实验 6：多进程模型

姓名：姜洋帆　　学号：17341068
院系：数据科学与计算机学院　专业：17 级计算机科学（大数据）

【实验题目】多进程模型

【实验目的】

1. 了解操作系统多进程的原理。
2. 了解多进程的简单实现方式：时间片轮转。

【实验要求】

1. 在 c 程序中定义进程表，进程数量为 4 个。
2. 完内核一次性加载 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。

【实验方案】

一、 硬件及虚拟机配置：Lenovo PC ； Oracle Virtual Box

二、 软件工具及作用：

Notepad++ ：写代码

Nasm：编译引导程序的 asm 文件到 bin 文件

Tasm：编译 16 位的内核代码

Tcc：编译 16 位 C 语言，实现 C 语言与 x86 联合编译

Sublime ：查看、编辑二进制文件，将编译完成的程序机器码写入软盘

C 程序：

1. 自己编写的一个 C 语言程序，用来将编译生成的二进制文件填充为 1.44M 大小，并将后缀改为.img，生成 1.44M 的软盘
2. 内核程序在之前的版本上修改，汇编部分增加了切换进程上下文的代码，C 语言部分定义了进程切换需要的相关数据结构（PCB）和一些调度函数，另外还扩展了命令行的功能，可以实现指定程序的并发执行。

86 汇编：

1. Myos5.asm 实现引导程序，偏移量位 7c00h，将内核加载至内存并跳转进入内核程序
2. Kernel.asm、ckernal.c 为内核程序的主要部分，自定义了一些需要的中断，完成内核管理机器的一些基本功能，进程切换的底层实现在这里完成。
3. Kliba.asm 将原本用来显示风火轮的时钟中断改造成用 TIME SHARING 方式实现多进程的代码，当没有正在并发执行的进程时，会恢复原时钟中断，显示风火轮。

三、 程序功能

Myos1 作为引导程序，设置好内核的偏移和段地址，加载内核程序，然后跳入内核程序，进行下一步操作。在保留了之前实验的全部功能的前提下，扩展出一个运行多进程的功能，可以选择一次并行执行全部 4 个程序，也可以通过特定的命令行参数-p 后跟程序号，并行地执行指定地几个程序。

本次实验主要的工作有：

1. 用 C 代码定义存储进程上下文的数据结构——PCB，在 C 代码实现高层的进程调度、初始化、保存等功能
2. 在汇编层，通过用户自定义时钟中断，用 TIME SHARING 的方式实现了一个简单的并行。
3. 扩展原操作系统的命令行，实现指定用户程序并发执行的功能。

四、 程序设计

程序结构与之前实验一致，在实验 5 的基础上，扩展内核部分代码。

实现多进程的关键在于进程上下文的切换，这里需要十分小心，确保所有寄存器的数据都保存在正确的位置。实现的方式与主流的课本上介绍的一样，采用进程控制块保存进程的上下文，当进程被切换出来后，将最终的寄存器上下文保存进 PCB 中，PCB 在 C 代码中定义，在 C 语言中表现为一个结构体。所以每次切换进程时做的工作就是将当前 CPU 的寄存器上下文保存进内存的指定位置，同时将即将被调度进 CPU 的进程的上下文读入寄存器中，然后设置正确的 IP 寄存器值，CPU 就会执行新的进程的代码。

PCB 结构体的定义比较简单，保存 86 汇编的寄存器值即可，用 int 型数据来保存，除此之外还需要额外定义表示进程状态的数据，可选的状态有就绪态和执行态。这里参考了老师文件里的原型代码，用 4 个进程状态，可能对后续的多进程相关实验有帮助。

除了数据结构的定义，还需要定义一些函数用来调度，首先需要一个保存的函数，在进程被切换出 CPU 之前，将 CPU 上下文保存进 PCB 中，这里 C 的实现很简单，传参赋值即可。而关键点在于汇编部分的实现，需要正确的将寄存器的值压入到栈的正确位置，才能确保正确的数据被 C 的函数调用、保存进 PCB 中，这里的实现比较麻烦。。。另外还需要实现进程调度，每次执行调度函数时，选择下一个进程即可（当前+1 模进程总数）。另外还有一些初始化用的函数，详情见第五部分。

汇编部分是实现并行的核心，主要思路是用时钟中断实现以 TIME-SHARING 为基础的并发。将时钟中断的频率改为 50ms 一次，即每 50ms 切换一次进程。在改写的时钟中断中定义一个 Finite 变量，用来计数，当该变量达到阈值时，就直接跳出并行，恢复原来的风火轮时钟中断。跳出并行的具体实现方式就是将当前进程切换成内核进程（0 号进程）。否则就正常切换进程，需要先将当前 CPU 寄存器上下文保存，这里要按照 C 代码中定义的函数参数的顺序（逆序）压栈，实现传参数的过程，由于调用中断时，系统自动将 FLAGS，CS，IP 寄存器入栈，所以这里只需要显示的压剩余的寄存器值就行了。执行调度函数后，ax 寄存器获取新进程 PCB 指针，然后再根据 C 代码中的定义，偏移指针取获取相关的数据就可以实现进程切换了。

主要代码如下：

五、主要代码

引导程序没有修改

内核程序（汇编部分，时钟中断）

调用 C 语言保存 PCB 的前置操作：

按照 C 代码中函数参数的顺序，逆序压栈即可，其中 ip，cs，flag 已经在中断执行前被压栈了，所以这里只需要将剩余的部分压入栈中即可。压栈后立即调用 C 函数，将当前进程的寄存器上下文保存进 PCB 中。C 代码根据 CurrentPCBno（内核中为 _CurrentPCBno）的数值即可判断具体保存进哪一个 PCB 中。之后再执行调度函数，更新当前_CurrentPCBno 的值以及进程状态。

```
    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 _Schedule
```

获取新进程的上下文：
这里需要注意的是 FLAGS,CS,IP 寄存器需要先压入栈，并且不用出栈，因为在最后调用 iret 时，会把 SP,SP+2,SP+4 一次出栈到 IP,CS,FLAGS 中，这样就实现了 PCB 数据的读取、赋值，完成了进程的切换。

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

call near ptr _Current_Process
mov bp, ax          ; 汇编 n 调用 C 代码时，C 函数的返回值保存在 AX 里

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

push word ptr ds:[bp+30]    ; FLAGS
push word ptr ds:[bp+28]    ; CS
push word ptr ds:[bp+26]    ; IP
                          ; 调用中断时会先将 FLAGS，CS，IP 入栈
push word ptr ds:[bp+2]     ; 执行 iret 后会把 SP,SP+2,SP+4 分别出栈到
push word ptr ds:[bp+4]     ; IP,CS,FLAGS 寄存器
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

    push ax
    mov al,20h
    out 20h,al
    out 0A0h,al
    pop ax
    iret
```

设置时钟中断间隔：
将中断间隔设置为 50ms 一次。

```
SetTimer:
    push ax
    mov al,34h   ; 设控制字值
    out 43h,al   ; 写控制字到控制字寄存器
    mov ax,29830 ; 每秒 20 次中断（50ms 一次）
    out 40h,al   ; 写计数器 0 的低字节
    mov al,ah    ; AL=AH
    out 40h,al   ; 写计数器 0 的高字节
    pop ax
    ret
```

其他部分主要实现一些逻辑上的控制（如控制中断 Finite 次后跳回内核），判断执行切换上下文的中断还是风火轮中断等等，这里不详细列出了。

内核程序（C 语言部分）

这一部分基本上照搬老师给的代码（实际上也没什么可以自己实现的，实现出来也就是保存那几个寄存器的值而已）

PCB

```
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;

void Save_Process(int gs,int fs,int es,int ds,int di,int si,int bp,
        int sp,int dx,int cx,int bx,int ax,int ss,int ip,int cs,int flags)
{
    pcb_list[CurrentPCBno].regImg.AX = ax;
    pcb_list[CurrentPCBno].regImg.BX = bx;
    pcb_list[CurrentPCBno].regImg.CX = cx;
    pcb_list[CurrentPCBno].regImg.DX = dx;

    pcb_list[CurrentPCBno].regImg.DS = ds;
    pcb_list[CurrentPCBno].regImg.ES = es;
    pcb_list[CurrentPCBno].regImg.FS = fs;
    pcb_list[CurrentPCBno].regImg.GS = gs;
    pcb_list[CurrentPCBno].regImg.SS = ss;

    pcb_list[CurrentPCBno].regImg.IP = ip;
    pcb_list[CurrentPCBno].regImg.CS = cs;
    pcb_list[CurrentPCBno].regImg.FLAGS = flags;

    pcb_list[CurrentPCBno].regImg.DI = di;
    pcb_list[CurrentPCBno].regImg.SI = si;
    pcb_list[CurrentPCBno].regImg.SP = sp;
    pcb_list[CurrentPCBno].regImg.BP = bp;
}


/* segement 是 es 段地址 （似乎没啥用，随便搞一个就行）,不同的程序区分开，
然后可以用统一的内存偏移地址 */
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;
}
```

初始化 4 个进程（5 号进程没有意义，实际上并没有加载，也没有执行，可以作为扩展用），其中 0 号进程为内核进程，用户进程内存偏移统一为 0XA100

```
void init_Pro()
{
    init(&pcb_list[0],64*1024/16,0x100);    /* 0 号进程是内核程序 */
    init(&pcb_list[1],0x2000,0xa100);
    init(&pcb_list[2],0x3000,0xa100);
    init(&pcb_list[3],0x4000,0xa100);
    init(&pcb_list[4],0x5000,0xa100);
    init(&pcb_list[5],0x6000,0xa100);

    Program_Num = 4;
}
```

实现根据命令行输入，并发执行不同的用户程序
a 和 n 根据用户的输入参数来构造，具体见下方说明

```
void runPar(int *a, int n)
{
    for (index = 0; index < n; index++)
    {
        init(&pcb_list[index + 1],Segement[a[index] - 1],Offset);
    }
    Program_Num = n;
}
```

定义在 ckernal.c 文件中，这一部分解析用户命令行输入，实现指定多进程程序的执行。

```
else if(CMDline[4] == '-' && CMDline[5] == 'p') {  /* run -p 1234  */
           num = 7;
           pnum = 0;
           while(CMDline[num]>='1' && CMDline[num]<='4') {
               pinfo[pnum++] = CMDline[num] - '0';
               num++;
           }
           cls();
           runPar(pinfo, pnum);
           num = 3;
           pnum = 0;
       }
```

【实验过程】

功能展示

在非并发执行程序时，保留原本的所有功能（无敌风火轮，程序运行时的键盘响应中断）

1. 输入 **run all**，内核调用 init_Pro()函数初始化 4 个 PCB，将四个用户程序的相关数据（内存偏移，段地址）加载（实现用 13 号中断加载到指定

位置）。同时更改 Program_Num 为 4，这样就可以直接触发多进程的执行。效果如下：

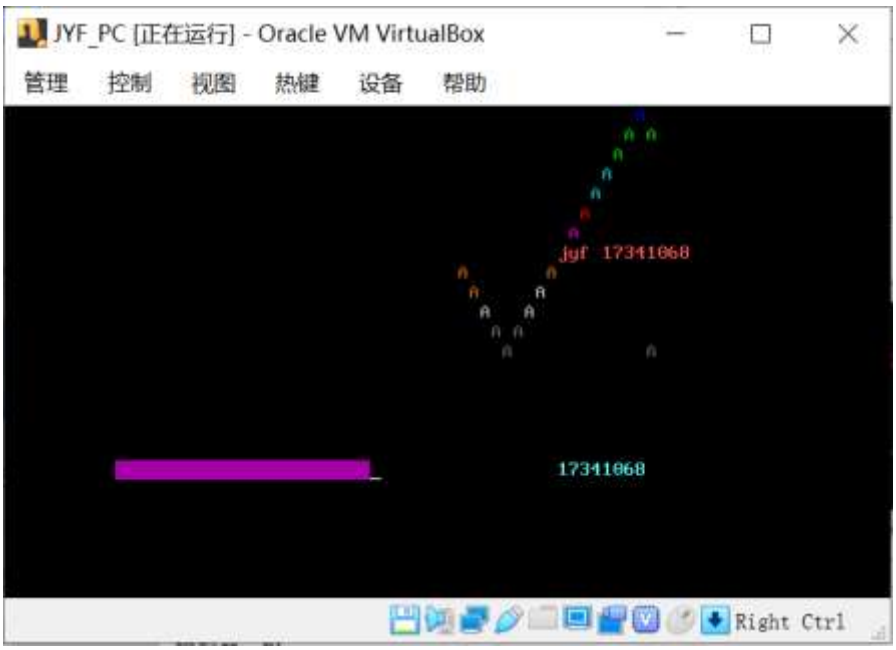
程序并行运行一段时间后（根据 **Finite** 值决定）返回主界面。

2. 指定程序并行执行，命令格式为 **run -p ###**：

   输入： **run -p 13**

输入 run -p 123

输入 run -p 234



3. 测试原中断

执行原本的 **run** 命令后，有风火轮和键盘中断响应。（风火轮在右下角，截图看得不是很清楚，下方字体为键盘中断响应，敲击键盘时才会显示）

【实验总结】

这次实验非常的综合，涉及到汇编和 C 的函数交叉调用，中断设置，虽然每一个小部分都已经做过实验，但是综合在一起还是容易出现问题的，debug 也不太方便，会花费不少时间。

多进程并发执行采用的时 TIME-SHARING 技术，即将 CPU 时间分片，每一片段时间执行不同的程序的代码。这里涉及到最关键的部分就是切换程序代码涉及到的一系列操作。简单地来讲，只需要保存寄存器上下文，将数据写入 PCB 中即可。然而这里的操作比较精细，压栈的顺序不能出错，压栈的时间也不能乱选（要保证栈里没有不小心压入的其他无关的数据），这里是比较麻烦的。还有一个就是利用中断实现时间片轮转，在调用中断时，系统会自动将 IP,CS,FLAGS 寄存器压栈。

实现程序多进程的具体方法是，先将用户程序装载进指定的段地址和内存偏移中，这里修改了一下用户程序，将内存偏移统一定位在 0A00H，否则太小的话，直接从内核调用单个用户程序时，偏移为 CS+OFFSET，可能还是落在内核区域内，会出现问题。

由于自己做的时实模式，并且老师给了大量的参考代码，所以实现起来比较直观，思想也比较简单，甚至可以直接将内核程序的状态装进 PCB 中，将内核程序和用户程序对等看待，这给了我们很大的方便。而实际在保护模式下，内核重入可能会有一些问题，所以正真的多进程实现肯定比这个实验要复杂许多倍，这里只是提供了一个简单的模型让我们实现，加深了理论课程中的多进程部分知识概念的理解。