

# 实验 8：进程同步机制

姓名：姜洋帆 学号：17341068

院系：数据科学与计算机学院 专业：17 级计算机科学（大数据）

**【实验题目】**进程控制与通信

**【实验目的】**

1. 实现基本的进程同步机制

**【实验要求】**

1. 内核实现 `do_p()`原语，在 c 语言中用 `p(int sem_id)`调用
2. 内核实现 `do_v()`原语，在 c 语言中用 `v(int sem_id)`调用
3. 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
4. 内核实现 `do_freeseem(int sem_id)`,在 c 语言中用 `freeseem(int sem_id)`调用

**【实验方案】**

一、硬件及虚拟机配置：Lenovo PC ； Oracle Virtual Box

二、软件工具及作用：

Notepad++ ：写代码

Nasm：编译引导程序的 asm 文件到 bin 文件

Tasm：编译 16 位的内核代码

Tcc：编译 16 位 C 语言，实现 C 语言与 x86 联合编译

Sublime ：查看、编辑二进制文件，将编译完成的程序机器码写入软盘

三、程序功能

在内核模式下，命令行输入 `test`，运行指定的用户程序（使用信号量机制，实现父进程与两个子进程的通信协作）。首先在父进程申请一个信号量 `s`，然后 `fork` 出两个子进程，两个儿子分别不断得提供水果和祝福，只有父亲获取到一个祝福和一个水果时，父亲才会去享用。

#### 四、程序设计

这次试验基于 fork 实验得基础上，在内核新增信号量 semaphore 数据结构以及一些原子操作，并封装相关操作供用户程序使用。如果在之前的实验中做好资源调度和五状态模型，只需要做一些简单的扩展就可以了，实现比较简单。

信号量数据结构包括计数器 count，代表可用资源的数量、used 标志，代表该信号量是否被申请使用、block\_list，代表被阻塞的进程列表，当信号量相对应的资源可用时，从阻塞队列里找一个进程（队列头）唤醒。阻塞队列使用简单的数组结构实现，定义两个头尾指针下标，实现一个简单的循环队列。

Get 和 free 操作主要是修改 semaphore 的 used 字段，其中 get 操作需要先判断是否有可用的信号量，若无则返回-1，报错，否则就进行一些简单的初始化，重启当前被中断的进程（在调用前需要保存用户程序 PCB，在 syscall.asm 中实现）。

Semblock 和 semwakeup 函数将当前进程放入或者移出阻塞队列，主要在 pv 操作时，资源用完或者有可用资源时被调用。

Do\_p,do\_v 判断资源状况（count），然后决定下一步操作。

#### 五、主要代码

##### 1. semaphore 数据结构

包括 count（资源数），used，阻塞队列 block\_List，和头尾指针 front，tail

```
typedef struct semaphore {
    int count;
    int used;
    int block_list[100];
    int front, tail;
}semaphore;
```

##### 2. 信号量列表:

代表内核中可用的信号量数量，这里假定为 100 个，实验实际上只用到了 1 个。

```
semaphore sem_list[100];
```

### 3. 获取信号量操作:

用户程序申请一个信号量，调用内核的 `do_getsem` 过程，主要的逻辑是查询当前是否有可用的信号量，若有则将 `used` 位置为 1，进行简单的初始化，最后给 `AX` 寄存器赋值，返回被选中的信号量的下标。

```
int do_getsem(int value) {
    int i = 0;
    while(sem_list[i].used == 1 && i < sem_num)
        i++;
    /* get a free semaphore */
    if (i < sem_num) {
        sem_list[i].used = 1;
        sem_list[i].count = value;
        sem_list[i].front = 0;
        sem_list[i].tail = 0;
        pcb_list[CurrentPCBno].regImg.AX = i;
        PCBrestart();
        return i;
    }
    else {
        pcb_list[CurrentPCBno].regImg.AX = -1;
        PCBrestart();
        return -1;
    }
}
```

### 4. 释放信号量

直接修改 `used` 字段值为 0 即可。

```
int do_freeseem(int sem_id) {
    sem_list[sem_id].used = 0;
}
```

### 5. 阻塞当前进程

当调用 `P` 操作，发现资源不够用时，会引发进程阻塞，此时需要调用该过程。首先将进程状态改为 `WAIT`，然后判断阻塞队列是否满，若满则报错返回，否则将该进程号放入队尾，调整 `tail` 变量。

代码如下:

```
void sem_block(int s) {
    int tail;
    tail = sem_list[s].tail;
    pcb_list[CurrentPCBno].Process_Status = WAIT;
    if((sem_list[s].tail + 1) % sem_num == sem_list[s].front) {
        printf("ERROR: the block list has been full\n\r");
        return;
    }
}
```

```

    }
    sem_list[s].block_list[tail] = CurrentPCBno;
    sem_list[s].tail = (tail + 1) % sem_num;
}

```

## 6. 唤醒阻塞队列中的进程:

在执行 V 操作，若 count 为负数（代表正在等待资源的进程数），则需要唤醒一个阻塞队列中的进程（队列头的进程），修改进程状态并调整头指针 front 即可。

```

void sem_wakeup (int s) {
    int pcbnum;
    if (sem_list[s].front == sem_list[s].tail) {
        printf("Block list is empty!\n\r");
        return;
    }
    pcbnum = sem_list[s].block_list[sem_list[s].front];
    pcb_list[pcbnum].Process_Status = READY;
    sem_list[s].front = (sem_list[s].front + 1) % sem_num;
}

```

## 7. P 操作

将资源 count 减 1，若 count 变为负数，表示资源不够用，需要将当前进程放入阻塞队列，调用 sem\_block 函数实现。然后再进行一次调度，重启新进程。

```

void do_p(int s) {
    sem_list[s].count--;
    if (sem_list[s].count < 0) {
        sem_block(s);
        Schedule();
    }
    PCBrestart();
}

```

## 8. V 操作

将 count 加 1，若 count<=0，代表此时有正在等在资源的进程，需要调用 sem\_wakeup 过程，将一个进程从阻塞队列中唤醒。

```

void do_v(int s) {
    sem_list[s].count++;
    /* 正在等待资源的进程数（负数） */
}

```

```

        if (sem_list[s].count <= 0) {
            sem_wakeup(s);
            Schedule();
        }
        PCBr restart();
    }
}

```

## 9. 封装为相关的系统调用

将上面实现的内核过程封装为系统调用。由用户程序跳转进内核前，需要保存当前进程的状态信息，即保存 PCB，这些与 fork 的相关操作实现类似，以封装 P 操作为例（具体代码见 syscall.asm 文件）：

```

system_call:
;中断调用的参数
    sti

    push bx    ;将参数 ax 传进来，并弹出到 bx 寄存器中
    mov bx,ax

    push dx
    push ax
    push cx
    push bp
    push ds
    push es

    mov ax,cs
    mov ds,ax
    mov es,ax

    mov ax,bx    ; 将传进来的 ax 参数赋值给 ax
    cmp al,11
    jz to_p

to_p:
    pop es
    pop ds
    pop bp
    pop cx
    pop ax
    pop dx

```

```
pop bx  
jmp p
```

```
p:  
    .386  
    push ss  
  
    .8086  
    push ax  
    push bx  
    push cx  
    push dx  
    push sp  
    push bp  
    push si  
    push di  
  
    .386  
    push ds  
    push es  
    push fs  
    push gs  
    .8086  
  
    mov ax,cs  
    mov ds, ax  
    mov es, ax  
    call near ptr _Save_Process  
  
    ;mov bx,ax  
    ;push bx  
    call near ptr _do_p  
    ;pop bx  
    iret
```

## 10. wakeup

```
void wakeup() {
```

```

        PCB* curr_pro = Current_Process();
        curr_pro->Process_Status = READY;
        Schedule();
        PCBrestart();
    }

```

## 11. 用户程序封装系统调用

将系统调用封装为供用户程序调用的函数：

```

public _getsem
_getsem proc
    mov al,9
    int 21h
    ret
_getsem endp

```

```

public _freeseem
_freeseem proc
    mov al,10
    int 21h
    ret
_freeseem endp

```

```

public _p
_p proc
    mov al,11
    int 21h
    ret
_p endp

```

```

public _v
_v proc
    mov al,12
    int 21h
    ret
_v endp

```

## 12. 用户程序实现 delay 函数

使用 15h 中断（ah=86h），实现延迟，方便查看执行结果 cx、dx 分别为延迟（毫秒数）的高位和低位。这里大约延迟一秒

```
public _delay
_delay proc
    mov ah,86h.
    mov cx,000fh
    mov dx,4240h
    int 15h
_delay endp
```

## 13. 测试用户程序

按照实验 PPT 要求以及给出的用户程序代码，做了一些修改：共 5 中水果，循环放入水果盘中（随机选择太难实现）。两个子进程每次增加资源时，都会执行 V 操作增加资源数量，并等待 1 秒左右，父进程连续执行两次 P 操作，当两项资源都有时才会进一步操作。由于两个子进程都是延迟一秒，所以可用保证当有两项资源时，一定是水果和祝福各一份。

```
extern int fork();
extern void wait();
extern void exit();
extern void printChar();
extern void cls();
extern int getsem();
extern int freesem();
extern void p();
extern void v();

char words[100];
int fruit_disk = 0;
int s;
int f;

void printf(char *str) {
    while(*str!='\0') {
        printChar(*str);
        str++;
    }
}

void printint(int n) {
    int i = 0;
    int tmp = 0;
```



```

    int len = 0;
    char ans[100];
    char out[100];
    if (n == 0) {
        printf("0");
        return;
    }

    while(n) {
        tmp = n % 10;
        n /= 10;
        ans[len++] = '0' + tmp;
    }
    for(i=0 ;i<len; i++) {
        out[i] = ans[len -1 - i];
    }
    out[i] = '\0';
    printf(out);
}

void putwords(char *p) {
    int i = 0;
    while(*p != '\0') {
        words[i++] = *p++;
    }
}

void putfruit() {
    fruit_disk %= 5;
    fruit_disk += 1;
}

void main()
{
    int pid;
    /* s:words, f: fruit */
    s = getsem(0);
    f = getsem(0);
    printf("\n\rUser program: forking...\n\r");
    pid = fork();
    /* father */
    if(pid) {
        while(1) {

```

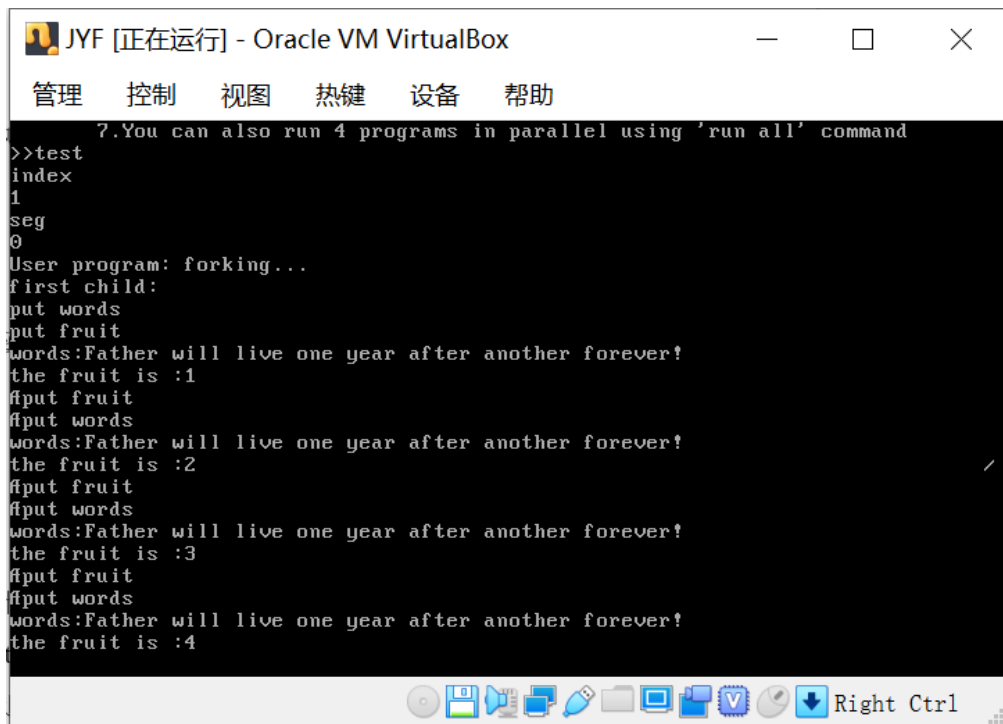
```

        p(s);
        p(s);
        printf("words:");
        printf(words);
        printf("\n\r");
        printf("the fruit is :");
        printint(fruit_disk);
        printf("\n\r");
    }
}
/* first child */
else {
    printf("first child:\n\r");
    pid = fork();
    /* first child */
    if(pid) {
        while(1) {
            printf("put words\n\r");
            putwords("Father will live one year after another
forever!");
            v(s);
            delay();
        }
    }
    /* second child */
    else {
        while(1) {
            printf("put fruit\n\r");
            putfruit();
            v(s);
            delay();
        }
    }
}
}
}

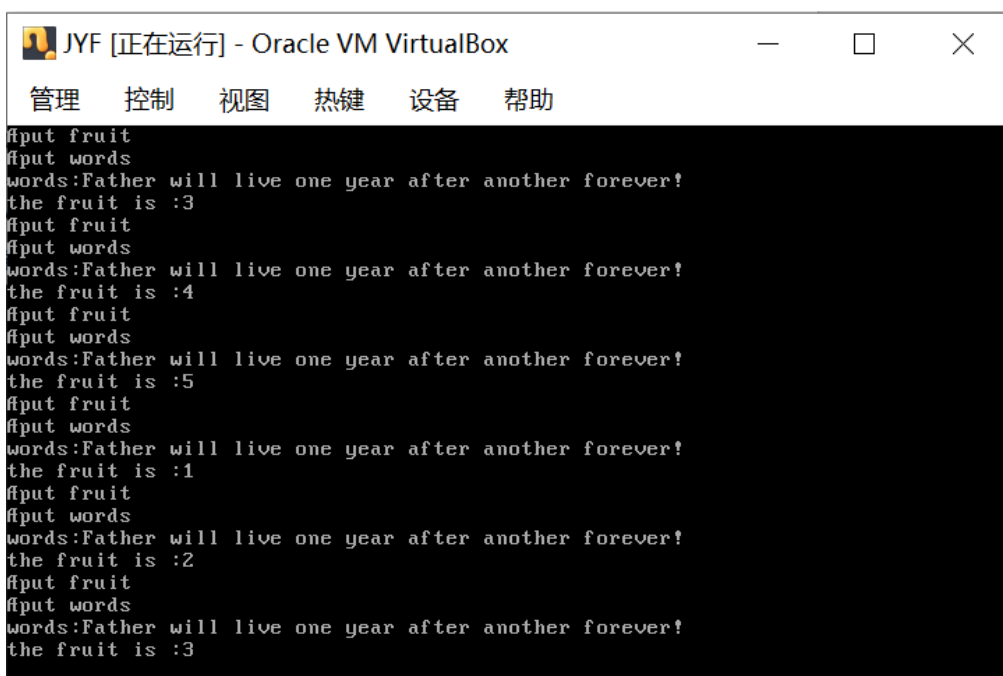
```

## 【实验过程】

在控制台输入 test 命令，执行本次实验的测试：



```
JYF [正在运行] - Oracle VM VirtualBox
管理 控制 视图 热键 设备 帮助
7.You can also run 4 programs in parallel using 'run all' command
>>test
index
1
seg
0
User program: forking...
first child:
put words
put fruit
words:Father will live one year after another forever!
the fruit is :1
ffput fruit
ffput words
words:Father will live one year after another forever!
the fruit is :2
ffput fruit
ffput words
words:Father will live one year after another forever!
the fruit is :3
ffput fruit
ffput words
words:Father will live one year after another forever!
the fruit is :4
```



```
JYF [正在运行] - Oracle VM VirtualBox
管理 控制 视图 热键 设备 帮助
ffput fruit
ffput words
words:Father will live one year after another forever!
the fruit is :3
ffput fruit
ffput words
words:Father will live one year after another forever!
the fruit is :4
ffput fruit
ffput words
words:Father will live one year after another forever!
the fruit is :5
ffput fruit
ffput words
words:Father will live one year after another forever!
the fruit is :1
ffput fruit
ffput words
words:Father will live one year after another forever!
the fruit is :2
ffput fruit
ffput words
words:Father will live one year after another forever!
the fruit is :3
```

可以看出，只用当子进程的 putfruit 和 putwords 都执行后，父进程才会执行后续指令，打印出祝福和水果的相关信息。表明信号量正常工作，实现了父子进程的同步通信。

## 【实验总结】

本次实验相较 `fork` 实验，难度下降了不少。主要是实验 7 踩的坑可用避开了。实验主要要我们实现信号量的一套数据结构以及操作。由于在并行计算和操作系统课上都讲过相应的内容，所以实现起来也没有太大的疑惑，基本上就是按照信号量的定义，以及参考课件上的实现方式，直接编写代码就好了。

其中阻塞队列自己做了一些简化，没有采用真正的队列（否则需额外增加数据结构，太过繁琐），直接使用数组存贮进程号来表示队列，用 `front` 和 `tail` 两个变量表示头尾的下标，实现循环队列。其它实现就比较容易，如果在之前的实验确实实现了五状态模型，做好资源调度的话，几乎是直接增加一些代码就可以实现这次的功能了，不需要做额外的修改。

另外发现 PPT 提供的用户程序代码，可能并不能真正实现描述的功能：若某个子进程因为某种原因，被系统连续多次调度，可能会连续执行两次 `V` 操作，但是此时另一项资源是没有的，而父进程只是用连续执行两次 `P` 操作来判断自由是否充足，不能确保一定有一个水果和祝福，所以自己在子进程每次循环时，增加了一个时延（1 秒），这样既可以方便观察结果，又可以确保两项资源交替产生。