《操作系统实验》

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）：	17 级计算机类教务 3 班
学 生 姓 名 ：	姚森舰
学　　号 ：	17341189
时　　间 ：	2019　年　6　月　25　日

实验目的

通过信号量实现进程同步机制。

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。

实验方案

实验环境

Windows10 +VMware

实验工具

NASM + Winhex + Tasm + Tcc + Dosbox + Notepad++

主要思路

本次实验是在实验7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`getsem(int value)`，即调用`do_getsem()`原语；释放信号量函数`freesem(int s)`，即调用`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)`；调用`do_p()`原语的函数`p(int s)`；调用`do_v()`原语的函数`v()`；实现初始化信号量队列的函数`initsema()`。

在`setint.asm`文件中，在21h中断系统调用服务中增加第12,13,14,15号服务，分别为`do_p(), do_v(), freesem(), getsem()`.

在`proces2.asm`文件中，封装调用21h中断第12,13,15号功能，分别作为`p()、v()、getsem()`函数供用户程序使用，因为`do_freesem(int sem_id)`原语在测试中没有用到，故没有封装。注意asm文件命名**不能超过7个字符**，否则会出现找不到文件的错误，这在之前的实验中已经提到，这里提醒一下。

最后在`test2.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，当有水果和祝福时，父进程将吃水果。

由于实验8是在实验7的基础上进行的，所以很多关于`fork()`等操作的实现和实验7相同，这里不再赘述。

`PCB.H`

新增的一些常量和数据结构

```
#define max_semaphore 8                 /* 最多可用信号量 */
#define max_blocked_pcb 8               /* 阻塞进程队列长度 */

typedef struct semaphoretype {
    int count;                          /* 资源数量 */
    int blocked_pcb[max_blocked_pcb];     /* 循环队列 */
    int used;                           /* 该信号量是否被使用 */
    int front, tail;                    /* 循环队列的头尾指针 */
} semaphoretype;

semaphoretype semaphorequeue[max_semaphore];      /* 定义信号量的数组 */
```

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

```
/* 将当前进程阻塞并放入信号量s的阻塞队列中 */
void semaBlock(int s) {
    pcb_list[CurrentPCBno].Process_Status = BLOCKED;
    /* 如果阻塞队列已满，什么都不做 */
    if ((semaphorequeue[s].tail + 1) % max_blocked_pcb == semaphorequeue[s].front) {
        return;
    }
    /* 将该进程放到队列最后，循环队列需要注意一下 */
    semaphorequeue[s].blocked_pcb[semaphorequeue[s].tail] = CurrentPCBno;
    semaphorequeue[s].tail = (semaphorequeue[s].tail + 1) % max_blocked_pcb;
}
```

`semaWakeUp(int s)` 唤醒信号量s的阻塞队列中的一个进程。

```
/* 唤醒信号量s的阻塞队列中的一个进程 */
void semaWakeUp(int s) {
    int t;
    /* 如果没有阻塞进程，什么都不做 */
    if (semaphorequeue[s].tail == semaphorequeue[s].front) {
        return;
    }
    /* 拿出队首进程并唤醒 */
    t = semaphorequeue[s].blocked_pcb[semaphorequeue[s].front];
    pcb_list[t].Process_Status = READY;
    semaphorequeue[s].front = (semaphorequeue[s].front + 1) % max_blocked_pcb;
}
```

do_p()

P操作，向系统申请资源。对信号量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。

```
/* 信号量的P操作，申请资源 */
void do_p(int s) {
    /* 可用资源减少 */
    semaphorequeue[s].count--;
    /* 没有可用资源则阻塞当前进程 */
    if (semaphorequeue[s].count < 0) {
        semaBlock(s);
        Schedule();
    }
    Switch();
}
```

`Switch()` 和 `Schedule()` 都是实验7中的函数，前者就是切换进程后重新开始的实现，后者为调度函数，详见实验7.

do_v()

V操作，向系统申请资源。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。

```
/* 信号量的P操作，释放资源 */
void do_v(int s) {
    semaphorequeue[s].count++;
    /* 如果有被阻塞的进程，则可将其唤醒 */
    if (semaphorequeue[s].count <= 0) {
        semaWakeUp(s);
        Schedule();
    }
    Switch();
}
```

do_freesem()

释放对应信号量，在测试中并没有用到。

```
/* 释放对应的信号量 */
void do_freesem(int s) {
    semaphorequeue[s].used = 0;
}
```

do_getsem()

向内核申请一个内核可用信号量。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。

```
/* 向内核申请一个内核可用信号量，并将count初始化为value */
int do_getsem(int value) {
    int i = 0;
    /* 找没被用的信号量 */
    while (semaphorequeue[i].used == 1 && i < max_semaphore) { ++i; }
    if (i < max_semaphore) {
        semaphorequeue[i].used = 1;
        semaphorequeue[i].count = value;
        semaphorequeue[i].front = 0;
        semaphorequeue[i].tail = 0;
        pcb_list[CurrentPCBno].regImg.AX = i;       /* 告知父进程 */
        Switch();
        return i;                                   /* 返回用于定位信号量的值 */
    }
    else {
        pcb_list[CurrentPCBno].regImg.AX = -1;      /* 为找到可用的信号量 */
        Switch();
        return -1;
    }
}
```

`setint.asm`

将这4个函数作为系统调用，因为此部分实现相似，故只列出`do_v()`，其余见代码：

```
    cmp ah,11
    jz helpJumpFun11

    cmp ah,12
    jz helpJumpFun12

    cmp ah,13
    jz helpJumpFun13

    cmp ah,14
    jz helpJumpFun14

    cmp ah,15
    jz helpJumpFun15
    ; 实际跳转区间超出jz可行的范围，所以引入中间跳板，之前的实验报告中已说明
helpJumpFun11:

    jmp fun11
```

```
helpJumpFun12:
    jmp fun12
helpJumpFun13:
    jmp fun13
helpJumpFun14:
    jmp fun14
helpJumpFun15:
    jmp fun15
    ...

fun12:
    jmp semaping
    iret
fun13:
    jmp semaving
    iret
fun14:
    jmp semafreeing
    iret
fun15:
    jmp semageting
    iret
    ...

extern _do_getsem:near
extern _do_freesem:near
extern _do_p:near
extern _do_v:near
semaving:
    push ss
    push ax
    push bx
    push cx
    push dx
    push sp
    push bp
    push si
    push di
    push ds
    push es
    .386
    push fs
    push gs
    .8086

    mov ax,cs
    mov ds, ax
    mov es, ax
    ; 向Save_Process()传参
    call near ptr _Save_Process
    mov bx,ax
    push bx

    call near ptr _do_v
```

```
    pop bx
    iret
```

其他相关或者新增部分代码

`syscll.asm`

加载实验7或实验8进程的函数

```
public _run_test_of_7_or_8
_run_test_of_7_or_8 proc
    push ax
    push bp

    mov bp,sp

    mov ax,[bp+6]       ;段地址 ; 存放数据的内存基地址
    mov es,ax           ;设置段地址（不能直接mov es,段地址）
    mov bx,100h         ;偏移地址; 存放数据的内存偏移地址
    mov ah,2            ;功能号
    mov al,2            ;扇区数 实验7的测试程序要两个扇区
    mov dl,0            ;驱动器号 ; 软盘为0，硬盘和U盘为80H
    mov dh,1            ;磁头号 ; 起始编号为0
    mov ch,0            ;柱面号 ; 起始编号为0   *************
    mov cl,[bp+8]       ;起始扇区号 ; 起始编号为1
    int 13H             ; 调用中断

    pop bp
    pop ax
    ret
_run_test_of_7_or_8 endp
```

`kernel.c`

修改测试指令`t`，

```
-t1: to run test program for lab7.
-t2: to run test program for lab8.
```

指令调用以下函数，实现测试：

```
void run_test_program_of_8()
{
    run_test_of_7_or_8(Segment,7*2-1);
    Segment += 0x1000;
    Program_Num=1;
    pcb_list[1].Used = 1;
}
```

`process2.asm`

封装`do_p()、do_v()、do_getsem()、do_freesem(int sem_id)`,过程相似，只举一个例子，例子如下，详见代码。

```
public _p
_p proc
    mov ah, 12
    int 21h
    ret
_p endp
```

测试

测试代码：

```
#include "process2.h"
extern void print();
extern void printchar();
extern void delay();
extern int getsem();
extern void p();
extern void v();
extern int fork();
extern void wait();
extern int exit();

main() {
   int s, cid;
   s = getsem(0);                          /* 初始化信号量 */
   print("\r\nUser: forking...\r\n");
   cid = fork();                           /* 创建子进程1 */
   if(cid) {
       while(1) {
           p(s);                           /* 同时有水果和祝福时吃水果 */
           p(s);

           print("Father enjoys the fruit.\r\n");
```

```
        }
    }
    else {                                  /* 子进程1，送祝福 */
        print("User: forking again...\r\n");
        cid = fork();                       /* 创建子进程2 */
        if(cid) {
            while(1) {
                print("Father will live forever!\r\n") ;
                v(s);
                delay();
            }
        }
        else {                              /* 子进程2，送水果 */
            while(1) {
                print("Put one fruit onto the plate.\r\n") ;
                v(s);
                delay();
            }
        }
    }
}
```

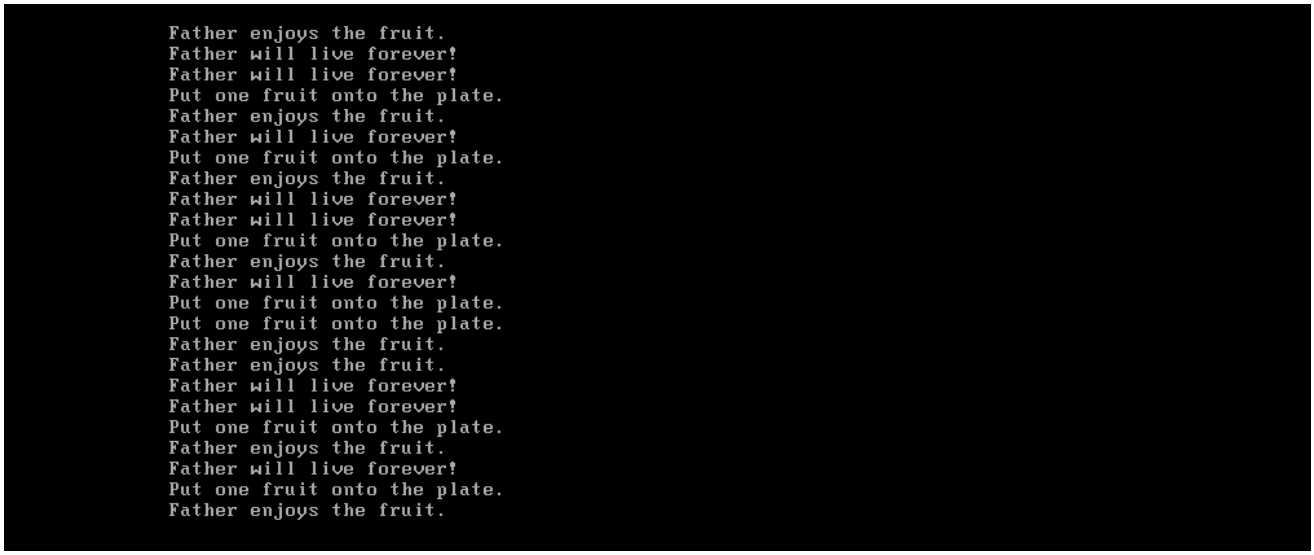
测试结果

```
 ------------------------------------------------------------------------
|                            Welcome to YaoOS                            |
 ------------------------------------------------------------------------

Command help:
-l: to get some help information.
-r n: (n is the index of user program.) to run the user program with index 'n'.
    For example: input "r 1 3" to run user program1 and then program3.
-h: to get some help information.
-p: to run multi-process program."p 12" to run prog1 and prog2 at the same time.

-t1: to run test program for lab7.
-t2: to run test program for lab8.
-s: shut down my OS.
-c: clear screen.
-q: quit.
-init.cmd: to run the batch commmand in the disk.
user $: t2_
```

```
User: forking...
User: forking again...
Father will live forever!
Put one fruit onto the plate.
Father enjoys the fruit.
Father will live forever!
Father will live forever!
Put one fruit onto the plate.
Father enjoys the fruit.
Father will live forever!
Put one fruit onto the plate.
Father enjoys the fruit.
Father will live forever!
Father will live forever!
Put one fruit onto the plate.
Father enjoys the fruit.
Father will live forever!
Put one fruit onto the plate.
```



可以看到，只有当子进程分别释放资源(也就是做了v操作)时，父进程才会输出。

总结

1. 本次的实验在实验7的基础上完成，关键的地方在实验7已经完成，实现的这几个函数和实验七中的fork、wait、exit这三个操作十分相似，流程都是先保护现场，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，只要对原理理解了，做起来相对来说就要简单一些。不过还好是实验7中可能出现的问题并没有在本次实验中出现，所以感觉比较幸运。创建子进程并调度那部分就像是搭积木，修改了一点，有可能就全部崩塌了，所以每次修改都是如履薄冰。
2. 实验7实验8的实验报告中删减了一些不重要步骤，比如创建虚拟机、编译这两部分，这些在之前的实验报告中已写明，为节省时间不在赘述。
3. 测试这一部分，并没有看懂老师的PPT的意思，就稍做了修改，直接输出语句，比较直观。父进程要做2次p操作才可以输出，所以只有等两个子进程分别释放资源(或者说产生资源)后才能输出，子进程在输出信息后会释放资源，观察对应的输出就知道此刻是哪个进程在运行。但是感觉这样做不是很好的方案，但是已经在操作系统实验上花了太多时间，临近期末，怕再改动出了问题，再DEBUG就没法复习了。
4. 总的来说，这学期的操作系统实验课是真的硬核，但也能让我学到很多，让我对操作系统的实现框架有了一定的认识。除了与操作系统相关的知识外，也给了我很多教训，让我知道了细节的重要性，在之前的实验中，常常因为小细节的问题花费很多时间，发现问题之后真的是懊恼不已。另外这些实验也让我知道了规范编码的重要性，当代码文件多且大的时候，不写注释和不规范命名、编码都会给阅读带来很大的影响，开始的几个实验我就写的不规范，到后面想改都不敢改了，一是时间不多，二是怕修改的时候漏了什么地方就很麻烦。不过，尽管过程很痛苦，最终还是受益匪浅，感谢老师和助教的帮助和付出！