lab4实验报告

Semaphore

新增数据结构Semaphore（Device同理）

```
struct Semaphore {
    int state; // 0: not in use; 1: in use;
    int value;  // >=0: no process blocked; -i: i process blocked;
    struct ListHead pcb; // link to all pcb ListHead blocked on this semaphore
};
```

根据双向链表结构，将current线程加到信号量i的阻塞列表，和从信号量i上阻塞的进程列表取出一个进程，可封装成如下代码实现：

```
#define DEV 0
#define SEM 1
void process_into_block(int choice, int current, int i){
    if(choice == SEM){
        pcb[current].blocked.next = sem[i].pcb.next;
        pcb[current].blocked.prev = &(sem[i].pcb);
        sem[i].pcb.next = &(pcb[current].blocked);
        (pcb[current].blocked.next)->prev = &(pcb[current].blocked);
    }
    else{
        pcb[current].blocked.next = dev[i].pcb.next;
        pcb[current].blocked.prev = &(dev[i].pcb);
        dev[i].pcb.next = &(pcb[current].blocked);
        (pcb[current].blocked.next)->prev = &(pcb[current].blocked);
    }
}
ProcessTable* process_out_block(int choice, int i){
    ProcessTable *pt = NULL;
    if(choice == SEM){
        pt = (ProcessTable*)((uint32_t)(sem[i].pcb.prev) -
                (uint32_t)&(((ProcessTable*)0)->blocked));
        sem[i].pcb.prev = (sem[i].pcb.prev)->prev;
        (sem[i].pcb.prev)->next = &(sem[i].pcb);
    }
    else{
        pt = (ProcessTable*)((uint32_t)(dev[i].pcb.prev) -
                (uint32_t)&(((ProcessTable*)0)->blocked));
        dev[i].pcb.prev = (dev[i].pcb.prev)->prev;
        (dev[i].pcb.prev)->next = &(dev[i].pcb);
    }
    return pt;
}
```

实现格式化输入函数

keyboardHandle

1. 将读取到的 keyCode 放入到 keyBuffer 中
2. 唤醒阻塞在 dev[STD_IN] 上的一个进程

```
void keyboardHandle(struct StackFrame *sf) {
    ProcessTable *pt = NULL;
    uint32_t keyCode = getKeyCode();
    if (keyCode == 0) // illegal keyCode
        return;
    // putChar(getChar(keyCode));
    keyBuffer[bufferTail] = keyCode;
    bufferTail=(bufferTail+1)%MAX_KEYBUFFER_SIZE;

    if (dev[STD_IN].value < 0) { // with process blocked
        // TODO: deal with blocked situation
        dev[STD_IN].value ++;
        pt = process_out_block(DEV, STD_IN);
        pt->state = STATE_RUNNABLE;
        pt->sleepTime = 0;
    }
    return;
}
```

syscallReadStdIn

1. 如果 dev[STD_IN].value == 0， 将当前进程阻塞在 dev[STD_IN] 上
2. 进程被唤醒，读 keyBuffer 中的所有数据

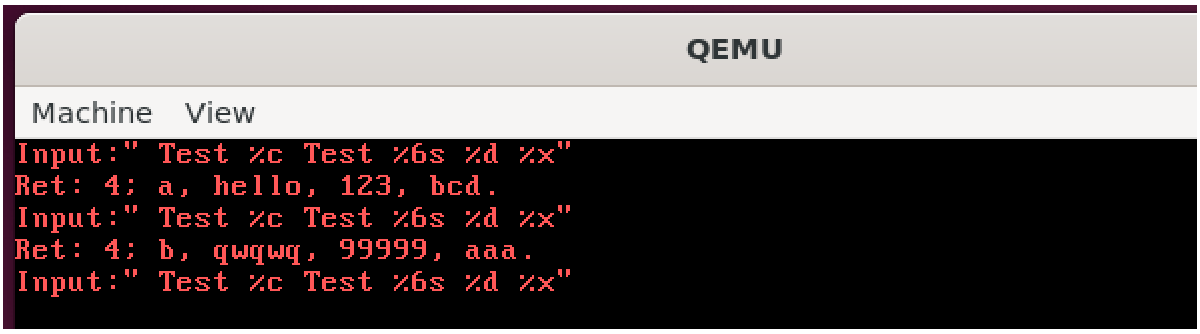
最多只能有一个进程被阻塞在 dev[STD_IN] 上，多个进程想读，那么后来的进程会返回 -1 ， 其他情况 scanf 的返回值应该是实际读取的字节数

```
void syscallReadStdIn(struct StackFrame *sf) {
    // TODO: complete `stdin`
    if(dev[STD_IN].value == 0){
        dev[STD_IN].value --;
        process_into_block(DEV, current, STD_IN);
        pcb[current].state = STATE_BLOCKED;
        pcb[current].sleepTime = -1;
        asm volatile("int $0x20");
        int sel = sf->ds;
        char *str = (char *)sf->edx;
        int size = sf->ebx;
        int i = 0;
        char character = 0;
        asm volatile("movw %0, %%es"::"m"(sel));
        while(i < size-1) {
            if(bufferHead == bufferTail) break;
            character=getChar(keyBuffer[bufferHead]);
            bufferHead=(bufferHead+1)%MAX_KEYBUFFER_SIZE;
            putChar(character);
```

```
            if(character != 0) {
                asm volatile("movb %0, %%es:(%1)"::"r"(character),"r"(str+i));
                i++;
            }
        }
        asm volatile("movb $0, %%es:(%0)"::"r"(str+i));
        pcb[current].regs.eax = i;
    }
    else if (dev[STD_IN].value < 0) {
        pcb[current].regs.eax = -1;
    }
}
```

Test scanf

```
Test a Test hello 123 0xbcd
Test b Test qwqwq 99999 0xaaa
```



实现信号量

SemInit

初始化信号量，其中参数 value 用于指定信号量的初始值，初始化成功则返回 0 ，指针 sem 指向初始化成功的信号量，否则返回 -1

```
int sem_init(sem_t *sem, uint32_t value) {
    *sem = syscall(SYS_SEM, SEM_INIT,  value, 0, 0, 0);
    if (*sem != -1)
        return 0;
    else
        return -1;
}
```

因为sem[] 是用数组形式存放，所以可以直接用下标 i 来作为返回值，表示初始化成功的信号量*sem

```
void syscallSemInit(struct StackFrame *sf) {
    // TODO: complete `SemInit`
    int i = 0;
    for(i = 0; i < MAX_SEM_NUM; i++){
        if(sem[i].state == 0){
            sem[i].state = 1;
            sem[i].value = sf->edx;
```

```
            sem[i].pcb.next = &(sem[i].pcb);
            sem[i].pcb.prev = &(sem[i].pcb);
            pcb[current].regs.eax = i;
            return;
        }
    }
    pcb[current].regs.eax = -1;
    return;
}
```

SemWait

对应信号量的 P 操作，使得 sem 指向的信号量的 value 减一，若 value 取值小于 0 ，则阻塞自身，否则进程继续执行，若操作成功则返回 0 ，否则返回 -1

```
void syscallSemWait(struct StackFrame *sf) {
    // TODO: complete `SemWait` and note that you need to consider some special
situations
    int i = (int)sf->edx;
    if (sem[i].state == 1) {
        pcb[current].regs.eax = 0;
        sem[i].value --;
        if (sem[i].value < 0) {
            pcb[current].state = STATE_BLOCKED;
            process_into_block(SEM, current, i);
            asm volatile("int $0x20");
        }
        return;
    }
    else {
        pcb[current].regs.eax = -1;
        return;
    }
}
```

SemPost

对应信号量的 V 操作，其使得 sem 指向的信号量的 value 增一，若 value 取值不大于 0 ，则释放一个阻塞在该信号量上进程（即将该进程设置为就绪态），若操作成功则返回 0 ，否则返回 -1

```
void syscallSemPost(struct StackFrame *sf) {
    int i = (int)sf->edx;
    ProcessTable *pt = NULL;
    if (i < 0 || i >= MAX_SEM_NUM) {
        pcb[current].regs.eax = -1;
        return;
    }
    // TODO: complete other situations
    if (sem[i].state == 1) {
        pcb[current].regs.eax = 0;
        sem[i].value ++;
```

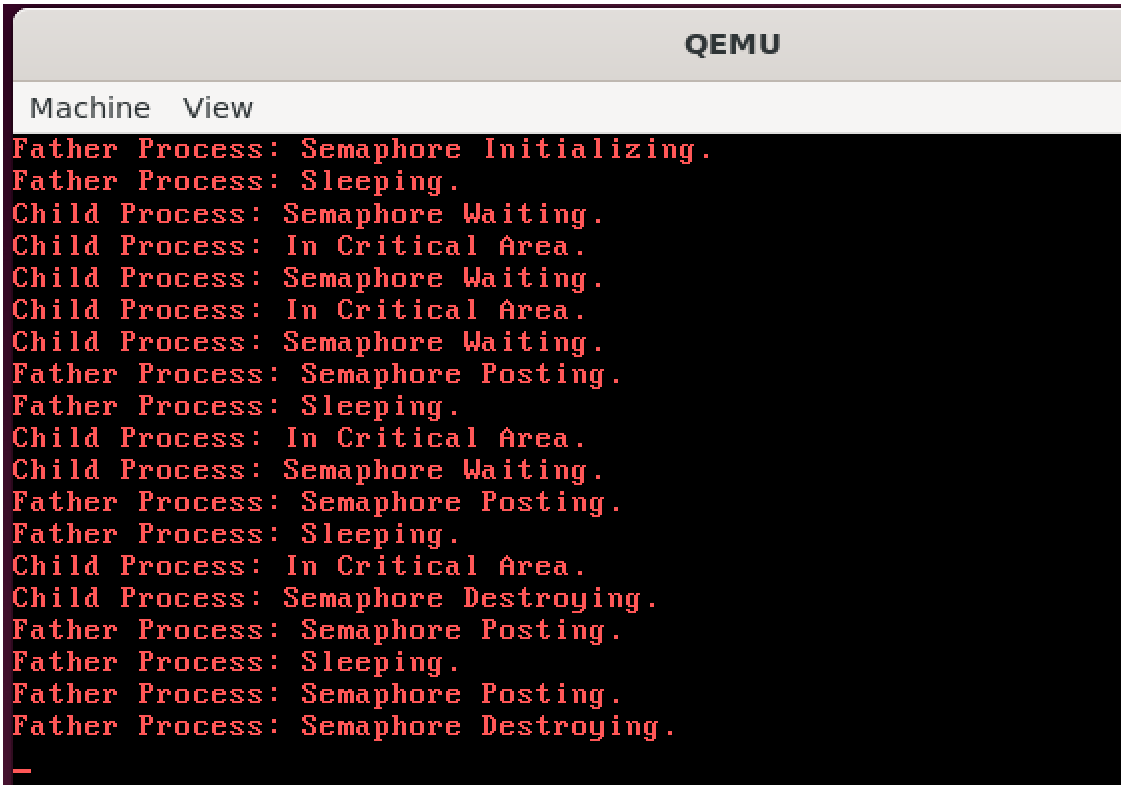
```
        if (sem[i].value <= 0) {
            pt = process_out_block(SEM, i);
            pt->state = STATE_RUNNABLE;
            pt->sleepTime = 0;
        }
        return;
    }
    else {
        pcb[current].regs.eax = -1;
        return;
    }
}
```

SemDestroy

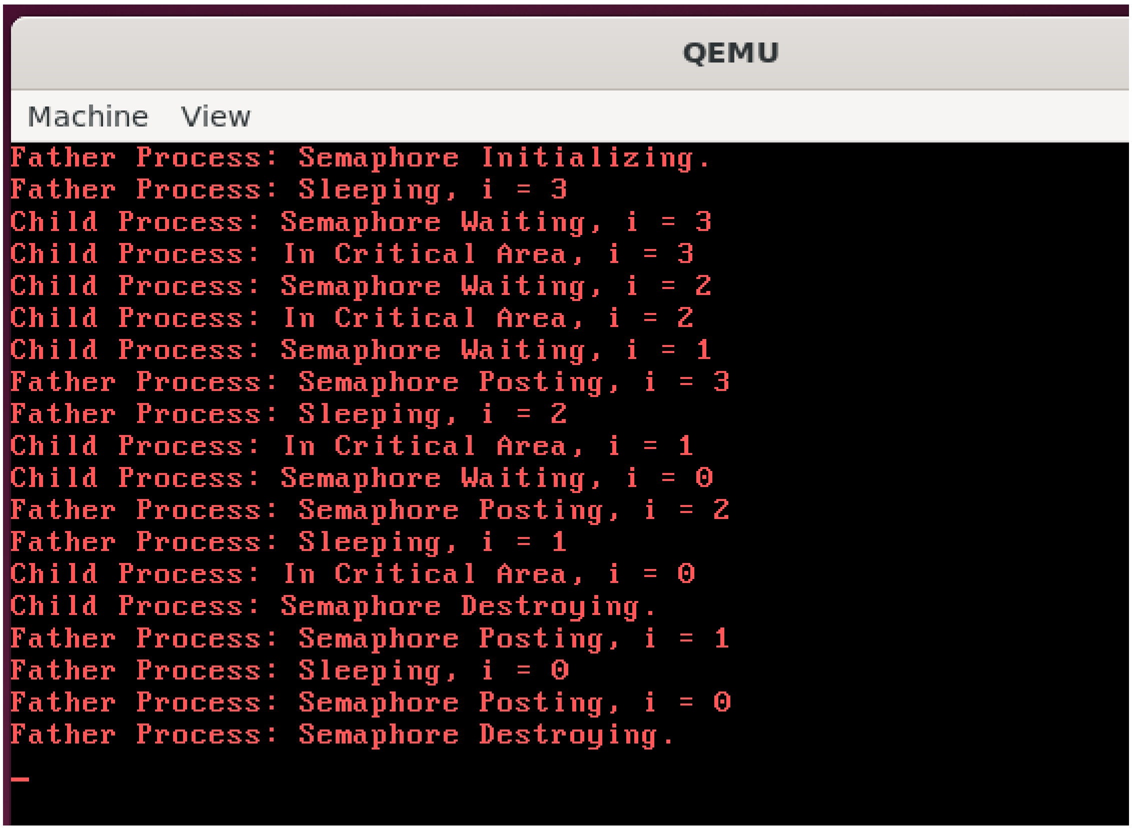
用于销毁 sem 指向的信号量，销毁成功则返回0，否则返回-1，若尚有进程阻塞在该信号量上，可带来未知错误

```
void syscallSemDestroy(struct StackFrame *sf) {
    // TODO: complete `SemDestroy`
    int i = (int)sf->edx;
    if (sem[i].state == 1) {
        pcb[current].regs.eax = 0;
        sem[i].state = 0;
        asm volatile("int $0x20");
    }
    else {
        pcb[current].regs.eax = -1;
    }
    return;
}
```

Test Semaphore



添加了 i 的版本



child先申请(P)再进入临界区，father先sleep再释放资源(V)

资源总数为2，开始child i=3 2 直接申请2个资源，之后child i = 1 和 i = 0 分别获得 father i = 3 和 i = 2 释放的资源，之后father i = 1 和 father i = 0 归还2个资源

解决进程同步问题

生产者-消费者问题

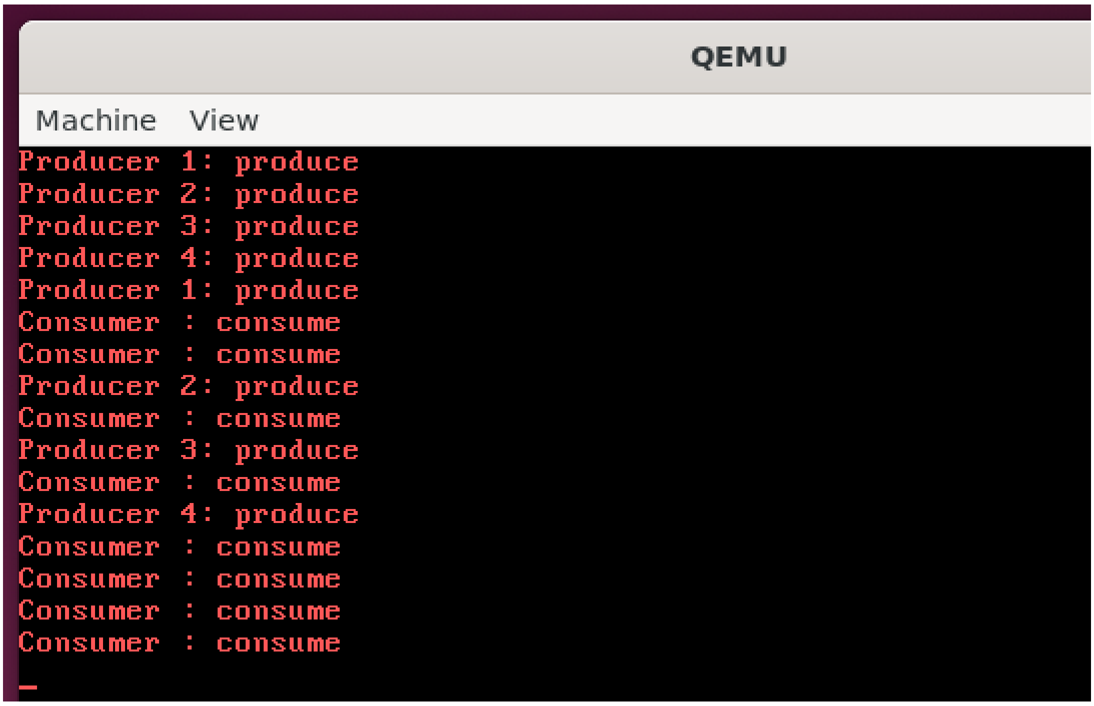
四个生产者在生产数据后放在一个缓冲区里；单个消费者从缓冲区取出数据处理；任何时刻只能有一个生产者或消费者可访问缓冲区

任何时刻只能有一个线程操作缓冲区（互斥访问） 缓冲区空时，消费者必须等待生产者（条件同步） 缓冲区满时，生产者必须等待消费者（条件同步）

信号量： mutex，fullBuffers，emptyBuffers

```
void producer_consumer(){
    int i=0, ret=0;
    sem_t empty, full, mutex;
    sem_init(&empty, 5);
    sem_init(&full, 0);
    sem_init(&mutex, 1);
    for(i=0;i<4;i++){
        if(ret==0)ret = fork();
        else if(ret>0) break;
    }
    // 1 : consumer ; 2 3 4 5 : producer
    int id = getpid();
    if(id > 1){ //producer
        for(int i=0;i<2;i++){
            sem_wait(&empty); //emptyBuffers->P();
            sem_wait(&mutex); //mutex->P();
            printf("Producer %d: produce\n", id-1);
            sleep(128);
            sem_post(&mutex); //mutex->V();
            sem_post(&full); //fullBuffers->V();
        }
    }
    else if(id == 1){ //consumer
        for(int i=0;i<8;i++){
            sem_wait(&full); //fullBuffers->P();
            sem_wait(&mutex); //mutex->P();
            printf("Consumer : consume\n");
            sleep(128);
            sem_post(&mutex); //mutex->V();
            sem_post(&empty); //emptyBuffers->V();
        }
    }
    exit();
}
```

运行效果：



哲学家就餐问题

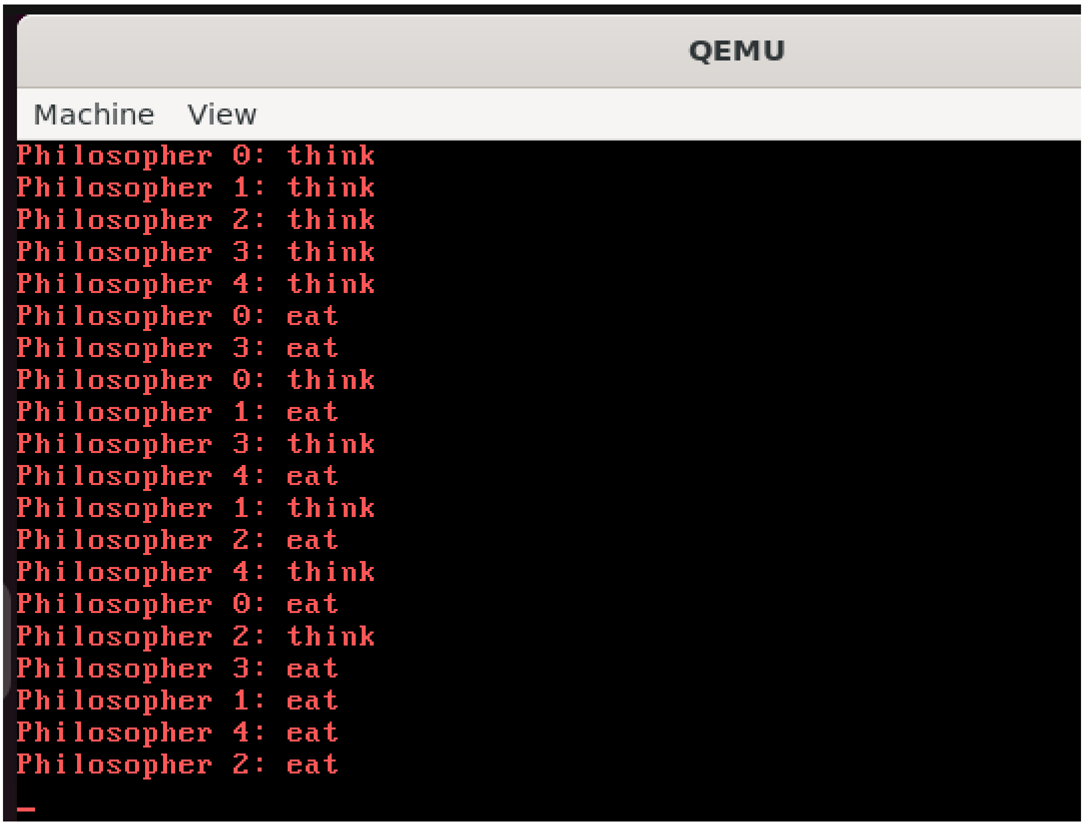
5个哲学家围绕一张圆桌而坐，桌子上放着5支叉子 每两个哲学家之间放一支叉子。哲学家的动作包括思考和进餐 ，进餐时需要同时拿到左右两边的叉子，思考时将两支叉子返回原处

用实验讲义中的方案3，偶数哲学家先拿左叉子，奇数哲学家先拿右叉子，没有死锁，可以实现多人同时就餐

```
void philosopher(){
    int i=0, ret=0;
    sem_t fk[5];
    for(int i=0;i<5;i++)sem_init(&fk[i], 1);
    for(i=0;i<4;i++){
        if(ret==0)ret = fork();
        else if(ret>0) break;
    }
    // philosopher 0-4, pid 1-5
    int id=getpid(); id-=1;
    for(int i=0;i<2;i++){
        printf("Philosopher %d: think\n", id);
        sleep(128);
        if(id%2==0){
            sem_wait(&fk[id]);
            sem_wait(&fk[(id+1)%5]);
        }
        else{
            sem_wait(&fk[(id+1)%5]);
            sem_wait(&fk[id]);
        }
        printf("Philosopher %d: eat\n", id);
        sleep(128);
        sem_post(&fk[id]);
        sem_post(&fk[(id+1)%5]);
    }
    if(id!=0)exit();
```

```
    for(int i=0;i<5;i++)sem_destroy(&fk[i]);
    exit();
}
```

运行效果：



读者-写者问题

同一时刻，允许有多个读者同时读，没有写者时读者才能读，没有读者时写者才能写，没有其他写者时写者才能写。

信号量WriteMutex，控制读写操作的互斥，初始化为1；读者计数Rcount，正在进行读操作的读者数目，初始化为0；信号量CountMutex，控制对读者计数的互斥修改，初始化为1，只允许一个线程修改Rcount计数

```
void reader_writer(){
    // 3 reader 3 writer
    // how to share the Rcount between processes?
    sem_t WriteMutex, CountMutex;
    // int Rcount = 0;
    sem_init(&WriteMutex, 1);
    sem_init(&CountMutex, 1);
    int i=0, ret=0;
    for(i=0;i<5;i++){
        if(ret==0)ret = fork();
        else if(ret>0) break;
    }
    // reader 1 2 3 writer 4 5 6
    int id = getpid();
    if(id>3){ //writer
        for(int i=0;i<2;i++){
            sem_wait(&WriteMutex);
```

```
                printf("Writer %d: write\n", id-3);
                sleep(128);
                sem_post(&WriteMutex);
            }
        }
        else { //reader
            for(int i=0;i<2;i++){
                sem_wait(&CountMutex);
                if(Rcount == 0)sem_wait(&WriteMutex);
                ++Rcount;
                sem_post(&CountMutex);
                printf("Reader %d: read, total %d reader\n", id, Rcount);
                sleep(128);
                sem_wait(&CountMutex);
                --Rcount;
                if(Rcount == 0)sem_post(&WriteMutex);
                sem_post(&CountMutex);
            }
        }
        if(id!=1)exit();
        sem_destroy(&WriteMutex);
        sem_destroy(&CountMutex);
        exit();
    }
```

但是不知道怎么在进程间共享Rcount数据

附录：一些额外的修改

添加getpid函数

> ../lib/lib.h

```
#define SYS_PID 7
pid_t getpid();
```

> ../lib/syscall.c

```
pid_t getpid() {
    return syscall(SYS_PID, 0, 0, 0, 0, 0);
}
```

> ../kernel/kernel/irqHandle.c

```
#define SYS_PID 7
case SYS_PID:
    syscallPid(sf);
    break; // for SYS_PID
void syscallPid(struct StackFrame *sf) {
    pcb[current].regs.eax = current;
    return;
}
```

扩充MAX_SEM_NUM 4->9

> ../kernel/include/x86/memory.h

```
#define NR_SEGMENTS      20           // GDT size
```

运行相关

> ../bootloader/boot.c

```
    // int phoff = 0x34;
```

> ../bootloader/Makefile

```
    objcopy  -S -j .text  -O binary bootloader.elf bootloader.bin
    chmod +x ../utils/genBoot.pl
```

> ../kernel/Makefile

```
    chmod +x ../utils/genKernel.pl
```

> ../Makefile

```
run:
    make clean
    make os.img
    make play
```

输出调试

> ../kernel/device/serial.h

```
void putStr(char *);
void putNum(int);
```

> ../kernel/kernel/serial.c

```
void putStr(char *ch){
    while(ch && (*ch) && (*ch)!='\0'){
        putChar(*ch);
        ch++;
    }
}
void putNum(int num){
    if (num == 0){ putChar('0'); return;}
    if (num < 0){ putChar('-'); num = -num;}
    while(num){
        char ch = (num % 10) + '0';
        putChar(ch);
        num /= 10;
    }
}
```