操作系统实验报告

实验六：具有进程调度的新原型操作系统

学院：　　数据科学与计算机学院

班级：　　16级计算机科学与技术 教务2班

姓名：　　郑映雪

学号：　　16337327

完成时间：　　2018.4.27

一、实验目的

1、在内核实现多进程的二状态模型，理解简单进程的构造方法和时间片轮转调度过程。

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基础。

二、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、在 c 程序中定义进程表，进程数量至少 4 个。

2、内核一次性加载多个用户程序运行时，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。

三、实验方案

1、相关原理

a.关于二状态进程模型

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，它可以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。在本实验中的模型是二状态模型，即阻塞-运行两种状态，通过时钟中断实现时间片轮转。

b.进程表和进程控制块 PCB

在每个进程中，可以由 CPU 分配不同的逻辑 CPU。当同一个计算机内并发执行多个不同的用户程序时，操作系统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。

PCB 包括了进程标识和逻辑 CPU 模拟，具体的实现方法我思考了我参考的原型。它包括了进程标识和逻辑 CPU 模拟。在逻辑 CPU 中，包括所有寄存器和标志寄存器。当并发执行时，通过时钟中断，逻辑 CPU 轮流映射到物理 CPU，从而可以实现多道程序的并发执行。

c.如何实现并发执行多道程序

利用上一个实验中我们认识的时间中断，打断执行中的用户程序实现 CPU 在进程之间交替。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。

d.保护现场的重要性（*老师的 PPT 里讲得非常有道理*）

在时钟中断发生，标志寄存器、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。

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。

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。

e.关于进程切换（*来源于老师 PPT*）

restart 函数可以用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。

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。

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。

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。

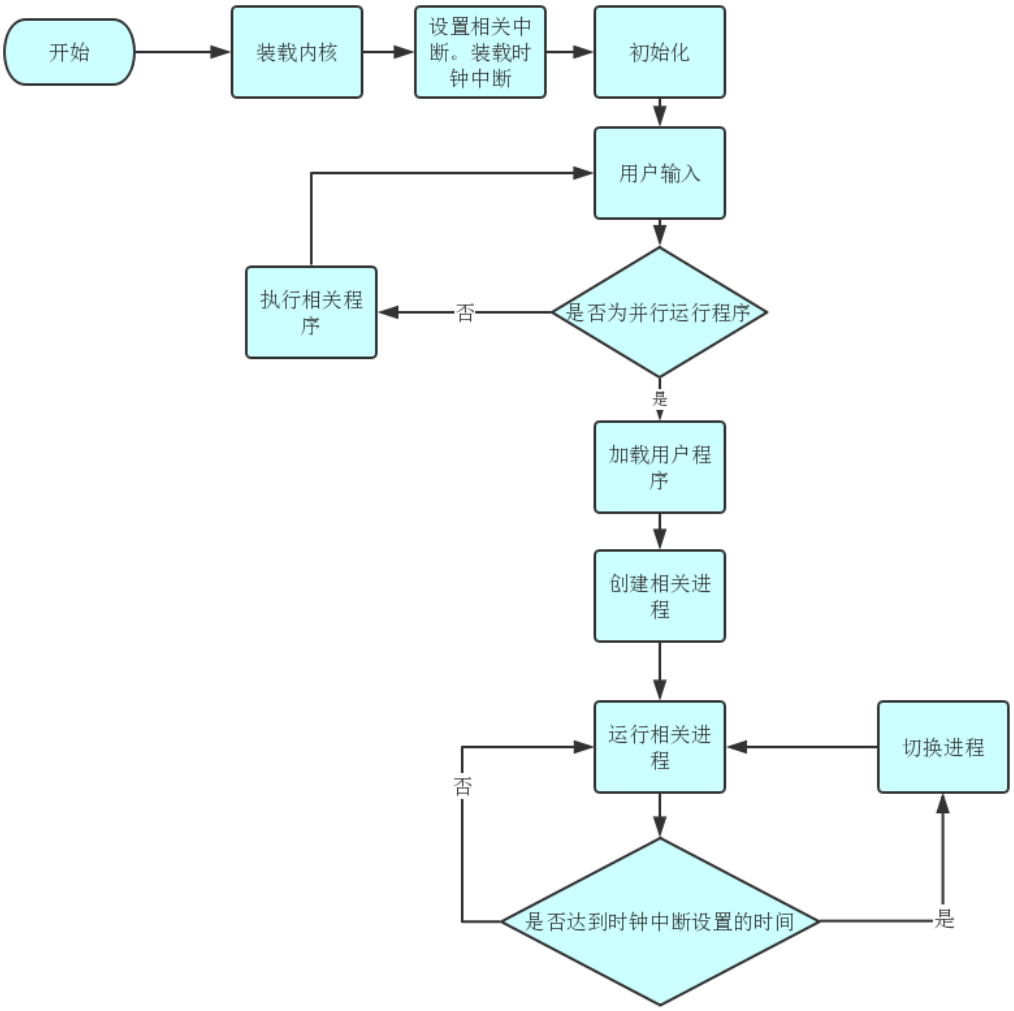
f.本实验中的做法

本实验中进程表和 PCB 由 C 语言编写。我参考了一个模型(参考来源见最后)，将进程表和 PCB 通过 C 语言中数组、struct、函数等来实现，相对来说更加简洁明了。系统启动时，用户程序加载，建立相应的 PCB，当时钟中断时，首先保护现场，然后用户程序就将控制权交给下一个用户程序，再加上中断恢复的过程，实现轮转。

本实验中，我的参考模型实现的方法让我耳目一新。首先，它采用了一个变量来控制内核态/用户态，时钟中断在内核态时依然可以显示右下角的动画，在用户态时即用来实现用户程序的并行。在用户态中，每次交接控制权时，将所有寄存器的值和 flag 压栈，然后储存在 PCB 中。之后通过 restart 恢复中断时的上下文。同时我发现这个模型中运行用户程序的函数也比我之前写得更加简洁，所以我对之前的运行用户程序模块也进行了改进。同时删除了一些以前实现过的、跟这个实验没什么关系的多余的功能，使我的内核在这个实验里更加精简了一些。

2、程序流程图

在本实验中，我的流程图如下：其中，“是否为并行运行程序”中“否”的分支为其它功能。“切换进程”的主要步骤在前面的实验原理中已经提及。



3、程序关键模块

a. PCB 和进程表实现模块

在这个模块中实现了 PCB 和进程表。其中逻辑 CPU 中的寄存器以结构体的形式存储，以此结构体构造进程表。参考模型以函数的形式模拟调度、保护现场、restart，具体各个函数的功能见注释。

```
typedef enum PCB_STATUS{PCB_READY, PCB_EXIT, PCB_RUNNING,
PCB_BLOCKED} PCB_STATUS;

typedef struct Register{
	int ss;
	int gs;
	int fs;
	int es;
	int ds;
	int di;
	int si;
	int sp;
	int bp;
	int bx;
	int dx;
	int cx;
	int ax;
	int ip;
	int cs;
	int flags;
} Register;

typedef struct PCB{
	Register regs;
	PCB_STATUS status;
	int ID;
} PCB;

int MAX_PCB_NUMBER = 8; /*限制最大数目为 8*/

PCB PCB_LIST[8];

PCB *current_process_PCB_ptr;

int first_time;
int kernal_mode = 1;
int process_number = 0;
int current_seg = 0x1000;

int current_process_number = 0;

PCB *get_current_process_PCB() { /*获取当前进程地址*/
	return &PCB_LIST[current_process_number];
}
```

```
void save_PCB(int ax, int bx, int cx, int dx, int sp, int bp, int si, int di, int ds, int es, int fs, int gs, int ss, int ip, int cs, int flags) {    /*保护现场*/
	current_process_PCB_ptr = get_current_process_PCB();

	current_process_PCB_ptr->regs.ss = ss;
	current_process_PCB_ptr->regs.gs = gs;
	current_process_PCB_ptr->regs.fs = fs;
	current_process_PCB_ptr->regs.es = es;
	current_process_PCB_ptr->regs.ds = ds;
	current_process_PCB_ptr->regs.di = di;
	current_process_PCB_ptr->regs.si = si;
	current_process_PCB_ptr->regs.sp = sp;
	current_process_PCB_ptr->regs.bp = bp;
	current_process_PCB_ptr->regs.bx = bx;
	current_process_PCB_ptr->regs.dx = dx;
	current_process_PCB_ptr->regs.cx = cx;
	current_process_PCB_ptr->regs.ax = ax;
	current_process_PCB_ptr->regs.ip = ip;
	current_process_PCB_ptr->regs.cs = cs;
	current_process_PCB_ptr->regs.flags = flags;
}

void schedule() { /*进程调度*/
	if (current_process_PCB_ptr->status == PCB_READY) {
		first_time = 1;
		current_process_PCB_ptr->status = PCB_RUNNING;
		return;
	}
	current_process_PCB_ptr->status = PCB_BLOCKED;
	current_process_number++;
	if (current_process_number >= process_number) current_process_number = 1;
	current_process_PCB_ptr = get_current_process_PCB();
	if (current_process_PCB_ptr->status == PCB_READY) first_time = 1;
	current_process_PCB_ptr->status = PCB_RUNNING;
	return;
}

void PCB_initial(PCB *ptr, int process_ID, int seg) { /*初始化 PCB*/
	ptr->ID = process_ID;
	ptr->status = PCB_READY;
	ptr->regs.gs = 0x0B800;
	ptr->regs.es = seg;
	ptr->regs.ds = seg;
	ptr->regs.fs = seg;
```

```
        ptr->regs.ss = seg;
        ptr->regs.cs = seg;
        ptr->regs.di = 0;
        ptr->regs.si = 0;
        ptr->regs.bp = 0;
        ptr->regs.sp = 0x0100 - 4;
        ptr->regs.bx = 0;
        ptr->regs.ax = 0;
        ptr->regs.cx = 0;
        ptr->regs.dx = 0;
        ptr->regs.ip = 0x0100;
        ptr->regs.flags = 512;
}

void create_new_PCB() {/*创建新的进程*/
        if (process_number > MAX_PCB_NUMBER) return;
        PCB_initial(&PCB_LIST[process_number], process_number, current_seg);
        process_number++;
        current_seg += 0x1000;
}

void initial_PCB_settings() {
        process_number = 0;
        current_process_number = 0;
        current_seg = 0x1000;
}
```

b. 时钟中断模块

这个模块主要架构可以参考上一个实验实现时间中断的架构。在系统态中我还是觉得|-/\更好看一些，所以就改了回来。在用户态中实现了相应的保护现场、交接控制权和 restart，对于模拟的入栈出栈在参考模型上也比较简洁易懂。

```
public _set_clock
_set_clock proc
        push es
        call near ptr _set_timer
        xor ax, ax
        mov es, ax
        mov word ptr es:[20h], offset Timer
        mov word ptr es:[22h], cs
        pop es
        ret
  set  clock endp
```

```
Timer:
        cmp word ptr [_kernal_mode], 1
        jne process_timer
        jmp kernal_timer

process_timer:
        .386
        push ss
        push gs
        push fs
        push es
        push ds
        .8086
        push di
        push si
        push bp
        push sp
        push dx
        push cx
        push bx
        push ax

        cmp word ptr [back_time], 0
        jnz time_to_go
        mov word ptr [back_time], 1
        mov word ptr [_kernal_mode], 1
        push 512
        push 800h
        push 100h
        iret

time_to_go:
        inc word ptr [back_time]
        mov ax, cs
        mov ds, ax
        mov es, ax
        call _save_PCB
        call _schedule

store_PCB:
        mov ax, cs
        mov ds, ax
        call _get_current_process_PCB
        mov si, ax
```

```
        mov ss, word ptr ds:[si]
        mov sp, word ptr ds:[si+2*7]
        cmp word ptr [_first_time], 1
        jnz next_time
        mov word ptr [_first_time], 0
        jmp start_PCB

next_time:
        add sp, 11*2

start_PCB:
        mov ax, 0
        push word ptr ds:[si+2*15]
        push word ptr ds:[si+2*14]
        push word ptr ds:[si+2*13]

        mov ax, word ptr ds:[si+2*12]
        mov cx, word ptr ds:[si+2*11]
        mov dx, word ptr ds:[si+2*10]
        mov bx, word ptr ds:[si+2*9]
        mov bp, word ptr ds:[si+2*8]
        mov di, word ptr ds:[si+2*5]
        mov es, word ptr ds:[si+2*3]
        .386
        mov fs, word ptr ds:[si+2*2]
        mov gs, word ptr ds:[si+2*1]
        .8086
        push word ptr ds:[si+2*4]
        push word ptr ds:[si+2*6]
        pop si
        pop ds

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

对于用户态的实现在上一个实验里已经做过，为了避免杂乱，这个实验就不放上来了。

c. 对照参考模型对于自己以前写的函数的修改

在进行这个实验的时候，我以前写的一个包含了输入功能和返回字符串长度的“strlen”功能为这个实验带来了不少的麻烦。但在这个参考模型中我发现了一个简洁明了的 getline 写法，所以进行了修正。

修改前：

```
public _strlen
_strlen proc
	push bp
	push cx
	push dx
	push bx
	mov bp, sp
	xor dx,dx
	mov ax, offset [bp+10] ;传参中，参数的地址存入堆栈中，但前面 push 了 4 个寄存器，故使 bp+10，下同。
	mov bx,ax ;bx 存储参数的地址
now:
	mov ah,0
	int 16h ;调用 16h 中断，输入字符

	cmp al,13
	je exit ;回车即结束长度统计
	cmp al,8
	jne save ;退格就不会计入长度

	mov ax, 0
	cmp dx, ax
	je first
	jmp back

first:
	push bx
	push dx

	mov bh,0 ;显示第 0 页的信息
	mov ah,3
	int 10h    ;此处调用功能号为 03h 的 10h 号中断，读取光标信息，行和列分别保存在 dh 和 dl 中

	mov bh,0 ;在光标处显示空格，即可以让光标仿照真正的 shell 一样在后面
	mov ah,2
	int 10h
```

```
        mov bh,0
        mov al,' '
        mov bl,07h
        mov cx,1
        mov ah,9
        pop dx
        pop bx
        jmp next

save:
        mov byte ptr [bx],al
        push bx
        push dx
showch@: ;  显示键入字符
        mov ah,0eh
        mov bl,0
        int 10h
        mov ah,0
        pop dx
        pop bx
        inc bx
        inc dx
next:
        jmp now
exit:
        mov ax,dx ;调用函数的返回值储存在 ax 中
        mov sp, bp
        pop bx
        pop dx
        pop cx
        pop bp
        ret
_strlen endp


;退格时存储空格字符在光标处
back proc
        sub bx,1
        sub dx,1
        push bx
        push dx
        mov bh,0
        mov ah,3
        int 10h
```

```
        sub dl,1
        mov bh,0
        mov ah,2
        int 10h
        mov bh,0
        mov al,' '
        mov bl,7
        mov cx,1
        mov ah,9    ;9 号功能的 10h 号中断作用是在当前光标处显示字符
        int 10h
        pop dx
        pop bx
        jmp next
back endp
```

修改后：

```
void get_input(char *ptr, int length) {
        int count = 0;
        if (length == 0) {
                return;
        }
        else {
                getChar();
                while (input != 13) {
                        printChar(input);
                        ptr[count++] = input;
                        if (count == length) {
                                ptr[count] = '\0';
                                printf("\n\r");
                                return;
                        }
                        getChar();
                }
                ptr[count] = '\0';
                printf("\n\r");
                return;
        }
}
```

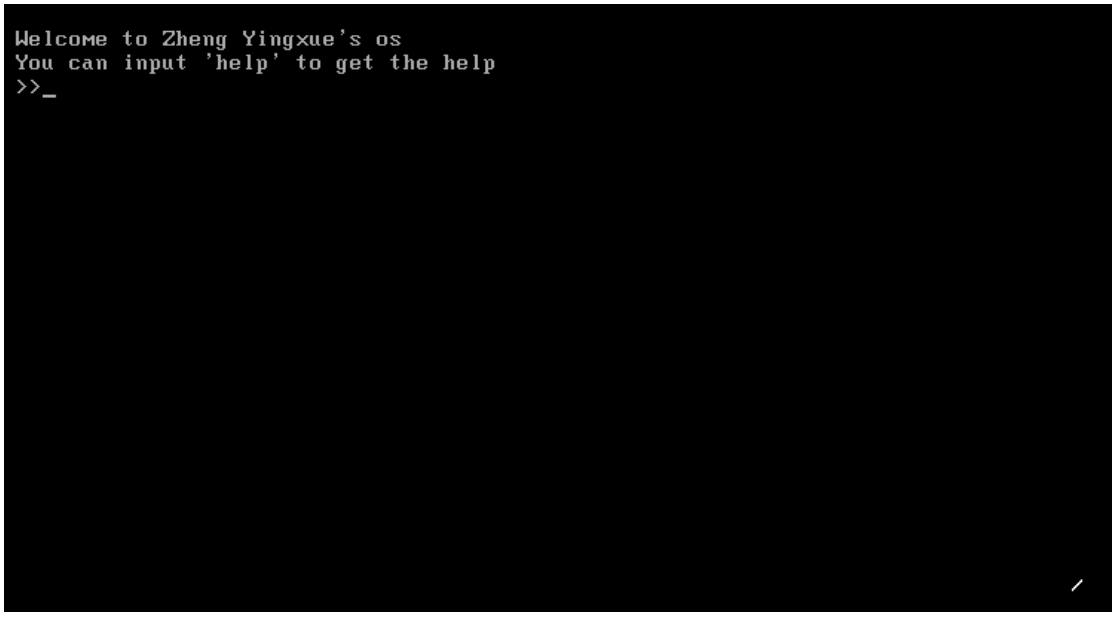
就这么二三十行代码就搞定了，虽然时间紧迫没有添加我原有的退格功能，但是与以前那个“集输入和计数与一身”的输入函数带来的麻烦相比，还是实现

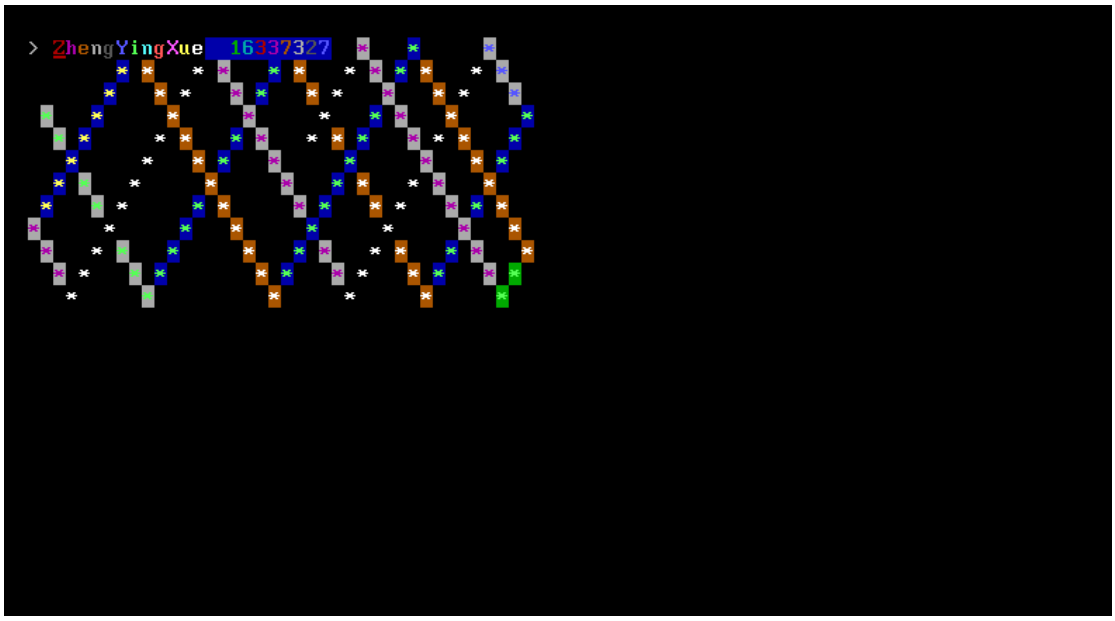
实验目的放在第一位。

四、实验过程

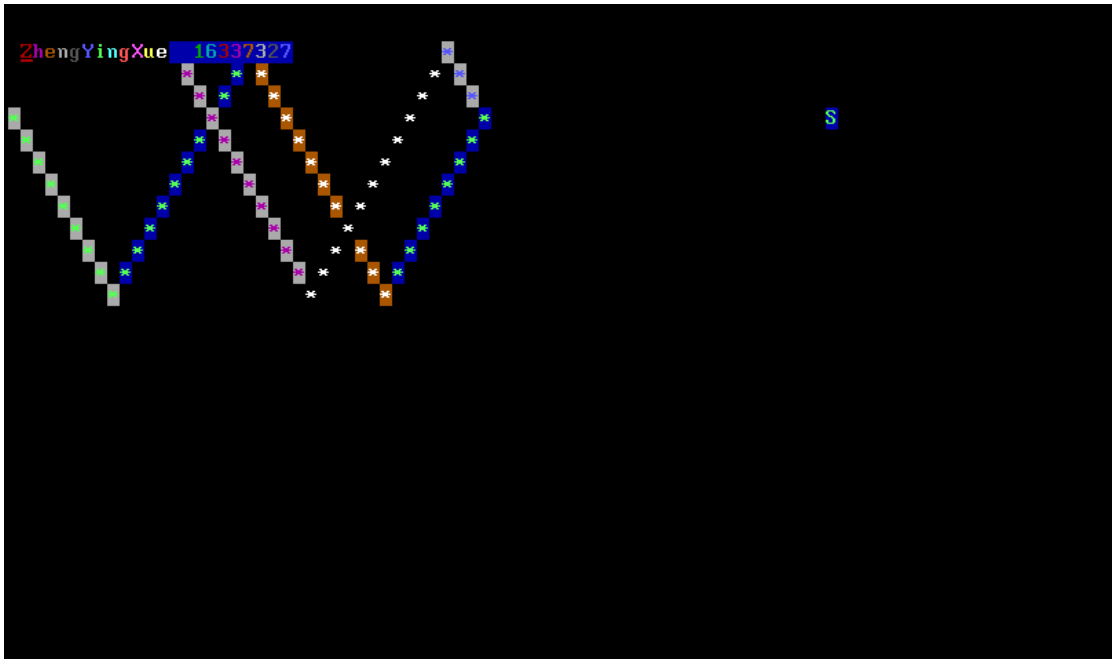
注：如果显示“wrong input”或者显示突然有减少，请重新输入一下，注意不要输错。

1、进入主界面，输入“go a”，我们可以看到第一个进程在运行。

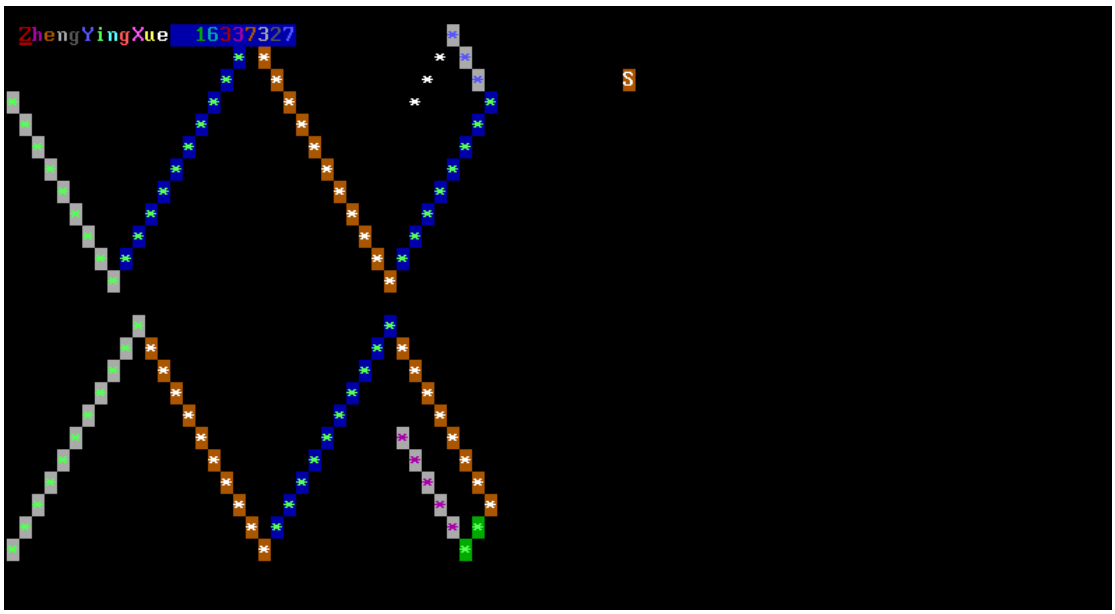




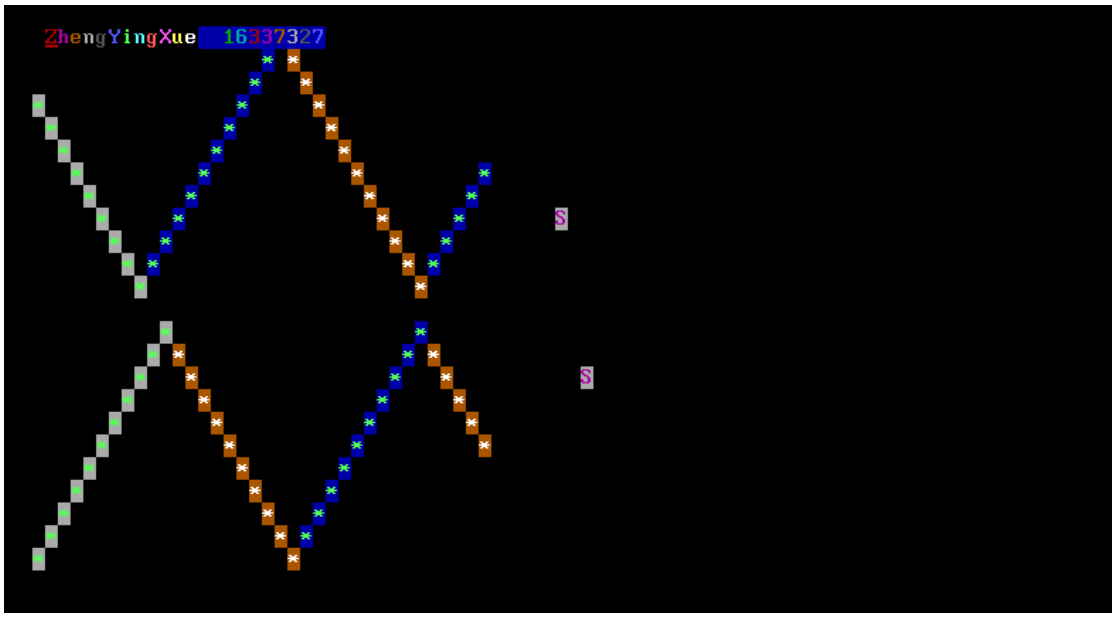
2、关闭虚拟机重新进入，输入“go ab”，我们可以看到两个进程在同时运行。



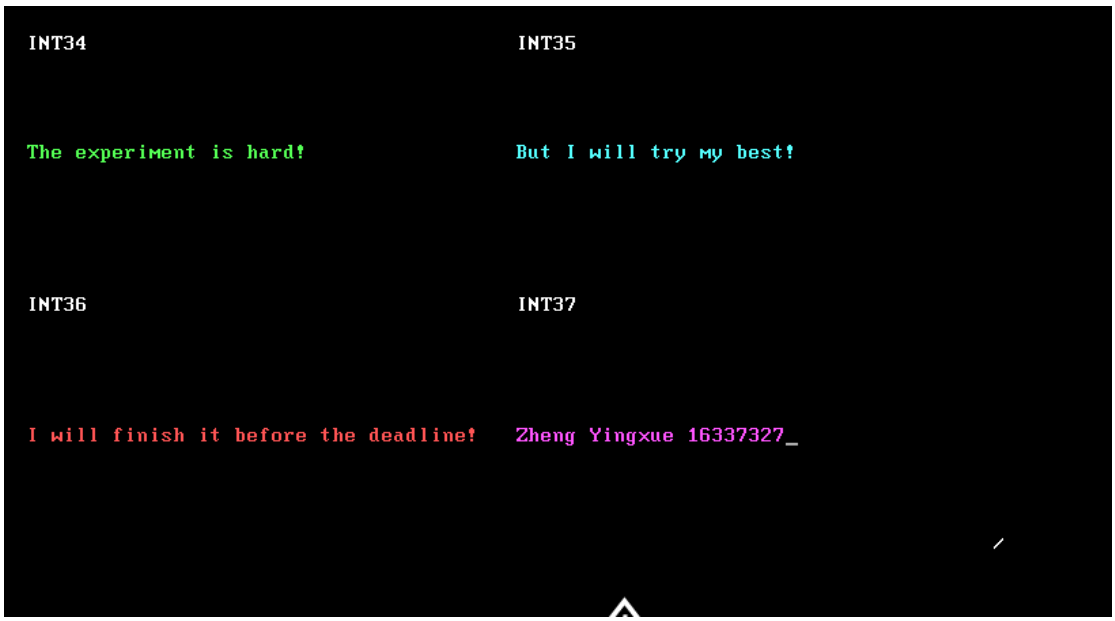
3、关闭虚拟机重新进入，输入“go abc”，我们可以看到三个进程在同时运行。



4、关闭虚拟机重新进入，输入“go abcd”，我们可以看到三个进程在同时运行。轮转成功!!



5、关闭虚拟机，重新进入，输入“int”，我们可以看到还是可以调用原本设置的 34、35、36、37 中断。



五、实验总结

这次的实验真的是一波三折。因为这几个星期许多大项目扎堆，所以实验六我在 ddl 的前几天才开始做。在有参考原型的情况下还是问题多多，离 ddl 还有十个小时的时候编译环境出了问题，真的是快要急哭了。这次最让我觉得抓狂的

问题解决是环境问题。比如在 tcc 调用头文件时，会出现联合编译时函数不能调用的问题，因为它总是显示没有定义。在去掉头文件整合进 C 内核的时候就不会报这个错了。（到现在也不知道这个错的原因）

问题又来了，在这种情况下，会显示如下信息:

我最后只分析出了引起这个错误的原因是原型里面的宏定义。可是我明明只写了一条，为什么会显示“too long”，抓耳挠腮不得其解。还有另一个办法：设全局变量！于是把宏定义改成全局变量以后不会报这个错了。可是有一堆外部函数显示“未定义”的问题在调用头文件的情况下没有得到解决，我开始怀疑是我自己环境的问题了。于是我写了一个简短的调用头文件的程序测试——报错+1。

得，这下应该是我自己的问题了。在这里感谢朋友借他的电脑给我做实验。（可能是我自己的电脑不知道什么时候被我改了环境了）这时候我学乖了，不调用头文件而是直接把头文件整合在了 C 程序里。

后来我明白了为什么调用头文件会出错了，当我调用头文件时，生成的 obj 文件很小，只有 2KB，而 asm 文件生成的 obj 文件都有 3KB，C 文件比 asm 文件复杂得多，但是生成的对象比 asm 生成的对象还小？这是不可能的。我以二进制打开了 kernal.obj 文件，发现只有零星几个函数读了进去，还都是头文件的……喵喵喵？你在逗我？这让我怎么解决呢？这一看就是在编译的时候内存的一部分被某个神奇的东西吃掉了（当然，我是找不到它的）。没办法，还是按照之前的那样，放弃头文件，把那些进程调度的相关函数全部写到 kernal 文件吧。

好不容易编译通过，又出现了进不去进程调用函数的问题。boches 一下发现原来我根本没进入创建进程的模块。后来发现原因是我参考的原型里的 strlen 和我自己在实验三中实现的 strlen 函数是不一样的。我实现的 strlen 函数包括了输入、回显、退格的功能，只因为它的返回值是字符串的最终长度我才把它定义为一个广义的 strlen。但是在参考原型里，它是一个简单的返回长度的函数，我再使用它实际上使用的是我之前写的广义的 strlen 函数，会修改字符的值。自然不能通过 switch 分支进入创建进程了。同理，参考原型中有一个变量与我在之前的实验中设置的全局变量重名，也会对实验造成影响。不过解决这个问题

的方法也比较简单，就是改一下重复变量名和函数名就好啦~

接下去麻烦又来了，我刚刚说的一个“广义的 strlen 函数”在不同的地方给我带来了许多麻烦，其中就有一个无论输入什么都是“wrong input”的问题，调试 boches 许久也找不到问题在哪里。于是我打开了参考原型的输入——几行简介明了的 getinput，除了没实现退格，别的不知道比我高到哪里去啦~于是我照着它的 getinput 的写法对程序进行了优化。我觉得一切额外的功能相比于达到实验目的来说都是次要的，尤其是在时间紧迫的环境下。

在进入用户态时，弹弹弹的字母不出现，整个界面卡死了。在这问题上我发现了两个代码上的问题，一个是控制权交给内核的时候，内核所放置的位置会把我用户程序放置的位置覆盖掉。修改掉段值后，它依然处于卡死的状态。百思不得其解，最后才发现我没有跳转进用户程序——又是在改变读取程序中断的程序后忘记去改掉用户程序一开始的 org……（觉得自己很粗心，每次都在这种细节上出错）

只出现了一次之后就卡死了，然后明明运行 4 个程序，只有前三个程序卡死在了这里，第四个干脆就不显示了。最后发现是进栈出栈的顺序问题造成了寄存器的值混乱了。

另外我发现了同时运行 1、2、3、4 个程序，由于轮转的问题，运行的速度是递减的，所以我将速度改成了在 2、3 个程序时较为适中，所以会出现运行第一个程序时飞快、运行第 4 个程序时慢吞吞的现象。

因为时间中断有原型可以参考，所以遇到的问题解决得都比较快。这次实验里我用 boches 较多，对一些指令例如‘c’（继续运行）、‘b’（设置断点）、‘u’（反汇编，还可以控制汇编多少句代码）等都有了一些实践使用。

当然，我这次实验除了学习到知识和调试方法外，还明白了一个道理——当好几个大项目碰到一起时，os 碰到的错误可能需要修改很长时间，一定要先做 os 实验呀……由于这次时间分配的失误，我上一个实验中说要在这个实验中实现中断中退出的话还是没有实现，非常自责。下次我一定要合理安排好自己的时间。

六、参考文献

1、凌老师的课堂课件（很重要！帮了大忙！）

2、参考原型来源：

https://github.com/AngryHacker/OS_Lab/tree/master/Lab06

（也很重要！这次参考除了让我可以参考了 PCB 和进程表的写法以外，还帮我优化了很多以前写得不够好的地方，比如带来许多麻烦的输入功能(╥﹏╥)）