School of Data and Computer Science, Sun Yat-sen University

《操作系统实验》

实验六：进程控制与通信

17341190 叶盛源

2019 年 6 月 15 日

Contents

操作系统实验报告

1 实验目的:

在内核中实现五状态进程模型,实现一些进程控制原语，进程创建fork()，进程退出exit()和父子进程通信wait()

2 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。

4) 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。

3 实验方案:

3.1 实验环境：

1) 实验运行环境：Windows10

2) 虚拟机软件：VMware Function和DOSBox

3) TCC+Tasm+TLink 混合编译链接

4) NASM 编译器

3.2 实验工具：

1) 汇编语言：NASM、TASM

2) 文本编辑器：VScode、notepad++

3) 软盘操作工具：WinHex

3.3 实验原理和思想：

1) 五状态进程模型：

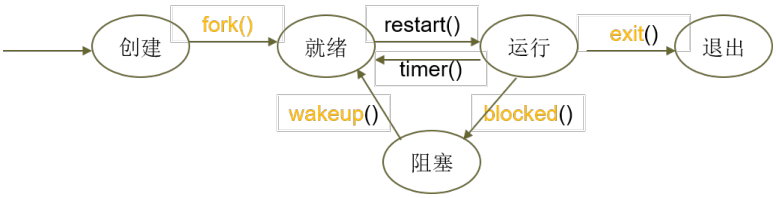

Figure 1: 五状态进程模型

i) 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

ii) 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

iii) 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。

2) 进程控制的基本操作

i) 扩展PCB结构，增加必要的数据项

ii) 进程创建do_fork()原语，在c语言中用fork()调用

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

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

v) 进程唤醒wakeup原语（内核过程）

vi) 进程唤醒blocked原语（内核过程）

3) 进程创建do_fork()原语

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，放在进程PCB的ax寄存器。

i) 寻找一个自由的PCB块，如果没有，创建失败，调用返回－1；

ii) 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。

iii) 产生一个唯一的ID作为子进程的ID，存入至PCB的相应项中。

iv) 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；

v) 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];

vi) 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。

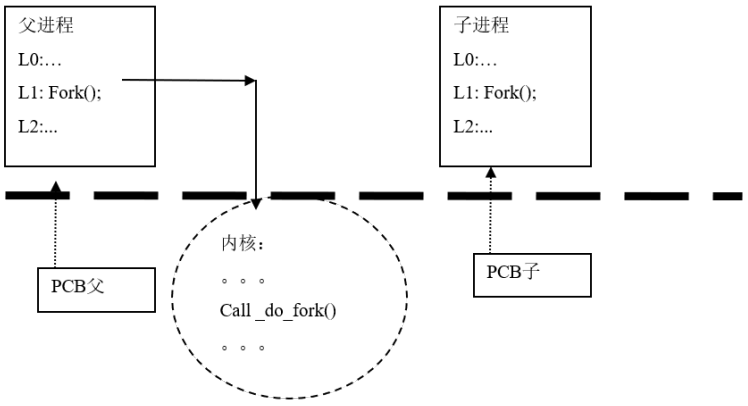

Figure 2: 进程创建

4) 进程等待do_wait()原语

i) 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。

ii) 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。

iii) 相应调度模块也要修改，禁止将CPU交权给阻塞状态的进程。

5) 进程退出do_exit()原语

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。

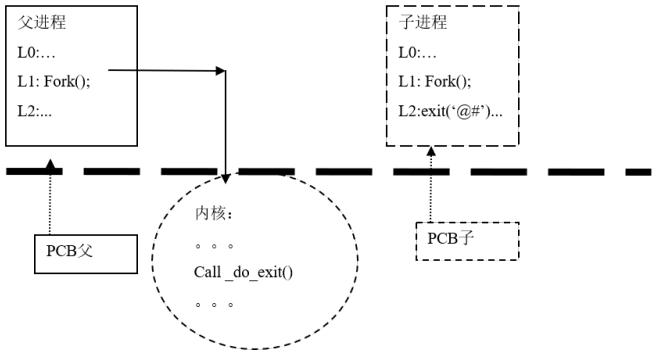

Figure 3: 进程退出

4 实验过程：

1) 修改后PCB块结构

和原来的PCB块大致类似，但为了方便实现五状态的模型，我们需要增加一些属性值：首先要增加一个Used位，它用来标志这个PCB块是否被使用了，因为新增了状态，存在了退出态和阻塞态，我们不能像原来那样直接找下一个PCB块来调度运行，而是找下一个准备好的进程来运行。还增加了一个FatherID，用来表示这个进程的父进程是谁，一般由内核调度的默认初始化为0号，又进程fork出来的就是fork出它的父进程的PCB块ID号。

```
typedef struct RegisterImage{
        int SS;
        int GS;
        int FS;
        int ES;
        int DS;
        int DI;
        int SI;
        int BP;
        int SP;
        int BX;
```

```
        int DX;
        int CX;
        int AX;
        int IP;
        int CS;
        int FLAGS;
}RegisterImage;

typedef struct PCB{
        RegisterImage regImg;
        int Process_Status;
        int Used;
        int FatherID;
}PCB;
```

2）修改后的调度函数

因为新增了五种状态，所以调度函数有很大的区别。如果原来是运行的，我们就要将他变为就绪态。并继续往下找到下一个准备好的块，如果都找不到就调度原来的进程，但如果找到了就调度新的进程。同时第一个运行的NEW状态还是需要单独讨论，在被调度后在汇编中实现将它变会RUNNING状态。假如此时没有用户程序在运行，调度函数会恢复内核状态，交回给内核操控。如下：

```
void Schedule()
{
    int End = CurrentPCBno;
    if (pcb_list[CurrentPCBno].Process_Status == RUNNING)
        pcb_list[CurrentPCBno].Process_Status = READY;
    CurrentPCBno += 1;
    if (CurrentPCBno >= MAX_PCB_NUMBER)
            CurrentPCBno = 1;
    while (CurrentPCBno != End)
    {
        if (pcb_list[CurrentPCBno].Used == 1)
        {
            if (pcb_list[CurrentPCBno].Process_Status == READY)
            {
              pcb_list[CurrentPCBno].Process_Status = RUNNING;
```

```
                return ;
            }
            else if (pcb_list[CurrentPCBno].Process_Status ==
                                                        NEW)
                return ;
        }
        CurrentPCBno++;
        if (CurrentPCBno >= MAX_PCB_NUMBER)
            CurrentPCBno = 1;
    }
    if (pcb_list[CurrentPCBno].Used == 1 &&
            pcb_list[CurrentPCBno].Process_Status == READY)
        pcb_list[CurrentPCBno].Process_Status = RUNNING;
    else
        CurrentPCBno = 0;
}
```

3) fork函数的实现

这个函数是这个实验新增的函数。这个函数的目的是从当前进程中创建一个子进程，子进程和父进程有一模一样的寄存器值，栈的值，除了栈的指针不一样。所以他们会从同一个地方完全一样的开始，但因为他们的ID不同，会导致他们的命运不同，这也是这个实验的核心目的。

在实现这个函数前，我们需要实现两个辅助函数，一个是拷贝PCB块的函数，一个是拷贝栈的函数。拷贝PCB块的函数我们在C语言中实现，我们通过直接拷贝一个指针的PCB块值给另外一个实现，因为SS寄存器在PCB块初始化的时候已经为它找好了一个空闲的段，则不需要再给它赋值，因为它要和父进程的栈区别。如下:

```
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.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;
}
```

拷贝栈的函数我们在汇编语言中实现，利用loop循环反复拷贝栈的值，实现目标，如下：

```
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]
    mov ds, ax
    mov ax, word ptr [bp+16]
    mov es, ax
    mov di, 0
    mov si, 0
    mov cx, 100h-4
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
```

接着我们就可以开始实现fork函数了，我们先找到一个空余的PCB块（如果找不到返回-1），否则将父亲的PCB块值都拷贝到新的PCB块中，然后拷贝栈。最后我们需要设置好子进程的ID值，并利用AX返回值，还要把子进程标注为USED状态，代码实现如下：

```
int do_fork()
{
    int i = Findempty();
    if (i == -1){
        pcb_list[CurrentPCBno].regImg.AX = -1;
        return ;
    }
    Program_Num++;
    PCBcopy(&pcb_list[i], &pcb_list[CurrentPCBno]);
    stackcopy(pcb_list[i].regImg.SS,
                        pcb_list[CurrentPCBno].regImg.SS);
    pcb_list[i].FatherID = CurrentPCBno;
    pcb_list[i].Used =  1;
    pcb_list[i].regImg.AX = 0;
    pcb_list[CurrentPCBno].regImg.AX = i;
    pcb_list[CurrentPCBno].Process_Status = READY;
    Switch();
}
```

4) exit函数的实现

exit函数是结束一个进程，它需要将进程调整为退出态，同时初始化这个PCB块。而且如果这个进程是一个非内核进程派生的子进程，我们还需要把它的父进程调整成READY状态唤醒。操作结束后重新调度并运行函数

```
int do_exit(int ch)
{
    int k;
```

```
    int FatherID = pcb_list[CurrentPCBno].FatherID;
    pcb_list[CurrentPCBno].Process_Status = EXIT;
    init(&pcb_list[CurrentPCBno], (CurrentPCBno+1)*0x1000,
                                                    0x100);
    if (FatherID != 0){
        pcb_list[FatherID].Process_Status = READY;
        pcb_list[FatherID].regImg.AX = ch;
    }
    Program_Num --;
    Segment=0x2000;
    for (k=0; k<15; ++k)
        delay();
    Schedule();
    Switch();
}
```

5) 将上述原子操作加入系统调用int 21h

因为我们要求上述的三个操作都是原子操作，所以我们必须在运行的时候不会被时钟中断打断，最简单的方法就是将他们加入int 21h系统中断调用中，因为这样就可以保证不会被时钟中断，也可以在后面的测试过程中，让测试函数调用。封装过程和以前一样，都是比较ah值后利用jmp函数跳转到目标位置再call C库中的外部函数实现。代码举例int 9如下：

```
next8:
    cmp ah,9
    jnz next9
    jmp forking
next9:
...
...
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
    mov ds, ax
    mov es, ax
    call near ptr _Save_Process
    call near ptr _do_fork
    iret
```

5 实验测试和结果：

接下来我们需要按照老师的要求编写测试文件来测试我们的进程创建等操作编写正确性。参考了老师的代码，我先编写了一个C语言的主测试函数，然后又编写了一个汇编的主程序，我们用它来调用C语言的函数来进行测试。同时因为我们需要单独把这几个文件链接编译成一个com文件，所以我还编写了两个小规模的C语言库和汇编语言的库供我调用输出字符串或是测试正确性。

```
if (pid)
{
    print("Father process: This is the father process.\r\n");
    print("Father process: My subprocess's ID is: ");
    printnumber(pid);
    print("\r\n");
    wait();/* 等待子进程运行 */
    print("Father process: The numbers of letters in string is: ");
    printnumber(44);
    print("\n\r");
    delay();
    exit(0);
}
```

Figure 4: 父进程路线

```
/* 子进程传回来的fork是0 执行以下函数 */
else
{
    print("Subprocess: This is the subprocess.\r\n");
    for (i=0; str[i]; ++i)
        letterNr++;
    print("Subprocess: The result that subprocess get after calculating is ");
    printnumber(letterNr);
    print("\r\n");
    exit(0);
}
```

Figure 5: 子进程路线

C语言主函数中我先在父进程中输出了一些语句，然后用fork派生一个子进程。子进程利用上面提到的AX会返回值，根据值的关系我们可以区分它是子进程还是父进程或者是创建失败，子进程和父进程有两条不一样的路线，父进程

在创建完后会利用wait等待子进程结束，子进程会输出字符串的长度并在结束后会唤醒父进程，同时父进程也输出字符串的长度的准确值，比对后就可以知道这几个原子函数是否实现成功。

在实现完测试函数之后，我们还需要在主程序中增加一个命令来调用这个新的测试用户程序，我用test指令来代表，输入只有就会运行这个用户程序并输出结果，代码如下：

```
/* 测试程序默认放在第一个PCB块中 */
void run_testPro_EX7()
{
    sector_number=7;
    RunTestProm(Segment,sector_number);
    Segment+=0x1000;
    Program_Num=1;
    pcb_list[1].Used = 1;
}
```

Figure 6: 编写测试指令1

```
else if(com[0]=='-'&&com[1]=='t'
{
    cls();
    run_testPro_EX7();
    Delay2();
    cls();
}
```

Figure 7: 编写测试指令2

最后我们就可以在内核中调用测试指令来查看运行结果，结果如下图。可以看到父进程在输出了一些字符串之后，就wait等待子进程结束，子进程循环计算了字符串长度后输出，然后exit并让父进程恢复，于是父进程继续运行并输出了字符串正确长度，比对后一致，可以知道fork等函数功能实现成功。

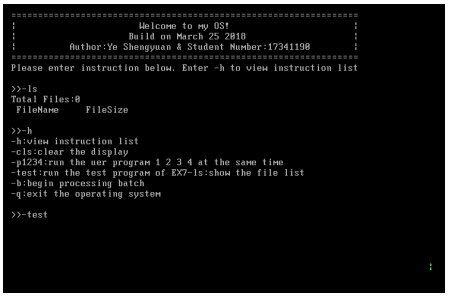

Figure 8: 测试结果1

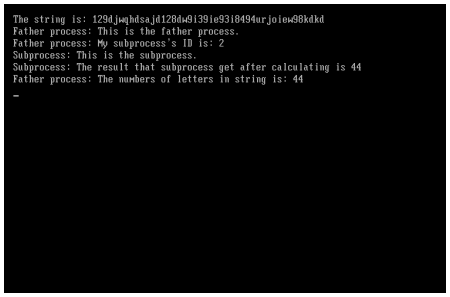

Figure 9: 测试结果2

6 实验心得和总结：

本次实验是要在上一次实验的基础上实现一个五进程的模型，这次实验比起上一次来说还是算比较简单的，可能是因为读懂了上次老师给的二进程切换

的代码后，感觉对操作系统的理解已经更进了一步，对整个框架已经有了了解，所以编写的时候更流畅，不同文件之间切换编写也比较明确，也不会漏掉很多的细节，因此debug的时间没有之前要长。

但做实验的过程中也并不是十分顺利的，因为要从二进程到五进程的模型也是需要一个巨大的改变，因为原来为了简单，在二进程的实现上有一些偷工减料，但这次五进程模型已经是一个很严谨的框架，导致这次有很多地方需要改动，比如我在请教了老师同学后知道了在PCB块中增加一个USED状态来表示是否被使用，而这个在上一次的实验中是多不必要的，但这次有了这个会让我的实现更简单清晰。同时上次的调度函数十分的简单，其实就是讨论了第一次调度的进程和不是第一次调度的进程的区别，但这次不一样，我需要考虑很多的状态，并让他们自然的切换，在一些努力和查看了一些参考文献后，渐渐明白了编写的原理并最后取得了成功。在编写这次新增原子函数的时候，我也遇到了很多困难，其中fork函数更是十分的困难，因为在实现fork前我需要实现一些辅助函数，虽然老师的ppt上已经有了模版代码，但我还是需要理解才能把它灵活的用在我的代码中，经过仔细思考分析，我才明白了栈拷贝和PCB拷贝这两个函数的写法，在解决了这两个函数其他问题也几乎迎刃而解，之后的wait函数和exit函数其实就是C语言的使用过程了。

最后在测试函数上我也花费了一些功夫，因为编写测试函数的时候一开始想到直接用C语言就可以了，但发现并不会用C语言弄成com文件，而且网上的资料也不多，最后还是决定了模仿刚开学编写的loader函数，用它来引入我们的C函数，这样就可以用老方法链接成obj文件再生成com文件。当com文件被引入内存，这个时候就可以使用int 21h系统调用，查中断向量表并调用我们在内核中实现的fork等函数实现最终的测试目标。当然测试结果也比较满意，符合预期的结果。

因为这次实验主要目的是实现fork等函数，所以我将没有必要的开机批处理程序和一些无关紧要会影响效率的代码暂时注释掉了，方便老师和助教检查查阅，这次实验又学到了很多，感谢老师和助教的付出，期待下一个实验，继续加油!

7 参考文献:

1) nasm手册

2) https://blog.csdn.net/longintchar/article/details/79511747

3) https://blog.csdn.net/cielozhang/article/details/6171783/

4) https://blog.csdn.net/lulipeng_cpp/article/details/8161982

5) https://www.cnblogs.com/alwaysking/p/7789282.html

6) https://blog.csdn.net/cielozhang/article/details/6171783

7) https://blog.csdn.net/lulipeng_cpp/article/details/8161982 https://stackoverflow.com/questions/12882342/override-default-int-9h