中山大学数据科学与计算机学院本科生实验报告

（2016 学年秋季学期）

课程名称：操作系统实验　　任课教师：凌应标　　教学助理（TA）：

年级	15 级	专业（方向）	软件工程（移动信息工程）
学号	15352461	姓名	宗嘉希（组长）
学号	15352443	姓名	钟凌山
学号	15352448	姓名	周禅城
电话	18022724490	Email	zongjx@mail2.sysu.edu.cn
开始日期	2017.06.01	完成日期	2016.06.24

【实验题目】

五状态进程模型

【实验目的】

实现五状态进程模型，其中包括：

（1）拓展 PCB 结构，增加必要的数据项

（2）进程创建 do_fork()原语，在 c 语言中用 fork()调用

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

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

（5）进程唤醒 wakeup 原语（内核过程）

（6）进程唤醒 blocked 原语（内核过程）

【实验要求】

在实验五或更后的原型基础上，进化原型操作系统，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。

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

【实验方案】

一、 硬件及虚拟机配置

硬件：操作系统为 win10 的笔记本电脑

虚拟机配置：无操作系统，10MB 硬盘，4MB 内存，启动时连接软盘

二、 软件工具及作用

Nasm:用于编译汇编程序，生成.bin 文件

WinHex:用于向软盘写入程序

VMware Workstation 12 Player：用于创建虚拟机，模拟裸机环境

Notepad++: 用于编辑汇编语言文件

TCC：用于编译 C 文件，生成 OBJ 文件

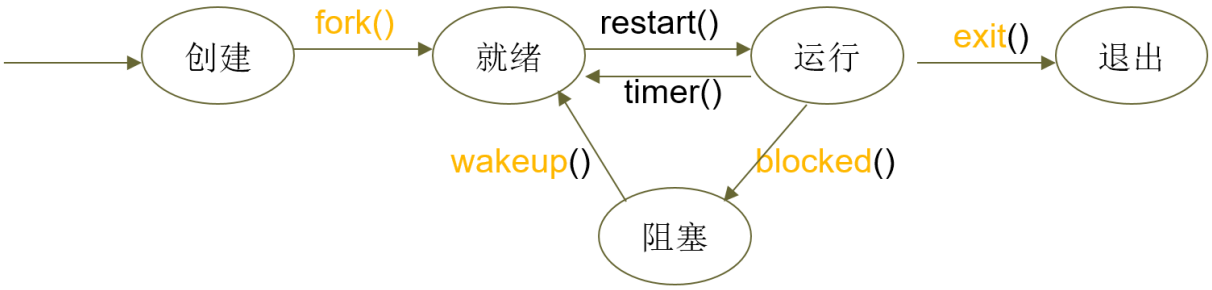
TASM：用于编译 ASM 文件，生成 OBJ 文件

TLINK：用于把两个 OBJ 文件连接，生成 COM 文件

Dosbox：用于提供 16 位运行环境，为 TCC+TASM+TLINK 编译链接提供一个环境

【实验过程】

相比二状态进程模型（运行和就绪两个状态），五状态进程模型增多了三个状态，包括了创建、就绪、运行、阻塞和退出五个状态。具体状态转换的关系如下图：



这样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，可以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。

对比上一次的实验 ,我们只需要在上一次的原型操作系统的基础上添加三个函数 fork()、exit()以及 wait()函数就可以满足实验要求。

因此，我们这一次的实验就直接修改我们上一次的二状态进程模型实验的代码，保留上次实验的功能，然后再写一个新的用户程序去测试我们的 fork()、exit()和 wait()函数是否正常运行。在这次的实验中，我们省略掉了 wakeup()和 block()函数，但是其功能都整合到了 wait()和 exit()函数里面。

我对这一次实验，这整一个的过程其实都有了非常深刻的理解，可惜我们这一次的实验做了 3 周都没有办法做出来，由于期末考临近，所以我们不得不放弃，但是我们会在这一次的实验报告中呈现我们对这一次实验内容的理解与我们在这一次实验中遇到的各种问题。

这一次的五状态进程模型相对于前一次的二状态进程模型来说多出了三个状态，而本次实验要求我们编写函数去实现。首先是创建，当一个父进程想要创建一个新的子进程的时候，首先要把父进程的状态保存下来，然后通过 fork()函数，找到一个空的 pcb 块，创建一个新的子进程。那么子进程的状态又是从何而来的呢？显然 ,是由父进程复制产生的。当前父进程把其状态(8086 中的 16 个寄存器的数据) 完全复制到新的 pcb 块中，这样子就可以产生出一个新的子进程 pcb 块。然而由于是子进程，完全复制父进程的话会导致数据混乱，也就是说，子进程应该有一个独立的栈空间去存放数据，这一个独立的占空间必须要跟父进程分开，这样才不会影响到父进程的数据。fork()函数的作用就是这样，找到一个空的 pcb 块，复制父进程的数据到子进程，再找一个新的栈空间，把父进程的堆栈里面的内容完全复制到子进程的栈中 (其实这一个过程就是让子进程遗传了父进程的一切信息) , 然后为子进程做一个标记，标记

其父进程的位置，以便在子进程结束（死去）的时候恢复到父进程中工作。但是要注意，fork()函数的最后，运行的还是父进程，操作系统的权力还是在父进程手中，fork()的过程仅仅是为父进程产生一个子进程而已。那么到底是什么时候才能够让子进程运行呢？这就是我们要实现的第二个函数 wait()函数了。wait()函数的作用是把父进程进行阻塞，然后通过 schedule()函数，把操作系统的权力转移到其子进程处。把父进程阻塞是什么意思呢？由于我们在运行子进程的时候，父进程不需要并行运行，因此在子进程运行的过程中，父进程是不能够运行的，因此我们需要把父进程排斥在轮转以外，因此我们给它增加一种新的状态叫做 BLOCKED，意思是把父进程阻塞掉，暂时不允许在 schedule()的过程中把其转换为 RUNNING 状态。那么，wait()函数的意思就是把父进程阻塞掉，然后轮转其他的 pcb，运行其他的用户程序。什么时候恢复父进程呢？那就要等到子进程结束的时候才能够恢复了，子进程的结束又是怎么样操作的呢？我们可以通过我们要实现的第三个函数 exit()去把子程序“杀死”，然后恢复子进程。exit()函数里面主要做的事就是通过之前记录在子进程中的父进程的位置，找到被阻塞的父进程，把它从 BLOCKED 状态转换为 READY 的状态，就是说取消了父进程的阻塞状态，使父进程可以继续运行。在恢复父进程的同时，也要把子进程“杀死”，因此把子进程的状态变为 EXIT，然后通过 schedule，使程序切换到另外一个 pcb 块，恢复寄存器数据。

以上就是五状态进程的整个过程。

由于种种的压力以及紧迫的时间，我们没有办法及时的做出一个非常完善的五状态进程模型操作系统，但是我们做出了一个符合本次实验要求的程序，使用老师给出的代码（用户程序），去测试我们的五状态进程模型。以下为老师的代码：

```
char str[80]="129djwqhdsajd128dw9i39ie93i8494urjoiew98kdkd";
int LetterNr=0;
void main() {
  int pid;  char ch;  pid=fork();
if (pid==-1) printf("error in fork!");
    if (pid) {  ch=wait(); printf("LetterNr="); ntos(LetterNr); }
Else {  CountLetter(str);  exit(0);}
}
```

老师的测试代码很简单的体现出了五状态进程的思想，就像我在是上面分析的一样，首先在程序刚开始的时候进行 fork，产生子进程，父进程返回值非 0，进入条件语句，进行 wait，切换到子进程，同时阻塞父进程，子进程返回的 pid 值是 0，然后进入另一个条件语句，进行对字符串的计数，保留在全局变量中，然后通过 exit“杀死”子进程，同时恢复父进程，通过轮转让父进程重新运行，由于在子进程中已经把字符串的长度计算出来了，因此就能够在父进程没有对字符串进行计数的情况下输出字符串的长度。

因此我们这一次就基于上次二状态进程模型实验，进行修改，保留所有的功能（除了 33、34、35 号中断，我把这三个中断改成用于用户程序调用 fork()、wait()和 exit()三个函数的时候调用的中断了）再加上本次的实验内容。

本次实验的关键就是在于 fork()、wait()和 exit()三个函数，还有轮转函数的正确。

首先我们来看一下我们写的 fork 函数，由于我们的程序不是很完善（期末考将近，时间都花到复习上面了，不能把精力都放在操作系统实验上，万分抱歉），我们就做了一个只 fork 一次的模型，程序如下图所示：

```
int do_fork()
{
    if(CurrentPCBno != 1)
    {return 0;}
else{
    int i;
    PCB* p;
    p=&pcb_list[1];
    while(p<=&pcb_list[7]&&p->Process_Status!=NEW) {p++;}            /*寻找空的pcb块*/

    if( p > &pcb_list[7] )
        pcb_list[CurrentPCBno].regImg.AX = -1;                       /*超出范围，返回-1*/
    else
    {
        Program_Num++;
        memcopy( pcb_list + CurrentPCBno, p );                       /*复制*/
        /*memcpy( pcb_list[CurrentPCBno].regImg.SS, p ->regImg.SS , 0x100 );*/

        for(i=0;i<2000;i++){
            stacks[4000-i]=stacks[2000-i];                           /*堆栈的复制*/
        }
        p->regImg.SP = pcb_list[CurrentPCBno].regImg.SP + 4000;      /*改变sp的值*/
        p->pcbid = p - pcb_list;
        p->f_pcbid = CurrentPCBno;
        p->Process_Status = READY;
        pcb_list[CurrentPCBno].regImg.AX = p->pcbid;
        p->regImg.AX = 0;
        return pcb_list[CurrentPCBno].regImg.AX;
    }
}
}
```

我们的父进程都是写在 1 号 pcb 里面的，因此，我们如果只进行一次 fork，就只需要对 1 号 pcb 进行 fork，其他的跳过。进入 fork 以后，首先寻找一个空闲的 pcb 块，如果找不到，就返回-1；找到了的话，就开始进行寄存器内容的复制了，首先把寄存器的值一一复制：

```
void memcopy( PCB* F_PCB, PCB* C_PCB )                /*复制寄存器*/
{
    /*C_PCB -> regImg.SP = F_PCB -> regImg.SP;*/
    C_PCB -> regImg.SS = F_PCB -> regImg.SS;
    C_PCB -> regImg.GS = F_PCB -> regImg.GS;
    C_PCB -> regImg.ES = F_PCB -> regImg.ES;
    C_PCB -> regImg.DS = F_PCB -> regImg.DS;
    C_PCB -> regImg.CS = F_PCB -> regImg.CS;
    C_PCB -> regImg.FS = F_PCB -> regImg.FS;
    C_PCB -> regImg.IP = F_PCB -> regImg.IP;
    C_PCB -> regImg.AX = F_PCB -> regImg.AX;
    C_PCB -> regImg.BX = F_PCB -> regImg.BX;
    C_PCB -> regImg.CX = F_PCB -> regImg.CX;
    C_PCB -> regImg.DX = F_PCB -> regImg.DX;
    C_PCB -> regImg.DI = F_PCB -> regImg.DI;
    C_PCB -> regImg.SI = F_PCB -> regImg.SI;
    C_PCB -> regImg.BP = F_PCB -> regImg.BP;
    C_PCB -> regImg.FLAGS = F_PCB -> regImg.FLAGS;
}
```

寄存器的值复制完以后再复制堆栈里面的内容，就是把数组里面的内容复制，再改变 sp 的值，用父进程的 sp 加上数组的大小乘以 2，因为数据类型的不一样，所以要乘上 2。至此，子进程的复制就完成了。

接下来是 wait()函数了。

```
int do_wait() {
    pcb_list[CurrentPCBno].Process_Status = BLOCKED;          /*阻塞父进程*/
    Schedule();                                               /*轮转到子进程*/
    /*PCB_Restart();*/
    return pcb_list[CurrentPCBno].regImg.AX;
}
```

wait 函数的结构很简单，其实就是阻塞父进程，先让父进程“暂停运行”，然后把通过轮转把 cpu 的权利转交给子进程。

最后就是 exit()函数，exit 函数的作用其实就是把阻塞掉的父进程恢复回来，然后再永久地结束子进程，不过子进程可以留下一个“遗言”给父进程：就是把传进来的数据放到父进程的 ax 寄存器中，在返回的时候就可以调用作为返回值了。结构如下：

```
int do_exit(char ch) {
    pcb_list[CurrentPCBno].Process_Status = EXIT;
    pcb_list[pcb_list[CurrentPCBno].f_pcbid].Process_Status = READY;
    pcb_list[pcb_list[CurrentPCBno].f_pcbid].regImg.AX=ch;
    Schedule();
    /*PCB_Restart();*/
    return CurrentPCBno;
}
```

至此，主要的那三个函数就已经呈现完毕了。

然后说一下我是如何把用户程序里的 fork()、wait()和 exit()连接到内核里面的。我是使用了中断的方法来进行连接的。我本来是使用 21 号中断来调用的，但是中间出现了一些奇怪的错误，我没有办法能够解决，所以不得已的情况下改用了 33、34、35 号中断来进行封装。

在用户程序中的封装如下：

```
public _fork
_fork proc
    int 33h
    ret
_fork endp

public _wait
_wait proc
    int 34h
    ret
_wait endp

public _exit
_exit proc
    int 35h
    ret
_exit endp
```

在内核中的封装如下：

```
public _int33h
_int33h proc
    cli
    push es
    push ax
    ;es置零
    xor ax,ax
    mov es,ax
    ;填充中断向量
    mov word ptr es:[0cch],offset _Int33
    mov ax,cs
    mov word ptr es:[0ceh],ax
    pop ax
    pop es
    sti
    ret
_int33h endp

 ; ****************************************
;* Int 33h                                *
; ****************************************
public _Int33
_Int33 proc
    call Save1
    call near ptr _do_fork
    jmp Restart1


_Int33 endp
```

```
public _int34h
_int34h proc
    cli
    push es
    push ax

    xor ax,ax
    mov es,ax
    mov word ptr es:[0d0h],offset _Int34
    mov ax,cs
    mov word ptr es:[0d2h],ax

    pop ax
    pop es
    sti
    ret
_int34h endp
 ; ****************************************
;* Int 34h                                *
; ****************************************
public _Int34
_Int34 proc
    call Save1
    call near ptr _do_wait
    jmp Restart1

_Int34 endp
```

```
public _int35h
_int35h proc
    cli
    push es
    push ax

    xor ax,ax
    mov es,ax
    mov word ptr es:[0D4h],offset _Int35
    mov ax,cs
    mov word ptr es:[0d6h],ax

    pop ax
    pop es
    sti
    ret
_int35h endp
 ; ***************************************
;* Int 35h                               *
; ***************************************
public _Int35
_Int35 proc
    call Save1
    call near ptr _do_exit
    jmp Restart1

_Int35 endp
```

由于上一次的二状态进程实验中，我是基于原型代码进行修改的，但是原型代码中的 save 和 restart 在本次实验中不适用，因为本次的用户程序有数据要压入堆栈中，也就是要进行数据的传输，这一个保存和恢复的过程一定要小心翼翼的。不能够有任何的差错，所以我就重新写了一个 save 和 restart，如下：

```
    ret_save dw 0
Save1:
    pop word ptr [ret_save]

    push ss
    push ax
    push bx
    push cx
    push dx

    mov ax,sp
    add ax,16
    push ax

    push bp
    push si
    push di
    push ds
    push es
    .386
    push fs
    push gs
    .8086

    mov ax,cs
    mov ds, ax
    mov es, ax

    call near ptr _Current_Process
    mov bp,ax
```

```
    pop word ptr ds:[bp+2]
    pop word ptr ds:[bp+4]
    pop word ptr ds:[bp+6]
    pop word ptr ds:[bp+8]
    pop word ptr ds:[bp+10]
    pop word ptr ds:[bp+12]
    pop word ptr ds:[bp+14]
    pop word ptr ds:[bp+16]
    pop word ptr ds:[bp+20]
    pop word ptr ds:[bp+22]
    pop word ptr ds:[bp+18]
    pop word ptr ds:[bp+24]
    pop word ptr ds:[bp+0]
    pop word ptr ds:[bp+26]
    pop word ptr ds:[bp+28]
    pop word ptr ds:[bp+30]
    jmp [ret_save]
```

```
Restart1:
    call near ptr _Current_Process
    mov bp, ax

    mov ss,word ptr ds:[bp+0]
    mov ax,word ptr ds:[bp+16]
    mov sp,ax

    push word ptr ds:[bp+30]
    push word ptr ds:[bp+28]
    push word ptr ds:[bp+26]

    push word ptr ds:[bp+2]
    push word ptr ds:[bp+4]
    push word ptr ds:[bp+6]
    push word ptr ds:[bp+8]
    push word ptr ds:[bp+10]
    push word ptr ds:[bp+12]
    push word ptr ds:[bp+14]
    push word ptr ds:[bp+18]
    push word ptr ds:[bp+20]
    push word ptr ds:[bp+22]
    push word ptr ds:[bp+24]

    pop ax
    pop cx
    pop dx
    pop bx
    pop bp
    pop si
    pop di
    pop ds
    pop es
    .386
    pop fs
    pop gs
    .8086
    iret
```

除此之外，还有轮转的函数需要有一些改动：

```
void Schedule(){

    if(pcb_list[CurrentPCBno].Process_Status == RUNNING)
    pcb_list[CurrentPCBno].Process_Status = READY;

    CurrentPCBno ++;
    if( CurrentPCBno > Program_Num )
        CurrentPCBno = 1;
    while(pcb_list[CurrentPCBno].Process_Status == BLOCKED || pcb_list[CurrentPCBno].Process_Status == EXIT) {
        CurrentPCBno ++;
    }

    if( pcb_list[CurrentPCBno].Process_Status != NEW )
        pcb_list[CurrentPCBno].Process_Status = RUNNING;
}
```

以上显示的内容基本上就是最重要的内容了。

接下来就开始介绍我用于测试的用户程序了。

我的用户程序是使用 c 和汇编写的，然后通过 tlink，编译出来一个 com 文件，放进虚拟软盘中。

在 c 文件中，写的主要是测试的主函数，还有各种功能函数等等，如下：

```
char str[80]="129djwqhdsajd18087";
int LetterNr=0;
void dmain() {
    int pid;
    int ch;
    pid=fork();
    if (pid==-1) {Print("error in fork!");exit(-1);}
    if (pid)  {
    Print("LetterNr=?        ");
    ch=wait();
    Print("LetterNr=");
    PrintNum(LetterNr);
}
else {
    LetterNr =CountLetter(str);
    Print("Counting.....                                                    ");
    exit(0);
     }
    Print("  Program finished");
}
```

以上是测试的主函数。

```
void Print( char* word )
{
    while( *word != '\0' )
    {
        printChar( *word );
        word ++ ;
    }
}
void PrintNum(int num){
    char s[20];
    int cnt=0;
    do{
        s[cnt]=num%10+48;
        num/=10;
        cnt++;
    }while (num>0);
    s[cnt]=0;
    while (cnt--){
        printChar(s[cnt]);
    }
}
```

以上是打印字符串和数字的函数。

```
int CountLetter(char *str){
    int cnt=0;
    while (*str!=0){
        cnt++;
        str++;
    }
    return cnt;
}
```

以上是对字符串进行计数的函数。

下面是汇编代码的主体：

```
start:
call near ptr _dmain
jmp $

public _printChar
_printChar proc
    push bp
        mov bp,sp
        ;***
        mov al,[bp+4]
        mov bl,0
        mov ah,0eh
        int 10h
        ;***
        mov sp,bp
    pop bp
    ret
_printChar endp

public _fork
_fork proc
    int 33h
    ret
_fork endp

public _wait
_wait proc
    int 34h
    ret
_wait endp

public _exit
_exit proc
    int 35h
    ret
_exit endp
```

分别用 TCC 和 TASM 把 c 文件和汇编文件编译成 obj 文件，然后使用 TLINK 连接生成 com 文件，放进虚拟软盘中。完成测试的用户程序。

另外，我还在操作系统的主函数中添加了一些代码，专门用于运行测试用户程序：

```
    else if( check( Buffer,"test" ) )
    {
        cls();
        load_test();
        Delay();
        cls();
        Initial();
    }
```

```
void load_test()
{
    Program_Num ++;
    another_load(Segment,16);
    Segment += 0x1000;
}
```

运行的时候输入 test 就可以运行本次的用户程序了。

以下介绍编译的过程：就像之前的实验一样，首先使用 DOSBOX 先把内核编译，再使用 nasm 把用户程序和引导程序编译出来，放到一个虚拟软盘的对应的位置。

```
D:\>TCC -LINCLUDE CMAIN
Turbo C  Version 2.01  Copyright (c) 1987, 1988 Borland International
cmain.c:
Warning kdata.h 117: Parameter 'num' is never used in function init
Turbo Link  Version 2.0  Copyright (c) 1987, 1988 Borland International
c0s.obj : unable to open file

        Available memory 412588

D:\>TASM MYOS.ASM -O MYOS.OBJ
Turbo Assembler  Version 4.1  Copyright (c) 1988, 1996 Borland International

Assembling file:   MYOS.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  454k
```

```
D:\>TASM KLIBC.ASM -O KLIBC.OBJ
Turbo Assembler  Version 4.1  Copyright (c) 1988, 1996 Borland International

Assembling file:   KLIBC.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  461k


D:\>TLINK /3 /T MYOS.OBJ KLIBC.OBJ CMAIN.OBJ,OS.COM,,
Turbo Link  Version 2.0  Copyright (c) 1987, 1988 Borland International
```

```
D:\>TCC -LINCLUDE PROG5
Turbo C  Version 2.01  Copyright (c) 1987, 1988 Borland International
prog5.c:
Warning prog5.c 50: 'ch' is assigned a value which is never used in function dma
in
Turbo Link  Version 2.0  Copyright (c) 1987, 1988 Borland International
c0s.obj : unable to open file

        Available memory 452868

D:\>TASM PROG5R.ASM -O PROG5R.OBJ
Turbo Assembler  Version 4.1  Copyright (c) 1988, 1996 Borland International

Assembling file:   PROG5R.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  467k


D:\>TLINK /3 /T PROG5R.OBJ PROG5.OBJ,PROG5.COM,,
Turbo Link  Version 2.0  Copyright (c) 1987, 1988 Borland International
```

以上是内核，和测试的用户程序的编译过程，引导盘和其他用户程序的编译过程在此略过。

运行的结果如下：

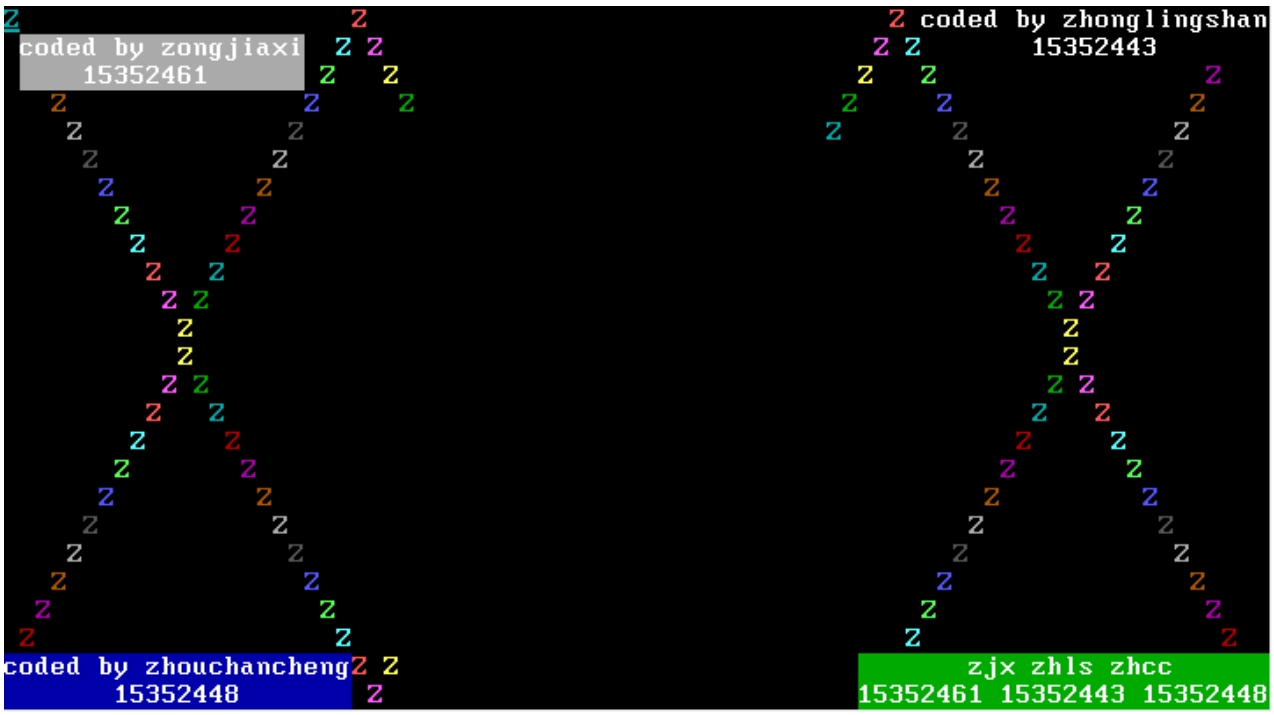
可以看见，我在程序中设定的字符为 `char str[80]="129djwqhdsajd18087";`

输出的结果是 18，符合。而且从输出的内容上看，程序运行的顺序也是正确的。首先进入了 pid=1 的情况，输出了一句“LetterNr= ？”，然后进入 wait，换到子进程中，输出“Counting……”把值赋到全局

变量中保存好，然后调用 exit 结束，回到了父进程中 wait 的下一句，输出“LetterNr=18”，然后到了主函数的结尾输出“Program Finish”。

不过由于时间问题，没能够改成输出字符的形式，有点可惜，如果想要修改字符串的话就只能够在 c 中进行修改，在重新编译了。

然而，这次的实验还保留了之前的功能，可以看到轮转程序依然能够正常运行。



我在修改一下用户程序中的字符串的长度，再做一次测试：

```
char str[80]="wrq32423r32rerq3rq23r23rwqr3r2r23r23r3r3r3qr32";
```

可以看出，输出的结果也是正确的。

【实验总结】

这一次的实验其实比上一次的实验难度大一点，主要是因为我门在这一次的实验中遇到了不少奇怪的 bug，本来按照代码的逻辑来说是应该输出正确的结果的，但是却弄了很久都不行，一直找不到错误在哪里，后来发现了很多可疑的问题，比如说在 fork 之前没有成功把进程的状态保存到 pcb 里面，或者是恢复的时候没有做好，导致 PCB 里面的寄存器的值没有放回到寄存器中，导致无法运行子进程。后来才发现，原来有很多地方都是很讲究的，比如说 save 和 restart 就不能够使用原型中的代码作为参考，因为在压栈的时候会出现一些错误，要是思维出现混乱的话很容易混淆。这一次五状态进程模型实验，我本以为只是简单的在二状态进程上添加 3 种状态就可以了，但是实际上却是差得很远。如果上一次二状态进程实验中做得不够好，那么这一次的实验就有许多需要补足的地方。只有深刻的理解到进程与进程间

的切换到底是一个什么样的过程，还有对于进程的复制到底又有哪些要做的事。复制一个进程不可能直接把所有寄存器直接复制，因为在进程运行的过程中，会产生数据，并且会发生多次的压栈和出栈操作，那么在内存中该进程的栈段中就会产生新的数据。进行复制的时候，如果是完全复制的话，那么两个进程其实就只相当于一个进程了，因为两个进程是一模一样的，数据段是一样的，栈顶的位置也是一样的，当一个进程在运行的时候，另一个进程的数据也会受到改变（其实只是针对堆栈里面的数据来理解，因为毕竟寄存器的值都存在 pcb 中，可是其存放的那些本应该独立分开，不能够被改变的数据却被别的进程给修改了，那么复制子进程这一个过程将变得毫无意义了）。因此，复制产生子进程一定要重新找一个空间，然后把父进程的堆栈的内容完全复制，这样子就可以避免上述的情况，不会相互影响到了。总而言之，这一次的实验里面，真的学习到了很多，这会又使我理解操作系统理论课上学到的知识得到了更加深刻的巩固了。

附录：

源代码：

用户程序 ：

prog1.asm

prog2.asm

prog3.asm

prog4.asm

int_caller.asm

prog5.c

prog5r.asm

内核代码 ：

klibc.asm

kliba.asm

myos.asm

kdata.h

cmain.c

引导扇区代码：

boot.asm

编译文件：

prog1.bin

prog2.bin

prog3.bin

prog4.bin

int_caller.bin

klibc.obj

myos.obj

cmain.obj

os.com

boot.bin

prog5.obj

prog5r.obj

prog5.com

镜像文件：

144mb.img