《操作系统原理实验》

实验报告

（实验八）

学 院 名 称：	数据科学与计算机学院
专业（班级）：	16 计科 2 班
学 生 姓 名：	朱志儒
学 号：	16337341
时 间：	2018 年 6 月 14 日

实 验 八 ： 进程同步机制

一． 实验目的

通过信号量实现进程同步机制

二． 实验要求

1、 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。

2、 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。

三． 实验方案

1、 虚拟机配置

使用Vmware Workstation配置虚拟机，虚拟机的配置：核心数为1的处理器、4MB的内存、10MB的磁盘、1.44MB的软盘。

2、 软件工具与作用

Notepad++：编写程序时使用的编辑器；

16位编辑器WinHex：可以以16进制的方式打开并编辑任意文件；

TAMS汇编工具：可以将汇编代码编译成对应的二进制代码；

NAMS汇编工具：可以将汇编代码编译成对应的二进制代码；

TCC编译器：可以将c代码编译成对应的二进制代码；

TLINK链接器：将多个.obj文件链接成.com文件；

Bochs：可调试操作系统

WinImage：可以创建虚拟软盘。

3、 基础原理

（1）进程基本的的同步机制

即用于互斥和同步的计数信号量机制，在这个项目中，我们完善进程模型。多个进程能够利用计数信号量机制实现临界区互斥。合作进程在并发时，利用计数信号量，可以按规定的时序执行各自的操作，实现复杂的同步，确保进程并发的情况正确完成使命

（2）信号量机制

即计数信号量机制，信号量是指一个整数和一个指针组成的结构体，在内核可以定义若干个信号量，统一编号。内核实现do_p()原语，在c语言中用p(int sem_id)调用；内核实现do_v()原语，在c语言中用v(int sem_id)调用；内核实现do_getsem()原语，在c语言中用getsem(int)调用，参数为信号量的初值；内核实现do_freesem(int sem_id),在c语言中用freesem(int sem_id)调用。

信号量是内核实现的。每个信号量是一个结构体，包含两个数据域：数值count和阻塞队列头指针next。我们在内核定义一个信号量数组，同时，内核还要实现p操作和v操作。我们设置4个系统调用为用户使用信号量机制。函数SemGet(int value)向内核申请一个内核可用信号量，并将其count域初始化为value，返值就是内核分配的一个可用信号量在数组的下标，如果没有可用的信

号量，返值为－1。

4、 方案思想

本次实验是在实验七的基础上进行的，加入了计数信号量机制，实现了do_p()、do_v()、do_getsem()、do_freesem(int sem_id)原语，并将这些原语封装进C库中，供用户程序使用。

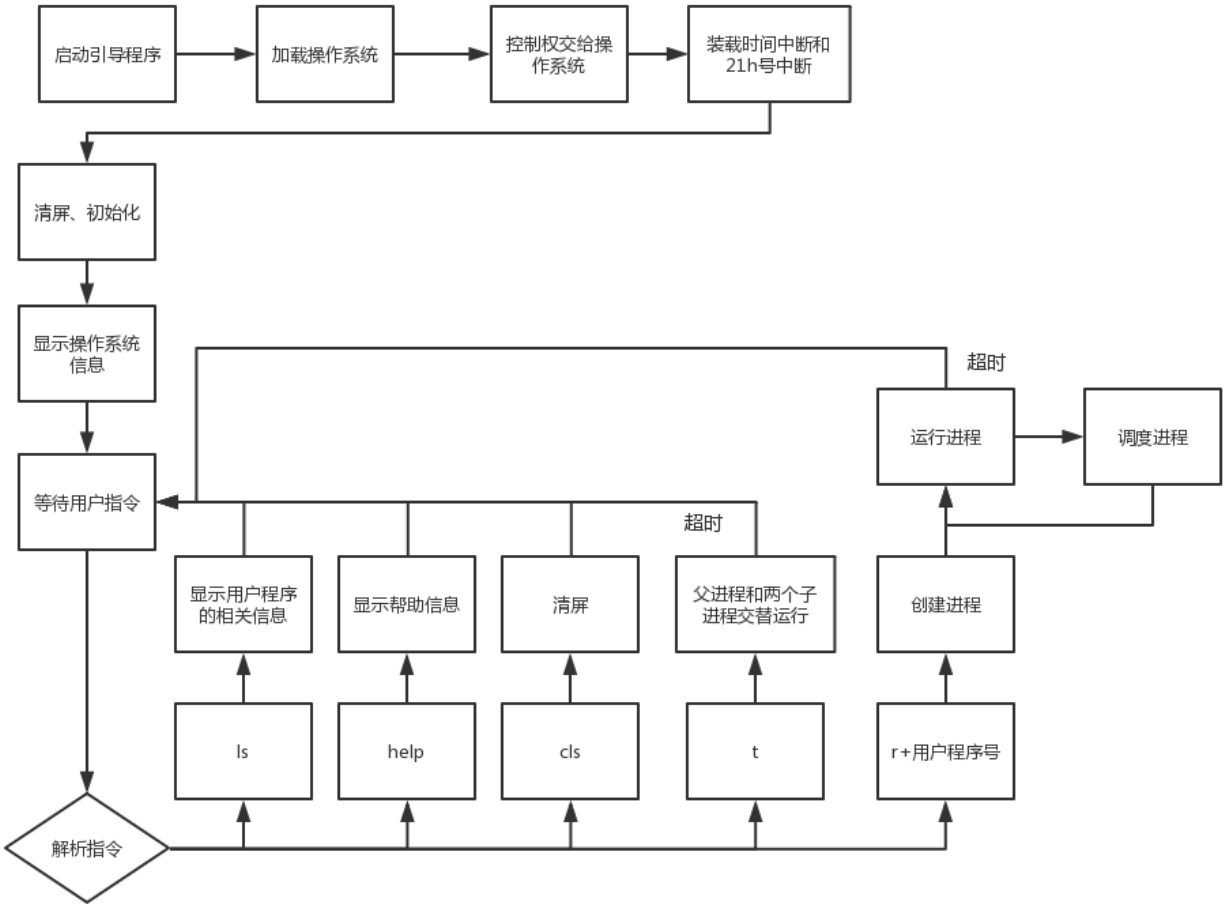
在PCB.h文件中，申明信号量的结构semaphoretype，信号量数组semaphorequeue,向内核申请一个内核可用信号量函数semaGet(int value)，即实现do_getsem()原语；释放信号量函数semaFree(int s)，即实现do_freesem(int sem_id)原语；实现将当前进程阻塞并放入信号量s的阻塞队列函数semaBlock(int s)；实现唤醒信号量s的阻塞队列中的一个进程函数semaWakeUp(int s)；实现do_P()原语的函数semaP(int s)；实现do_V()原语的函数semaV()；实现初始化信号量队列的函数initsema()。

在kliba.asm文件中,在21h中断系统调用服务中增加第7、8、9、10号服务；实现第7号semageting服务、第8号semafreeing服务、第9号semaping服务、第10号semaving服务。

在user_lib.asm文件中，封装调用21h中断第7、9、10号功能，分别作为semaGet()、semaP()、semaV()函数供用户程序使用。

最后在userc.c文件中，实现父进程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)。

5、 程序流程



6、 算法与数据结果

信号量结构:

（1）一个整数count，表示资源量；

（2）一个数组，表示阻塞队列，一个循环队列；

（3）标志位used，表示是否使用过；

（4）队头front、队尾tail。

算法:

semaGet(int value)：申请信号量。在信号量队列中找到一个未使用的信号量，完成初始化操作后，将信号量下标返回。

semaFree(int s)：释放信号量。根据传递的参数决定释放的信号量。

semaP(int s)：P操作。对信号量的count--，如果小于0则将当前进程放到阻塞队列末尾，并重新调度。

semaV(int s)：V操作。对信号量的count++，如果小于等于0，则唤醒阻塞队列里面的第一个进程。

semaWakeUp(int s)：唤醒信号量s的阻塞队列中的一个进程。

semaBlock(int s)：将当前进程阻塞并放入信号量s的阻塞队列中。

7、 程序关键模块

（1）在PCB.h文件中，信号量结构，代码如下：

```
typedef struct semaphoretype {
    int count;
    int blocked_pcb[nrpcb];
    int used, front, tail;
} semaphoretype;
```

semaGet(int value)函数，代码如下：

```
int semaGet(int value) {
    int i = 0;
    while (semaphorequeue[i].used == 1 && i < nrsemaphore) { ++i; }
    if (i < nrsemaphore) {
        semaphorequeue[i].used = 1;
        semaphorequeue[i].count = value;
        semaphorequeue[i].front = 0;
        semaphorequeue[i].tail = 0;
         PCB_LIST[current_process_number].regs.ax = i;
         PCB_Restore();
         return i;}
    else {
         PCB_LIST[current_process_number].regs.ax = -1;
         PCB_Restore();
         return -1;}}
```

semaFree(int s)函数，代码如下：

```
void semaFree(int s) { semaphorequeue[s].used = 0; }
```

semaBlock(int s)函数，代码如下：

```
void semaBlock(int s) {
    PCB_LIST[current_process_number].status = PCB_BLOCKED;
    if ((semaphorequeue[s].tail + 1) % nrpcb ==
semaphorequeue[s].front) {
        print("kernal: too many blocked processes\r\n");
        return;}
    semaphorequeue[s].blocked_pcb[semaphorequeue[s].tail] =
current_process_number;
    semaphorequeue[s].tail = (semaphorequeue[s].tail + 1) % nrpcb;}
```

semaWakeUp(int s)函数，代码如下：

```
void semaWakeUp(int s) {
    int t;
    if (semaphorequeue[s].tail == semaphorequeue[s].front) {
        print("No blocked process to wake up\r\n");
        return;}
    t = semaphorequeue[s].blocked_pcb[semaphorequeue[s].front];
    PCB_LIST[t].status = PCB_READY;
    semaphorequeue[s].front = (semaphorequeue[s].front + 1) % nrpcb;}
```

semaP(int s)函数，代码如下：

```
void semaP(int s) {
    semaphorequeue[s].count--;
    if (semaphorequeue[s].count < 0) {
        semaBlock(s);
        schedule();}
    PCB_Restore();}
```

semaV(int s)函数，代码如下：

```
void semaV(int s) {
    semaphorequeue[s].count++;
    if (semaphorequeue[s].count <= 0) {
```

```
        semaWakeUp(s);
        schedule();}
    PCB_Restore();}
```

Initsema()函数，代码如下：

```
void initsema() {
    int i;
    for (i = 0; i < nrsemaphore; ++i) {
        semaphorequeue[i].used = 0;
        semaphorequeue[i].count = 0;
        semaphorequeue[i].front = 0;
        semaphorequeue[i].tail = 0;}}
```

（2）在kliba.asm文件中，Semaving功能代码如下：

```
semaving:
    .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
    mov ax,cs
    mov ds, ax
    mov es, ax
    call _save_PCB
    mov bx,ax
    push bx
    call near ptr _semaV
    pop bx
    iret
```

semaping功能代码如下：

```
semaping:
    .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
    mov ax,cs
    mov ds, ax
    mov es, ax
    call _save_PCB
    mov bx,ax
    push bx
    call near ptr _semaP
    pop bx
    iret
```

semafreeing功能代码如下：

```
semafreeing:
    .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
    mov ax,cs
    mov ds, ax
    mov es, ax
    call _save_PCB
    mov bx,ax
    push bx
    call near ptr _semaFree
    pop bx
    iret
```

semageting功能代码如下：

```
semageting:
    .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
    mov ax,cs
    mov ds, ax
    mov es, ax
    call near ptr _save_PCB
    mov bx,ax
    push bx
    call near ptr _semaGet
    pop bx
    iret
```

（3）在user_lib.asm文件中，封装semaGet()函数部分代码如下：

```
public _semaGet
_semaGet proc
    mov ah, 7
    int 21h
    ret
_semaGet endp
```

封装semaP()函数部分代码如下：

```
public _semaP
_semaP proc
    mov ah, 9
    int 21h
    ret
_semaP endp
```

封装semaV()函数部分代码如下：

```
public _semaV
_semaV proc
    mov ah, 10
    int 21h
    ret
_semaV endp
```

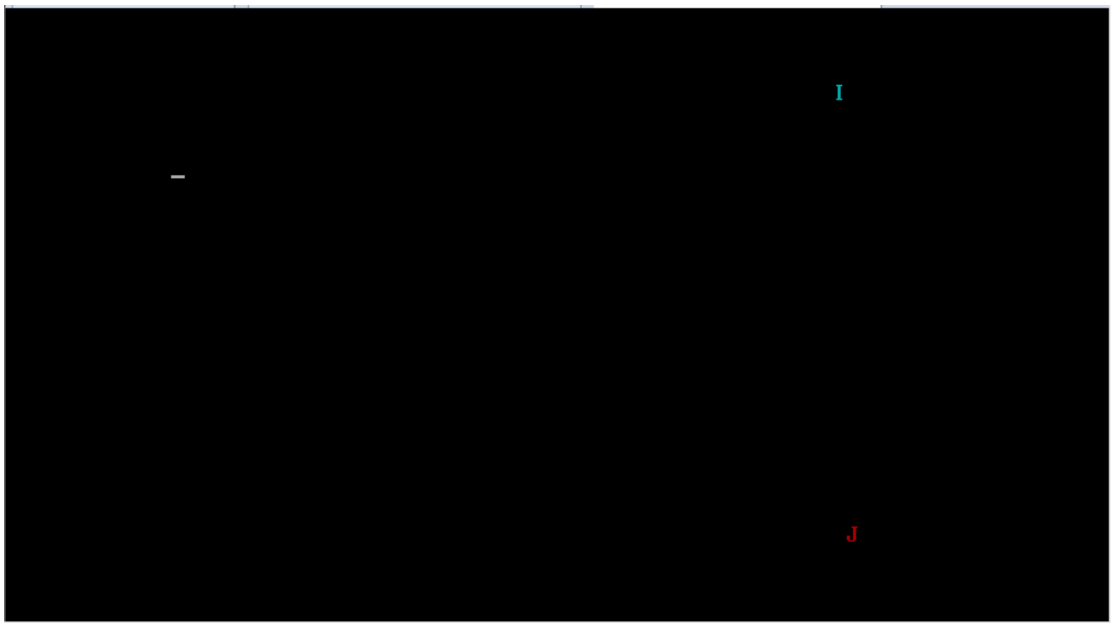
（4）在userc.c文件中实现测试程序，代码如下：

```
#include "user_lib.h"
char words[100];
int be_change = 0, fruit_disk, rand = 0;
void write(char *p) {
    int i = 0;
    while(*p != '\0') {
        words[i++] = *p;
        p++;}}
void putfruit() { fruit_disk = rand++ % 10; }
main() {
   int s, tmp;
   s = semaGet(0);
```

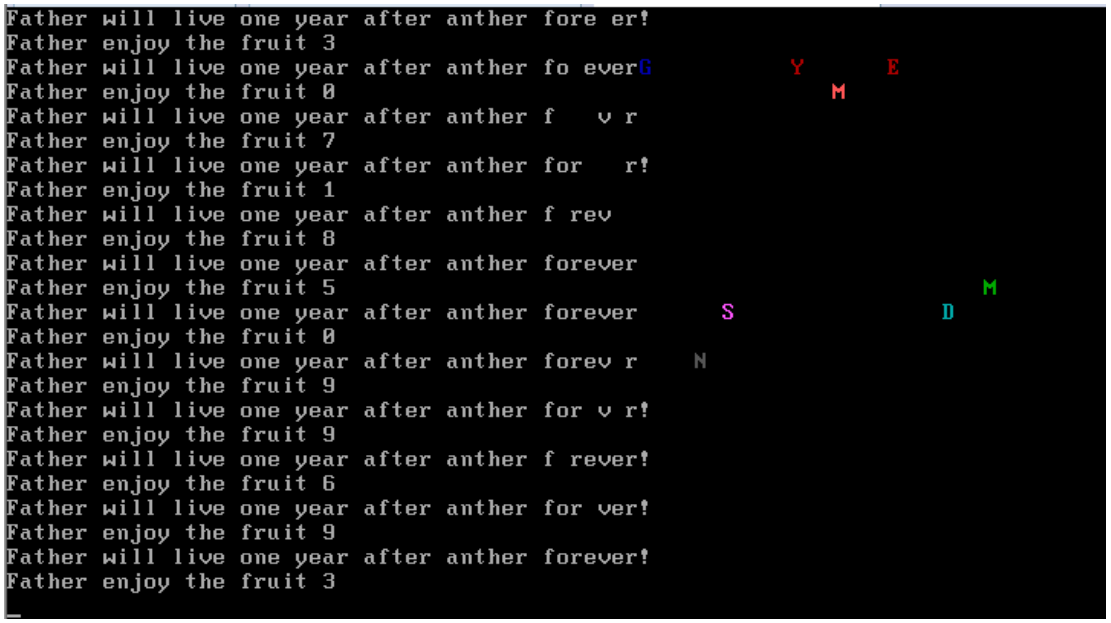
```
   print("\r\nUser: forking...\r\n");
   tmp = fork();
   if(tmp) {
       while(1) {
           semaP(s);
           semaP(s);
           if(be_change) {
               print(words);
               be_change = 0;}
           print("Father enjoy the fruit ");
           printInt(fruit_disk);
           print("\r\n");
           fruit_disk = 0;}}
   else {
       print("User: forking again...\r\n");
       tmp = fork();
       if(tmp) {
           while(1) {
               be_change = 1;
               write("Father will live one year after anther
forever!\r\n");
               semaV(s);
               delay(5000);}}
       else {
           while(1) {
               putfruit();
               semaV(s);
               delay(5000);}}}}
```

四． 实验过程和结果

1、 输入指令r 24，程序2、4并发执行，一段时间后返回内核，从右下角动态的‘-’可观察到，如图所示。（并发运行2、4两个程序的目是体现测试程序是并发执行的，测试时可以不运行2、4程序。）



2、 返回后按下回车即可再次输入指令，输入指令t，运行测试信号量程序。由于步骤1的程序并未退出，此时共有5个进程在运行，分别是程序2、4和测试程序t及其两个子进程。如图所示，可以看到当两个子进程分别将一个祝福写到共享数据和一个水果放进果盘后，父进程才去享受，即从共享数据取出一个祝福和吃一个水果，如此反复进行。



3、 运行一段时间后返回内核，从右下角动态的‘-’可观察到。再次按下回车即可输入指令。

五． 实验总结

总结:

通过这次实验，我对信号量有了更加深刻的理解。

为了防止出现因多个进程同时访问一个共享资源而引发的一系列问题，我们使用信号量机制，在任一时刻只能有一个执行进程访问代码的临界区域。我们使用信号量来实现互斥，防止多个进程因同时访问共享数据而引发RC问题。

本次实验，我们使用信号量机制实现多进程通过共享来合作。例如，父进程f创建两个子进程s和d，这两个子进程分别将一个祝福和一个水果写入共享数据后，父进程从共享数据中取出一个祝福并吃一个水果。我们使用信号量S来实现这三个进程的合作。父进程f执行两次P操作，因为它既要等待s进程送的水果，又要等待d进程送的祝福。子进程s和d分别送一次水果和一次祝福后，也要分别执行一次P操作。这样就实现了这三个进程通过共享数据来合作。

上述的只是用户程序部分，最重要的是如何在内核中实现这信号量机制。

有了实验七的基础，这次实验的难度并不是特别大。在汇编代码部分中，semaV、semaP、semaGet、semaFree这四个操作和实验七中的fork、wait、exit这三个操作十分相似，均是保护现场，将进程的上下文存入进程控制块中，再进入内核进行相关操作。

在内核操作中，我觉得难度较大的几点是如何实现信号量阻塞队列、如何唤醒信号量的阻塞队列中的一个进程、如何将当前进程阻塞并放入信号量s的阻塞队列中。

六． 参考文献

1、 《x86 PC汇编语言，设计与接口》

2、 Windows 下BOCHS的使用

http://blog.51cto.com/liyuelumia/1562508