

实验报告 实验七

姓名：王钦 学号：13349112 班级：计科二班

实验目的

1. 学习进程模型知识，掌握进程模型的实现方法。
2. 利用时钟中断，设计时钟中断处理进行进程交替执行
3. 扩展MyOS，实现多进程模型的原型操作系统

实验内容

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。
- 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。

参考程序如下：

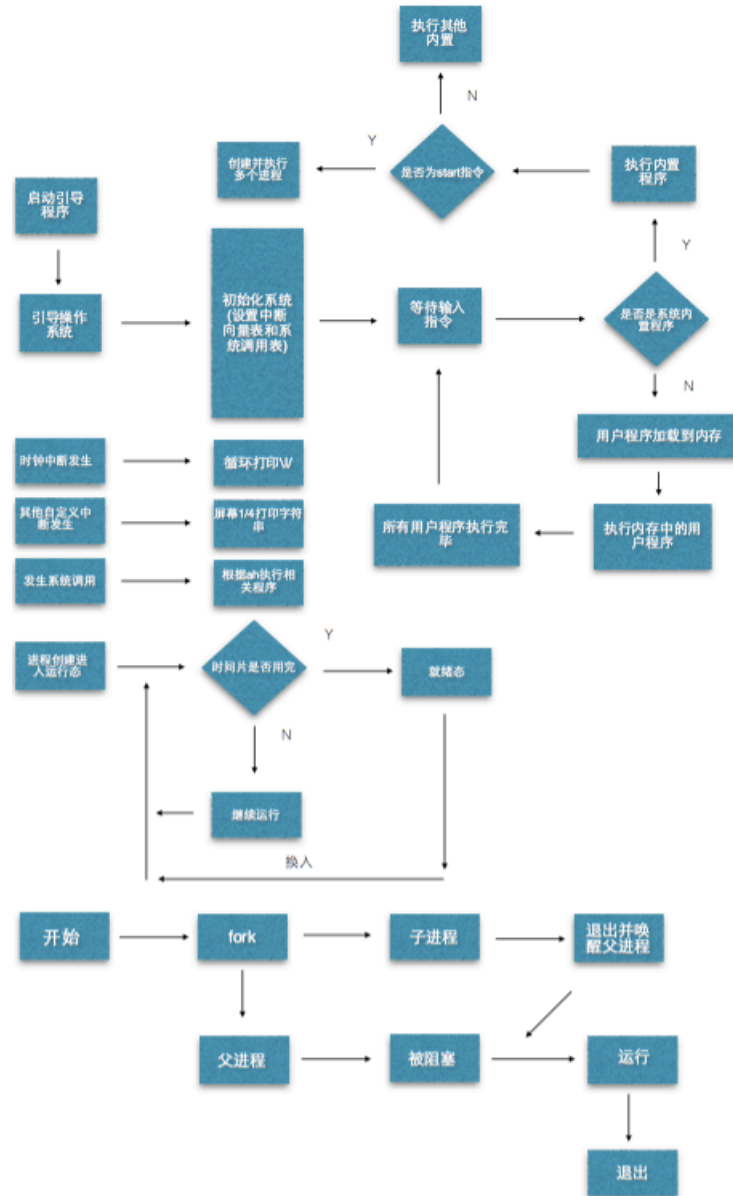
```
1 char str[80]="129djwqhdsajd128dw9i39ie93i8494urjoiew98kdkd";
2 int LetterNr=0;
3 void main() {
4     int pid;
5     char ch;
6     pid=fork();
7     if (pid==-1) printf("error in fork!");
8     if (pid) {
9         ch=wait();
10        printf("LetterNr=");
11        ntos(LetterNr);
12    }
13    Else {
14        CountLetter(str);
15        exit(0);
16    }
17 }
```

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。

实验平台

xxd+dd+gcc+ld+nasm+Linux+vim

算法流程图



功能一览

1. 系统内置功能:

terminal, 装载内核shell, 为用户提供一个与操作系统交互的工具, 开机后自动进入, 以下所有功能都在terminal中交互

date, 显示当前日期

time, 显示当前时间

asc, 显示一个字符的asc码

clear, 清除当前屏幕所有字符, 刷新屏幕

help, 显示系统帮助信息

man, 显示内置函数的帮助信息, 比如 `man date`, 显示date的

相关帮助

`python` , `python` 扩展, 类似`python`命令行工具, 可以使用这个工具输入计算表达式返回计算结果, 目前只支持加法减法

`start`, 开始创建并执行四个进程并且每秒18.2次的调度, 分别在屏幕1/4处打印一些个性化信息(不同配置的虚拟机动画速度不一样, 建议使用`vmware`测试)

2. 用户程序:

`run` ,软盘中含有两个用户程序,输入`run 12`,可分别执行两个用户程序, 当然也可以通过改变执行序列来改变执行的顺序

3. 自定义中断:

时钟中断: 通过PTR每秒发出18.2次的信号来从8592芯片的RT0引脚发出终端号`int 08h`来触发的用户时钟软中断 `int 1ch`, 实现在terminal的右下角 一个横杠在转动.

另外有自定义中断`int 33h`,`int 34h`,`int 35h`,`int 36h`分别在屏幕四分之一的位置打印个性化信息

4. 进程调度:

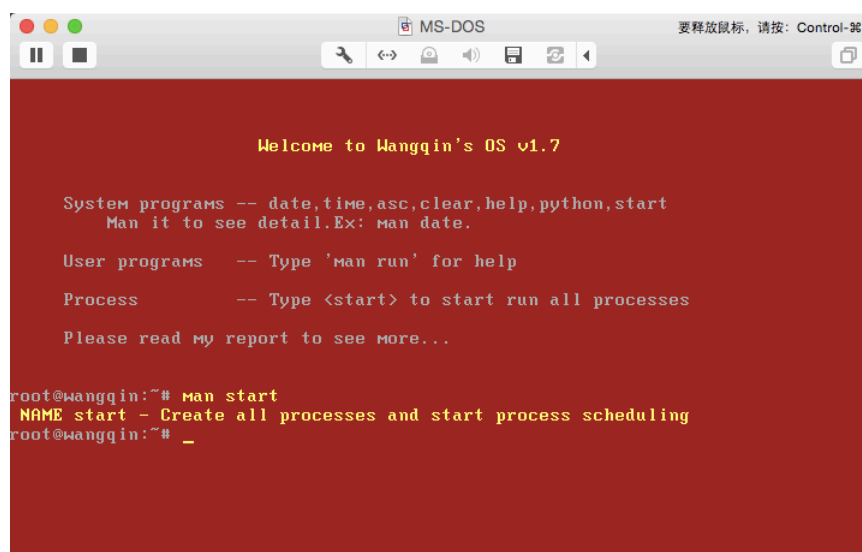
软盘中共存放了用于展示进程调度的六个应用程序, 其中一个应用程序为监听用户键盘事件然后退出多进程调度状态回到 `terminal`每个应用程序分别代表一个进程。开启装载进程并进行进程调度由1中系统内置功能的 `start`指令激活。

5. 进程`fork`,`wait`,`exit`:

这个功能也是本次实验要求实现的, 具体使用在第六个进程中, 执行`start`后即可看到包括第六个进程在内全部进程执行的结果。

实验步骤及效果图

1. 编辑修改ASM 文件,和C文件
2. 使用`make`命令配合`makefile`文件进行编译
3. 运行`bochs` or `vmware`虚拟机进行测试,输入`man start`,查看此条指令的说明



```

Welcome to Wangqin's OS v1.7

System programs -- date,time,asc,clear,help,python,start
Man it to see detail.Ex: man date.

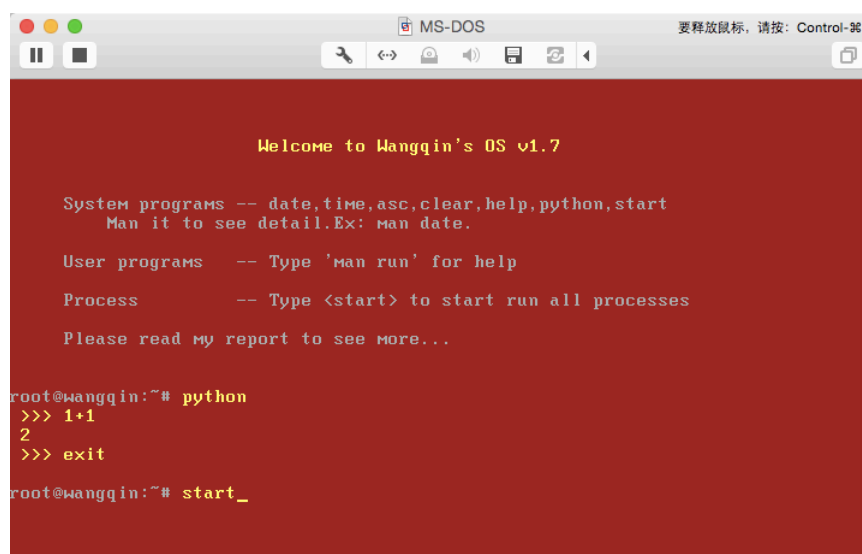
User programs   -- Type 'man run' for help

Process         -- Type <start> to start run all processes

Please read my report to see more...

root@wangqin:~# man start
NAME start - Create all processes and start process scheduling
root@wangqin:~# _
```

4. 输入start 指令测试本次实验要求实现的功能



```

Welcome to Wangqin's OS v1.7

System programs -- date,time,asc,clear,help,python,start
Man it to see detail.Ex: man date.

User programs   -- Type 'man run' for help

Process         -- Type <start> to start run all processes

Please read my report to see more...

root@wangqin:~# python
>>> 1+1
2
>>> exit
root@wangqin:~# start_
```

5. 执行 start后开始执行的进程包括四个分别在1/4屏幕打印个性信息的进程和一个监听任意按键退出的进程最后一个就是本次实验要测试的含有fork等函数的c语言进程,一共6个进程,下面是这个c进程的详细代码

```

1 process_counter.c
2
3 #include "muti_process.h"
4
5 char str[ 80] = "129djwqhdsajd128dw9i39ie93i8494urjoiew98kdkd";
6 int LetterNr = 0;
7 void main() {
8     _asm_ ( "sti");
9     int pid;
10    char ch;
11    printf( "\r\nBefore fork \r\n");
12    printf( "fork start..");
13    pid = fork();
14    if ( pid == -1) printf( "error in fork!\0");
15    if ( pid){
16        printf( "\r\nFather process:after fork pid is ");
17        printf( pid);
```


7. 下面具体解释一下执行的详细过程,可以参照第5步贴的代码,绿色框内是执行fork之前输出的字符串,黄色框代表fork后父进程输出的代码,紫色是fork后子进程输出的。可以看到fork完之后父进程运行了一段时间后被阻塞,然后子进程继续运行直到退出并唤醒父进程。父进程再次执行并把子进程执行countletter的结果打印出来,最后父进程退出

```

before fork
fork start..
father process:after fork pid is 7
father process:running...THE
father process:blocked
Sub process:after fork pid is 0
Sub process:running...
Sub process:exit
father process:runni9g...
father process:Lette9Nr=27
father process:exit_9
WORLD
WAR
44444444444444444444444444444444

```

8. 再执行的任何时间都可以按下任意按键退出模拟多进程调度的程序回到terminal。

```

Welcome to Wangqin's OS v1.7

System programs -- date,time,asc,clear,help,python,start
Man it to see detail.Ex: man date.

User programs   -- Type 'man run' for help

Process         -- Type <start> to start run all processes

Please read my report to see more...

root@wangqin:~# _

```

内存和软盘存储管理

1. 引导程序加载到内存0x7c00处运行
2. 引导程序将操作系统加载到0x7e00处运行
3. 操作系统讲用户程序加载到0x1000处运行
4. 软盘第0个柱面的第一个扇区存储操作系统引导程序
5. 软盘第0个柱面剩下所有扇区2~36扇区存储操作系统内核
6. 软盘第1,2,3,4,5,6柱面分别存储六个进程代码
7. 软盘第7,8柱面分别存储两个用户程序的程序代码

栈结构

内核栈:从内存的 0xffff开始向下扩展

用户栈:从内存的 0x1000开始向下扩展

进程栈:第i个进程对应的进程栈从内存的 $i * 0x800$ 开始向下扩展

更多细节信息请阅读我的Makefile文件

系统目录架构

```
.
├── Makefile makefile 文件
├── README
├── bochsrc
├── boot.asm 引导程序
├── disk.img
├── kernel
│   ├── os.asm 主要为os.c提供函数实现.
│   ├── os.c 为内核主要控制模块
│   ├── os.h 主要为os.c提供函数实现.
│   ├── os_syscall.asm
│   ├── oslib.c 为oslib.c提供更底层的函数封装
│   ├── oslib.asm 初始化系统调用和设置系统调用相关模块
│   ├── pcb.h
│   ├── process.c 进程调度创建相关代码
│   ├── terminal.c 装载shell的工具
│   └── terminal.h
├── muti_process.h fork,wait,wakeup,printf等声明实现
├── oslib_share.c 内核和用户的共享库(使用户程序体积减少)。
├── oslib_share.asm 内核和用户的共享库(使用户程序体积减少)。
├── process1.asm 第一个进程
├── process2.asm 第二个进程
├── process3.asm ..
├── process4.asm ..
├── process_counter.c 统计字母个数的进程
├── process_wait_key.asm 监听退出进程
├── python_extension.c Python扩展
├── report 实验报告目录
│   ├── Illustrations 插图
│   │   ├── back.png
│   │   ├── flow.png
│   │   ├── fork_begin.png
│   │   ├── fork_done.png
│   │   ├── intro.png
│   │   ├── man_start.png
│   │   └── start.png
│   ├── report.aux
│   ├── report.log
│   ├── report.pdf
│   ├── report.tex latex源码
│   └── tree
├── snapshot.txt
├── usr1.asm 用户代码
└── usr2.asm
```

1 directory, 28 files

主要函数模块

1. process.c: do fork 函数模块, 这个模块是本次试验要求实现 fork 的核心代码, 注意的地方就是要在复制父进程的上下文 tss 之前要保存一下父进程现在的状态到 pcb, 另外就是串操作复制父进程栈的时候要注意寻址方式 $ss*16+sp$, w_is_r 表示当前正在运行进程在数组中的 index

```
1
2 extern void restore_flags();
3 short int sub_stack, fa_stack;
4 void do_fork(){
5     process_num++;
6     //add index for new sub process
7     PCB_queue[ process_num].f_pid = w_is_r;
8     //sub -> father
9     PCB_queue[ process_num].process_id = process_num;
10    //sub->pid = new index
11    //copy_fa_Tss
12
13    update_fa(); //save father registers to pcb
14    restore_flags(); //to fix some stack bug
15    __asm__("pop %cx");
16    // update fa end
17    PCB_queue[ process_num].tss = PCB_queue[ w_is_r].tss;
18    //sub.tss=father.tss
19    PCB_queue[ process_num].tss.SP = _sp + 0x1000;
20    //sub.stack_pointer = father.stack_pointer+0x1000
21    PCB_queue[ process_num].tss.AX = 0;
22    //father process return pid is 0
23    PCB_queue[ process_num].tss.Stack_END=PCB_queue[ w_is_r].tss.Stack_END+0x1000;
24    //ss
25
26    sub_stack = (PCB_queue[ process_num].tss.Stack_END-0x200)/16;
27    //sub_stack.segmentaddress-> find address ss*16+sp
28    fa_stack = (PCB_queue[ w_is_r].tss.Stack_END-0x200)/16;
29    //father_stack_segment_addr
30    __asm__("mov $0x104,%cx"); //movsw length
31    copy_stack(); //movsw
32    __asm__("pop %cx");
33
34    PCB_queue[ process_num].process_status = READY;
35    //sub process into READY queue
36
37    restore_ax_pid();
38    // sub process return pid is self pid
39    __asm__("pop %bx");
40
41    __asm__("pop %bx");
42    __asm__("pop %bx");
43    __asm__("pop %bx");
44    __asm__("jmp *%bx");
45    //end of syscall
46 }
```

2. copy_stack 模块, 使用串操作复制父进程栈的内容到子进程栈

```
1 global copy_stack
2 extern sub_stack, fa_stack
3 copy_stack:
4     push ax
5     push es
6     push ds
7     push di
8     push si
9     push cx
10    mov ax,[ sub_stack]
11    mov es,ax
12    mov edi,4
13
14    mov byte [es:di],0x12 ;3df0
15
16    mov ax,[ fa_stack]
17    mov ds,ax
18    mov esi,4
19    cld
20    rep movsw ;ds:si -> es:di
```

```

21     pop cx
22     pop si
23     pop di
24     pop ds
25     pop es
26     pop ax
27     ret

```

3. do_wait和 do_exit模块，这两个模块实现起来就比较简单，改变一下进程的状态就好了

```

1     global copy_stack
2     void do_wait(){
3         PCB_queue[ w_is_r].process_status = BLOCK;
4         __asm__( "int $0x1c" );
5         __asm__( "pop %ax" );
6         __asm__( "pop %ax" );
7         __asm__( "pop %ax" );
8         __asm__( "jmp *%ax" );
9     }
10    void do_exit(){
11        PCB_queue[ w_is_r].process_status = DONE;
12        PCB_queue[ PCB_queue[ w_is_r].f_pid].process_status = READY;
13        __asm__( "int $0x1c" );
14        __asm__( "pop %ax" );
15        __asm__( "pop %ax" );
16        __asm__( "pop %ax" );
17        __asm__( "jmp *%ax" );
18    }

```

4. 进程调度模块，这个模块对上一个实验中的 schedule进行了修改，主要修改是调度的时候调整了对不同进程状态的筛选，例如一个进程要被送上cpu的时候必须是就绪态等。

```

1     int fin_times = 0;
2     void schedule(){
3         saveall_reg();           //hurry not inclue sp
4         __asm__( "pop %cx" );
5         __asm__( "pop %eax" );    //junk
6
7         if( PCB_queue[ w_is_r].process_status == RUNNING){
8             PCB_queue[ w_is_r].process_status = READY;
9         }
10        while(1){
11            if( w_is_r == 0)
12                nw_is_r = start_process_num;
13            else
14                nw_is_r = w_is_r + 1;
15
16            if( nw_is_r > process_num){
17                nw_is_r = start_process_num;
18            }
19            if( PCB_queue[ nw_is_r].process_status == READY) break;
20            w_is_r = nw_is_r;
21        }
22
23        PCB_queue[ nw_is_r].process_status = RUNNING;
24
25        saveToqueue();           //code order don't change
26
27        //-----set ip cs flag-----
28        __asm__( "pop %ax" );
29        __asm__( "pop %dx" );
30        __asm__( "pop %cx" );
31
32        saveall_reg_seg();       //include sp
33        __asm__( "pop %cx" );
34
35        if( _di == 0x1234){
36            isProcessRun = 0;           //shut down process
37            nw_is_r = 0;
38            process_num --;
39            backto_os();
40        }else{
41            isProcessRun = 1;
42        }
43
44
45        PCB_queue[ w_is_r].tss.SP = _sp;

```

```

46 PCB_queue[ w_is_r].tss.IP = _ip;
47 PCB_queue[ w_is_r].tss.CS = _cs;
48 PCB_queue[ w_is_r].tss.Flags = _flags;
49
50 //-----end-----
51 _ip = PCB_queue[ nw_is_r].tss.IP;
52 _cs = PCB_queue[ nw_is_r].tss.CS;
53 _flags = PCB_queue[ nw_is_r].tss.Flags;
54 _sp = PCB_queue[ nw_is_r].tss.SP;
55
56 restore_reg_seg(); //include sp
57 __asm__("pop %cx");
58
59 queueTodata(); // ax bx cx...
60
61 w_is_r = nw_is_r; //change now running process
62 restore_reg();
63 __asm__(" pop %di"); //don't use di in any process is dangerous
64
65 __asm__(" jmp schedule_end");
66 while(1);
67 }

```

5. muti_process.h中的 fork,wait,exit,分别对应系统功能号为 6,7,8的系统调用.

```

1
2 extern char return_ax_Tpid();
3 char pid;
4 char fork(){
5     __asm__(" cli");
6     __asm__("mov $6,%ah");
7     __asm__(" int $0x80");
8
9     pid = return_ax_Tpid();
10    __asm__("pop %cx");
11    __asm__(" sti");
12    return pid;
13 }
14
15 char wait(){
16     __asm__(" cli");
17     __asm__("mov $7,%ah");
18     __asm__(" int $0x80");
19     __asm__(" sti");
20     return pid;
21 }
22
23 void exit( char x){
24     __asm__(" cli");
25     __asm__("mov $8,%ah");
26     __asm__(" int $0x80");
27     __asm__(" sti");
28     while(1);
29 }
30 }

```

6. Countletter 函数实现, 这个比较简单没什么好说的就是从给出的一串字符串中统计里面字母的个数

```

1 extern int LetterNr;
2 void inline CountLetter( char *str){
3     char i;
4     for ( i = 0;i<80;i++){
5         if( str[ i] <='z' && str[ i] >='a'){
6             LetterNr++;
7         }
8     }
9 }

```

实验心得及仍需改进之处

实验心得:

本次实验过程还算轻松，没有遇到一些难以解决的bug，不过也是有一些地方卡了很久，比如说上次实验的进程栈指针是从0开始隔0x1000一个，这次因为进程比较多，fork完后是7个，又由于前面五个进程根本用不到这么大的栈所以改成了隔 0x800为一个进程栈，我当时写代码的时候以为 $0x1000/2 = 0x500$ ，然后设置栈指针的时候设置为 $0x500*i$ ，后来调了半天发现 $0x1000/2 = 0x800$ 并不是0x500，细节上稍微一不注意，这种低级错误就会发生，写汇编的时候就要抛弃原来十进制的思维方式。

另外就是要注意fork时复制之前一定要保存一下父进程的状态到pcb，并不是只有到被调度的时候才保存，fork的时候如果没有更新pcb就复制父进程的pcb，显然不能复制到父进程最新的pcb，导致fork出错。

do_wait, do_exit 的时候要注意，改变完进程状态后马上手动触发下 0x1c 时钟中断，将本进程根据进程状态进行处理。否则并不能起到马上改变进程状态的作用，因为要等待下一次时钟中断发生。

实验仍需改进之处:

仍需完善细节，比如说python的命令行工具加入乘法，除法完全是几行代码的问题

可以考虑将用户程序做成elf格式，动态链接系统的代码库。目前的情况是用户程序自己带着一份和操作系统一样的代码库，分别联合编译

精简操作系统代码，减少冗余代码

增加文件系统功能

调度算法和进程控制模块的存储有待优化，使用链表来实现pcb

如果时间片很小，父进程在执行到 wait 之前就被调度，接下来子进程完成了整个程序的执行，然后exit，然后父进程重新开始执行，那么此时父进程执行了 wait，变成阻塞态，接下来一直无法运行，也没有子进程来更改父进程的状态，父进程就会无限等待下去。所以，这个问题应该要靠使用信号量的PV操作来解决