

# 进程调度实验报告

## part 1 设计调度算法

### 1.1 算法分析

本实验中的调度算法要求相当于规定了一个运行周期的概念。一个运行周期是指：所有的进程或者已经用尽了时间片，或者由于等待信号量、睡眠、等待消息等原因没有办法运行。每当一个进程用尽其时间片之后，无论其优先度高低，都没有办法在本轮运行周期之中再次得到调度，只能等到所有可运行的进程都耗尽了时间片，当前运行周期结束，进入到下一个周期，才有被调度的机会。

要评价一种调度算法，主要考察这种算法是否能做到兼顾效率和公平性，同时还应当尽力避免恶意抢占和饥饿发生。本实验所采用的调度算法，其优点在于，通过对高优先级的进程在调度时给予一定的限制，来充分保证不会发生饥饿，对公平性提供了非常大的保障。而缺点在于，这种调度算法仅靠一个 readylist 是无法完成的，还需要加入其他的辅助数据结构，并且在每一个运行周期结束之后需要额外增加一块开销用于维护，所以实际上导致了调度开销上升。

与Xinu原来的调度算法比较，这种算法的优势在于避免了饥饿现象。假如一个高优先级的进程是永远无法完成的，那么在原来的调度算法之下，由于如下所示的结构，会导致任何一个其他进程都会处于饥饿状态而无法获得CPU：

```
if (ptold->prstate == PR_CURR) { /* Process remains eligible */
    if (ptold->prprio > firstkey(readylist)) {
        return;
    }
    ...
}
```

### 1.2 实现步骤

- 需要在readylist之外再新建一个nextroundlist，当某一个进程用尽其时间片之后，被resched () 调度下来的时候，如果当前进程的状态仍然是PR\_CURR,就需要将该进程移入到nextroundlist之中，而不能加入readylist。下面是/include/queue.h之中的修改

```
#ifndef NQENT
#define NQENT    (NPROC + 8 + NSEM + NSEM)
#endif
```

下面是在/system/ready.c之中的定义

```
qid16 nextroundlist;
```

- 在Xinu之中，所有的queue都是使用隐式链表结构来实现的，也就是queuetab。而queuetab的表项是被严格确定的NQENT个，NQENT=NPROC+4+NSM+NSM，他的意义是每一个进程有一个表项，每一个信号量的头和尾有一个表项，以及readylist和sleepq的头节点和尾节点也各有一个表项。如果需要在之外额外的增加表项，那么就需要增加NQENT至少2，否则的话使用newqueue()会返回SYSERR。

- 同时，如果一个进程在没有用完时间片之前就被调度下来了，比如说在等待信号量或者睡眠等，那么需要用个变量或者结构来记录一下它在被调度下来之前已经使用了多少时间，所以我在procent的结构体定义之中又加入了一项remaintime，用来保存这个进程由于除时间片之外的原因被调度下来时还剩余多少时间可以使用，因此修改/include/process.h如下：

```
struct procent {          /* Entry in the process table      */
    uint16 prstate;       /* Process state: PR_CURR, etc.    */
    pri16 prprio;         /* Process priority                */
    char *prstkptr;       /* Saved stack pointer            */
    char *prstkbase;      /* Base of run time stack         */
    uint32 prstklen;      /* Stack length in bytes          */
    char prname[PNMLEN];  /* Process name                   */
    sid32 prsem;          /* Semaphore on which process waits */
    pid32 prparent;       /* ID of the creating process      */
    umsg32 prmsg;         /* Message sent to this process    */
    bool8 prhasmsg;       /* Nonzero iff msg is valid       */
    int16 prdesc[NDESC];  /* Device descriptors for process  */
    uint32 remaintime;
};
```

- 当readylist为空或者readylist中只有NULLPROC的时候，就需要使用while循环，将nextroundlist中所有的进程全部移入到readylist之中，将nextroundlist进行清空，并再次添加NULLPROC到readylist之中。这个时候需要添加空进程的原因是，相当于是一个哨兵，用来检查readylist是否为空，一个运行调度周期是否结束。其实，readylist在一般状况下不会为空。
- 每一次选定了新进程之后，如果新进程的remaintime不为0（默认值），证明这个进程是不因为时间片用尽而自然结束的，那么就令preempt为remaintime，同时将remaintime重置为0；否则，就按照该进程对应的优先级设置好时间片即可。

整体将resched () 修改如下：

```
void resched(void)      /* Assumes interrupts are disabled */
{
    struct procent *ptold; /* Ptr to table entry for old process */
    struct procent *ptnew; /* Ptr to table entry for new process */

    /* If rescheduling is deferred, record attempt and return */

    if (Defer.ndefers > 0) {
        Defer.attempt = TRUE;
        return;
    }

    /* Point to process table entry for the current (old) process */

    ptold = &proctab[currpid];

    if (ptold->prstate == PR_CURR) {
        ptold->prstate = PR_READY;
        insert(currpid, nextroundlist, ptold->prprio);
    }
    /* Force context switch to highest priority ready process */
    if (isempty(readylist)){
        while (!isempty(nextroundlist)){
            pid32 tempid = dequeue(nextroundlist);
```

```

        insert(tempid, readylist, proctab[tempid].prprio);
    }
    currpid = dequeue(readylist);
} else {
    currpid = dequeue(readylist);
    if(currpid==NULLPROC){
        while(!isempty(nextroundlist)){
            pid32 tempid = dequeue(nextroundlist);
            insert(tempid, readylist, proctab[tempid].prprio);
        }
        insert(NULLPROC, readylist, 0);
        currpid = dequeue(readylist);
    }
}

//reset preempt according to their priority
kprintf("%s\n", proctab[currpid].prname);
ptnew = &proctab[currpid];
ptnew->prstate = PR_CURR;
if(ptnew->remaintime!=0){
    preempt = ptnew->remaintime;
    ptnew->remaintime = 0;
} else {
    if (ptnew->prprio == PRIO1)
        preempt = QUANTUM1;
    else if (ptnew->prprio == PRIO2)
        preempt = QUANTUM2;
    else if (ptnew->prprio == PRIO3)
        preempt = QUANTUM3;
    else
        preempt = QUANTUM3;
}
//preempt = QUANTUM; /* Reset time slice for process */
ctxsw(&ptold->prstkptr, &ptnew->prstkptr);

/* old process returns here when resumed */

return;
}

```

- 为了记录下来每一个进程在由于睡眠和等待信号量而失去CPU时已经使用的时间，需要修改/system/sleep.c添加语句：

```

proctab[currpid].prstate = PR_SLEEP;
proctab[currpid].remaintime = preempt;

```

以及在/system/wait.c之中添加：

```

if (--(semptr->scount) < 0) {          /* If caller must block */
    prptr = &proctab[currpid];
    prptr->prstate = PR_WAIT;          /* Set state to waiting */
    prptr->prsem = sem;                /* Record semaphore ID */
    prptr->remaintime = preempt;
    enqueue(currpid, semptr->squeue); /* Enqueue on semaphore */
    resched();                        /* and reschedule */
}

```

### 1.3实验效果分析

第一个测试样例，为了验证是否实现了运行调度周期，我们使用的主函数main()如下所示：

```
process main(void)
{
    resume(create(test1, 1024, 30, "test1", 0));
    resume(create(test2, 1024, 30, "test2", 0));
    resume(create(test3, 1024, 30, "test3", 0));
}
```

其中三个进程所使用的测试函数如下,分别输出50个A, B, C:

```
void test1(){
    int i, j,k,count=0;
    while(1){
        for (i = 0; i < 100000;i++)
            ;
        putc(CONSOLE, 'A');
        count++;
        if(count==50)
            break;
    }
}

void test2(){
    int i, j,k,count=0;
    while(1){

        for (i = 0; i < 100000;i++)
            ;

        putc(CONSOLE, 'B');
        count++;
        if(count==50)
            break;
    }
}

void test3(){
    int i, j,k,count=0;
    while(1){

        for (i = 0; i < 100000;i++)
            ;

        putc(CONSOLE, 'C');
        count++;
        if(count==50)
            break;
    }
}
```

得到的测试结果如下图所示：

```
QEMU
Startup process
resume
Main process
resume
test1
AAAAAAAAAAAAAAAAtime slice run out
test1
AAAAAAAAAAAAAAAAtime slice run out
Startup process
Main process
resume
test2
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBtest1
AAAAAAAAAAAAAAAAtime slice run out
Main process
resume
test3
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCtest1
AAAAAMain process

All user processes have completed.
```

首先start up create+resume main()之后被抢占，main()又在create+resume test1()之后被抢占，在test1耗尽自己的时间片之后，一个调度周期结束，nextroundlist之中的所有进程全部都加入到readylist之中，顺序是test1-startup-main，test1在又一次用尽自己的时间片之后，虽然优先级最高，但仍然没有继续执行，这就证明了调度周期。此外，在最后还可以看到，test1()用完自己的周期，转换到main()resume了test3，调用了resched()之后，test1没有被调度，也能够证明调度周期地存在。

为了测量我的设备上一个周期的忙等待参数，我对main()和test进行了如下修改：

```
int32 count1=0;
void test1(){
    int32 i;
    while (1)
    {
        for (i = 0; i < 10000;i++)
            ;
        count1++;
    }
}
process main(void)
{
    pid32 pid1, pid2;
    int32 count = 0;
    int32 sum = 0;
    resume(pid1 = create(test1, 1024, 10, "test1", 0));
    while(1){
        count1 = 0;
        sleep(0);
        count++;
        sum += count1;
        if (count >= 100)
        {
            kprintf("total:%d\n", sum);
            kill(pid1);
            break;
        }
    }
}
```

```
}  
}
```

结果显示:

- 在长度为10的时间片内, 如果空循环次数是10000, 那么大概可以进行自加操作5-6次左右;
- 在长度为20的时间片内, 如果空循环次数是10000, 那么大概可以进行自加操作11-12次左右;

下面, 我们创建了5个测试函数test1-5, 分别用来声明5个进程, 并在这些进程之中分别调用睡眠、等待信号量等方式, 观察他们的调度方式, 5个函数如下:

```
void test1(){  
    int32 i;  
    while (1)  
    {  
        for (i = 0; i < 10000;i++)  
            ;  
        count1++;  
        if(count1==6)  
            sleepms(5);  
    }  
}  
void test2(){  
    int32 i;  
    while(1){  
        for (i = 0; i < 10000;i++)  
            ;  
        count2++;  
        if(count2==12)  
            sleepms(5);  
    }  
}  
void test3(){  
    int32 i;  
    while(1){  
        for (i = 0; i < 10000;i++)  
            ;  
        wait(sim);  
        count3++;  
        signal(sim);  
    }  
}  
void test4(){  
    int32 i;  
    while(1){  
        for (i = 0; i < 10000;i++)  
            ;  
        wait(sim);  
        count4++;  
        signal(sim);  
    }  
}  
void test5(){  
    int32 i;  
    while(1){  
        for (i = 0; i < 10000;i++)  
            ;  
    }  
}
```

```

        wait(sim);
        count5++;
        signal(sim);
    }
}

```

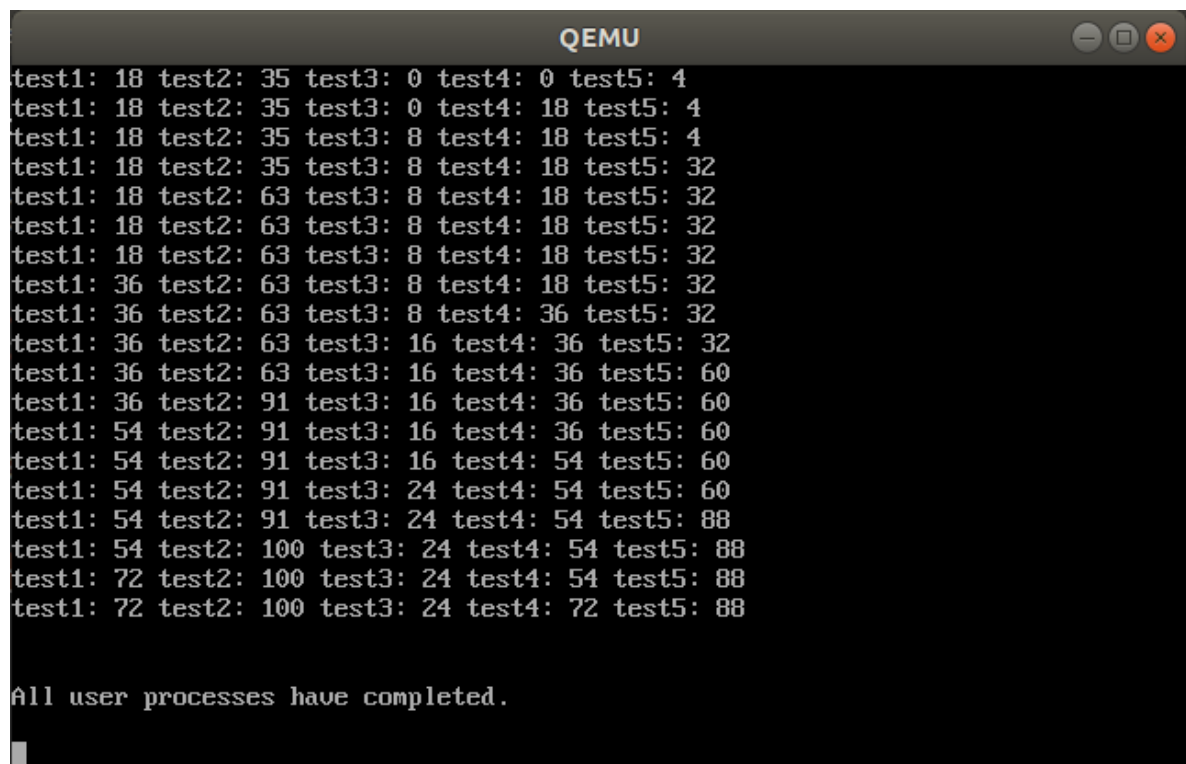
并在main之中采用延迟调度的方式，给五个进程赋予不同的优先级：

```

process main(void)
{
    pid32 pid1, pid2, pid3, pid4, pid5;
    sim = semcreate(2);
    resched_cntl(DEFER_START);
    resume(pid1 = create(test1, 1024, 20, "test1", 0));
    resume(pid2 = create(test1, 1024, 30, "test2", 0));
    resume(pid3 = create(test1, 1024, 10, "test3", 0));
    resume(pid4 = create(test1, 1024, 20, "test4", 0));
    resume(pid5 = create(test1, 1024, 30, "test5", 0));
    resched_cntl(DEFER_STOP);
    sleepms(200);
    kill(pid1);
    kill(pid2);
    kill(pid3);
    kill(pid4);
    kill(pid5);
}

```

对各个进程的记录运行时间，在每一次调用resched()进行调度的时候打印出来各个进程的运行时间，最终结果如下图所示：



```

QEMU
test1: 18 test2: 35 test3: 0 test4: 0 test5: 4
test1: 18 test2: 35 test3: 0 test4: 18 test5: 4
test1: 18 test2: 35 test3: 8 test4: 18 test5: 4
test1: 18 test2: 35 test3: 8 test4: 18 test5: 32
test1: 18 test2: 63 test3: 8 test4: 18 test5: 32
test1: 18 test2: 63 test3: 8 test4: 18 test5: 32
test1: 18 test2: 63 test3: 8 test4: 18 test5: 32
test1: 36 test2: 63 test3: 8 test4: 18 test5: 32
test1: 36 test2: 63 test3: 8 test4: 36 test5: 32
test1: 36 test2: 63 test3: 16 test4: 36 test5: 32
test1: 36 test2: 63 test3: 16 test4: 36 test5: 60
test1: 36 test2: 91 test3: 16 test4: 36 test5: 60
test1: 54 test2: 91 test3: 16 test4: 36 test5: 60
test1: 54 test2: 91 test3: 16 test4: 54 test5: 60
test1: 54 test2: 91 test3: 24 test4: 54 test5: 60
test1: 54 test2: 91 test3: 24 test4: 54 test5: 88
test1: 54 test2: 100 test3: 24 test4: 54 test5: 88
test1: 72 test2: 100 test3: 24 test4: 54 test5: 88
test1: 72 test2: 100 test3: 24 test4: 72 test5: 88

All user processes have completed.

```

## part 2 生产者、消费者、摇摆者模拟

### 2.1 实现原理及步骤

为了让摇摆者进行正确的操作，首先我们要计算出生产者和消费者的期望运行时间。跟据实验指导可知，期望运行时间由两部分组成，一部分是生产者和消费者在当前调度周期内已经执行的时间，另外一部分是生产者和消费者在当前调度周期内的剩余执行时间。定义变量pexe,prem,cexe,crem分别来表示和存储生产者和消费者已经执行时间和剩余时间。

对pexe和cexe的更新应当发生在resched()每一次将旧进程调度下来时，依照旧进程的类型进行更新，为此，需要在/system/resched.c之中加入如下操作：

```
if(ptold->prname[0]=='p')
    pexe += ptold->prprio - preempt;
else if(ptold->prname[0]=='c')
    cexe += ptold->prprio - preempt;
else if(ptold->prname[0]=='s' && ptold->prname[1]=='w'){
    if(ptold->prmsg==1)
        pexe+=ptold->prprio - preempt;
    else if(ptold->prmsg==2)
        cexe += ptold->prprio - preempt;
}
```

而对crem和prem的更新则比较麻烦，因为每一次对readylist队列的插入操作都会导致剩余执行时间增加，这需要在ready中每一次insert之后对crem和prem进行处理；并且，对于睡眠和等待的进程来说，如果他们恢复就绪状态的时候仍然处于当前的调度周期，那么他们的剩余时间片是接着睡眠和等待之前继续使用的；如果他们回复就绪状态的时候已经处于下一个调度周期，那么他们的剩余时间片被重置，优先级也会被重置。也就是说，如果一个进程的时间片是N，在等待信号/睡眠之前使用过的时间片是X，那么它恢复就绪之后的时间片是N-X（当轮恢复）或者N（下一轮恢复）。为此，在/system/ready.c之中，insert语句之前，加入下面的语句。首先判断是否有剩余时间，如果有的话就按照剩余时间来统计，否则就按照一个完整的时间片来统计：

```
if(proctab[pid].prname[0]=='p')
    prem += proctab[pid].remaintime ? proctab[pid].remaintime :
(uint32)proctab[pid].prprio;
if(proctab[pid].prname[0]=='c')
    crem += proctab[pid].remaintime ? proctab[pid].remaintime :
(uint32)proctab[pid].prprio;
```

同时，resched()调度选定一个新进程之后，这个进程的时间片应当从剩余时间之中去掉。在/system/resched.c即将调用ctxsw()之前加入下述语句：

```
if(ptnew->prname[0]=='p')
    prem -= preempt;
if(ptnew->prname[0]=='c')
    crem -= preempt;
```

同时，由于结束一个调度周期之后生产者和消费者的优先级会发生互换，所以每次一个调度周期结束之后，都需要将nextroundlist之中的内容加入到readylist之中，并全部重置prem、crem、pexe、cexe，在/system/resched.c之中修改如下：

```
if(isempty(readylist)){
    kprintf("round end\n");
    cycle_count++;
    cycle_count %= 100;
    prem = crem = pexe = cexe = 0;
    while (!isempty(nextroundlist))
    {
```



```

        pid32 tempid = dequeue(nextroundlist);
        proctab[tempid].remaintime = 0;
        proctab[tempid].prprio = 40 - proctab[tempid].prprio;
        if(proctab[tempid].prname[0]=='p')
            prem += proctab[tempid].prprio;
        else if(proctab[tempid].prname[0]=='c')
            crem += proctab[tempid].prprio;
        insert(tempid, readylist, proctab[tempid].prprio);
    }
    currpri = dequeue(readylist);

} else {
    currpri = dequeue(readylist);
    if(currpri==NULLPROC){
        kprintf("round end\n");
        cycle_count++;
        cycle_count %= 100;
        prem = crem = pexe = cexe = 0;
        while (!isempty(nextroundlist))
        {
            pid32 tempid = dequeue(nextroundlist);
            proctab[tempid].remaintime = 0;
            proctab[tempid].prprio = 40 - proctab[tempid].prprio;
            if(proctab[tempid].prname[0]=='p')
                prem += proctab[tempid].prprio;
            else if(proctab[tempid].prname[0]=='c')
                crem += proctab[tempid].prprio;
            insert(tempid, readylist, proctab[tempid].prprio);
        }
        insert(NULLPROC, readylist, 0);
        currpri = dequeue(readylist);
    }
}
}

```

其中，cycle\_count是一个resched.c之中的一个全局变量，用来记录当前是第几个调度周期，以100为上限，超过则重新开始计数。之所以需要这个全局变量，是因为我们需要了解当一个进程从PR\_SLEEP或者PR\_WAIT之中恢复就绪状态时，是仍然处于当前调度周期还是处于下一调度周期。为此，我们需要在include/process.h之中修改procent结构体如下：

```

struct procent {
    /* Entry in the process table */
    uint16 prstate; /* Process state: PR_CURR, etc. */
    pri16 prprio; /* Process priority */
    char *prstkptr; /* Saved stack pointer */
    char *prstkbase; /* Base of run time stack */
    uint32 prstklen; /* Stack length in bytes */
    char prname[PNMLEN]; /* Process name */
    sid32 prsem; /* Semaphore on which process waits */
    pid32 prparent; /* ID of the creating process */
    umsg32 prmsg; /* Message sent to this process */
    bool8 prhasmsg; /* Nonzero iff msg is valid */
    int16 prdesc[NDESC]; /* Device descriptors for process */
    uint32 remaintime; /*remaining time slice after sleep or suspend*/
    uint32 cyclestamp; /*to show which cycle the process is in*/
};

```

其中，cyclestamp可以通俗理解为周期戳，当一个进程因为wait或者睡眠 而让出CPU时，需要记录两样东西：一样是当时的剩余时间片，一样是当时所在的调度周期，也就是在/system/wait.c以及/system/sleep.c之中调度之前加入如下两行：

```
prptr->remaintime = preempt;
prptr->cyclestamp = cycle_count;
```

在procent之中有了这两个变量，我们就可以在进程从睡眠或者等待状态恢复时进行判断和相应的处理。下面是在wakeup.c和signal.c之中,在调用ready将相应的进程加入到readylist之前加入如下代码：

```
if ((semptr->scount++) < 0) { /* Release a waiting process */
    pid32 tempid;
    tempid = dequeue(semptr->squeue);
    if(proctab[tempid].cyclestamp!=cycle_count){
        proctab[tempid].remaintime = 0;
        proctab[tempid].prprio = 40 - proctab[tempid].prprio;
    }
    ready(tempid);
}
```

跟据周期戳来进行判断。如果匹配成功，代表着仍处于当前调度周期之内，时间片还没有用完，可以继续使用；如果匹配失败，则代表着已经进入到下一周期之中，需要将优先级和时间片进行重置。

为了让摇摆者可以进行正确的操作，需要resched发现即将获得CPU的新进程是摇摆者时，就向该进程发送一个消息，通过这个消息来告知摇摆者应当进行何种操作。

```
if (ptnew->prname[0] == 's' && ptnew->prname[1] == 'w')
{
    if(prem+pexe<=crem+cexe)
        send(currpid, 1);
    else
        send(currpid, 2);
}
```

## 2.2 实现效果

首先，为了测试一个10ms或者20ms的时间片之内可以读写连续内存地址多少次，我使用如下代码和main()函数进行测试：

```
void test1()
{
    int i;//test how many slots can producer write:
    while (1){
        for (i = 0; i < 10000;i++)
            ;
        writeindex++;
        if(writeindex>=N)
            writeindex %= N;
        buffer[writeindex] = 1;
        cnt++;
    }
}

process main(){
    int cnt_sum = 0;
    cnt = readindex =writeindex= 0;
```

```

int i,j=0;
for (i = 0; i < N;i++)
    buffer[i] = 0;
pid32 test1pid;
resume(test1pid=create(test2, 1024, 10, "test1", 0));
while(1){
    cnt_sum += cnt;
    cnt = 0;
    j++;
    if(j==100){
        kill(test1pid);
        break;
    }
    resched();
}
kprintf("avg: %d", cnt_sum);
}

```

测试结果显示：

- 如果采用10000次空转进行忙等待，那么一个10ms的时间片之内可以读写5.75个数组内存地址
- 如果采用50000次空转进行忙等待，那么一个10ms的时间片之内可以读写1个数组内存地址。

基于以上的测试结果，我设计生产者函数如下：

```

void producer(){
    int32 i;
    int32 count = 0;
    while (1)
    {
        for (i = 0; i < 10000;i++)
            ;
        wait(psem);
        wait(pmux);
        writeindex++;
        if(writeindex>=N)
            writeindex %= N;
        buffer[writeindex] = 1;
        signal(pmux);
        signal(csem);
        count++;
        if(count==5){
            count = 0;
            sleepms(5); //only if producer's priority > consumer
        }
    }
}

```

注意，一共需要使用四种锁，分别是信号量psem，csem和互斥锁pmux，cmux。在这里，只要有多个producer，使用互斥锁时必须的，否则可能发生这样一种情况：一个producer在写buffer的writeindex下标处时，调度到另外一个producer的位置，两个producer都写在了同一个writeindex处，不仅中间有一个没有数据的空槽，而且还造成了数据的覆盖。

按照互斥锁加锁顺序规则，先等待psem，在等待pmux，尽可能缩小临界区，从而尽可能地减少持锁时间。完成写入之后，要先释放pmux，然后释放csem，否则就可能造成死锁。

如果连续写了五个位置，按照之前在本机测试的结果，差不多已经用了10ms的基本时间片，因此令这个进程睡眠10ms，让出CPU。

consumer()的实现逻辑同样如此，不再赘述。

```
void consumer(){
    int32 data;
    int32 i;
    int32 count = 0;
    while (1)
    {
        for (i = 0; i < 10000; i++)
            ;
        wait(csem);
        wait(cmux);
        readindex++;
        if(readindex>=N)
            readindex %= N;
        data = buffer[readindex];
        signal(cmux);
        signal(psem);
        count++;
        if(count==5){
            count = 0;
            sleepms(5);
        }
    }
}
```

swayer摇摆者被调度时,首先接受消息，然后按照消息的指示，从事消费者的活动或者是生产者的活动，如下图所示：

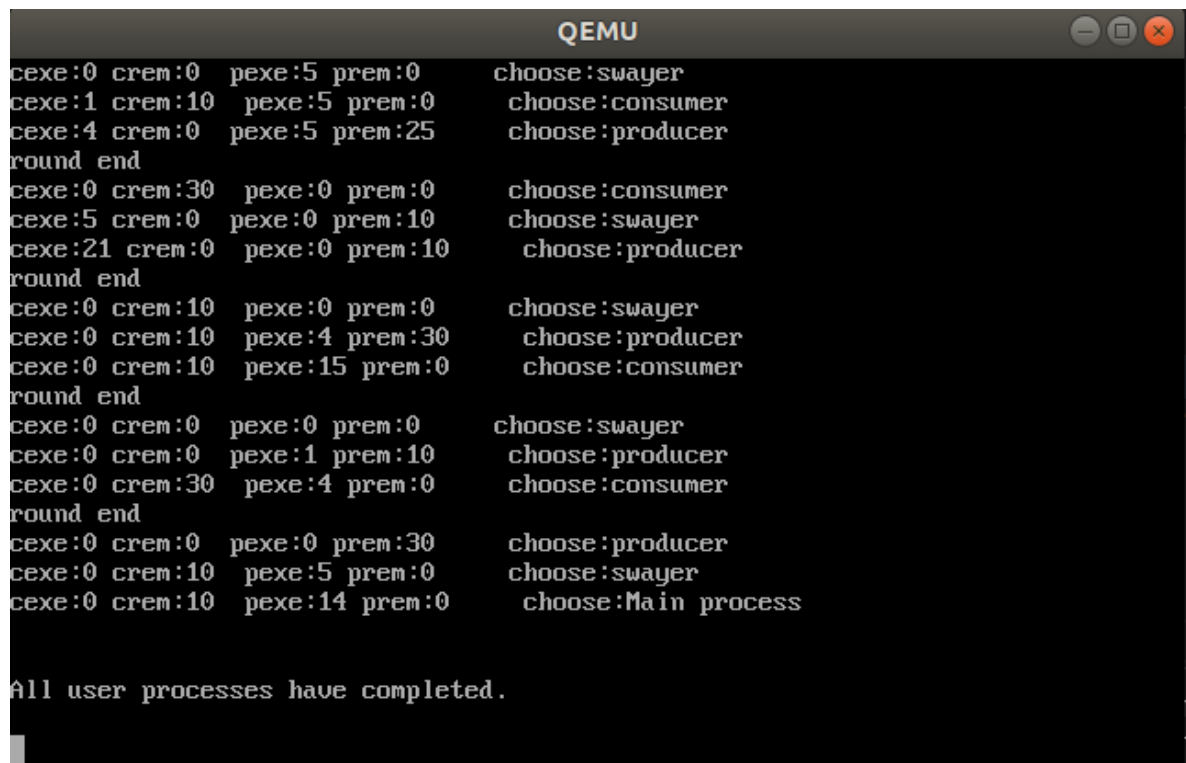
```
void swayer(){
    int32 i;
    while (1)
    {
        int32 count = 0;
        umsg32 msg = receive();
        /*umsg32 msg;
        if(pexe+prem<=crem+cexe)
            msg = 1;
        else
            msg = 2;
        */
        if (msg == 1)
        {
            while(1){
                for (i = 0; i < 10000;i++)
                    ;
                wait(psem);
                wait(pmux);
                writeindex++;
                if(writeindex>=N)
                    writeindex %= N;
                buffer[writeindex] = 2;
                signal(pmux);
                signal(csem);
            }
        }
    }
}
```

```

        count++;
        if(count==5)
            sleep(0);
    }
}
else if(msg==2){
    int32 data;
    while (1)
    {
        for (i = 0; i < 10000;i++)
            ;
        wait(csem);
        wait(cmux);
        readindex++;
        if(readindex>=N)
            readindex%=N;
        data = buffer[readindex];
        signal(cmux);
        signal(psem);
        count++;
        if(count==5)
            sleep(0);
    }
}
}
}
}

```

第一轮测试，只声明了三个进程，producer，consumer,swayer各一个，main函数如下所示：其执行效果如下图所示：



The image shows a QEMU terminal window with a black background and white text. The text displays the execution of a program with three processes: cexe, pexe, and choose. The output is organized into rounds, with each round showing the state of these processes. The final message indicates that all user processes have completed.

```

QEMU
cexe:0 crem:0 pexe:5 prem:0 choose:swayer
cexe:1 crem:10 pexe:5 prem:0 choose:consumer
cexe:4 crem:0 pexe:5 prem:25 choose:producer
round end
cexe:0 crem:30 pexe:0 prem:0 choose:consumer
cexe:5 crem:0 pexe:0 prem:10 choose:swayer
cexe:21 crem:0 pexe:0 prem:10 choose:producer
round end
cexe:0 crem:10 pexe:0 prem:0 choose:swayer
cexe:0 crem:10 pexe:4 prem:30 choose:producer
cexe:0 crem:10 pexe:15 prem:0 choose:consumer
round end
cexe:0 crem:0 pexe:0 prem:0 choose:swayer
cexe:0 crem:0 pexe:1 prem:10 choose:producer
cexe:0 crem:30 pexe:4 prem:0 choose:consumer
round end
cexe:0 crem:0 pexe:0 prem:30 choose:producer
cexe:0 crem:10 pexe:5 prem:0 choose:swayer
cexe:0 crem:10 pexe:14 prem:0 choose:Main process

All user processes have completed.

```

第二轮测试，声明了三个producer，三个consumer，一个swayer总共七个测试进程，main函数如下所示：

```

process main(void)
{

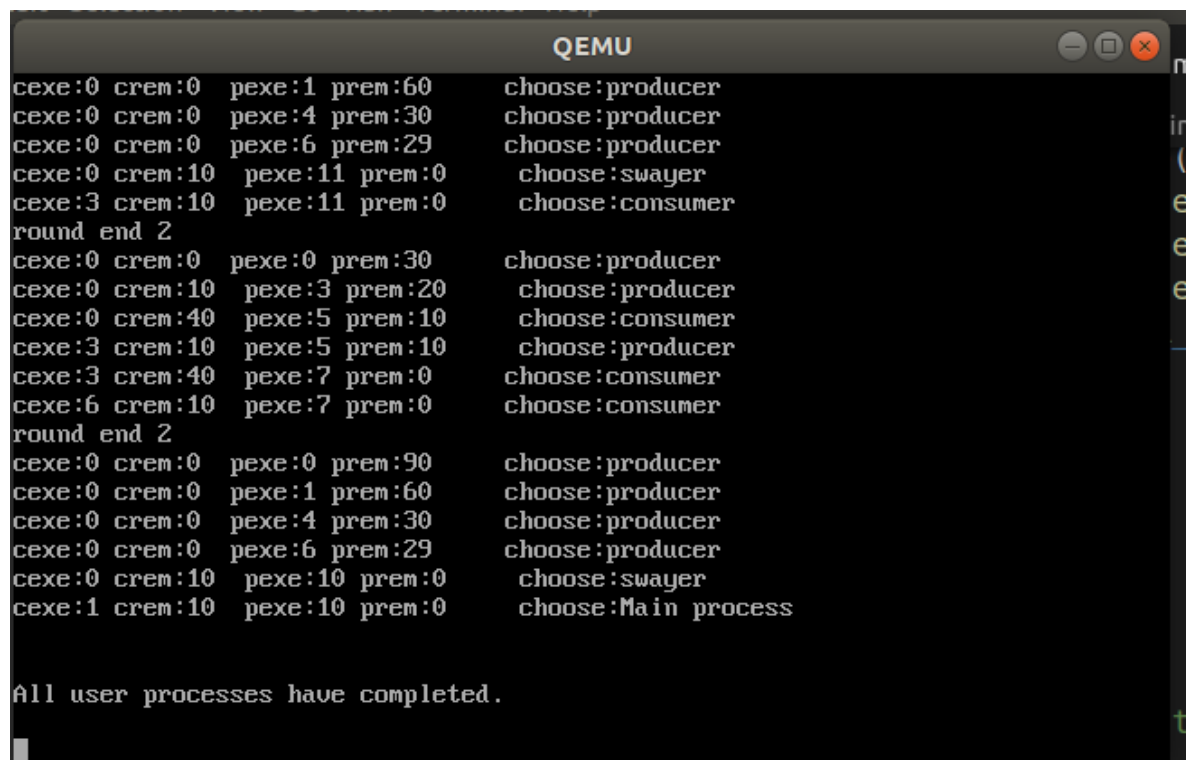
```

```

pid32 ppid1,ppid2,ppid3, cpid1,cpid2,cpid3,spid;
pmux = semcreate(1);
cmux = semcreate(1);
psem = semcreate(N);
csem = semcreate(0);
readindex = -1;
writeindex = -1;
resched_cntl(DEFER_START);
resume(ppid1=create(producer, 1024, 30, "producer", 0));
resume(ppid2=create(producer, 1024, 30, "producer", 0));
resume(ppid3=create(producer, 1024, 30, "producer", 0));
resume(spид=create(swayer, 1024, 20, "swayer", 0));
resume(cpid1=create(consumer, 1024, 10, "consumer", 0));
resume(cpid2=create(consumer, 1024, 10, "consumer", 0));
resume(cpid3=create(consumer, 1024, 10, "consumer", 0));
resched_cntl(DEFER_STOP);
sleepms(800);
kill(ppid1);
kill(ppid2);
kill(ppid3);
kill(spид);
kill(cpid1);
kill(cpid2);
kill(cpid3);
}

```

其执行效果和各个进程的CPU用时如下图所示：



```

QEMU
cexe:0 crem:0 pexe:1 prem:60 choose:producer
cexe:0 crem:0 pexe:4 prem:30 choose:producer
cexe:0 crem:0 pexe:6 prem:29 choose:producer
cexe:0 crem:10 pexe:11 prem:0 choose:swayer
cexe:3 crem:10 pexe:11 prem:0 choose:consumer
round end 2
cexe:0 crem:0 pexe:0 prem:30 choose:producer
cexe:0 crem:10 pexe:3 prem:20 choose:producer
cexe:0 crem:40 pexe:5 prem:10 choose:consumer
cexe:3 crem:10 pexe:5 prem:10 choose:producer
cexe:3 crem:40 pexe:7 prem:0 choose:consumer
cexe:6 crem:10 pexe:7 prem:0 choose:consumer
round end 2
cexe:0 crem:0 pexe:0 prem:90 choose:producer
cexe:0 crem:0 pexe:1 prem:60 choose:producer
cexe:0 crem:0 pexe:4 prem:30 choose:producer
cexe:0 crem:0 pexe:6 prem:29 choose:producer
cexe:0 crem:10 pexe:10 prem:0 choose:swayer
cexe:1 crem:10 pexe:10 prem:0 choose:Main process

All user processes have completed.

```

```
QEMU
cexe:7 crem:10 pexe:9 prem:0 choose:consumer
round end
cexe:0 crem:0 pexe:0 prem:30 choose:swayer
cexe:2 crem:10 pexe:0 prem:30 choose:producer
cexe:2 crem:40 pexe:2 prem:20 choose:consumer
cexe:4 crem:10 pexe:2 prem:20 choose:producer
cexe:4 crem:40 pexe:4 prem:10 choose:consumer
cexe:5 crem:10 pexe:4 prem:10 choose:producer
cexe:5 crem:38 pexe:7 prem:0 choose:consumer
cexe:8 crem:10 pexe:7 prem:0 choose:consumer
round end
cexe:0 crem:0 pexe:0 prem:90 choose:producer
cexe:0 crem:0 pexe:1 prem:60 choose:producer
cexe:0 crem:0 pexe:2 prem:30 choose:producer
cexe:0 crem:30 pexe:4 prem:0 choose:consumer
cexe:0 crem:0 pexe:4 prem:29 choose:producer
cexe:0 crem:0 pexe:5 prem:29 choose:producer
cexe:0 crem:10 pexe:9 prem:0 choose:swayer
cexe:2 crem:20 pexe:9 prem:0 choose:Main process

All user processes have completed.
```

```
QEMU
cexe:0 crem:50 pexe:5 prem:0 choose:consumer
cexe:0 crem:20 pexe:5 prem:0 choose:consumer
cexe:0 crem:10 pexe:5 prem:0 choose:consumer
round end
cexe:0 crem:0 pexe:0 prem:30 choose:swayer
cexe:0 crem:10 pexe:2 prem:30 choose:producer
cexe:0 crem:40 pexe:5 prem:20 choose:consumer
cexe:1 crem:10 pexe:5 prem:20 choose:producer
cexe:1 crem:40 pexe:6 prem:10 choose:consumer
cexe:2 crem:10 pexe:6 prem:10 choose:producer
cexe:2 crem:39 pexe:8 prem:0 choose:consumer
cexe:3 crem:10 pexe:8 prem:0 choose:consumer
round end
cexe:0 crem:0 pexe:0 prem:90 choose:producer
cexe:0 crem:10 pexe:3 prem:60 choose:producer
cexe:0 crem:40 pexe:4 prem:30 choose:producer
cexe:0 crem:40 pexe:5 prem:0 choose:consumer
cexe:0 crem:10 pexe:5 prem:0 choose:swayer
cexe:0 crem:20 pexe:7 prem:0 choose:Main process

All user processes have completed.
```