**《操作系统实验》**

**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** | **：** | 杨志成 | | | | | |
| **学号** | **：** | 16337281 | | | | | |
| **时间** | **：** | 2018 | 年 | 5 | 月 | 24 | 日 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  | **成绩** | **:** |  |
| **实验七** | **：** | 五状态进程模型 | | | |

**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**

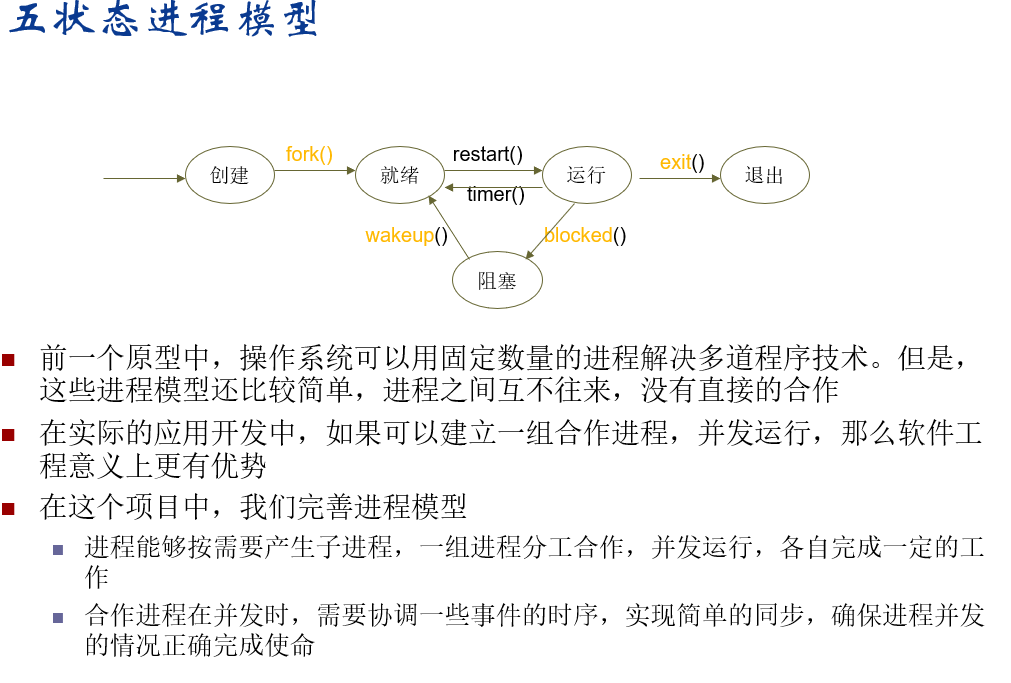
(1)实现控制的基本原语do\_fork()、 do\_wait()、do\_exit()、blocked()和wakeup()。

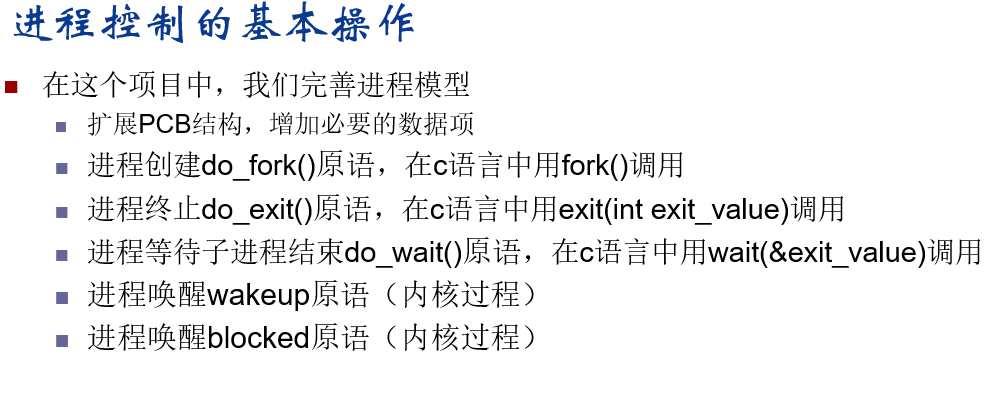
(2)内核实现三系统调用fork()、 wait()和exit() ，并在c库中封装相关的系统调用.

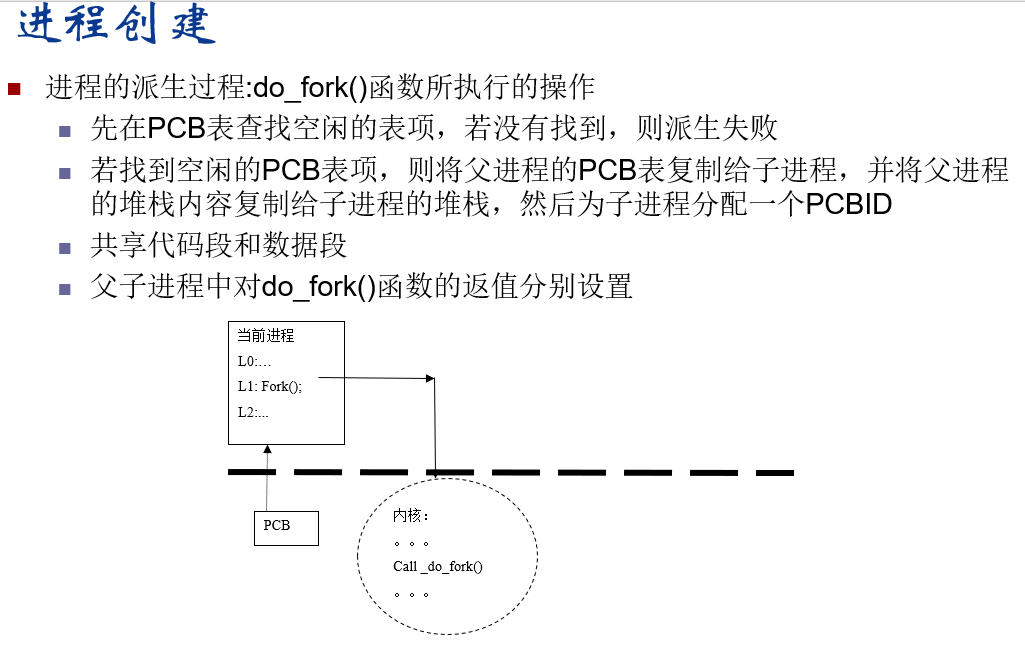
(3)编写一个c语言程序，实现多进程合作的应用程序。

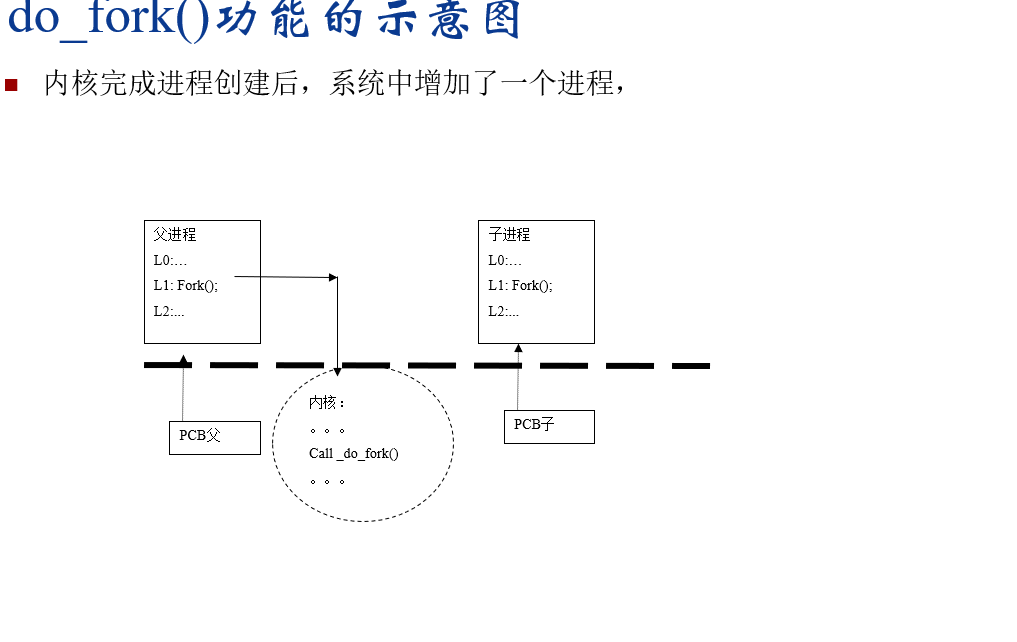
**三 实验方案**

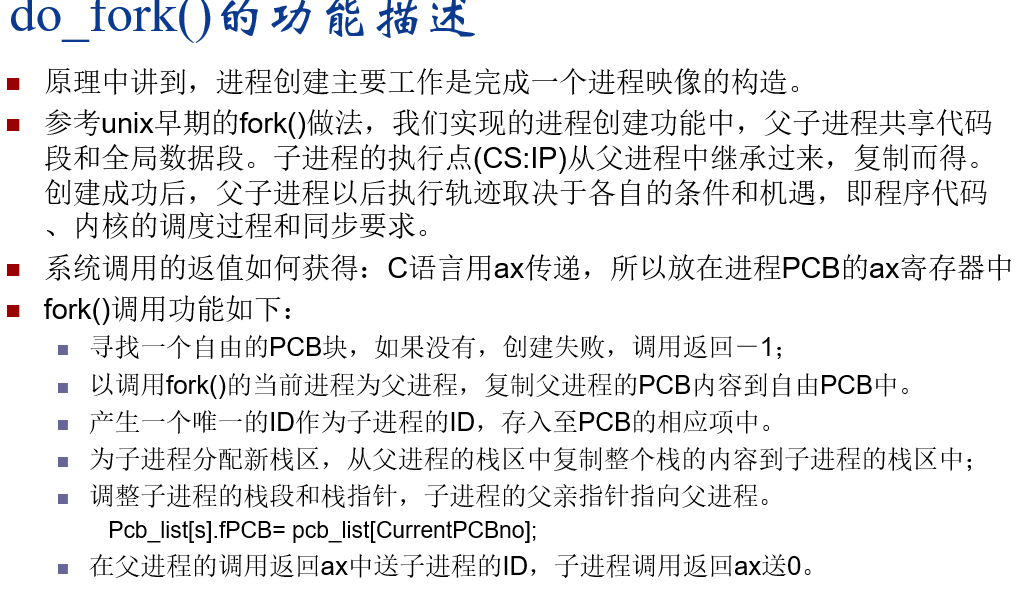
(1)基本原理

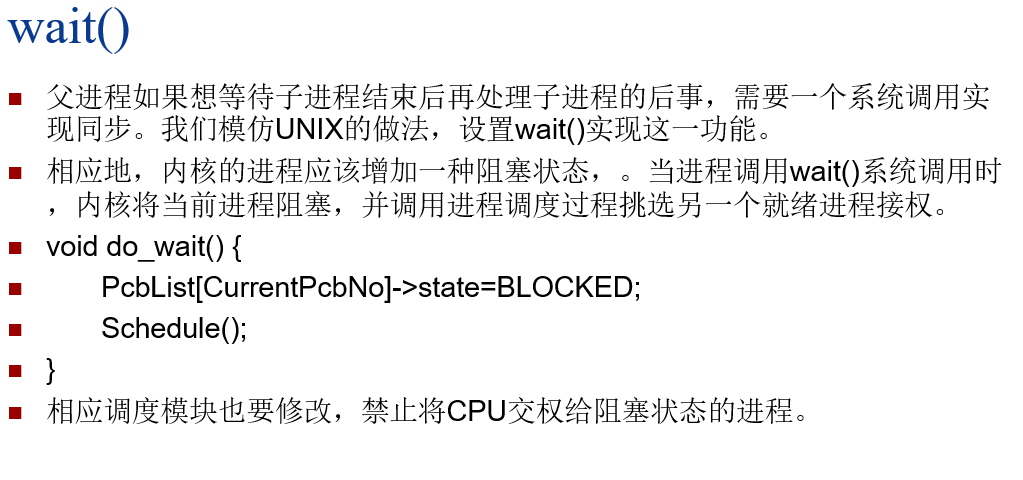


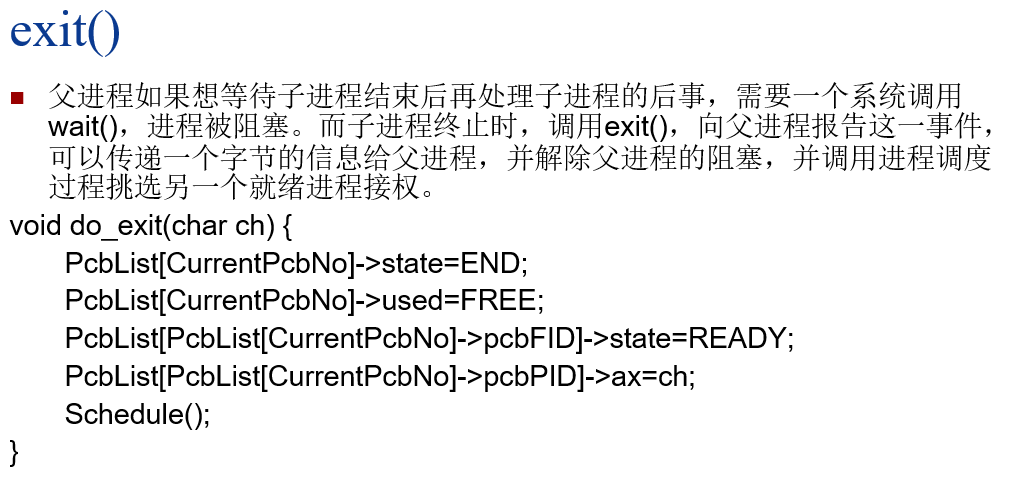












**2）实验工具环境**

实验支撑环境

硬件：个人计算机

主机操作系统：Windows/Linux/Mac OS/其它

虚拟机软件：VMware/VirtualPC/Bochs/其它

PC虚拟机裸机/DOS虚拟机/其它

实验开发工具

汇编语言工具：x86汇编语言

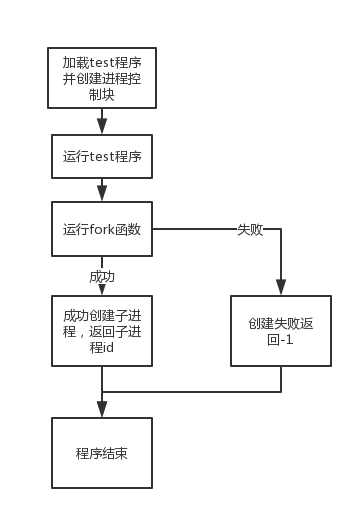
高级语言工具：标准c语言

磁盘映像文件浏览编辑工具

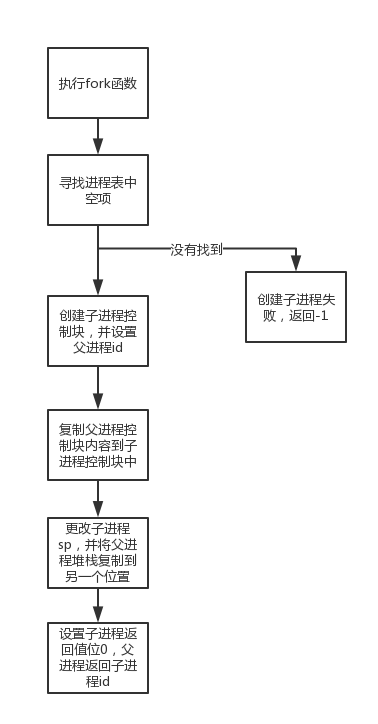
调试工具：Bochs

**（3）程序流程示意图：**

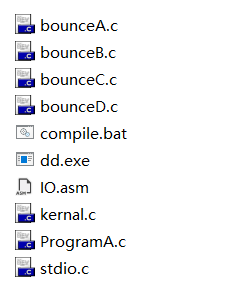
**总体程序流程图：**



**Fork函数执行流程：**



**（4）代码文档组成**

**、**

**--leader.asm 引导程序**

**--IO.asm 汇编语言库**

**--stdio.c c语言基本输入输出库**

**--bounceA/B/C/D.c 弹跳小球的用户程序**

**--ProgramA.c 老师提供的测试fork函数的程序**

**（5）代码模块**

新的schedule函数：  
**void** Schedule()

{

Cur\_num = CurrentProc->id;

    if(CurrentProc->status == Exit)

        CurrentProc->status = Empty;        *//exit进程*

    else if(CurrentProc->status == Run)

        CurrentProc->status = Ready;        *//正常进程切换*

    while(1){

        Cur\_num = (Cur\_num + 1) % 5;

        if(PCBlist[Cur\_num].status == Ready){

            CurrentProc = &PCBlist[Cur\_num];

            CurrentProc->status = Run;

            break;

        }

    }

}

因为是五进程模型，所以新的进程调度模块需要做相应的调整，我们这里的策略是将当前进程状态位status先变为Ready，然后在整个进程控制表中寻找第一个状态为Ready的进程，然后将CurrentProc更新为指向当前进程，并将状态位置为Run，注意同一个时刻，进程控制表中，只有一个进程的状态位Run

**do\_fork()函数**

**void** do\_fork(){

**struct** pcb \* son\_p = CurrentProc ;

**int** son\_id = 0;

    for(son\_id = 1 ; son\_id < 5 ; son\_id++) if(PCBlist[son\_id].status == Empty) break;

if(son\_id == 5) CurrentProc->ax = -1;*//进程已满，创建子进程失败，返回-1；*

else {

printstr("father process id:");printint(CurrentProc->id);CR();

printstr("son process id:");printint(son\_id);CR();

son\_p = &PCBlist[son\_id]; *// p 指向 子进程PCB*

memcopy(CurrentProc, son\_p);

son\_p->f\_id = CurrentProc->id;

CurrentProc->ax = son\_id; *//设置父进程fork返回值*

son\_p->status = Ready;

son\_p->ax = 0; *//设置子进程fork返回值*

son\_p->id = son\_id;

**int** key = son\_p->cs >> 12; *// cs段寄存器的值前4位 如0x1000*

son\_p->sp = EMPTY\_TABLE[key];

EMPTY\_TABLE[key] -= 0x1000;

STACK\_TABLE[son\_id] = son\_p->sp;

son\_p->sp = stackcopy(CurrentProc->sp , STACK\_TABLE[CurrentProc->id] , son\_p->sp);

return;

}

}

该函数可以说是本次实验的核心内容，它的运行过程我们已经在之前的程序流程图中说明过了。注意stackcopy函数是我在汇编语言中所实现的内容，它的主要功能是将父进程中的堆栈复制到子进程的堆栈中。

**do\_exit（）函数**

**void** do\_exit(**int** s){

    CurrentProc->status = Exit;*//结束当前进程*

    if(CurrentProc->f\_id != -1 && PCBlist[CurrentProc->f\_id].status == Block){

        wakeup(CurrentProc->f\_id);

        if (s == 0)

            PCBlist[CurrentProc->f\_id].ax = 'T';

        else

            PCBlist[CurrentProc->f\_id].ax = 'F';

    }

**\_\_asm\_\_** ("int $0x8\n"); *// 调用时钟中断*

}

这个源语的内容就是为了结束相应的子进程，即更改当前进程控制块的状态为exit，然后判断它是否是子进程，并且其父进程是否状态为block，在此之后，唤醒父进程，并设置函数返回值。接下来调用时钟中断进行轮换

**另外的三个基本原语**

**void** wakeup(**int** id){

    PCBlist[id].status = Ready;

}

**void** block(**int** id){

    PCBlist[id].status = Block;

}

**void** do\_wait(){

**\_\_asm\_\_** ("cli\n");

    block(CurrentProc->id);

    Schedule();

}

这三个原语功能很明确我就不细说了。

**Int33-35中断更改**

    xor ax,ax

    push ds

    push es

    mov ax,cs

    mov es,ax

    mov ds,ax

    mov word[es:0xCC],INT\_33

    mov word[es:0xCE],ax

    mov word[es:0xD0],INT\_34

    mov word[es:0xD2],ax

    mov word[es:0xD4],INT\_35

    mov word[es:0xD6],ax

    pop es

    pop ds

    pop ecx

    jmp cx

这三个终端中，分别封装了do\_fork() , do\_wait() , do\_exit() 三个函数

比如当我们调用fork函数时，函数会先调用int33，然后int33中执行do\_fork函数完成相应的功能，这地方就体现到了我们之前所写的系统功能调用的用处

**Stackcopy函数核心部分:**

copy:

    mov dh,byte[bx]

    mov byte[di],dh

    sub di,1

    sub bx,1

    cmp bx,ax

    jnz copy

*;复制结束*

该函数的核心功能就是为了将父进程的堆栈复制到子进程相应位置，上面这段代码就实现了类似功能，其中bx存储了堆栈起始位置，而ax存储了当前堆栈栈顶位置，该函数通过一个循环不断的将旧的堆栈中的值复制到新的堆栈中，从而实现我们所需要的功能

**(6)实现效果：**

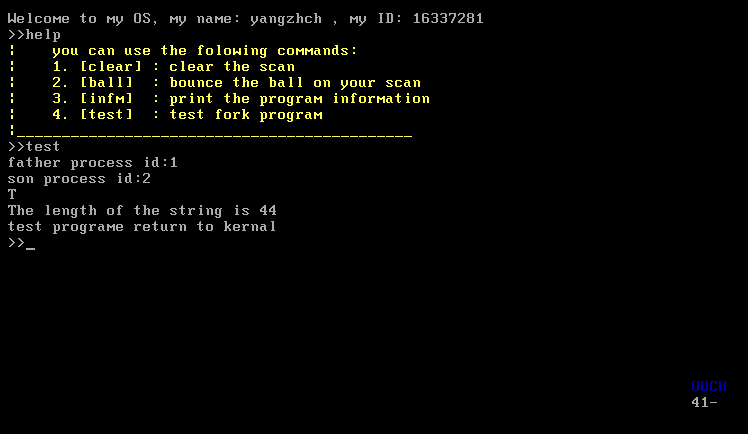
在本次实验中，我保留了之前实验的实验内容

要运行老师的测试代码，先输入test

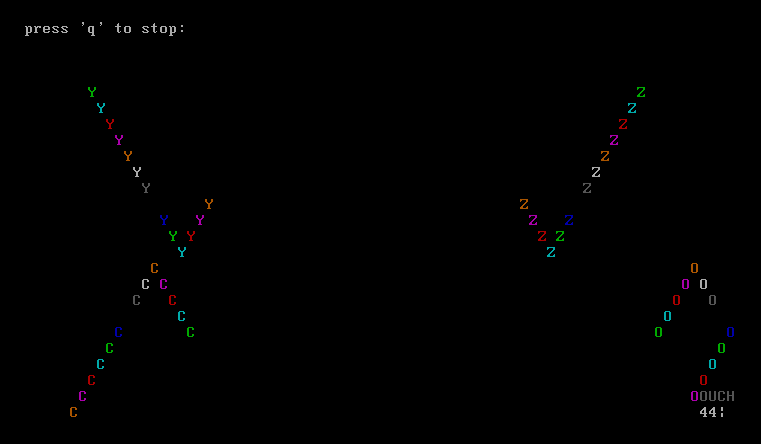


如图所示，代码正确运行，子进程id为2，父进程id为1，程序打印字符T，并且输出相应的字符串长度为44. 这说明我们的实验是成功的.

按q键退出当前进程，回到内核



接下来我们验证以前的功能依然是完好的：



如图所示，四个小球进程依然运行顺利

至此，说明我们此次实验成功.

**五 实验过程及其总结**

在本次实验中，我发现我的内核过于臃肿，以至于当其以7e00开头时，第一个扇区不够使用，因此我做了很多简化内核的工作，这也是了我的实验7进展比较缓慢，现在才交的原因。在本次实验中，我首先是理清楚了几个原语间的作用关系，然后才去实现的代码，再加上老师的ppt写的十分详细，因此本次实验中遇到的问题并不算多，接下来还是列举一些我在本次实验中遇到的问题：

**问题1：运行fork程序后，创建的子进程堆栈与父进程堆栈冲突**

这样的类似问题我在实验6中也遇到过，当时我打算将两个进程放在同一个段上，结果一不小心将两个进程的堆栈放到一起了，引起堆栈冲突导致程序崩溃。

在本次实验中，也出现了这样的问题，此类问题并不容易发现根本的原因，因此并不容易debug，但是有了实验6的基础，我马上意识到了这个问题的原因，于是我给我新创建了两个数组：

**int** STACK\_TABLE[5];

**int** EMPTY\_TABLE[5];

分别用来记录父进程堆栈位置和新的子进程堆栈位置

在fork函数中，我加入这样的代码：

**int** key = son\_p->cs >> 12; *// cs段寄存器的值前4位 如0x1000*

son\_p->sp = EMPTY\_TABLE[key];

EMPTY\_TABLE[key] -= 0x1000;

STACK\_TABLE[son\_id] = son\_p->sp;

son\_p->sp = stackcopy(CurrentProc->sp , STACK\_TABLE[CurrentProc->id] , son\_p->sp);

上述代码通过父进程的cs判断出父进程的id，因为我设定的程序加载模式就是0x1000加载到第一个进程控制块，0x2000加载到第二个，0x3000加载到第三个……以此类推。因此将cs右移12位所得结果就是进程id。

然后调用stackcopy函数，将父进程的堆栈，一个个复制到子进程的新的堆栈位置中。

**问题2：memcopy函数不能够copy进程控制块的ax寄存器**

先贴上memcopy的代码：

**void** memcopy(**struct** pcb\* father, **struct** pcb\* son){

    son->bx = father->bx;

    son->cx = father->cx;

    son->dx = father->dx;

    son->si = father->si;

son->di = father->di;

    son->bp = father->bp;

    son->flags = father->flags;

    son->es = father->es;

son->ds = father->ds;

son->cs = father->cs;

    son->ss = father->ss;

    son->ip = father->ip;

}

一开始，我一不小心在第一行把ax寄存器也copy了一下，其造成的直接结果就是把fork函数返回给父进程的值也copy给了子进程，本来子进程返回值应该位0，但在这个操作中，0被覆盖了。

这一个小点坑了我很久。。。。。。

**问题3：进程控制块没有初始化**

PCBlist[0].f\_id = -1;

    for(**int** i = 1 ; i < 5 ; i++){

        PCBlist[i].status=Empty;

        EMPTY\_TABLE[i]=0xffff;

    }

一开始我没有补上这样的代码，这串代码的操作就是为了初始化每一个进程控制块位empty，如果不进行这样的操作，容易产生代码在空的进程控制块间不断切换，导致操作系统死机

**问题4：内核过于臃肿**

在我的实验中，由于内核没有分离的问题，以及各个模块间的封装不够彻底，导致内核十分臃肿，就造成了“只要再多写一行代码，就会程序崩溃的问题”，究其原因，就是内核太大，导致第一个扇区不够用了，因此，在此次实验中，我部分化简了我的内核，但这次化简并不彻底。我会在下次实验进行改进。

**六 实验感想**

本次实验可能代码量不大，加上老师的ppt把原理讲述的很详细, 但是由于我需要重整内核的原因，所以我花了很多的功夫。在这次的实验中，我在网上找到了很多的资料，通过分析这些资料（源码），我得出了do\_fork和do\_exit程序的大致写法，可以见得网络对我们的重要性。

此外，有关一些十分隐藏的bug我们找不出来的时候，要善于询问同学，这次我就是有一个bug找不出来，问同学才问出来的。

本次实验的代码量大概200行，但是确实是十分的重要的.这次的实验，增强了我对于五状态进程模型的认识，也让我深刻的了解到了fork函数的原理，以及子进程父进程之间的关系。

其实这次实验对我的困难主要在于，我的内核过于臃肿，导致的一些莫名其妙的bug，不过当我把所有的bug修复好了以后，我感到十分欣喜并且很有成就感。操作系统这门课的确很有意义。