wakeup()函数，代码如下：

```
void wakeup() {
    if (process_number == 1)
        kernal_mode = 1;
    schedule();
    PCB_Restore();}
```

h) 实现blocked()函数，代码如下：

```
void blocked() {
    current_process_PCB_ptr->status = PCB_BLOCKED;
    schedule();
    PCB_Restore();}
```

（2）在kliba.asm文件中

a) 修改_run_process()函数，代码如下：

```
    public _run_process
    _run_process proc
        push es
```

```
    mov ax, word ptr [_current_seg]
    mov es, ax
    mov bx, 100h
    mov ah, 2
    mov al, byte ptr [_sector_size]
    mov dl, 0
    mov dh, 0
    mov ch, 0
    mov cl, byte ptr [_sector_number]
    int 13h
    call _create_new_PCB
    pop es
    ret
_run_process endp
```

b) 修改时钟中断程序

```
Timer:
    cmp word ptr [_kernal_mode], 1
    jne process_timer
    jmp kernal_timer
process_timer:
    .386
    push ss
    push gs
    push fs
    push es
    push ds
    .8086
    push di
    push si
    push bp
    push sp
    push dx
    push cx
    push bx
    push ax
    cmp word ptr [back_time], 800
    jnz time_to_go
    mov word ptr [back_time], 1
    mov word ptr [_current_process_number], 0
    mov word ptr [_kernal_mode], 1
    mov ax, 600h
    mov bx, 700h
    mov cx, 0
```

```
        mov dx, 184fh
        int 10h
        call _initial_PCB_settings
        call _PCB_Restore
    time_to_go:
        inc word ptr [back_time]
        mov ax, cs
        mov ds, ax
        mov es, ax
        call _save_PCB
        call _schedule
        call _PCB_Restore
        iret
```

c) 实现_PCB_Restore()函数，代码如下：

```
public _PCB_Restore
_PCB_Restore proc
    mov ax, cs
    mov ds, ax
    call _get_current_process_PCB
    mov si, ax
    mov ss, word ptr ds:[si]
    mov sp, word ptr ds:[si+2*7]
    cmp word ptr [_first_time], 1
    jnz next_time
    mov word ptr [_first_time], 0
    jmp start_PCB
next_time:
    add sp, 11*2
start_PCB:
    push word ptr ds:[si+2*15]
    push word ptr ds:[si+2*14]
    push word ptr ds:[si+2*13]
    mov ax, word ptr ds:[si+2*12]
    mov cx, word ptr ds:[si+2*11]
    mov dx, word ptr ds:[si+2*10]
    mov bx, word ptr ds:[si+2*9]
    mov bp, word ptr ds:[si+2*8]
    mov di, word ptr ds:[si+2*5]
    mov es, word ptr ds:[si+2*3]
    .386
    mov fs, word ptr ds:[si+2*2]
    mov gs, word ptr ds:[si+2*1]
    .8086
```

```
    push word ptr ds:[si+2*4]
    push word ptr ds:[si+2*6]
    pop si
    pop ds
process_timer_end:
    push ax
    mov al, 20h
    out 20h, al
    out 0A0h, al
    pop ax
    iret
endp _PCB_Restore
```

d) 实现_stackCopy()函数，代码如下：

```
    public _stackCopy
    _stackCopy proc
        push ax
        push es
        push ds
        push di
        push si
        push cx
        mov ax, word ptr [_sub_ss]
        mov es,ax
        mov di, 0
        mov ax, word ptr [_f_ss]
        mov ds, ax
        mov si, 0
        mov cx, word ptr [_stack_size]
        cld
        rep movsw
        pop cx
        pop si
        pop di
        pop ds
        pop es
        pop ax
        ret
    _stackCopy endp
```

e) 修改21h中断系统调用服务程序，部分代码如下：

```
    int_21h:
        push bp
        push ds
```

```
    push es
    mov bx, cs
    mov ds, bx
    mov es, bx
    cmp ah, 1
    je showstring1
    cmp ah, 2
    je showstring2
    cmp ah, 3
    je showstring3
    cmp ah, 4
    je to_forking
    cmp ah, 5
    je to_waiting
    cmp ah, 6
    je to_exiting
    jmp end21h
to_forking:
    pop es
    pop ds
    pop bp
    jmp forking
to_waiting:
    pop es
    pop ds
    pop bp
    jmp waiting
to_exiting:
    pop es
    pop ds
    pop bp
    jmp exiting
```

f) 实现21h中断4号功能forking，代码如下：

```
forking:
    .386
    push ss
    push gs
    push fs
    push es
    push ds
    .8086
    push di
    push si
```

```
    push bp
    push sp
    push dx
    push cx
    push bx
    push ax
    mov ax,cs
    mov ds, ax
    mov es, ax
    call _save_PCB
    call near ptr _do_fork
    iret
```

g) 实现21h中断5号功能waiting，代码如下：

```
waiting:
    .386
    push ss
    push gs
    push fs
    push es
    push ds
    .8086
    push di
    push si
    push bp
    push sp
    push dx
    push cx
    push bx
    push ax
    mov ax,cs
    mov ds, ax
    mov es, ax
    call _save_PCB
    call near ptr _do_wait
    iret
```

h) 实现21h中断6号功能exiting，代码如下：

```
    exiting:
        .386
        push ss
        push gs
```

```
push fs
push es
push ds
.8086
push di
push si
push bp
push sp
push dx
push cx
push bx
push ax
mov ax,cs
mov ds, ax
mov es, ax
call _save_PCB
call near ptr _do_exit
iret
```

（3）在kernal.c文件中

a) 修改create_process(char *comm)函数，代码如下：

```
    void create_process(char *comm) {
        int i, sum = 0, flag = 0;
        for (i = 1; i < strlen(comm); ++i) {
            if (comm[i] == ' ' || comm[i] >= '1' && comm[i] <= '4')
continue;
            else {
                print("invalid program number: ");
                printChar(comm[i]);
                print("\n\n\r");
                return;}}
        for (i = 1; i < strlen(comm); ++i)
            if (comm[i] != ' ') flag = 1;
        if (flag == 0) {
```

```
            print("invalid input\n\n\r");
            return;}
        for (i = 1; i < strlen(comm) && sum < MAX_PCB_NUMBER; ++i) {
            if (comm[i] == ' ') continue;
            sum++;
            sector_number = comm[i] - '0' + 10;
            sector_size = 1;
            run_process();}
        PCB_initial(&PCB_LIST[0], 1, 0x1000);
        kernal_mode = 0;}
```

b) 修改run_test()函数，代码如下：

```
void run_test() {
    sector_number = 15;
    sector_size = 2;
    run_process();
    kernal_mode = 0;}
```

（4）在user_lib.asm文件中

a) 封装fork()函数，代码如下：

```
public _fork
_fork proc
    mov ah,4
    int 21h
    ret
_fork endp
```

b) 封装wait()函数，代码如下

```
public _wait
_wait proc
    mov ah,5
    int 21h
    ret
_wait endp
```

c) 封装exit()函数，代码如下：

```
public _exit
_exit proc
    push bp
    mov bp,sp
    push bx
    mov ah,6
    mov bx,[bp+4]
```

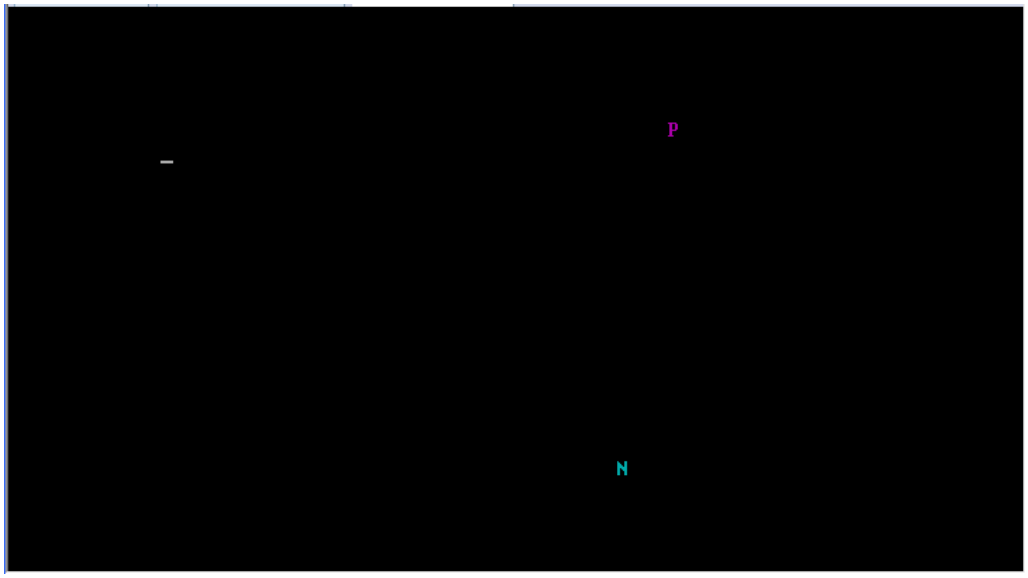
```
        int 21h
        pop bx
        pop bp
        ret
    _exit endp
```

（5）在userc.c文件中，实现多进程合作程序，代码如下：

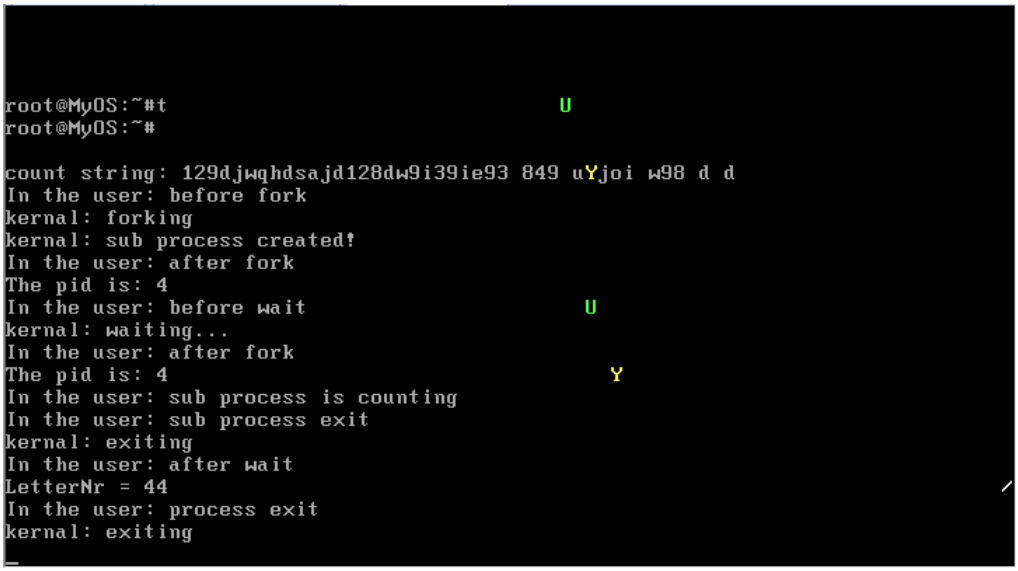
```
extern int fork();
extern int wait();
extern void exit();
#include "user_lib.h"
char str[80] = "129djwqhdsajd128dw9i39ie93i8494urjoiew98kdkd";
int LetterNr = 0;
void main() {
    int pid;
    print("\r\n\r\ncount string: ");
    print(str);
    print("\r\n");
    print("In the user: before fork\r\n");
    pid = fork();
    print("In the user: after fork\r\n");
    print("The pid is: ");
    printInt(pid);
    print("\r\n");
    if(pid == -1) {
        print("error in fork!\r\n");
        exit(-1); }
    if(pid) {
        print("In the user: before wait\r\n");
        wait();
        print("In the user: after wait\r\n");
        print("LetterNr = ");
        printInt(LetterNr);
        print("\r\n");
        print("In the user: process exit\r\n");
        exit(0);}
    else {
        print("In the user: sub process is counting\r\n");
        LetterNr = strlen(str);
        print("In the user: sub process exit\r\n");
        exit(0);}}
```

四． 实验过程和结果

1. 进入操作系统后输入指令r 24，程序2、4并发执行，一段时间后返回内核，从右下角动态的‘-’可观察到，如图所示。



2. 返回后按下回车即可再次输入指令，输入指令t，运行多进程合作程序。由于步骤1的程序并未退出，此时共有四个进程在运行，分别是程序2、4和测试程序t及其子进程，如图所示。



3. 运行一段时间后返回内核，从右下角动态的‘-’可观察到。再次按下回车即可输入指令。

五． 实验总结

Ⅰ．总结:

实验六中的历史遗留问题终于得到解决，我感到十分的开心。实验六中有一定几率会出现返回内核后用户按键没有任何响应的问题，为解决这个问题，我将内核作为一个独立的进程装入进程表，通过时钟中断超时返回或是通过调度函数返回内核。

在本次实验中，我再一次意识到正确地通过栈传参的重要性。这次实验我遇到过多压三个参数导致从中断返回时，程序并没有返回到调用中断的地点，而是跳到一个无效的地点，使整个系统崩溃了。

对于这个问题，我觉得有两个有效的解决办法。第一个方法就是使用全局变量传参，调用者只需将这些变量置为相应的参数即可，这一做法在参数很少的情况可有效使用，但当传递的参数较多时，需要设置很多的全局变量，这样的就很麻烦。第二个方法就是在编写代码的时候谨慎谨慎再谨慎，压栈和出栈时需要十分细致的考虑，这一方法我在传递进程上下文参数时使用，毕竟save_PCB函数需要较多的参数。

Ⅱ．遇到的问题及解决方法:

（1）测试多进程合作程序时，父进程成功创建一个子进程后，整个程序就进入死循环，如图1所示。我使用bochs调试后，发现程序跳转到一个无效的地址，如图2所示。于是，我就在forking()函数设置一个断点，以便查看寄存器值，压栈传参前查看堆栈如图3所示。将所有寄存器压栈传给save_PCB()函数后，将这些寄存器值全部保存到进程控制块，再调用do_fork()函数创建子进程，在do_fork()函数中，程序执行完stackCopy()函数复制堆栈后没有出现问题，进

入PCB_Restore()函数切换进程后出现问题了。查看栈如图4所示，通过iret跳转至子进程，CS:IP为2000:2000，如图5所示，该地点是一个无效地址。所以问题的关键所在是调用forking()函数前栈是有问题的，导致切换进程时访问到无效的地址。查看我编写的代码时，我发现调用forking()函数前进行了三次压栈操作，栈出现问题，最终导致这个问题的出现。进行相关修改后，调用forking()函数前正确的栈如图6所示。

图1

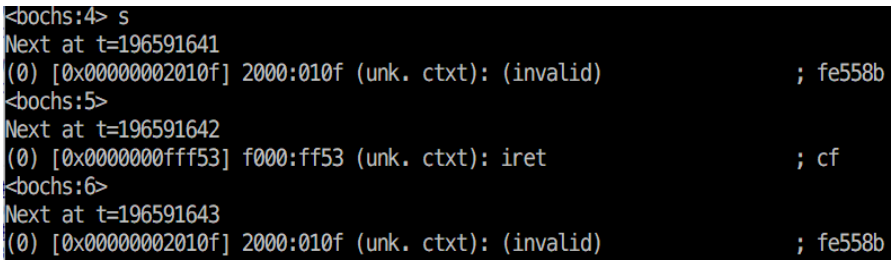


图2

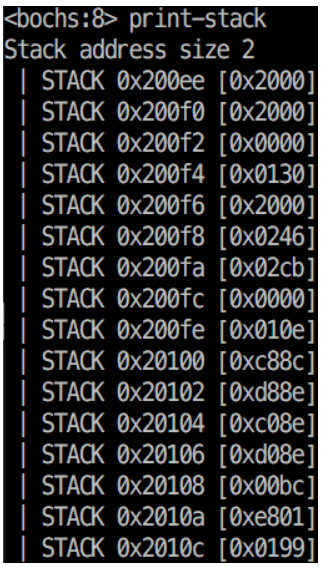


图3

```
Next at t=203254401
(0) [0x0000000082b9] 0800:02b9 (unk. ctxt): iret                    ; cf
<bochs:41> print-stack
Stack address size 2
 | STACK 0x200ee [0x2000]
 | STACK 0x200f0 [0x2000]
 | STACK 0x200f2 [0x0000]
 | STACK 0x200f4 [0x0130]
 | STACK 0x200f6 [0x2000]
 | STACK 0x200f8 [0x0246]
 | STACK 0x200fa [0x02cb]
 | STACK 0x200fc [0x0000]
 | STACK 0x200fe [0x010e]
 | STACK 0x20100 [0xc88c]
 | STACK 0x20102 [0xd88e]
 | STACK 0x20104 [0xc08e]
 | STACK 0x20106 [0xd08e]
 | STACK 0x20108 [0x00bc]
 | STACK 0x2010a [0xe801]
 | STACK 0x2010c [0x0199]
```

图4

```
<bochs:42> s
Next at t=203254402
(0) [0x000000022000] 2000:2000 (unk. ctxt): add byte ptr ds:[bx+si], al ; 0000
<bochs:43> s
Next at t=203254403
(0) [0x000000022002] 2000:2002 (unk. ctxt): add byte ptr ds:[bx+si], al ; 0000
```

图5

```
<bochs:4> print-stack
Stack address size 2
 | STACK 0x200f4 [0x0130]
 | STACK 0x200f6 [0x2000]
 | STACK 0x200f8 [0x0246]
 | STACK 0x200fa [0x02cb]
 | STACK 0x200fc [0x0000]
 | STACK 0x200fe [0x010e]
 | STACK 0x20100 [0xc88c]
 | STACK 0x20102 [0xd88e]
 | STACK 0x20104 [0xc08e]
 | STACK 0x20106 [0xd08e]
 | STACK 0x20108 [0x00bc]
 | STACK 0x2010a [0xe801]
 | STACK 0x2010c [0x0199]
 | STACK 0x2010e [0xfeeb]
 | STACK 0x20110 [0x8b55]
 | STACK 0x20112 [0x56ec]
```

图6

（2）解决上述的问题后，多进程合作程序并不能成功运行，该进程还是处于死循环状态，如图7所示。使用bochs调试如图8所示，发现程序在0800:084a到0800:087d间循环，经过推算发现，这一段是调度程序schedule()中

```
do {
      current_process_number++;
      if (current_process_number >= process_number)
          current_process_number = 1;
   } while (PCB_LIST[current_process_number].status == PCB_BLOCKED ||
PCB_LIST[current_process_number].status == PCB_EXIT);
```

这段C代码的汇编代码。经过思考后，我发现之前的调度程序代码存在逻辑漏洞，那就是当所有的进程均为退出态或是阻塞态时，这段代码将是一个死循环。于是，我在这段代码前加入一段判断，如果所有的进程均为退出态或是阻塞态，就执行内核进程，这样成功解决该问题。

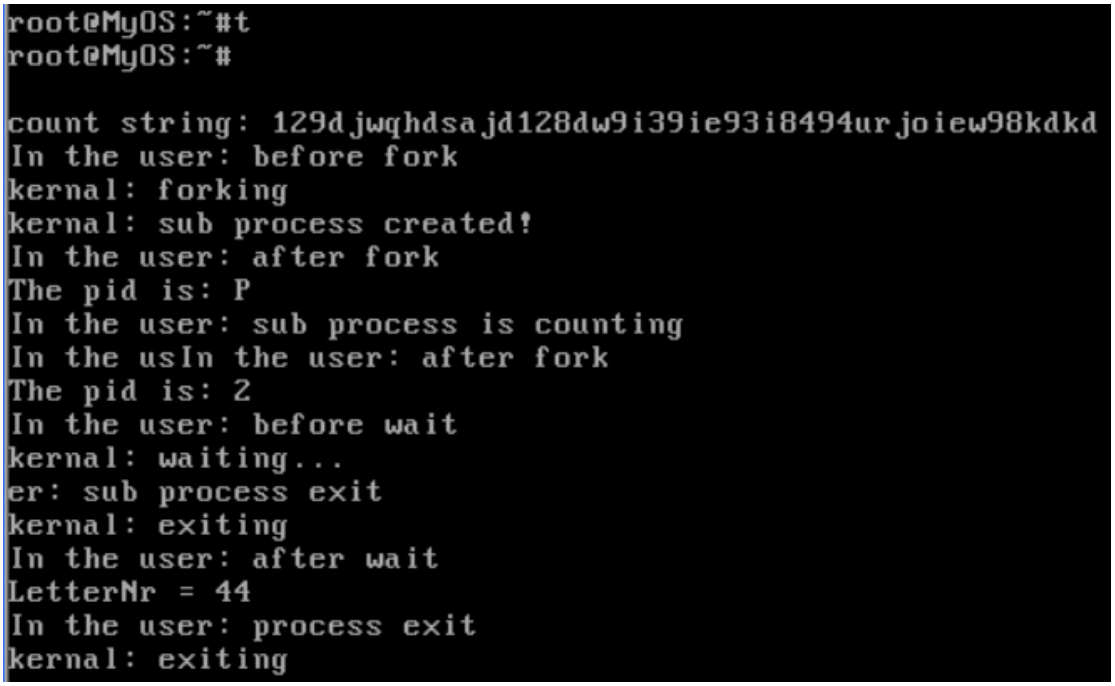


图7

```
(0) Breakpoint 1, 0x000000000000884a in ?? ()
Next at t=301247457
(0) [0x00000000884a] 0800:084a (unk. ctxt): inc word ptr ds:0x0e72    ; ff06720e
<bochs:60> u /20
0000884a: (                    ): inc word ptr ds:0x0e72    ; ff06720e
0000884e: (                    ): mov ax, word ptr ds:0x0e72 ; a1720e
00008851: (                    ): cmp ax, word ptr ds:0x0e6e ; 3b066e0e
00008855: (                    ): jl .+6                     ; 7c06
00008857: (                    ): mov word ptr ds:0x0e72, 0x0001 ; c706720e0100
0000885d: (                    ): mov ax, word ptr ds:0x0e72 ; a1720e
00008860: (                    ): mov dx, 0x0026             ; ba2600
00008863: (                    ): mul ax, dx                 ; f7e2
00008865: (                    ): mov bx, ax                 ; 8bd8
00008867: (                    ): cmp word ptr ds:[bx+4568], 0x0004 ; 83bfd81104
0000886c: (                    ): jz .-36                    ; 74dc
0000886e: (                    ): mov ax, word ptr ds:0x0e72 ; a1720e
00008871: (                    ): mov dx, 0x0026             ; ba2600
00008874: (                    ): mul ax, dx                 ; f7e2
00008876: (                    ): mov bx, ax                 ; 8bd8
00008878: (                    ): cmp word ptr ds:[bx+4568], 0x0002 ; 83bfd81102
0000887d: (                    ): jz .-53                    ; 74cb
0000887f: (                    ): call .-270                 ; e8f2fe
00008882: (                    ): mov word ptr ds:0x11b2, ax ; a3b211
00008885: (                    ): mov bx, word ptr ds:0x11b2 ; 8b1eb211
```

图8

六． 参考文献

1、《x86 PC汇编语言，设计与接口》

2、 Windows 下BOCHS的使用

http://blog.51cto.com/liyuelumia/1562508