

# 实验报告 实验八

姓名：王钦 学号：13349112 班级：计科二班

## 实验目的

1. 学习信号量机制的原理，掌握其实现方法。
2. 利用信号量机制，实现进程互斥和同步
3. 扩展MyOS，实现信号量机制

## 实验内容

如果内核实现了信号量机制相关的系统调用，并在c库中封装相关的系统调用，那么，我们的c语言就也可以实现多进程同步的应用程序了。例如：利用进程控制操作，父进程f创建二个进程s和d，大儿子进程s反复向父进程f祝福，小儿子进程d反复向父进程送水果(每次一个苹果或其他水果)，当二个进程分别将一个祝福写到共享数据a和一个水果放进果盘后，父进程才去享受：从数组a收取出一个祝福和吃一个水果，如此反复进行。参考程序如下：

```
1 Char fruit_disk; 苹果, 雪梨, ... =1=2
2 void main(){
3     int s;
4     s=GetSem(0);
5     if (fork())
6         while(1) { p(s); p(s); myprintf(words); fruit_disk=0;}
7     else
8         if(fork())
9             while(1) {
10                putwords "(Father will live one year after another for ever" !);
11                v(s)
12            }else
13                while(1) { putfruit(); v(s)}
14 }
15 Void putwords(char *w) { 将祝福一个词一个词放进 words[];}
16 Void putfruit() { 随机选择一个水果放进 fruit_disk; }如果顺利，编译连接这个用户程序，产生
17 oom
```

## 实验平台

xxd+dd+gcc+ld+nasm+Linux+vim  
实验报告编辑：Latex

## 算法流程图

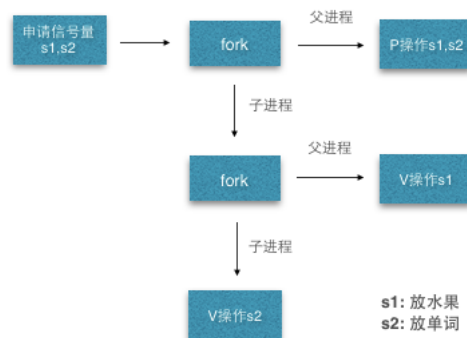


Figure 1: flow

## 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

clear, 清除当前屏幕所有字符, 刷新屏幕

help, 显示系统帮助信息

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bochs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:

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.

### 6. 使用信号量解决进程间的同步:

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. 运行 start命令，开始执行并调度所有进程

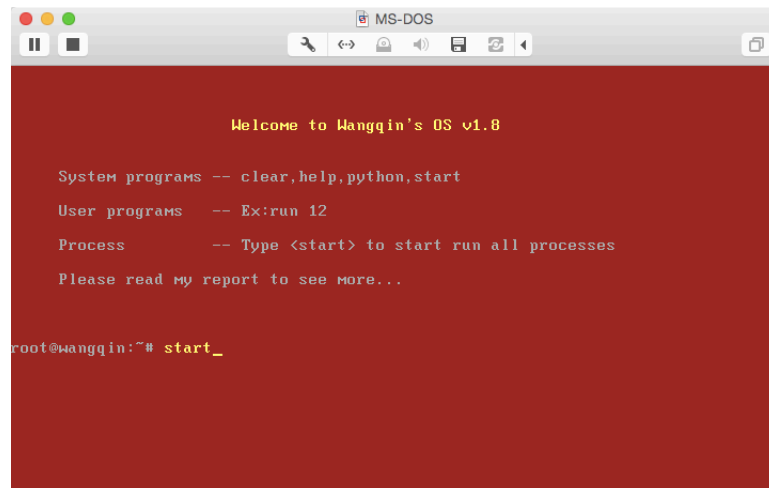


Figure 2: start

2. 下图为执行画面，利用信号量实现了进程间的同步关系，儿子放水果的时候随机放1或2（1代表苹果，2代表雪梨）可以看到每次都是两个儿子进程putfruit, putwords 之后再让第一个进程执行打印出句子和水果。

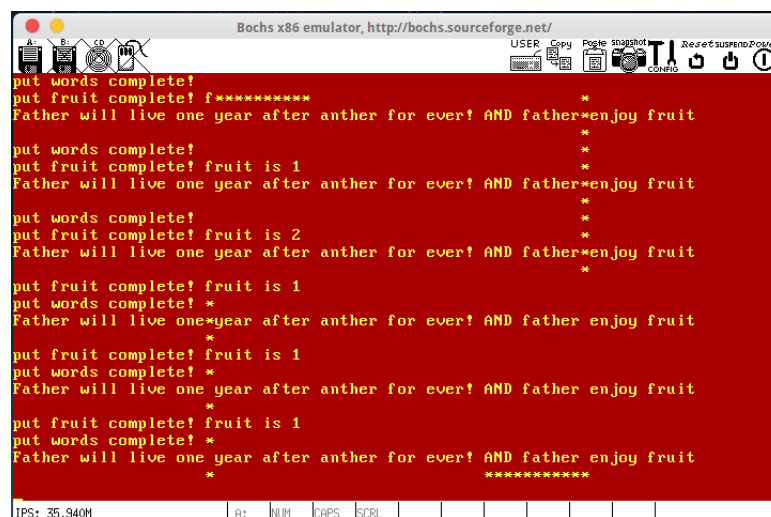


Figure 3: start

## 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

第六个进程之后会有点特殊每次进程栈会加0x1000（主要为了保障fork后的进程有足够大的栈），例如第六个是0x3000，第七个是0x4000,第八个是0x5000...

更多细节信息请阅读我的Makefile文件

## 系统目录结构

```

├── Makefile makefile 文件
├── README
├── bochsrc          bochs配置文件
├── boot.asm         引导程序
├── disk.img 镜像文件
├── kernel           目录中存放内核相关代码
│   ├── 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 系统调用相关代码
│   ├── 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 进程控制模块
│   ├── process.c     进程调度创建以及信号量相关代码
│   ├── 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_semaphore.c  father's gifts 进程(信号量解决)<本次实验要求>
├── 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
│   ├── flow.png
│   └── report.tex Latex文件
├── snapshot.txt
├── usr1.asm 用户程序1
├── usr2.asm \dots
└── usrlib.asm 用户程序专用汇编库(不与内核共享)

```

3 directories, 31 files

## 主要函数模块

1. pcb.h: semaphore,定义信号量结构体, block\_queue为阻塞队列, head,tail分别为阻塞的队列的头部和尾部, count表示 信号量的值, used表示是否正在被使用。

```

1      struct semaphore{
2          int count;
3          int block_queue[ process_num_MAX+1];

```

```

4         int used;
5         int head;
6         int tail;
7     };

```

2. muti\_process.h: GetSem09号系统调用获取信号量.

```

1     char GetSem( char value ){
2         __asm__( " cli " );
3         __asm__( " mov $9,%ah " );
4         __asm__( " int $0x80 " );
5         __asm__( " pop %ebx " );
6         __asm__( " pop %ebx " );
7         __asm__( " pop %ebx " );
8         __asm__( " sti " );
9         __asm__( " jmp *%bx " );
10    }

```

3. process.c: do\_getsem,获取信号量,同时标志此信号量已被使用, 返回信号量的主键,未找到可用的返回-1

```

1     void do_getsem(){
2         if_find = 0;
3         for(i=0;i<=sema_num_MAX;i++){
4             if( sema_array[ i ].used == 0){
5                 sema_array[ i ].used = 1;
6                 if_find = 1;
7                 sema_index = i;
8                 break;
9             }
10        }
11        if( !if_find){
12            return_sema(-1);
13        }
14        return_sema( sema_index);
15        __asm__( " pop %cx " );
16        __asm__( " pop %ecx " );
17
18        __asm__( " pop %bx " );
19        __asm__( " pop %bx " );
20        __asm__( " pop %bx " );
21        __asm__( " jmp *%bx " ); //back to os_syscall callrun
22    }

```

4. muti\_process.h: sema\_P,使用10号系统调用, stosi函数将信号量暂存在si寄存器中中断时传递给内核.

```

1     void sema_P( int s){
2         __asm__( " cli " );
3         stosi( s);
4         __asm__( " pop %cx " );
5         __asm__( " mov $10,%ah " );
6         __asm__( " int $0x80 " );
7         __asm__( " pop %ebx " );
8         __asm__( " pop %ebx " );
9         __asm__( " pop %ebx " );
10        __asm__( " sti " );
11        __asm__( " int $0x1c " );
12        __asm__( " jmp *%bx " );
13    }

```

5. process.c: do\_P,执行对信号量的P操作, 如果发现信号量小于0则阻塞当前进程。sitoindex()函数将之前保存在si中的信号量拷贝到s\_index中, 使用s\_index即可定位到对应的semaphore.阻塞队列使用循环队列的方式控制。w\_is\_r为当前正在运行的进程

```

1     void semaBlock( char s){
2         PCB_queue[ w_is_r ].process_status = BLOCK;
3         sema_array[ s ].block_queue[ sema_array[ s ].tail ] = w_is_r;
4         sema_array[ s ].tail++;
5         if( sema_array[ s ].tail>process_num_MAX)
6             sema_array[ s ].tail = 0;
7     }
8     char s_index;

```

---

```

9      void do_P(){
10         sitoindex();
11         __asm__( "pop %bx" );
12
13         sema_array[ s_index].count --;
14         if ( sema_array[ s_index].count<0){
15             semaBlock( s_index);
16         }
17
18         __asm__( "mov %bp,%sp" );
19         __asm__( "pop %bx" );
20         __asm__( "pop %bx" );
21         __asm__( "pop %bx" );
22         __asm__( "jmp *%bx" );    //back to os_syscall callrun
23     }

```

6. muti\_process.h: sema\_v,使用11号系统调用, stosi函数将信号量暂存在si寄存器中中断时传递给内核.

```

1      void sema_V( int s){
2         __asm__( " cli " );
3         stosi( s);
4         __asm__( "pop %cx" );
5         __asm__( "mov $11,%ah" );
6         __asm__( "int $0x80" );
7         __asm__( "pop %ebx" );
8         __asm__( "pop %ebx" );
9         __asm__( "pop %bx" );
10        __asm__( " sti " );
11        __asm__( "int $0x1c" );
12        __asm__( "jmp *%bx" );
13    }

```

7. process.c: do\_v,执行对信号量的V操作,如果发现信号量小于等于0则从阻塞队列中唤醒一个进程. sitoindex()函数将之前保存在si中的信号量拷贝到s\_index中,使用s\_index即可定位到对应的semaphore.阻塞队列使用循环队列的方式控制. w\_is\_r为当前正在运行的进程

```

1      void semaWakeUp( char s){
2         temp = sema_array[ s].block_queue[ sema_array[ s].head];
3         PCB_queue[ temp].process_status = READY;
4         sema_array[ s].head++;
5         if ( sema_array[ s].head > process_num_MAX)
6             sema_array[ s].head = 0;
7     }
8      void do_V(){
9         sitoindex();
10        __asm__( "pop %bx" );
11        sema_array[ s_index].count ++;
12        if ( sema_array[ s_index].count<=0){
13            semaWakeUp( s_index);
14        }
15        __asm__( "mov %bp,%sp" );
16        __asm__( "pop %bx" );
17        __asm__( "pop %bx" );
18        __asm__( "pop %bx" );
19        __asm__( "jmp *%bx" );    //back to os_syscall callrun
20    }

```

8. muti\_process.h: ReleaseSem,使用12号系统调用释放正在用的信号量。

```

1      void ReleaseSem( char value ){
2         __asm__( " cli " );
3         __asm__( "mov $12,%ah" );
4         __asm__( "int $0x80" );
5         __asm__( " sti " );
6     }

```

9. process.c: do\_free,释放信号量再次标志此信号量可用

```

1      void do_free(){
2         sitoindex();
3         __asm__( "pop %bx" );

```

```

4         sema_array[ s_index].count = 0;
5         sema_array[ s_index].used = 0;
6         sema_array[ s_index].head = 0;
7         sema_array[ s_index].tail = 0;
8         __asm__( "mov %bp,%sp" );
9         __asm__( "pop %bx" );
10        __asm__( "pop %bx" );
11        __asm__( "pop %bx" );
12        __asm__( "jmp *%bx" ); //back to os_syscall callrun
13    }

```

10. muti\_process.h: putfruit特别介绍一下这个函数，通过使用rdtsc指令来获取从开机到现在的cpu周期数然后除以2 得到一个小于2的余数以此来产生0到1的随机数，用来随机放雪梨还是苹果。

```

1    extern char fruit_disk;
2    void putfruit(){
3        fruit_disk = rdtsc_ax()%2+1;
4        printf( "put fruit complete! fruit is " );
5        printToscn( fruit_disk + 48);
6        printToscn( '\r' );
7        printToscn( '\n' );
8    }

```

11. process\_semaphore.c: main,这是本次试验展示使用信号量解决进程间同步的测试代码部分.为了便于观察效果 每次PV操作后都执行两次延时函数delay。结合本报告中流程图可以很容易理解一下代码的意思，大概就是一个父进程先fork出来一个子进程P1， 然后P1 再次fork出P2，P1和P2为父亲的两个儿子。

```

1    #include "muti_process.h"
2
3    #define DelayTime 120
4    char fruit_disk; // 苹果, 雪梨, ... =1=2
5    short int semaphore1, semaphore2;
6    char words[60];
7    char pid;
8    void main(){
9        semaphore1 = GetSem( 0);
10       semaphore2 = GetSem( 1);
11
12       if ( fork())
13           while(1) {
14               sema_P( semaphore1);
15               sema_P( semaphore2);
16               myprintf( words);
17               fruit_disk=0;
18
19               delay();
20               __asm__( "pop %ax" );
21               delay();
22               __asm__( "pop %ax" );
23
24           }
25       else{
26           if ( fork())
27               while(1) {
28                   putwords("Father will live one
29                   year after anther for ever! \0");
30                   sema_V(semaphore1);
31
32                   delay();
33                   __asm__( "pop %ax" );
34                   delay();
35                   __asm__( "pop %ax" );
36
37               }
38           else
39               while(1){
40                   putfruit();
41                   sema_V(semaphore2);
42
43                   delay();
44                   __asm__( "pop %ax" );
45                   delay();
46                   __asm__( "pop %ax" );

```

```
47     }  
48     }  
49 }
```

## 实验心得及仍需改进之处

### 实验心得:

本次实验难度与之前的实验难度相对来说跨度是最小的,从做实验的时间上来说,之前的实验所花费的时间是本次的三倍。其实就是在进程上加了个信号量来实现进程同步而已,主要是增加semaphore 结构体,以及信号量相关PV操作的实现。和之前的实验一样消耗时间最长的还是调试,本次增加的代码量并不多大部分时间都是花费在 调试上,很典型的情况就是在c中栈没有控制好导致调用的函数ret不回来,我花费了很大的时间去调整栈结构使函数能正常调用并返回。

由于实验一直在实模式下进行,我的操作系统代码量太过于庞大加载到内存的时候超过了0xb400(0x7e00 0xb400),很奇怪的是代码一超过0xb400就会出一些奇怪的错误。所以我把之前实验要求的几个系统内置功能time,date,asc删掉了,还有我自己个性化设置的查看帮助信息的man命令,这样减少了很多代码,操作系统也恢复到了正常状态,希望后面两个实验所增加的代码量不要太大,最保险的方法还是自己搞清楚为什么会出现这种问题,或者到保护模式下进行实验。本次唯一增加的一个文件就是usrlib.asm这个文件是只和用户程序进行联合编译的,并不参与内核的编译,与oslib\_share.asm的性质有些区别。

### 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

信号量池可以考虑使用链表进行存储,此外fruit\_disk,存在一个互斥的问题,当父亲在改写fruit\_disk=0的时候,儿子进程可能正在放水果,导致两个进程 对同一变量进行改写,这个问题通过多申请一个信号量很容易可以解决。