《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）：	17 级计算机类教务 3 班
学 生 姓 名 ：	姚森舰
学　　号 ：	17341189
时　　间 ：	2019　年　6　月　23　日

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(), blocked()和wakeup()`.

实验要求

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。

多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。

实验方案

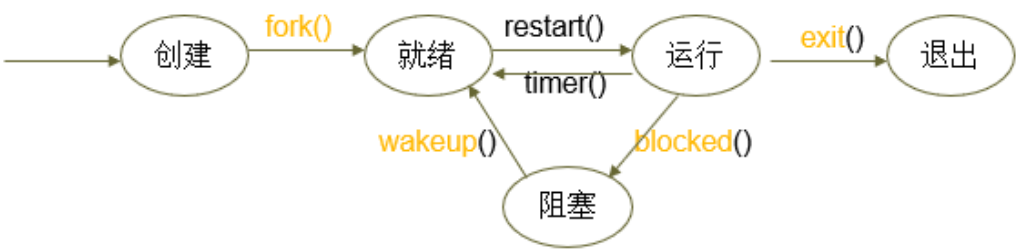
实验环境

Windows10 +VMware

实验工具

NASM + Winhex + Tasm + Tcc + Dosbox + Notepad++

主要思路



以上是状态转换图。

主要思路可以看看本次实验测试的流程，本次实验测试的大概流程为，在`test.c`中，调用封装好的`fork()`函数，`fork()`函数内部使用系统调用，系统调用`PCB.h`中的`do_fork()`，在`do_fork()`中，先在PCB中寻找空闲PCB块，将父进程的数据拷贝进去，新建子进程，并通过FatherID找打父进程，返回子进程的进程号，也就是PCB块号，拷贝时，栈需要特殊处理，将全部栈的内容拷贝过去。这使PCB表中就会有两个几乎相同的进程，但是他们的PID，也就是AX中的只不同，父进程的AX中是子进程号，子进程是0，通过PID，进行判断做不同的任务，这就完成了`fork()`操作。父进程在执行时遇到`wait()`函数就暂时将状态改为BLOCK，等待一段子进程完成后再继续执行。

调度时，就是在PCB表中找READY状态的进程执行，若最后只剩READY，或者BLOCK状态的进程，就将状态改为RUNNING并执行，执行完毕后返回内核，也就是PCB[0]。

fork()

fork()调用功能如下

1. 寻找一个自由的PCB块，如果没有，创建失败，调用返回 - 1；
2. 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。
3. 产生一个唯一的ID作为子进程的ID，存入至PCB的相应项中。
4. 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；
5. 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。
6. 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。

```
int do_fork()
{
    int i = Find_empty_PCB();       /* 寻找空闲进程 */
    if (i == -1){                  /* 没有空闲的PCB块 ax作为返回值 */
        pcb_list[CurrentPCBno].regImg.AX = -1;
        return ;
    }
    Program_Num++;                  /* 创建成功 */
    PCBcopy(&pcb_list[i], &pcb_list[CurrentPCBno]);                        /* 拷贝PCB */
    stackcopy(pcb_list[i].regImg.SS, pcb_list[CurrentPCBno].regImg.SS);    /* 特殊处理栈 */
    pcb_list[i].FatherID = CurrentPCBno;                                  /* 标识父进程 */
    pcb_list[i].Used =  1;                 /* 标识此进程被占用 */
    pcb_list[i].regImg.AX = 0;              /* 子进程本身返回0 */
    pcb_list[CurrentPCBno].regImg.AX = i;   /* 父进程返回进程号 */
    pcb_list[CurrentPCBno].Process_Status = READY;
    Switch();
}

int Find_empty_PCB()   /* 用于寻找空闲进程 */
{
    int index = 1;
    while (index < MAX_PCB_NUMBER)
    {
        if (pcb_list[index].Used != 1)     /* 若未被占用 */
            return index;
        index++;

    }
```

```
    return -1;
}
```

这里的`Switch()`实际上就是将实验六中的`Restart`部分相关代码写成一个函数，方便重用。

拷贝进程数据：

```
void PCBcopy(PCB* p1, PCB* p2)
{
    p1->regImg.AX = p2->regImg.AX;
    p1->regImg.BX = p2->regImg.BX;
    p1->regImg.CX = p2->regImg.CX;
    p1->regImg.DX = p2->regImg.DX;
    p1->regImg.CS = p2->regImg.CS;
    p1->regImg.IP = p2->regImg.IP;
    p1->regImg.DS = p2->regImg.DS;
    p1->regImg.ES = p2->regImg.ES;
    p1->regImg.GS = p2->regImg.GS;
    p1->regImg.FS = p2->regImg.FS;
    /*p1->regImg.SS = p2->regImg.SS;
    栈要做特殊处理，应该为Segment，这在初始化时已完成，其余见stackcopy() */
    p1->regImg.DI = p2->regImg.DI;
    p1->regImg.SI = p2->regImg.SI;
    p1->regImg.BP = p2->regImg.BP;
    p1->regImg.SP = p2->regImg.SP;
    p1->regImg.FLAGS = p2->regImg.FLAGS;
    p1->Process_Status = READY;
}
```

因为父子进程出现分支后执行不同的代码，所以他们的栈应该独立，但分支之前应相同，因此要拷贝全部栈：

```
public _stackcopy
_stackcopy proc
    push ds
    push es
    push di
    push si
    push cx
    push ax
    push bp
    mov bp, sp
    mov ax, word ptr [bp+18]   ; 源地址，也就是父进程
    mov ds, ax
    mov ax, word ptr [bp+16]   ; 目的地址,也就是子进程
    mov es, ax
    mov di, 0
    mov si, 0

    mov cx, 100h-4             ; 全部栈
```

```
loop_copy:
    mov al, byte ptr ds:[si]
    mov byte ptr es:[di], al
    inc di
    inc si
    loop loop_copy
    pop bp
    pop ax
    pop cx
    pop si
    pop di
    pop es
    pop ds
    ret
_stackcopy endp
```

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。

```
void do_wait()
{
    /* 阻塞进程 */
    blocked(CurrentPCBno);
    Schedule();
    Switch();       /* 切换PCB */
    delay();
}
```

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。 这部分要注意的是，当子进程执行完毕后，要向父进程传递执行完毕的信息。

```
int do_exit(int ch)
{
    int FatherID = pcb_list[CurrentPCBno].FatherID;
    pcb_list[CurrentPCBno].Process_Status = EXIT;
```

```
    /* 结束进程并初始化进程控制块 */
    init(&pcb_list[CurrentPCBno], (CurrentPCBno)*0x1000, 0x100);
    /* 如果当前退出进程的父进程不是内核，解除父进程的阻塞 */
    if (FatherID != 0){
        /* 唤醒父进程 */
         wakeup(FatherID);
        /* 用ax来传递信号 */
        pcb_list[FatherID].regImg.AX = ch;
    }
    Program_Num--;
    Segment=0x1000;
    delay();
    Schedule();
    Switch();
}
```

将这三个函数作为系统调用，因为此部分实现相似，故只列出`do_fork()`，其余见代码：

```
    cmp ah,9
    jz helpJumpFun9

    cmp ah,10
    jz helpJumpFun10

    cmp ah,11
    jz helpJumpFun11
    ...

    helpJumpFun9:   ; 实际跳转区间超出jz可行的范围，所以引入中间跳板，之前的实验报告中已说明
    jmp fun9

    helpJumpFun10:
        jmp fun10

    helpJumpFun11:
        jmp fun11
    ...
    fun9:            ; 这些其实是多余的，只是为了整体看起来规范
        jmp forking
        iret

    fun10:
        jmp waiting
        iret

    fun11:
        jmp exiting
        iret

extrn _do_fork:near
```

```
forking:
    push ss
    push ax
    push bx
    push cx
    push dx
    push sp
    push bp
    push si
    push di
    push ds
    push es
    .386
    push fs
    push gs
    .8086
    mov ax, cs                      ;ds es回到内核状态
    mov ds, ax
    mov es, ax                      ; 此时PSW CS IP 已经被压栈
    call near ptr _Save_Process     ; 保存现在的状态
    call near ptr _do_fork
    iret
```

schedule()

调度时，就是在PCB表中找READY状态的进程执行，若最后只剩READY，或者BLOCK状态的进程，就将状态改为RUNNING并执行，执行完毕后返回内核，也就是PCB[0]。

```
void Schedule(){
    int temp_count = MAX_PCB_NUMBER;          /* 循环次数 */
    if (pcb_list[CurrentPCBno].Process_Status == RUNNING)
        pcb_list[CurrentPCBno].Process_Status = READY;

    while (temp_count--)        /* 要么return 要么遍历整个PCB */
    {
        CurrentPCBno++;
        if (CurrentPCBno >= MAX_PCB_NUMBER)
            CurrentPCBno = 1;
        if (pcb_list[CurrentPCBno].Used == 1)    /*如果这个PCB块被占用 */
        {
            if (pcb_list[CurrentPCBno].Process_Status == READY)
            {
                pcb_list[CurrentPCBno].Process_Status = RUNNING;
                return ;
            }
            /* 如果是new状态的话要单独讨论一次 */
            else if (pcb_list[CurrentPCBno].Process_Status == NEW)
                {

                    /* 特殊处理了一下，因为多进程几乎结束了，不应该出现new的状态，忽略 */
```

```
                    /* Finite为调度的次数，与实验六不同的是，将其声明为了全局变量 */
                     if( Finite >= 118 ){
                        continue;
                    }
                    return ;
                }
            }
        }
        /* 如果没有其他运行的进程，就继续运行之前的进程 如果它也结束了就返回内核 */
        if (pcb_list[CurrentPCBno].Used == 1 && pcb_list[CurrentPCBno].Process_Status == READY)
            pcb_list[CurrentPCBno].Process_Status = RUNNING;
        /* 运行结束，返回内核 */
        else{
            CurrentPCBno = 0;
            Segment = 0x1000;
            Program_Num = 0;
        }

    }
```

blocked()

为方便实现，都传入了一个id参数：

```
void blocked(int id){
    if(pcb_list[id].Process_Status == RUNNING){
        pcb_list[id].Process_Status == BLOCKED;
    }
}
```

wakeup()

```
void wakeup(int id){
    if(pcb_list[id].Process_Status==BLOCKED){
        pcb_list[id].Process_Status == READY;
    }
}
```

其他相关或者新增部分代码

`PCB.h`

PCB新增属性

```
typedef struct PCB{
    RegisterImage regImg;       /* 各个寄存器的值，多个int组成 */
    int Process_Status;         /* 进程状态 */
    int Used;                   /* 是否被占用 */
    int FatherID;               /* 父进程ID */
}PCB;
```

一些变量的声明：

```
int NEW = 0;
int READY = 1;
int RUNNING = 2;
int BLOCKED = 3;
int EXIT = 4;
int Segment = 0x1000;             /* 用户程序段值与实验六不同的是，将这两个变量其声明为了全局变量 */
int Finite = 0;                  /* 调度次数 */
#define MAX_PCB_NUMBER 8          /*最多同时放这么多个进程 */
```

对应的初始化PCB函数`void init(PCB* pcb,int segement, int offset)`新增两行代码：

```
    pcb->Used = 0;          /* 初始化为未被占用 */
    pcb->FatherID = 0;      /* 初始父进程为主进程 */
```

`kernel.c`

新增测试指令`t`：

```
-t: to run test program for lab7.
```

指令的实现：

```
void run_test_program_of_7(){
    run_test_of_7(Segment,6*2-1);
    Segment += 0x1000;
    Program_Num = 1;
    pcb_list[1].Used = 1;    /* 注意 */
}
```

为方便debug写的输出数字的函数：

```
void printnumber(int number)
{
    int i = 0;
    char temp[1000];
    while (number)
    {
        temp[i++] = number % 10 + '0';
        number /= 10;
    }
    i -= 1;
    while (i >= 0)
        printchar(temp[i--]);
}
```

`syscll.asm`

加载测试程序：

```
public _run_test_of_7
_run_test_of_7 proc
    push ax
    push bp
    mov bp,sp
    mov ax,[bp+6]       ;段地址 ； 存放数据的内存基地址
    mov es,ax           ;设置段地址
    mov bx,100h         ;偏移地址;
    mov ah,2            ;功能号
    mov al,2            ;扇区数 实验7的测试程序要两个扇区
    mov dl,0            ;驱动器号 ； 软盘为0
    mov dh,1            ;磁头号 ； 起始编号为0
    mov ch,0            ;柱面号 ； 起始编号为0
    mov cl,[bp+8]       ;起始扇区号 ； 起始编号为1
    int 13H             ;调用中断
    pop bp
    pop ax
    ret
_run_test_of_7 endp
```

`process.asm`

封装`fork(), wait(), exit()`,过程相似，只举一个例子，例子如下，详见代码：

```
public _fork
_fork proc
    mov ah, 9
    int 21h
    ret
_fork endp
```

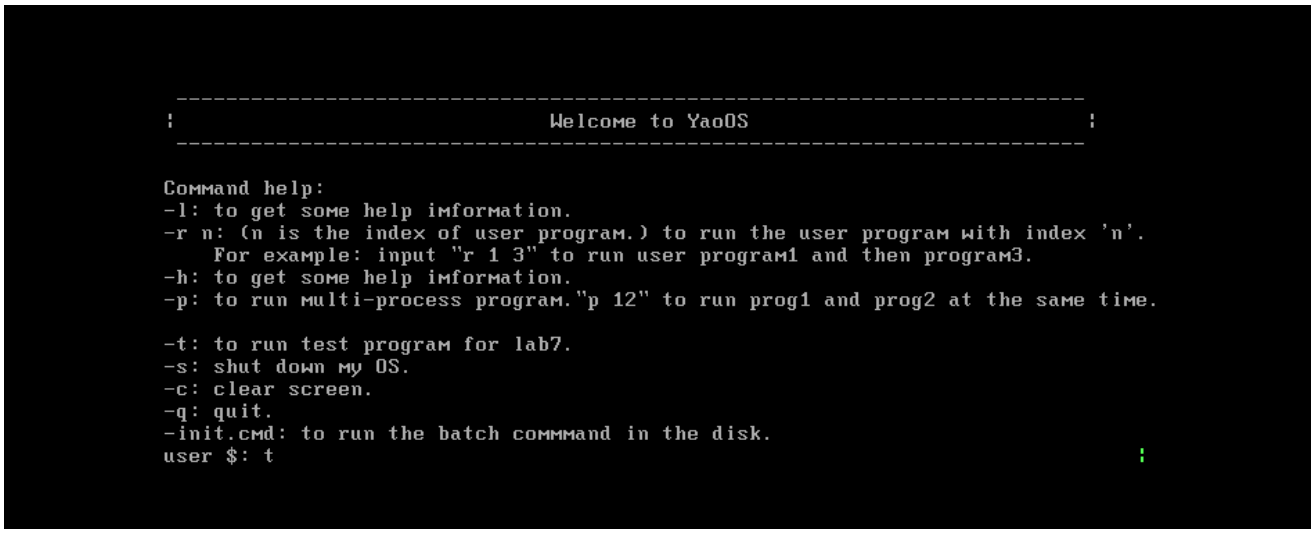
测试

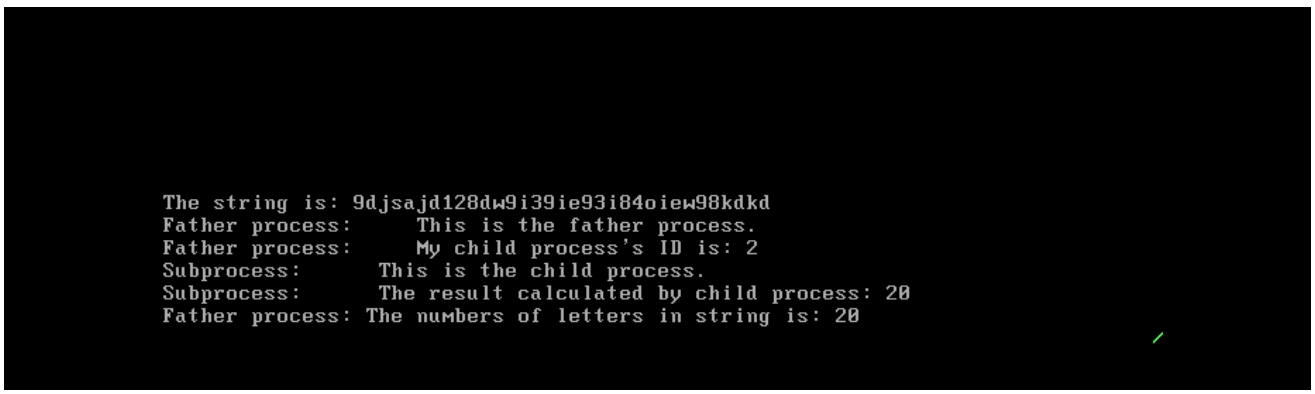
测试代码：

```
#include "process.h"
extern int fork();
extern void wait();
extern int exit();

char str[80] = "9djsajd128dw9i39ie93i84oiew98kdkd";
int letterNr = 0;
int ex;
void main()
{
    int pid=0;
    int i;
    print("The string is: ");
    print(str);
    print("\r\n");
    pid = fork();        /* 封装好的函数 */
    if (pid == -1)
    {
        print("error in fork!");
        exit(-1);
    }

    if (pid)            /* fork返回的pid是子进程的id */
    {
        int l = 0;
        print("Father process:     This is the father process.\r\n");
        print("Father process:     My child process's ID is: ");
        printnumber(pid);
        print("\r\n");
        wait();         /* 封装好的函数 */
        print("Father process: The numbers of letters in string is: ");
        for (i=0; str[i]; ++i)
            if((str[i]>='A'&&str[i]<='Z')||(str[i]>='a'&&str[i]<='z'))
                l++;
        printnumber(l);

        print("\n\r");
```

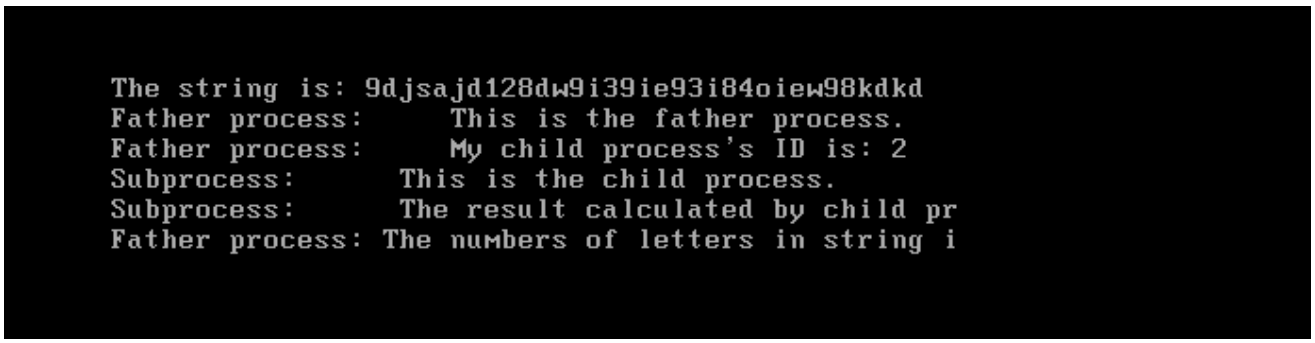
```
        delay();
        exit(0);
    }

    else                /* 父进程也就是主进程收到的pid为0 */
    {
        print("Subprocess:      This is the child process.\r\n");
        for (i=0; str[i]; ++i)
            if((str[i]>='A'&&str[i]<='Z')||(str[i]>='a'&&str[i]<='z'))
                letterNr++;
        print("Subprocess:      The result calculated by child process: ");
        printnumber(letterNr);
        print("\r\n");
        exit(0);
    }
}
```

测试结果





注意若是在测试时，多按了键，可能会弹出之前的`“ouch ouch”`，与这部分输出重叠，使得输出不完整，出现以下情况，这也是正常的：



临近期末，没有再这些小细节上花费太多时间。

总结

1. 遇到的第一个问题就是内核超出了分配的扇区，居然超过了十个扇区！由于又是利用批处理做得编译，所以找了很久才找出问题，实在是浪费了很多时间。
2. 在跑测试的时候，要把新建的PCB块的Used属性设置为1，否则就会出现很奇怪的现象，Program_num的确变为1了，但是在调度的时候是会忽略这个PCB块的，根据schedule()函数的代码，会忽略没有被占用的PCB块。
3. 由于之前部分代码，比如按键输出”ouch ouch“会和测试函数输出重叠，可能会出现部分字符或结果输出看起来有点问题，但是结果其实是没问题的，这里需要注意一下。做的时候也以为是代码写的不对，结果发现原因居然在这里，浪费许多时间。
4. 在schedule()函数处卡了很久，事实上，到现在我也没有彻底想清楚问题出在哪里，之前有时候是fork()部分可以成功运行，但是回到实验六的多进程时就不能正确的返回，要不然就是实验六的那部分正确，但是fork()又有问题。只有不断的修改schedule()函数，改成现在比较合理的样子之后就没有问题了，但是原因没找到，应该是在某些细节或者某些特殊情况没有考虑清楚，做了一些测试没有问题，但是还是不踏实，只能保证现在的代码没问题，能解决计数字母的问题，但是其他问题不敢保证。
5. 本次实验的原理，其实老师课上早就讲过，但实现起来的时候出现了太多细节的问题，最后找出来的时候感觉很懊恼，这些问题让人很急躁，也可能是期末临近的原因。操作系统有时候让人无奈的地方就是没有好的方法DEBUG，只能靠print了，后悔没有在一开始就用bochs.