中山大学数据科学与计算机学院本科生实验报告

（2016 学年第二学期）

课程名称：操作系统实验　　任课教师：凌应标

年级	15 级	专业（方向）	软件工程（移动信息工程）
学号	15352429 15352447 15352454	姓名	张梓坤 钟镇威 朱睿
电话	15989046646	Email	1243031595@qq.com
开始日期	2017.6.1	完成日期	2017.6.18

【实验题目】

1. 五状态进程模型

【实验目的】

一、　学习认识五状态进程模型

二、　学习并实现进程的创建，退出操作

三、理解进程交替执行原理，实现父子进程的通信

【实验要求】

1. 创建 PCB 结构，并创建 8 个空的 PCB 块
2. 实现控制的基本原语 do_fork()、 do_wait()、do_exit()、blocked()和 wakeup()。

3. 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.
4. 编写一个 c 语言程序,实现多进程合作的应用程序。
5. 多进程合作程序为：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

【实验软件】

一、实验软件工具

1．Notepad++:用于编写汇编代码生成.asm 文件。

2．Nasm：用于编辑.asm 文件，生成.bin 文件。

3．WinImage：用于创建虚拟软盘。

4．WinHex：用于编辑软盘的引导扇区数据。

5．VMware Workstayion Player：用于创建虚拟机，模拟裸机环境。

6．TCC：将 c 程序转化成 OBJ 目标文件

7．TASM：将汇编程序转化成 OBJ 目标文件

8．TLINK：链接器，将汇编与 c 生成的目标文件进行链接，生成可以运行的.COM 文件。

9．DOSBOX：用于运行 TCC, TASM 与 TLINK

二、实验硬件及虚拟机配置

硬件：操作系统为 win10 的笔记本电脑

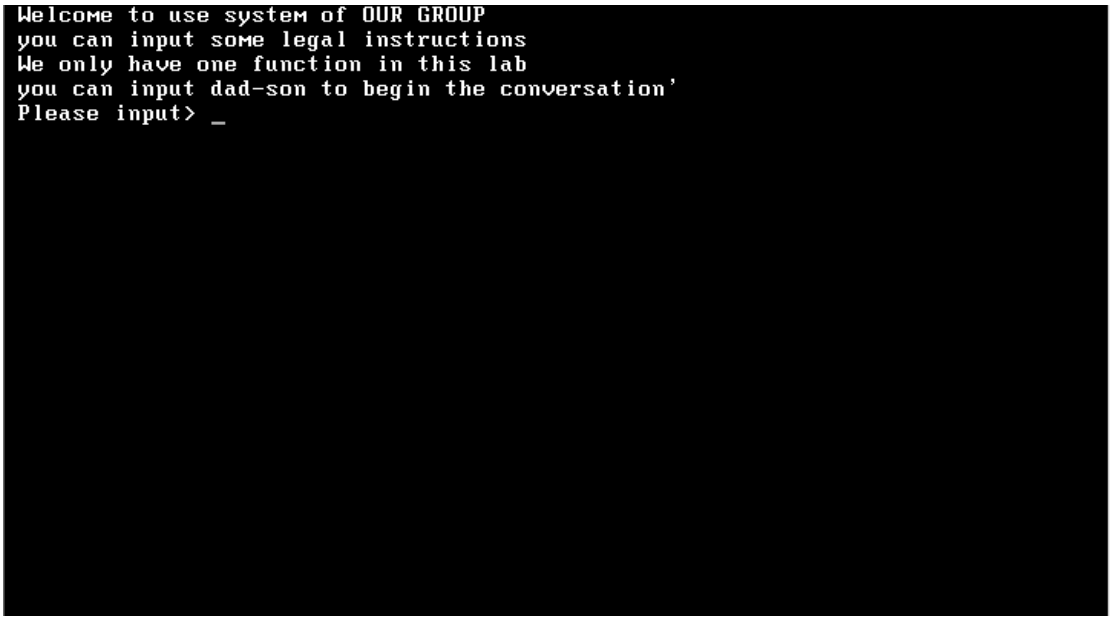
虚拟机设置：无操作系统，1G 硬盘，4MB 内存，

启动时连接软盘

三、程序功能描述与使用

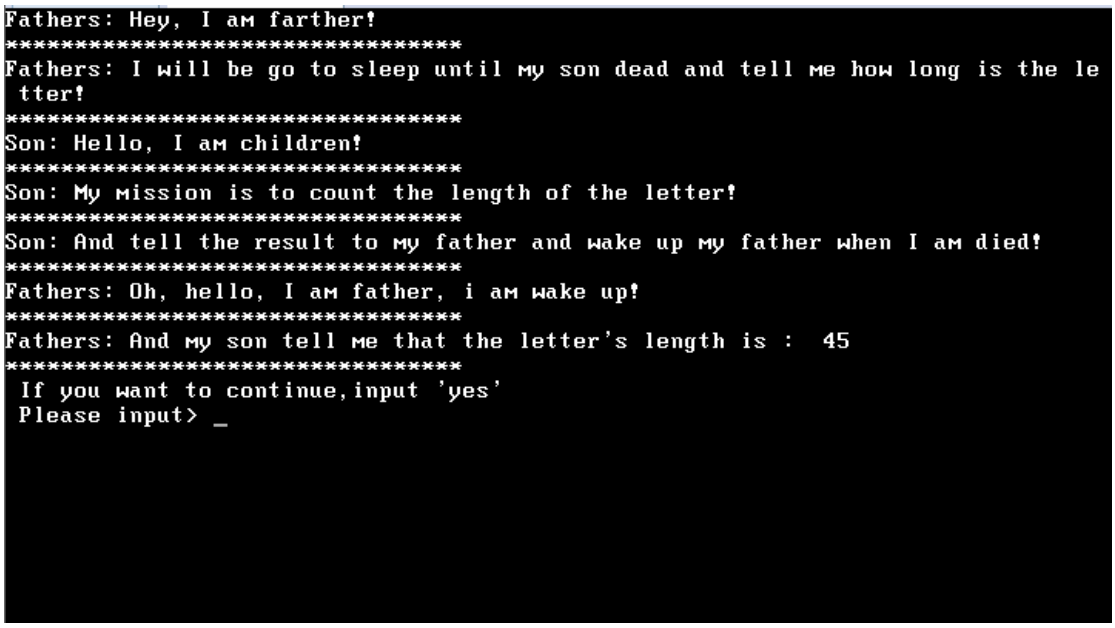
（将“BIN 与 com 文件”文件夹中的 myloader.bin 当作软盘加载进虚拟机里面即可运行我们的实验。）

内核界面



此次实验只有一个功能，就是实现多进程合作，其中首先由父进程创建一个子进程，然后用时钟中断让父子进程不断轮转，父进程进行 do_wait（），阻塞自己，子进程在数完一个字符串长度之后，执行 do_exit() 操作，把自己结束掉，然后唤醒父进程，并把自己的遗言（即数的字符串长度）告诉父进程， 然后父进程唤醒后，把子进程的遗言即字符串长度显示出来。

输入 dad-son 执行结果：



可以结合具体的实现函数来看：

```
77  char s1[80]="error in fork\n";
78
79  char s2[80]="Fathers: Hey, I am farther!\n";
80  char s6[100]="Fathers: I will go to sleep until my son dead and tell me how long is the letter!\n";
81  char s7[80]="Fathers: Oh, hello, I am father, i am wake up! \n";
82  char s8[80]="Fathers: And my son tell me that the letter's length is :  ";
83
84
85  char s3[80]="Son: Hello, I am children!\n";
86  char s4[80]="Son: My mission is to count the length of the letter!\n";
87  char s5[80]="Son: And tell the result to my father and wake up my father when I am died!\n";
88
89  char hang[80] = "**********\n";
```

```
father_and_children()
{
    /*设置局部变量PID，用于区分父子进程*/
    int pid;
    /*初始化所有PCB表的状态*/
    initstatus();
    /*开启时钟中断*/
    set_clock_interrupt();
    /*fork操作*/
    pid=forking();/*save do_fork restart*/

    if (pid==-1) printstring(s1);/*GUI*/
    if (pid){ /*父进程*/
        printstring(s2);
        printstring(hang);
        printstring(s6);
        printstring(hang);

        /*wait操作*/
        waiting();   /* save do_wait restart*/

        printstring(s7);
        printstring(hang);
        printstring(s8);
        exchange(LetterNr);
        printstring(str1);
        printstring("\n");
        printstring(hang);
        re_clock_interrupt();
    }
    else{  /*子进程*/
       printstring(s3);
       printstring(hang);
       printstring(s4);
       printstring(hang);
       printstring(s5);
       printstring(hang);
       count();
       /*exit操作*/
       exiting();/*save do_exit restart*/
    }
}
```

8、老师您可以看对应的输出的字符串，查看程序是否正确运行，运行程序图如上。

1、 首先，创建局部变量 pid，用于区分父子进程

2、 初始化所有 PCB 表的状态，以及开启时钟中断，不断的让状态为 ready 的进程不断轮转

3、 Fork 操作，创建子进程，此时就有两个进程为 ready 状态，那么时钟中断就不断在这两个进程之间轮转，在 fork 包括了 save，do_fork,restart 三步操作，在 restart 之后，此时 pid 的赋值为 AX 寄存器的值，pid 为局部变量，所以父子进程的 pid 更新为不同的值。

4、 当父进程接管时，先显示语句"Fathers: Hey, I am farther!\n"跟"Fathers: I will go to sleep until my son dead and tell me how long is the letter!\n"两句话以后，执行 wait 操作，阻塞了自己，那么时钟中断就不会再轮转到父进程了。这时 wait 操作后面的语句在他状态没变成 ready 之前是不会继续执行下去的。

5、 当子进程执行的时候，先输出"Son: Hello, I am children!\n";和"Son: My mission is to count the length of the letter!\n"和"Son: And tell the result to my father and wake up my father when I am died!\n"三句话，然后去数字符串长度，再执行 exit 操作，结束自己唤醒父进程

6、 刚刚父进程阻塞在 wait 操作以后，但是子进程一执行 exit 操作以后，就唤醒了父进程，这时父进程就会输出 wait 后面的语句，"Fathers: Oh, hello, I am father, i am wake up! \n"跟"Fathers: And my son tell me that the letter's length is : "两句话并显示出字符串长度，然后关闭时钟中断结束程序。

7、 整个程序体现了父子进程合作。

【实验过程】

一、 各个程序与各个模块的代码解释与介绍：

```
18  typedef struct RegisterImage{
19      int SS;    /*0*/
20      int GS;    /*2*/
21      int FS;    /*4*/
22      int ES;    /*6*/
23      int DS;    /*8*/
24      int DI;    /*10*/
25      int SI;    /*12*/
26      int BP;    /*14*/
27      int SP;    /*16*/
28      int BX;    /*18*/
29      int DX;    /*20*/
30      int CX;    /*22*/
31      int AX;    /*24*/
32      int IP;    /*26*/
33      int CS;    /*28*/
34      int FLAGS;  /*30*/
35  }RegisterImage;
36
37  typedef struct PCB{
38      RegisterImage regImg;
39      int Process_Status;  /*状态*/
40      int PCBID;           /*PCB编号*/
41      int F_PCBID;         /*父亲PCB编号*
42  }PCB;
```

首先要创建进程表控制块，即 PCB，创建该模块可以理解为将一个 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。
PCB 其中一部分是逻辑 CPU 模拟。

而逻辑 CPU 模拟要包含 8086CPU 的所有寄存器，根据我们已经写出来的程序，包含 16 个 8086CPU 的寄存器，分别是 ：
SS/GS/FS/ES/ES/DS/DS/SI/BP/SP/BX/DX/CX/AX/IP/CS/FLAGS

下面就是与上次实验不同的地方，增加了进程状态标识：Process_Status; 当前的 PCB 表编号 PCBID，以及父亲 PCB 编号 F_PCBID

下面是本次实验的主要工作函数：

首先是 do_fork 函数：

```
/*执行fork操作*/
int do_fork(){
    PCB *p;
    p=&pcb_list[0];    /*从第0个PCB表开始*/
    while(p<=&pcb_list[7]&&p->Process_Status!=EXIT) p=p+1;
    /*寻找空的PCB表*/
    if(p>&pcb_list[7]){
        pcb_list[CurrentPCBno].regImg.AX=-1;
    }
    /*如果没有找到空的PCB块就退出，把当前进程的AX寄存器赋值为-1*/
    /*如果找到，开始把父进程的PCB复制到子进程的PCB模块*/
    p->regImg.GS = pcb_list[CurrentPCBno].regImg.GS;
    p->regImg.ES = pcb_list[CurrentPCBno].regImg.ES;
    p->regImg.DS = pcb_list[CurrentPCBno].regImg.DS;
    p->regImg.CS = pcb_list[CurrentPCBno].regImg.CS;
    p->regImg.FS = pcb_list[CurrentPCBno].regImg.FS;
    p->regImg.IP = pcb_list[CurrentPCBno].regImg.IP;
    p->regImg.AX = pcb_list[CurrentPCBno].regImg.AX;
    p->regImg.BX = pcb_list[CurrentPCBno].regImg.BX;
    p->regImg.CX = pcb_list[CurrentPCBno].regImg.CX;
    p->regImg.DX = pcb_list[CurrentPCBno].regImg.DX;
    p->regImg.DI = pcb_list[CurrentPCBno].regImg.DI;
    p->regImg.SI = pcb_list[CurrentPCBno].regImg.SI;
    p->regImg.BP = pcb_list[CurrentPCBno].regImg.BP;
    p->regImg.FLAGS = pcb_list[CurrentPCBno].regImg.FLAGS;

    /*为子进程的PCBID赋值，直接赋值为该PCB的下标编号*/
    p->PCBID=p-&pcb_list[0];

    /*把父进程栈的全部内容复制到子进程的栈段内*/
    CopyStack(p->PCBID);

    /*子进程的SS寄存器与父进程相同*/
    p->regImg.SS =pcb_list[CurrentPCBno].regImg.SS;
    p->regImg.SP =pcb_list[CurrentPCBno].regImg.SP+2000*p->PCBID;
/*子进程的SP寄存器与父进程当前的SP的值上加上我设置的大小1000的偏移乘上当前PCB编号/
    /*但是由于int是两个字节的，所以加上1000*2乘上PCB编号相同*/

    p->F_PCBID=CurrentPCBno;
    /*子进程的父PCB编号设置*/

    p->regImg.AX=0;
    /*子进程的寄存器AX设置为0*/
    p->Process_Status=READY;
    /*子进程的状态设置为就绪*/
    pcb_list[CurrentPCBno].regImg.AX=p->PCBID;
    /*父进程的AX寄存器设置为子进程的PCB编号*/
}
```

Do_fork 函数我理解为实现三部分功能：

1、 寻找空的 PCB 块

2、 把父进程所有寄存器的内容复制到新的 PCB 块之中（SS 寄存器与 SP 寄存器另外赋值）

3、 为子进程分配新的栈段，并把父进程栈段内的所有内容复制到子进程栈段内，新栈段的分配无非是更改子进程 SS 寄存器跟 SP 寄存器，这一段的实验我觉得是本次实验的重点跟难点，我将在后面具体说明

4、 把子进程的 AX 寄存器送 0，F_PCBID 设置为它父亲的 PCBID，并把自己成的状态设置为就绪状态，父进程的 AX 寄存器设置为子进程的 PCBID。

下面具体说明一下我的栈段分配与实现

```
8   int Stack [9][1000];
9   int* StackForOs=&Stack[1][0];
10
11  void CopyStack(int NewPCBno){
12      for(i=0;i<1000;i++){
13          Stack[NewPCBno][i]=Stack[CurrentPCBno][i];
14      }
15  }
```

```
34  start:
35      mov  ax,  cs
36      mov  ds,  ax
37      mov  es,  ax
38      mov  ss, ax
39      mov  sp, word ptr _StackForOs
40
41      call int21h;
42      call near ptr _cmain
43      jmp $
44
```

1、我创建了一个二维数组，当作本次实验的所有进程的栈段使用，每个进程块我给了 1000 个位置给它们使用，比如 PCB[0]使用 Stack[0][0]---Stack[0][999]作为栈段，PCB[1]使用 Stack[1][0]---Stack[1][999]，作为栈段，以此类推。
2、其中的 StackForOs 是第一个栈段的开始位置，我把他给了刚开始运行的内核，可以看到上面的第二张图
3、栈段内的内容的复制简单粗暴，直接用一个循环直接复制，根据传入的 PCB 编号，把当前栈段的内容直接给复制过去。
4、新栈段的分配无非是设置新进程的 SS 跟 SP 寄存器，在本次实验中，我所有的进程都是用的同一个 SS，内核段地址，不同的是 SP，查看第一张图 SS 跟 SP 的寄存器的赋值可以看到，SS 保持相同不变，新进程的 SP 在父进程 SP 的基础之上，加上了 [偏移量（1000*2）* 新进程 PCB 编号]，为什么我原本栈的大小分配的是 1000，还要乘上个 2，因为 int 是两个字节大小的，这真是一个大坑，多亏问了别人才知道。
5、这一段的实现虽然做出来之后，感觉也没什么难的但是刚开始做的时候，真的是错漏百出。花了很多的时间来填这个坑。

Do_fork 操作实现大致如上，我是严格按照 PPT 上的思路打的。

下面是 do_wait 函数：

```
118 void do_wait(){
119     pcb_list[CurrentPCBno].Process_Status=BLOCKED;
120     /*把当前进程的状态设置为阻塞*/
121     schedule();
122     /*时间片轮转*/
123 }
```

做的操作十分简单，就是把当前进程阻塞，然后把轮转到其他状态为就绪的进程。

do_exit 函数：

```
125 void do_exit(){
126     pcb_list[CurrentPCBno].Process_Status=EXIT;
127     /*把当前进程状态设置为结束退出*/
128     pcb_list[pcb_list[CurrentPCBno].F_PCBID].Process_Status=READY;
129     /*把当前进程的父进程的状态设置为就绪*/
130     pcb_list[pcb_list[CurrentPCBno].F_PCBID].regImg.AX=0;
131     /*把当前进程的父进程的AX寄存器设置为0*/
132     schedule();
133     /*时间片轮转*/
134 }
```

Exit 函数，完成的操作有：
1、 把当前进程结束，状态变为退出
2、 把当前进程的父进程状态变为就绪
3、 把父进程的 AX 寄存器送 0
4、 时间片轮转，轮转到状态为 ready 的程序。

下面是这三个本次实验主题函数的封装：

```
234 ;跳转分支表
235 syscall_vector dw syscall0,syscall1,syscall2
236 ;系统调用入口，根据ah来判断具体功能
237 ;因为此处用到al,bx，所以al,bx不能作为系统调用的参数
238 mysyscall proc
239     cli
240     mov al,ah
241     xor ah,ah
242     shl ax,1  ;ax=ax*2
243     mov bx,offset syscall_vector
244     add bx,ax    ;中断向量加上偏移量，即可得到不同功能号的程序
245     mov ax,cs
246     mov es,ax
247     mov ds,ax
248     jmp word ptr [bx]
249 mysyscall endp
```

```
252 ;系统调用0，ah=0
253 syscall0 proc
254    call _save
255    call near ptr _do_fork
256    jmp _restart
257 syscall0 endp
258
259
260 ;系统调用1，ah=1
261 syscall1 proc
262
263    call _save
264    call near ptr _do_wait
265    jmp _restart
266 syscall1 endp
267
268 ;系统调用1，ah=2
269 syscall2 proc
270
271    call _save
272    call near ptr _do_exit
273    jmp _restart
274 syscall2 endp
```

1、我把 fork，wait，exit 三个函数分别封装到了 21 号中断的 0，1，2 三个功能号上

2、Sysyca110:fork 部分的封装，包括了 save,do_fork,restart 三个函数，是 21H 中断的 0 号功能

3、Sysyca111:wait 部分的封装，包括了 save，do_wait,restart 三个函数，是 21H 中断的 1 号功能

4、Sysca112：exit 部分的封装，也包括了 save,do_exit,restart 三个函数，是 21H 中断的 2 号功能，其实这部分的 save 函数可有可无，因为会结束掉当前进程，所以并不需要 save 过程。

```
196 public _forking
197 _forking proc
198     mov ah,0
199     int 21h
200     ret
201 _forking endp
202
203 public _waiting
204 _waiting proc
205     mov ah,1
206     int 21h
207     ret
208 _waiting endp
209
210 public _exiting
211 _exiting proc
212     mov ah,2
213     int 21h
214     ret
215 _exiting endp
```

上面中断的封装已经完成，下面把调用这些中断的语句写成 public 函数，在 C 部分调用即可。

时间片轮转函数：

```
137 void schedule(){
138     /*防止下一次进入循环直接退出*/
139     if(CurrentPCBno==7) CurrentPCBno=-1;
140     /*每次从当前进程的下一个编号开始轮转*/
141     for(i=CurrentPCBno+1;i<8;i++){
142         if(pcb_list[i].Process_Status==READY){
143             CurrentPCBno=i;
144             break;
145         }
146         if(i==7) i=-1;
147     }
148 }
```

这个函数就是轮转状态为就绪的进程，每次从当前进程的下一个编号开始搜索状态为就绪的进程，确保遍历全部 PCB 表。

下面是时钟中断相关的内容，上次实验已经体现，包括时钟中断

中断服务函数的修改，以及 save，restart 函数的实现，我直接把上次实验报告的内容嫁接过来。

```
1035  SetTimer:
1036      push ax
1037      mov al,34h    ; 设控制字值
1038      out 43h,al    ; 写控制字到控制字寄存器
1039      mov ax,29830 ; 每秒 20 次中断（50ms 一次）
1040      out 40h,al    ; 写计数器 0 的低字节
1041      mov al,ah     ; AL=AH
1042      out 40h,al    ; 写计数器 0 的高字节
1043      pop ax
1044      ret
1045
1046
1047
1048  ;设置时钟中断 08h
1049  public _set_clock_interrupt
1050  _set_clock_interrupt proc
1051      cli
1052      push es
1053      push ax
1054
1055      call SetTimer
1056
1057      xor ax,ax
1058      mov es,ax
1059      ;save the vector
1060      mov ax,word ptr es:[20h]
1061      mov word ptr [clock_vector],ax
1062      mov ax,word ptr es:[22h]
1063      mov word ptr [clock_vector+2],ax
1064      ;fill the vector
1065      mov word ptr es:[20h],offset Timer
1066      mov ax,cs
1067      mov word ptr es:[22h],ax
1068      pop ax
1069      pop es
1070      sti
1071      ret
1072  _set_clock_interrupt endp
```

SetTimer 是计时器函数，用该函数实现了每秒 20 次中断，即 50ms 执行一次中断。

设置时钟中断的函数与上次百分之九十以一摸一样的，都是修改时钟中断的中断向量表，添加中断服务程序。

这次的区别有：

1、 调用了计时器函数

2、 改变了事件中断的中断都唔程序。

```
1097  Timer:
1098  cli
1099  call save
1100  ;函数保存当前正在运行程序的栈到PCB表中
1101  call near ptr _Schedule
1102  ;调度程序，轮转到下一程序
1103  jmp restart
1104  ;restart函数切换栈，恢复各个状态寄存器。
1105
```

Save 函数：

```
1108  ret_save dw ?
1109  si_save dw ?
1110  kernal_sp dw ?
1111  kernal_ss dw ?
1112  kernal_cs dw ?
1113  bp_save dw ?
1114  ax_save dw ?
1115  sp_save dw ?
1116
1117  save:
1118      push ds
1119      ;当前栈顶：psw\cs\ip\call_ret\ds
1120      ;PCB:
1121      push cs
1122      ;当前栈顶：psw\cs\ip\call_ret\ds\cs
1123      ;PCB:
1124      pop ds          ;ds=cs
1125      ;当前栈顶：psw\cs\ip\call_ret\ds
1126      ;PCB:
1127      mov word ptr ds:[kernal_cs],ds
1128      ;此时ds=cs，存下ds的值即可存下内核cs的值
1129      pop word ptr ds:[ds_save]
1130      ;当前栈顶：psw\cs\ip\call_ret
1131      ;PCB:
1132
1133      pop word ptr ds:[ret_save]
1134      ;当前栈顶：psw\cs\ip
1135      ;PCB:
1136
1137      mov word ptr ds:[bp_save],bp
1138      mov word ptr ds:[ax_save],ax
1139      ;接下来要改变ax跟bp寄存器先存下来
1140
1141      call near ptr _Current_Process
1142      ;返回的pcb指针在ax寄存器中
1143      mov bp,ax    ;bp=ax
1144
1145      pop word ptr ds:[bp+26]  ;ip
1146      ;当前栈顶：psw\cs
1147      ;PCB:    ip
```

1、前面三步：
push ds
push cs
pop ds
可以那么 ds=cs，此时 ds 就会指向内核。

2、接下来用临时变量存下寄存器的值是因为接下来的操作中会用到该寄存器，并需要改变该寄存器的值，如程序中的 bp, ax，sp 寄存器

3、Psw,cs,ip,ss 这四个寄存器是需要手动入栈的即通过 mov 操作入栈。

4、栈切换可以通过修改 ss 寄存器以及 sp 寄存器实现

5、ss 寄存器在入栈以前，一定要是指向内核的。

6、调用_Curent_Process 函数时，返回值会存入 ax 寄存器中。该函数返回值是根据 PCB 标号返回 PCB 指针。

7、在每个 push 跟 pop 操作下面，我都在注释中表明了当前栈是哪个以及 PCB 跟当前栈内部结构

```
1145        pop word ptr ds:[bp+26]   ;ip
1146        ;当前栈顶: psw\cs
1147        ;PCB:    ip
1148
1149        pop word ptr ds:[bp+28]   ;cs
1150        ;当前栈顶: psw
1151        ;PCB:    cs\ip
1152        pop word ptr ds:[bp+30]   ;flag
1153        ;当前栈顶:
1154        ;PCB:    psw\cs\ip
1155
1156        mov word ptr ds:[bp+0], ss ;ss
1157        ;当前栈顶:
1158        ;PCB:    psw\cs\ip\ss
1159
1160        mov word ptr ds:[sp_save],sp
1161        ;在sp改变之前存下sp的值
1162
1163        add bp, 26
1164        mov sp,bp
1165        push cs
1166        pop ss
1167        ;进行栈的切换，当前栈变成了pcb中的栈
1168        ;为什么sp=bp+26?
1169        ;可以查看我的PCB数据结构
1170        ;这是为了让sp=pcb中sp的真实位置
1171        ;pcb中sp是pcb指针偏移26的位置
1172
```

1160-1166 中改变 sp 跟 ss 寄存器的值，通过改变 sp 跟 ss 寄存器即可进行栈的切换。

其中 sp=bp+26，bp 是 pcb 指针的位置，这步操作是为了让 sp 等于 pcb 中的真实位置，在 pcb 中，sp 寄存器的位置是 pcb 指针 偏移 26 位，具体可以看到上面的 PCB 结构。

.386 是告诉编译器 fs，gs 是 386 汇编中的寄存器

```
1173    push word ptr ds:[ax_save]
1174    ;当前栈（pcb）：psw\cs\ip\ax
1175    push cx
1176    ;当前栈（pcb）：psw\cs\ip\ax\cx
1177    push dx
1178    ;当前栈（pcb）：psw\cs\ip\ax\dx
1179    push bx
1180    ;当前栈（pcb）：psw\cs\ip\ax\dx\bx
1181    push word ptr ds:[sp_save]
1182    ;当前栈（pcb）：psw\cs\ip\ax\dx\bx\sp
1183    push word ptr ds:[bp_save]
1184    ;当前栈（pcb）：psw\cs\ip\ax\dx\bx\sp\bp
1185    push si
1186    ;当前栈（pcb）：psw\cs\ip\ax\dx\bx\sp\bp\si
1187    push di
1188    ;当前栈（pcb）：psw\cs\ip\ax\dx\bx\sp\bp\si\di
1189    push word ptr ds:[ds_save]
1190    ;当前栈（pcb）：psw\cs\ip\ax\dx\bx\sp\bp\si\di\ds
1191    push es
1192    ;当前栈（pcb）：psw\cs\ip\ax\dx\bx\sp\bp\si\di\ds\es
1193    .386
1194    push fs
1195    ;当前栈（pcb）：psw\cs\ip\ax\dx\bx\sp\bp\si\di\ds\es\fs
1196    push gs
1197    ;当前栈（pcb）：psw\cs\ip\ax\dx\bx\sp\bp\si\di\ds\es\fs\gs
1198    .8086
1199
1200    ;保存完毕
1201    mov sp, word ptr [kernal_sp]
1202    ;让sp变回内核的sp
1203
1204    mov ax, ds
1205    mov es,ax
1206    ;es指向数据段
1207
1208    mov ax, word ptr ds:[ret_save]
1209    jmp ax
1210    ;返回call的下一跳指令
```

Restart 函数：

```
1216  restart:
1217
1218      mov ax,word ptr ds:[kernal_cs]
1219      mov ds,ax
1220      mov es,ax
1221      ;ds跟es都指向内核
1222      call near ptr _Current_Process
1223      mov bp, ax
1224
1225      mov ss,word ptr [bp+0]
1226      mov sp,word ptr [bp+16]
1227      ;进行栈的切换
1228      ;ss,sp的值手动恢复
1229
1230      push word ptr [bp+30]
1231      ;当前栈:psw
1232      push word ptr [bp+28]
1233      ;当前栈:psw\cs
1234      push word ptr [bp+26]
1235      ;当前栈:psw\cs\ip
1236      push word ptr [bp+2]
1237      ;当前栈:psw\cs\ip\gs
1238      push word ptr [bp+4]
1239      ;当前栈:psw\cs\ip\gs\fs
1240      push word ptr [bp+6]
1241      ;当前栈:psw\cs\ip\gs\fs\es
1242      push word ptr [bp+8]
1243      ;当前栈:psw\cs\ip\gs\fs\es\ds
1244      push word ptr [bp+10]
1245      ;当前栈:psw\cs\ip\gs\fs\es\ds\di
1246      push word ptr [bp+12]
1247      ;当前栈:psw\cs\ip\gs\fs\es\ds\di\si
1248      push word ptr [bp+14]
1249      ;当前栈:psw\cs\ip\gs\fs\es\ds\di\si\bp
1250      push word ptr [bp+18]
1251      ;当前栈:psw\cs\ip\gs\fs\es\ds\di\si\bp\bx
1252      push word ptr [bp+20]
1253      ;当前栈:psw\cs\ip\gs\fs\es\ds\di\si\bp\bx\dx
1254      push word ptr [bp+22]
1255      ;当前栈:psw\cs\ip\gs\fs\es\ds\di\si\bp\bx\dx\cx
1256      push word ptr [bp+24]
1257      ;当前栈:psw\cs\ip\gs\fs\es\ds\di\si\bp\bx\dx\cx\ax
```

```
1258        pop ax
1259        pop cx
1260        pop dx
1261        pop bx
1262        pop bp
1263        pop si
1264        pop di
1265        pop ds
1266        pop es
1267        .386
1268        pop fs
1269        pop gs
1270        .8086
1271        ;恢复结束
1272
1273        push ax
1274        mov al,20h
1275        out 20h,al
1276        out 0A0h,al
1277        pop ax
1278        iret
1279        ;结束中断返回。
1280
1281
```

这种恢复状态寄存器的方式很好理解，可读性很强，在每一步 push 操作以后，我都注释了当前栈的的内部结构，在 pop 操作时，只要找对应的寄存器恢复就好了。

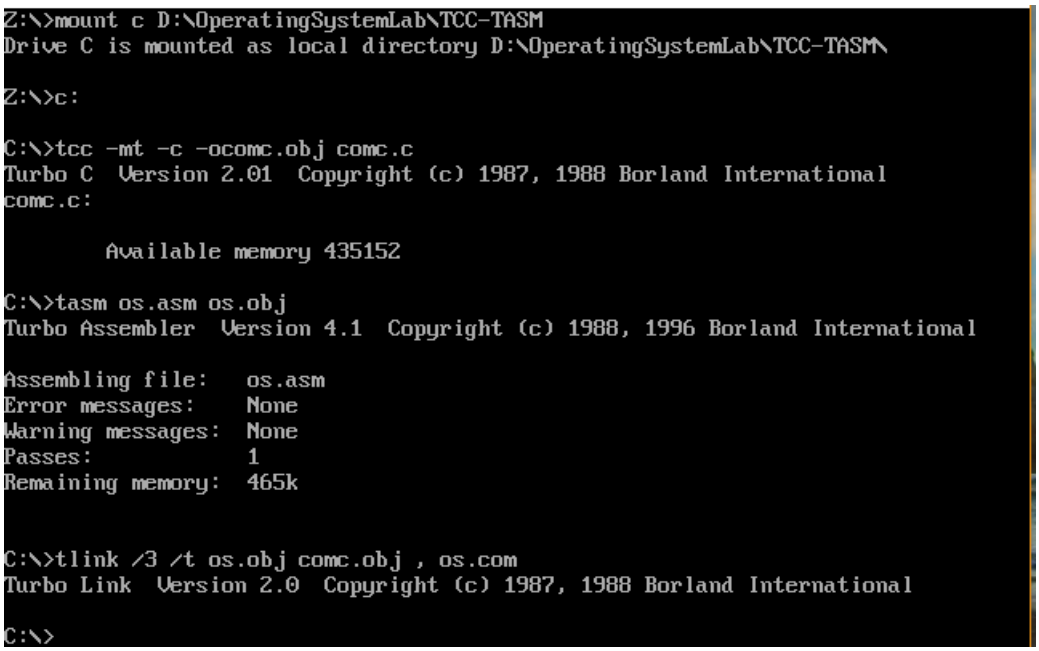
最后就是数字符串的长度函数，这些都比较简单：

```
146  char str[80]="129djwqhdsajd128dw9i39ie93i8494urjoiew98kdkd\n";
147  int LetterNr=0;
148  char str1[80]="";
149  int a;
150  int b;
151  /*把数字转化成字符串，便于输出*/
152  void exchange(int Letter)
153  {
154      i=0;
155      b= Letter % 10;
156      a= Letter / 10;
157      if(a!=0)
158      {
159          str1[i]=a+'0';
160          i++;
161      }
162      str1[i]=b+'0';
163      str1[i+1]='\0';
164  }
165  /*数字符串str的长度*/
166  void count()
167  {
168      LetterNr=0;
169      for(i=0;i<80;i++)
170      {
171          if(str[i]!='\0')
172              LetterNr++;
173          else
174              break;
175      }
176  }
```

以上便是我本次实验主要内容。

工具具体使用：

首先时 TCC,TASM,TLINK 的使用，这几个工具要在 dosbox 内使用。



实验总结

看看时间，这次实验，应该是本学期我能完成的最后一个实验了。总体来说，os 实验是本学期最为困难的一门实验，是我花了最多时间去完成的，但是完成了以后，也是最有成就感的一门实验。下面我还是讲一下我这次实验完成的过程。

每次实验例行流程，我还是先前前后后仔仔细细的看了看这次实验的 PPT，了解这次实验的目的，其实，这次试验是我看完 PPT 以后，对最后实现效果没有预想的实验，我看完之后，还不是很清楚实验完成之后，应该是怎样，脑子里只是依稀记得老师课堂上提过的父子进程协作等字眼。接下来的时间，我与同学一起对主函数进行探讨与学习，因为我发现这次实验，理清 PPT 上 mian 函数的执行流程以及效

果是根本，是最核心的东西。摸爬滚打，终于弄清楚 cmian 函数里，的各种细节流程，以及完成以后应该要有的效果。

接下来，开始具体去实现本次的实验，就是三个函数 do_fork,do_wait,do_exit 三个函数的书写以及封装。其中 do_wait 以及 do_exit 函数都比较简单完成，也不容易出错，难点就在与 do_fork 函数，首先在实验开始之前，我并不明白，为什要把这三个部分封装进系统调用里面，后来我才知道每当调用中断的时候，操作系统就会自动给当前进程的 FLAG,CS,IP 三个寄存器赋值，在 iret 之后，返回到调用中断的出口。这个问题理清之后，就是 do——fork 函数的书写，依照 PPT 上的思路，我大致打了出来，但是就是有错，根本无法成功，不能让父进程跟子进程两个线程依靠时钟中断不断轮转运行。后来我发现是栈段的问题，栈段本次实验实现过程中我觉的最困难的，首先一个因素是我对栈只是初步的了解，然后就是不知道怎么分配栈段，我查阅网上资料，也问了一些同学，知道了可以用数组代替栈段，在汇编中，每次压栈，SP 寄存器的值就会减 4，所以假如你要分配二维数组 stack[0][0]-stack[0][999]的区间作为一个进程的栈段，你要把 SP 的值指向 stack[0][999]的位置，因为每次压栈，SP 的值就会一直减下去。栈的复制过程就比较容易实现，直接用循环把数组当前区间的全部值复制到目标区间就好了。

每次都是做完实验之后，觉得并不是很困难，但是开始做的时候，过程总是很曲折。

最后，本学期也快结束了，谢谢老师，老师辛苦了！