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

《操作系统实验》
实验六：进程模型

17341190 叶盛源

2019 年 5 月 9 日

Contents

操作系统实验报告

1 实验目的:

1) 在内核中实现多进程的二状态模型，理解和使用进程和进程控制块的创建方法，还有时间片轮转的调度过程
2) 实现解释多进程的控制台命令，建立相应进程并能启动执行。
3) 至少一个进程可用于测试前一版本的系统调用，搭建完整的操作系统框架，为后续实验项目打下扎实的基础。

2 实验要求:

保留原型原有特征的基础上，设计满足下列要求的新原型操作系统：

1) 定义进程表，进程数量为4个。
2) 内核一次性加载4个用户程序运行时，采用时间片轮转调度进程运行，用户程序的输出各占1/4屏幕区域，信息输出有动感，以便观察程序是否在执行。
3) 在原型中保证原有的系统调用服务可用。再编写1个用户程序，展示系统调用服务还能工作。

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) 二状态进程模型：

i) 进程模型就是实现多道程序和分时系统的一个理想的方案，就是实现多个用户程序并发执行。在进程模型中，操作系统可以知道有几个用户程序在内存运行，每个用户程序执行的代码和数据放在什么位置，入口位置和当前执行的指令位置，哪个用户程序可执行或不可执行，各个程序运行期间使用的计算机资源情况等等。

ii) 二状态进程模型有两个状态，即执行态和等待态。目前进程的用户程序都是COM格式的，是最简单的可执行程序。进程仅涉及一个内存区、CPU、显示屏这几种资源，所以进程模型很简单，只要描述这几个资源。以后扩展进程模型解决键盘输入、进程通信、多进程、文件操作等问题。

2) 初级进程

i) 现在的用户程序都很小，只要简单地将内存划分为多个小区，每个用户程序操作系统原理实验占用其中一个区，就相当于每个用户拥有独立的内存。根据我们的硬件环境，CPU可访问1M内存，我们规定MYOS加载在第一个64K中，用户程序从第二个64K内存开始分配，每个进程64K。

ii) 对于键盘，我们先放后解决，即规定用户程序没有键盘输入要求，我们将在后继的关于终端的实验中解决

iii) 对于显示器，我们可以参考内存划分的方法，将25行80列的显示区划分为多个区域，在进程运行后，操作系统的显示信息是很少的，我们就将显示区分为4 个区域。如果用户程序要显示信息，就规定在其中一个区域显示。当然，理想的解决方案是用户程序分别拥有一个独立的显示器，这个方案会在关于终端的实验中提供。

iv) 文件资源和其它系统软资源，则会通过扩展进程模型的数据结构来实现，相关内容将安排在文件系统实验和其它一些相关实验中。

3) 进程表

初级的进程模型可以理解为将一个CPU模拟为多个逻辑独立的CPU。每个进程具有一个独立的逻辑CPU。同一计算机内并发执行多个不同的用户程序，MYOS 要保证独立的用户程序之间不会互相干扰。为此，内核中建立一个重要的数据结构：进程表和进程控制块PCB。现在的PCB包括进程

标识和逻辑CPU模拟。逻辑CPU中包含8086CPU的所有寄存器，AX、BX、CX、DX、BP、SP、DI、SI、CS、DS、ES、SS、IP、FLAG，这些用内存单元模拟。逻辑CPU轮流映射到物理CPU，实现多道程序的并发执行。可选择在汇编语言或是C语言中描述PCB。

4) 进程交替执行原理

i) 在以前的原型操作系统顺序执行用户程序，内存中不会同时有两个用户程序，所以CPU控制权交接问题简单，操作系统加载了一个用户到内存中，然后将控制权交接给用户程序，用户程序执行完再将控制权交接回操作系统，一次性完成用户程序的执行过程

ii) 采用时钟中断打断执行中的用户程序实现CPU在进程之间交替

iii) 简单起见，我们让两个用户的程序均匀地推进，就可以在每次时钟中断处理时，将CPU控制权从当前用户程序交接给另一个用户程序

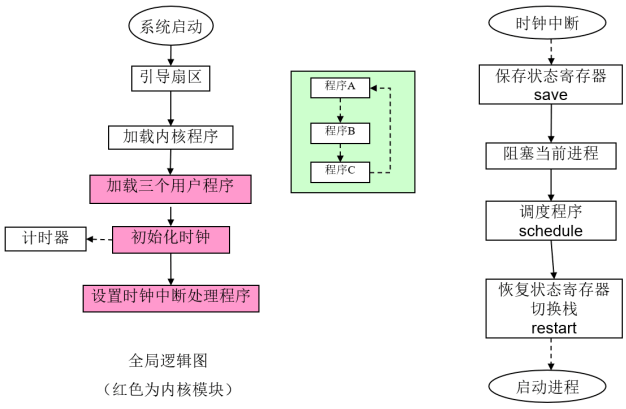

Figure 1: 实现进程模型的系统框架

5) 内核

i) 利用时钟中断实现用户程序轮流执行

ii) 在系统启动时，将加载两个用户程序A和B，并建立相应的PCB。

iii) 修改时钟中断服务程序每次发生时钟中断，中断服务程序就让A换B或B换A。要知道中断发生时谁在执行，还要把被中断的用户程序的CPU寄存器信息保存到对应的PCB中，以后才能恢复到CPU中保证程序继续正确执行。中断返回时，CPU控制权交给另一个用户程序

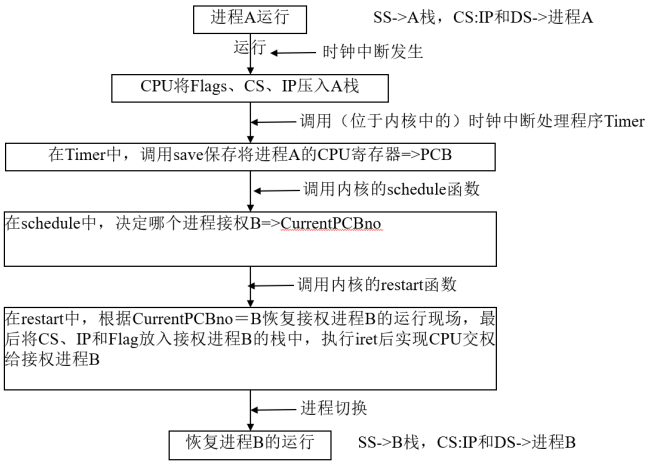

Figure 2: 时间中断服务程序

以下三个部分是这个多进程模型的核心部分!

6) **现场保护save过程**

i) Save是一个非常关键的过程，保护现场不能有丝毫差错，否则再次运行被中断的进程可能出错。

ii) 涉及到三种不同的栈：应用程序栈、进程表栈、内核栈。其中的进程表栈，只是我们为了保存和恢复进程的上下文寄存器值，而临时设置的一个伪局部栈，不是正常的程序栈

iii) 在时钟中断发生时，实模式下的CPU会将FLAGS、CS、IP先后压入当前被中断程序（进程）的堆栈中，接着跳转到（位于kernel内）时钟中断处理程序（Timer函数）执行。注意，此时并没有改变堆栈（的SS和SP），换句话说，我们内核里的中断处理函数，在刚开始时，使用的是被中断进程的堆栈

iv) 为了及时保护中断现场，必须在中断处理函数的最开始处，立即保存被中断程序的所有上下文寄存器中的当前值。不能先进行栈切换，再来保存寄存器。因为切换栈所需的若干指令，会破坏寄存器的当前值。这正是我们在中断处理函数的开始处，安排代码保存寄存器的内容

v) 我们PCB中的16个寄存器值，内核一个专门的程序save，负责保护被中断的进程的现场，将这些寄存器的值转移至当前进程的PCB中。

7) **现场恢复:restart过程**

i) 用内核函数restart来恢复下一进程原来被中断时的上下文，并切换到下一进程运行。这里面最棘手的问题是SS的切换。

ii) 使用标准的中断返回指令IRET和原进程的栈，可以恢复（出栈）IP、CS和FLAGS，并返回到被中断的原进程执行，不需要进行栈切换。

iii) 如果使用我们的临时（对应于下一进程的）PCB栈，也可以用指令IRET完成进程切换，但是却无法进行栈切换。因为在执行IRET指令之后，执行权已经转到新进程，无法执行栈切换的内核代码；而如果在执行IRET指令之前执行栈切换（设置新进程的SS和SP的值），则IRET指令就无法正确执行，因为IRET必须使用PCB栈才能完成自己的任务。

iv) 解决办法有三个，一个是所有程序，包括内核和各个应用程序进程，都使用共同的栈。即它们共享一个（大栈段）SS，但是可以有各自不同区段的SP，可以做到互不干扰，也能够用IRET进行进程切换。第二种方法，是不使用IRET指令，而是改用RETF指令，但必须自己恢复FLAGS和SS。第三种方法，使用IRET指令，在用户进程的栈中保存IP、CS和FLAGS，但必须将IP、CS和FLAGS 放回用户进程栈中，这也是我们程序所采用的方案。

8) **进程调度：Scheduler过程**

i) 我们采用时间片轮转法，每次运行时间到了就会调用schedule方法来切换到下一个要运行的用户程序或内核程序，具体的代码分析会在后面介绍。

4 实验过程和结果：

1) PCB块结构

我们的目标是建立一个寄存器的进程映象，这样才能让我们在切换进程前将上一个进程的所有必要的信息保存进它的PCB块中，这就是进程上下文，他将在我们restart这个进程的时候被重新加载进入各个寄存器中恢复上下文。定义一个PCB结构体用来装进程印象块和这个程序当前的运行状态。

```
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;
}PCB;
```

2) 在内存中安排进程

我们将四个用户进程存储在从0x2000地址空间开始的内存中，每个内存占了64k的空间，而内核单独存放。如下图。

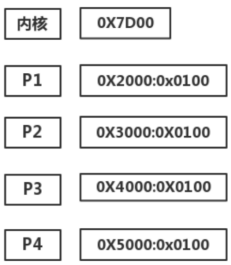

Figure 3: 内存中进程安排

我们需要在内核中定义一个初始化函数init来供主函数调用来初始化我们的所有PCB块，传入我们想要放的内存的位置和偏移量然后进行初始化。

```
void init(PCB* pcb,int segement, int offset)
{
        pcb->regImg.GS = 0xb800;
        pcb->regImg.SS = segement;
        pcb->regImg.ES = segement;
        pcb->regImg.DS = segement;
        pcb->regImg.CS = segement;
        pcb->regImg.FS = segement;
        pcb->regImg.IP = offset;
        pcb->regImg.SP = offset - 4;
        pcb->regImg.AX = 0;
        pcb->regImg.BX = 0;
        pcb->regImg.CX = 0;
        pcb->regImg.DX = 0;
        pcb->regImg.DI = 0;
        pcb->regImg.SI = 0;
        pcb->regImg.BP = 0;
        pcb->regImg.FLAGS = 512;
        pcb->Process_Status = NEW;
}
```

我们先将显存地址GS赋值为b800h，接着要把segment的值给到所有的段寄存器，而IP则赋值为偏移量，这里栈的指针要为sp-4，是为了避免栈在粗放数据的时候和代码段的内容出现重叠，影响进程的执行。

我们在主进程中也要定义一个函数用来给init传入要初始化的PCB块对应的内存地址。我们在多进程运行完后，一定要重新调用init函数重新初始化PCB块，因为PCB块的内容在程序运行的过程中很多都被改变，需要重新定位到新的内存地址空间才能运行另外的程序。

3) 加载和运行用户程序

因为实现了多进程的模型，那我原来实现的串行运行用户程序就可以被替换成多进程的方式运行。当我只运行一个程序的时候，其实就相当于一个进程在运行的多进程模型。代码如下：

```
else if(com[0]=='-'&&com[1]=='p')
{
    cls();
```

```
    Random_Load(com);
    Delay();
    cls();
}
```

当收到指令为-p加要运行的用户程序编号的时候，就会进入这个子分支，并调用在函数 Random_Load 函数。在这个函数中，我们将对应编号的用户程序加载到从0x2000开始的内存单元，偏移量为100h，每加载一个，内存单元就加0x2000(64k),同时 Program_Num++ 当这个值不为0的时候，在时钟中断中就会从内核模式跳转进入用户程序模式，这个具体在下一部分会详细描述。代码如下：

```
while(com[i]!='\0'&&(com[i]>=0+'0'&&com[i]<=9+'0'))
{
        num=com[i]-'0';
        if( Segment > 0x6000 )
        {
            printf("\r\nThere have been 4 Processes !");
            break;
        }
        sector_number=num;
        another_RunProm(Segment,sector_number);
        Segment+=0x1000;
        Program_Num++;
}
```

another_RunProm是一个是现在汇编系统调用库内的函数，他用来加载我们的用户程序，我们需要给它传入要加载到的段地址，和用户程序所在的扇区号，他就会将它加载进入内存。

同时我们的用户程序也需要做出修改：因为我们是通过跳转地址的方法运行用户程序，所以在用户程序中就不能再使用 ret 来返回，可以选择其他的方式替代比如说换成 jmp $ 让它在原地等待，防止它返回到奇怪的地方。同时最顶部要加上org 100h才能对应上100h的偏移地址。

4) **用时间中断实现进程切换**

首先我们需要设置时钟中断，我们先调用一个 SetTimer 的过程，将时钟中断的频率修改为每秒20次中断，50ms一次，然后再利用上一章学过的

修改中断向量表的方式，将我们自己编写的时钟中断程序的地址替换中断向量表中20h的位置，代码如下：

```
call near ptr _SetTimer
xor ax,ax
mov es,ax
mov word ptr es:[20h],offset Pro_Timer
mov ax,cs
mov word ptr es:[22h],cs
```

接着我们需要自己编写时钟中断用来实现进程之间的切换。当每个时钟中断到来的时候，我们就需要切换一次用户的进程，因此我们需要保护当前上下文并调度下一个用户程序的上下文来重新启动运行，时钟中断包含两个大部分，当 Program_Num 的大小为0的时候，就会进入内核模式并跳转到我们上次实验实现的风火轮中，这里不再详细描述代码。而当我们 Program_Num 的大小大于0的时候，我们就会进入用户模式，开始运行用户程序。

首先我们需要设置一个总的调度次数，没切换一次进程调度次数就会加一，当调度次数加到我们的指定次数的时候。我们就会执行一些初始化的指令并退出，例如将 Program_Num 的大小设置成0，这样接下来的时钟中断就不会再进入用户模式，还有将刚刚的调度次数计数器恢复成0等，这些操作是为了方便下一次用户输入指令后再执行其他的用户程序做好了准备工作。代码如下：

```
cmp word ptr[Finite],400
jnz Lee
mov word ptr[_CurrentPCBno],0
mov word ptr[Finite],0
mov word ptr[_Program_Num],0
mov word ptr[_Segment],2000h
jmp Pre
```

我们通过将要保护的内容push进入用户栈后，调用 Save_Process 函数来保存所有当前程序的上下文，接着我们需要一个调度函数 _Schedule ，来调度下一个进程，并获取它的PCB块。如果是new状态说明这个程序是第一次运行的我们需要单独讨论，其他情况我们只需要把ready状态修改成running状态，并将它的PCB块指针传来。

接着我们需要restart我们的下一个用户进程，我们利用offset从PCB块中找到对应的寄存器的偏移量，并把对应内存单元的值给到我们的寄存器中。因为我们不能影响上一个程序的用户栈内容，所以我们要切换到这个程序的用户栈中执行这些push和pop操作，因此我们需要先拿出sp和ss的值进入寄存器。

```
inc word ptr[Finite]
call near ptr _Current_Process
mov bp, ax
mov ss,word ptr ds:[bp+0]
mov sp,word ptr ds:[bp+16]
```

这个时候我们需要考虑，因为如果进程是第一次运行的话，它的sp和ss是初始化的时候初始的，一定是正确的，所以我们判断它的状态是new的时候就可以继续正常运行。但如果不是，我们会发现，在上一次这个用户程序运行完准备保存上下文的过程中，在传入sp的值进入 Save_Process 之前，我们先压入了一些其他的寄存器的值，还有cs和ip和psw。则上一次保存的时候保存的sp并不是准确的，我们需要加上16调整到该用户程序进入时钟中断前的那一刻的sp的值才是我们需要的准确的栈的值，当我们加上16再跳转回来才能正常的重启这个用户程序。代码如下

```
  jnz No_First_Time
        ...
        ...
No_First_Time:
  add sp,16
  jmp Restart
```

接着我们就可以正常的重启这个新的用户程序了，这个时候我们只需要正常的结束时钟中断，先发送AEOI，再用iret返回，返回的时候会把此时栈顶的cs ip和psw返回到寄存器中，这个时候就完全的恢复了要运行的用户程序的上下文，此时正常执行，并自动在下一个时钟中断进行相同操作，直到调度次数满了然后结束并恢复内核模式。

直接使用老师的风火轮会发现转速太快，于是这里还进行了修改，将风火轮设置为5次时钟中断才执行一次，这样就可以看清楚风火轮的运行情况了。代码如下：

```
  ccount db 5
```

```
  dec byte ptr [ds:ccount]
  jnz end1
        ...
        ...
end1:
  ret
```

5) 实验效果展示：

在刚进入内核的时候，原本会有一个开机程序，但因为串行的程序执行方式被串行所替代，所以我将它移入主函数在加载memu之前使用并行的方法运行批处理文件，修改后可以像往常串行一样执行单个或连续执行多个指令或用户程序，如下图：

Figure 4: 串行的开机程序

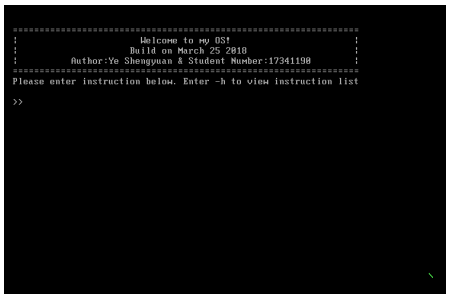

Figure 5: 批处理结束后进入内核

在内核中我们输入p1234同时运行四个用户程序，也可以输入4个数字的任意组合运行其中几个程序。下图展示效果：

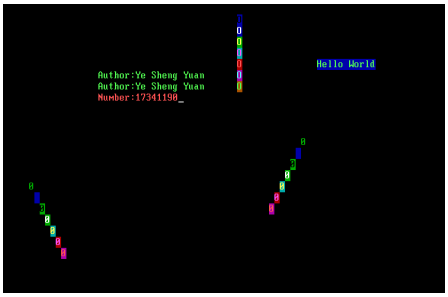

Figure 6: 多进程运行–1

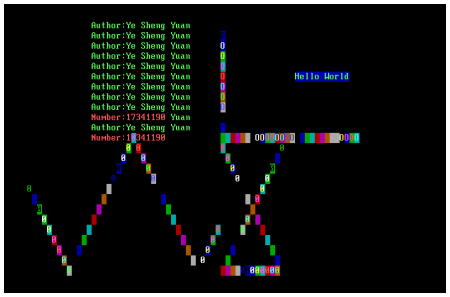

Figure 7: 多进程运行–2

当然我们原来的批处理串行功能还能使用，当然运行的模式已经从原来的简单加载用户程序并调用变成了现在的并行模型下的运行方式，如下图：

Figure 8: 串行运行指令输入

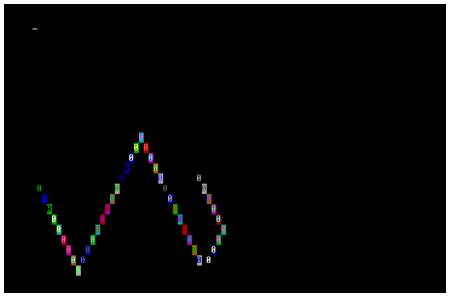

Figure 9: 串行运行单个程序–1

Figure 10: 串行运行单个程序–2

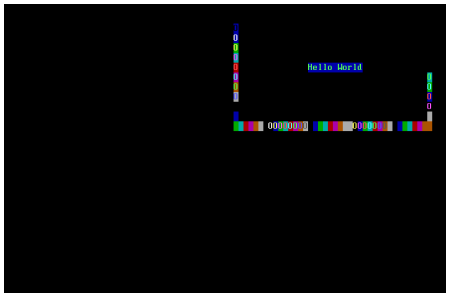

Figure 11: 串行运行单个程序–3

5 实验心得和总结：

这次实验和之前的相比更加的困难。他要我们将在理论课堂上学到的多进程的相关知识，运用在我们的实验里。最重要的是，我们不仅需要会写代码会debug，我们还需要会思考，我们要想清楚理论和逻辑再动手做。

刚开始，我尝试自己编写这个代码，但我太拘泥于用进程的状态来实现切换不同的进程，并且总想着让进程编号和进程控制块在数组中的下标序号对应，因此进程控制块总是在保存或者调度的时候出错，而我又没有去下载一些汇编的调试工具，全靠输出调试，因此搞了很久，并没有解决问题。后来我参考了

老师的代码，才明白可以这么灵活的处理这个问题。我们只需要每次都按照顺序把进程加载进入内存，不必在意进程的编号顺序，调度的时候只需要正常的让序号加一就可以，不用再做一些状态的判断等麻烦的操作，每个进程都是平等的，这样也大大提高了代码的可移植性，也可以很容易的就跑别人的用户程序了。因为自己的尝试花费了很多的时间和精力都没有成效，后面我就决定采用老师的方法，借鉴了老师代码，慢慢学习才理解到里面的精妙之处，也顺利的完成了这次实验内容

但这次实验还是遇到了很多问题的，例如经常出现这个用户程序可以正常运行，但另外一个就用不了的问题，这也是很难通过调试解决bug的，后来慢慢尝试研究发现，我需要把用户程序开头的org改成100h，同时我不能再用ret返回了，因为在调用的时候并没有使用call指令，所以也不会有cs ip 和psw存放在栈内，自然无法跳转会内核，反而会运行一些奇怪的代码导致程序出错，因此我做了修改，先把用户程序改成更多次的循环运行，其次去掉了ret改为jmp $，这样我顺利的解决了问题

还有就是在我尝试用多进程的模型来实现串行的时候，遇到了如果我一个一个程序运行，它就可以正常运行，但如果我使用批处理一次性运行就不行，我也想了很久这个问题，明明是几乎一样的步骤，怎么结果不一样，后面仔细检查就发现我在每次执行完指令回来会有一个初始化PCB块的操作，而批处理的时候就没有加上这一句，就是这个问题，导致了结果的不同，后面我调用了init函数后就解决了问题。

还遇到一些问题我在实验报告中已经有图文并茂的提及。但总的来说在借鉴了老师的代码使得我的整个实验过程比较的顺畅，遇到的困难也是可以一一化解的。但我发现这个实验真的要编写起来，并不是很难的过程，最难的其实是想到这样一套多进程模型的解决方案来实现我们要实现的目标，因此我花费了很多的时间在理解老师的代码上，比如为什么sp要是ip-4，为什么sp在不是第一次执行的用户程序调度时候要加上16等等。理解了这些之后，联系上理论课的知识和之前自己的尝试思考，才渐渐融汇，理解了整个代码的精妙之处，很多push pop的小细节比如要按照顺序push等等都是需要注意的，这也是我之所以在自己尝试建立一套自己的多进程理论失败的原因。但不管怎么样，这次实验也比较顺利的完成了，虽然没有研究出自己的方法，但在完全理解了老师的代码后也学到了非常多的知识，感谢老师和助教的耐心指导和帮助，继续努力，期待下一次实验项目!

6 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