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

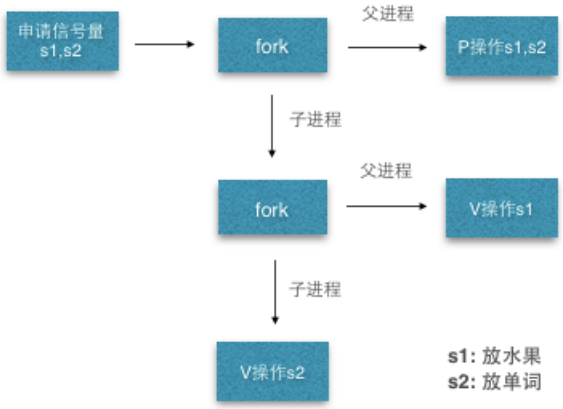
如果内核实现了信号量机制相关的系统调用，并在c库中封装相关的系统调用，那么，我们的c语言就也可以实现多进程同步的应用程序了。 例如：利用进程控制操作，父进程f创建二个子进程s和d，大儿子进程s反复向父进程f祝福，小儿子进程d反复向父进程送水果(每次一个苹果或其他水果)，当二个子进程分别将一个祝福写到共享数据a和一个水果放进果盘后，父进程才去享受：从数组a收取出一个祝福和吃一个水果，如此反复进行。 参考程序如下：

```
Char fruit_disk; 苹果，雪梨，。。。 =1=2
void main(){
        int s;
        s=GetSem(0);
        if (fork())
                while(1) { p(s); p(s); myprintf(words);fruit_disk=0;}
        else
                if(fork())
                while(1) {
                putwords “(Father will live one year after anther for ever”!);
                v(s)
                }else
                while(1) { putfruit(); v(s)}
}
Void putwords(char *w) { 将祝福一个词一个词放进 words[];}
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，产生一个文件，放进程原型操作系统映像盘中，试试有何结果?
com
```

实验平台

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

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

`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`:

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

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
│   ├── osclib.c               为osclib.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
├── osclib_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 \ldots
└── 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。

```
struct semaphore{
        int count;
        int block_queue[ process_num_MAX+1];
        int used;
        int head;
        int tail;
};
```

2. muti_process.h: GetSem09号系统调用获取信号量.

```
char GetSem( char value ){
        __asm__("cli");
        __asm__("mov $9,%ah");
        __asm__("int $0x80");
        __asm__("pop %ebx");
        __asm__("pop %ebx");
        __asm__("pop %bx");
        __asm__("sti");
        __asm__("jmp *%bx");
}
```

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

```
void do_getsem(){
        if_find = 0;
        for(i=0;i<=sema_num_MAX;i++){
                if( sema_array[ i].used == 0){
                        sema_array[ i].used = 1;
                        if_find = 1;
                        sema_index = i;
                        break;
                }
        }
        if( !if_find){
                return_sema(-1);
        }
        return_sema( sema_index);
        __asm__("pop %cx");
        __asm__("pop %ecx");

        __asm__("pop %bx");
        __asm__("pop %bx");
        __asm__("pop %bx");
        __asm__("jmp *%bx");  //back to os_syscall callrun
}
```

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.

```
void sema_P( int s){
        __asm__("cli");
        stosi( s);
        __asm__("pop %cx");
        __asm__("mov $10,%ah");
        __asm__("int $0x80");
        __asm__("pop %ebx");
        __asm__("pop %ebx");
        __asm__("pop %bx");
        __asm__("sti");
        __asm__("int $0x1c");
        __asm__("jmp *%bx");
}
```

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

```
void semaBlock( char s){
        PCB_queue[ w_is_r].process_status = BLOCK;
        sema_array[ s].block_queue[ sema_array[ s].tail] = w_is_r;
        sema_array[ s].tail++;
        if( sema_array[ s].tail >process_num_MAX)
                sema_array[ s].tail = 0;
}
char s_index;
void do_P(){
        sitoindex();
        __asm__("pop %bx");

        sema_array[ s_index].count --;
        if( sema_array[ s_index].count<0){
                semaBlock( s_index);
        }

        __asm__("mov %bp,%sp");
        __asm__("pop %bx");
        __asm__("pop %bx");
        __asm__("pop %bx");
        __asm__("jmp *%bx");  //back to os_syscall callrun
}
```

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.

```
void sema_V( int s){
        __asm__("cli");
        stosi( s);
        __asm__("pop %cx");
        __asm__("mov $11,%ah");
        __asm__("int $0x80");
        __asm__("pop %ebx");
        __asm__("pop %ebx");
        __asm__("pop %bx");
        __asm__("sti");
        __asm__("int $0x1c");
        __asm__("jmp *%bx");
}
```

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

```
void semaWakeUp( char s){
        temp = sema_array[ s].block_queue[ sema_array[ s].head];
        PCB_queue[ temp].process_status = READY;
        sema_array[ s].head++;
        if( sema_array[ s].head > process_num_MAX)
                sema_array[ s].head = 0;
}
void do_V(){
        sitoindex();
        __asm__("pop %bx");
        sema_array[ s_index].count ++;
        if( sema_array[ s_index].count<=0){
                semaWakeUp( s_index);
        }
        __asm__("mov %bp,%sp");
        __asm__("pop %bx");
        __asm__("pop %bx");
        __asm__("pop %bx");
        __asm__("jmp *%bx");  //back to os_syscall callrun
}
```

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

```
void ReleaseSem( char value ){
        __asm__("cli");
        __asm__("mov $12,%ah");
        __asm__("int $0x80");
        __asm__("sti");
}
```

9. process.c: do_free,释放信号量再次标志此信号量可用

```
void do_free(){
        sitoindex();
        __asm__("pop %bx");
        sema_array[ s_index].count = 0;
        sema_array[ s_index].used = 0;
        sema_array[ s_index].head = 0;
        sema_array[ s_index].tail = 0;
        __asm__("mov %bp,%sp");
        __asm__("pop %bx");
        __asm__("pop %bx");
        __asm__("pop %bx");
        __asm__("jmp *%bx");   //back to os_syscall callrun
}
```

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。

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

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.

```
#include "muti_process.h"

#define DelayTime 120
char fruit_disk; // 苹果，雪梨，。。。=1=2
short int semaphore1,semaphore2;
char words[60];
char pid;
void main(){
    semaphore1 = GetSem( 0);
    semaphore2 = GetSem( 1);

    if ( fork())
                while(1) {
                        sema_P( semaphore1);
                        sema_P( semaphore2);
                        myprintf( words);
                        fruit_disk=0;

                        delay();
                        __asm__("pop %ax");
                        delay();
                        __asm__("pop %ax");

                }
        else{
                  if( fork())
                          while(1) {
                                  putwords("Father will live one
                                  year after anther for ever! \0");
                                  sema_V(semaphore1);

                                  delay();
                                  __asm__("pop %ax");
                                  delay();
                                  __asm__("pop %ax");

                          }
                  else
                          while(1){
                                  putfruit();
                                  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。