《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

五状态进程模型

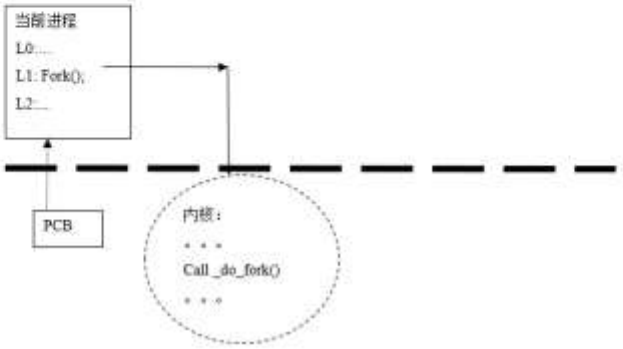
- 前一个原型中，操作系统可以用固定数量的进程解决多道程序技术。但是，这些进程模型还比较简单，进程之间互不往来，没有直接的合作
- 在实际的应用开发中，如果可以建立一组合作进程，并发运行，那么软件工程意义上更有优势
- 在这个项目中，我们完善进程模型
  - 进程能够按需要产生子进程，一组进程分工合作，并发运行，各自完成一定的工作
  - 合作进程在并发时，需要协调一些事件的时序，实现简单的同步，确保进程并发的情况正确完成使命

进程控制的基本操作

- 在这个项目中，我们完善进程模型
  - 扩展PCB结构，增加必要的数据项
  - 进程创建do_fork()原语，在c语言中用fork()调用
  - 进程终止do_exit()原语，在c语言中用exit(int exit_value)调用
  - 进程等待子进程结束do_wait()原语，在c语言中用wait(&exit_value)调用
  - 进程唤醒wakeup原语（内核过程）
  - 进程唤醒blocked原语（内核过程）

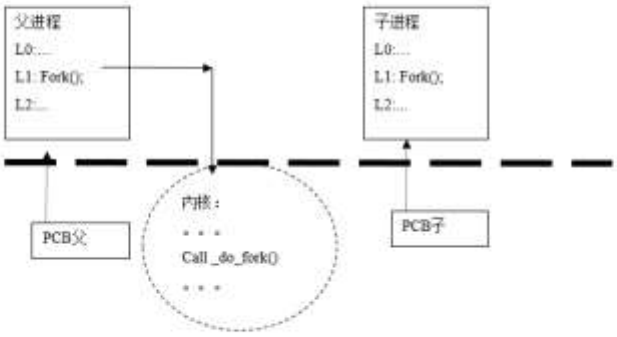
进程创建

- 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
  - 父子进程中对do_fork()函数的返值分别设置



do_fork()功能的示意图

- 内核完成进程创建后，系统中增加了一个进程，



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。
- 系统调用的返值如何获得：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：
  - 寻找一个自由的PCB块，如果没有，创建失败，调用返回－1；
  - 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。
  - 产生一个唯一的ID作为子进程的ID，存入至PCB的相应项中。
  - 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；
  - 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。

    Pcb_list[s].fPCB= pcb_list[CurrentPCBno];
  - 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。

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。
- void do_wait() {
- PcbList[CurrentPcbNo]->state=BLOCKED;
- Schedule();
- }
- 相应调度模块也要修改，禁止将CPU交权给阻塞状态的进程。

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。

```
void do_exit(char ch) {
    PcbList[CurrentPcbNo]->state=END;
    PcbList[CurrentPcbNo]->used=FREE;
    PcbList[PcbList[CurrentPcbNo]->pcbFID]->state=READY;
    PcbList[PcbList[CurrentPcbNo]->pcbPID]->ax=ch;
    Schedule();
}
```

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

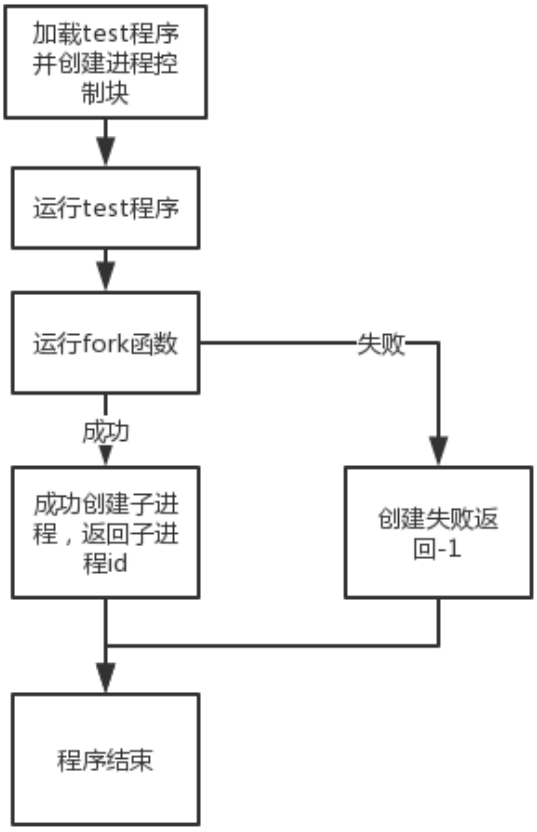
高级语言工具：标准 c 语言

磁盘映像文件浏览编辑工具

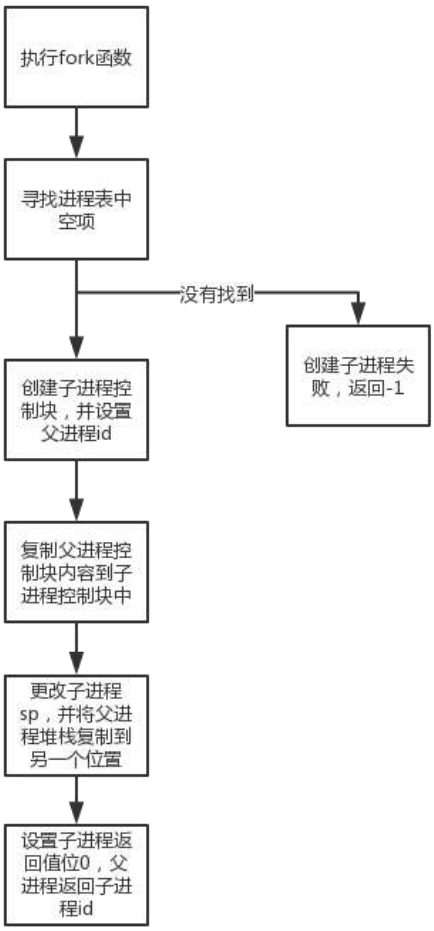
调试工具：Bochs

（3）程序流程示意图：

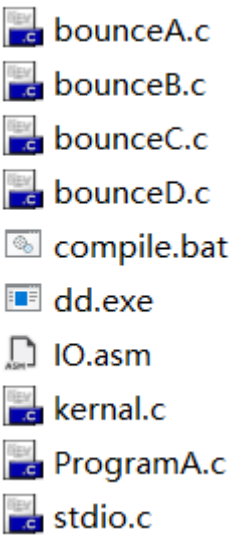
总体程序流程图：



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

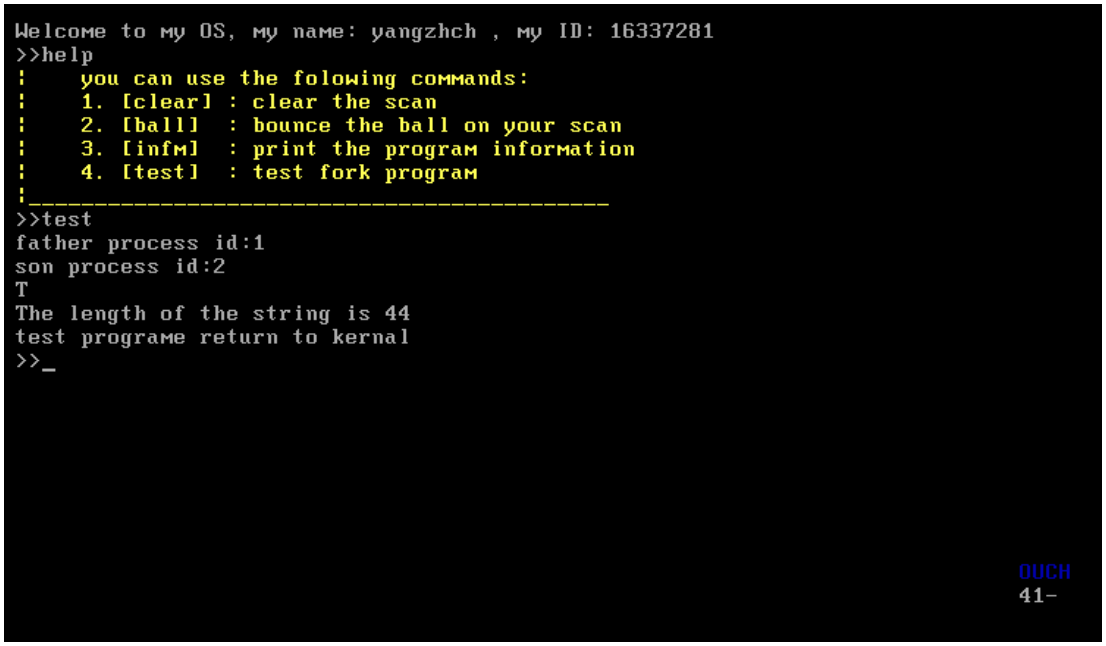
(6)实现效果：

在本次实验中，我保留了之前实验的实验内容

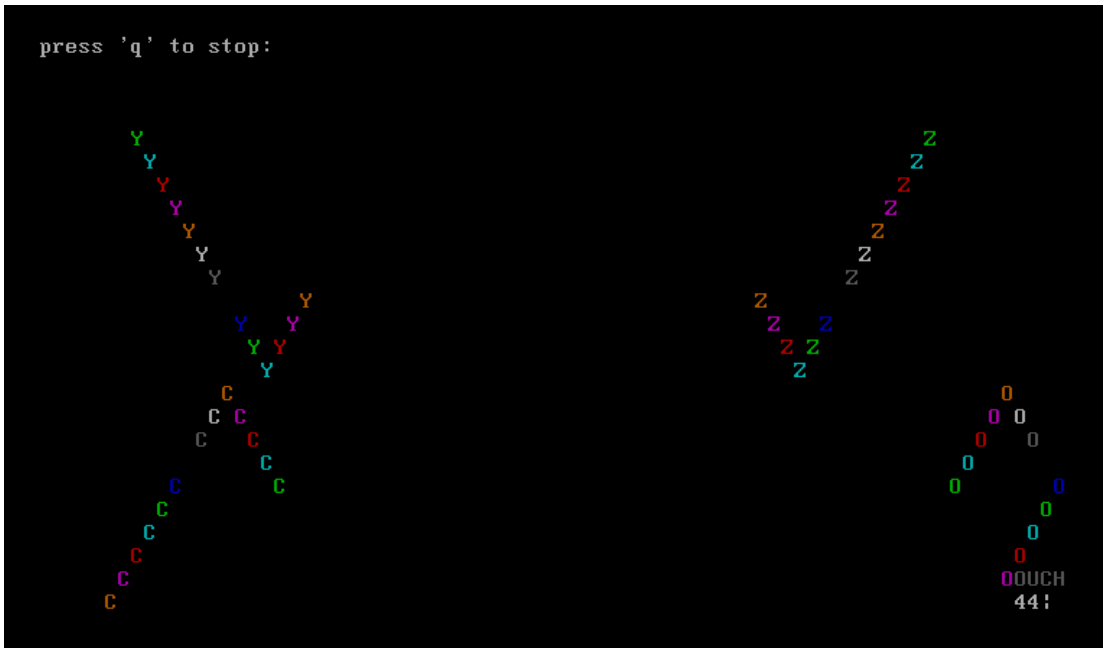
要运行老师的测试代码，先输入 test

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。