lab4实验报告

刘国涛 181860055 **计算机科学与技术系**

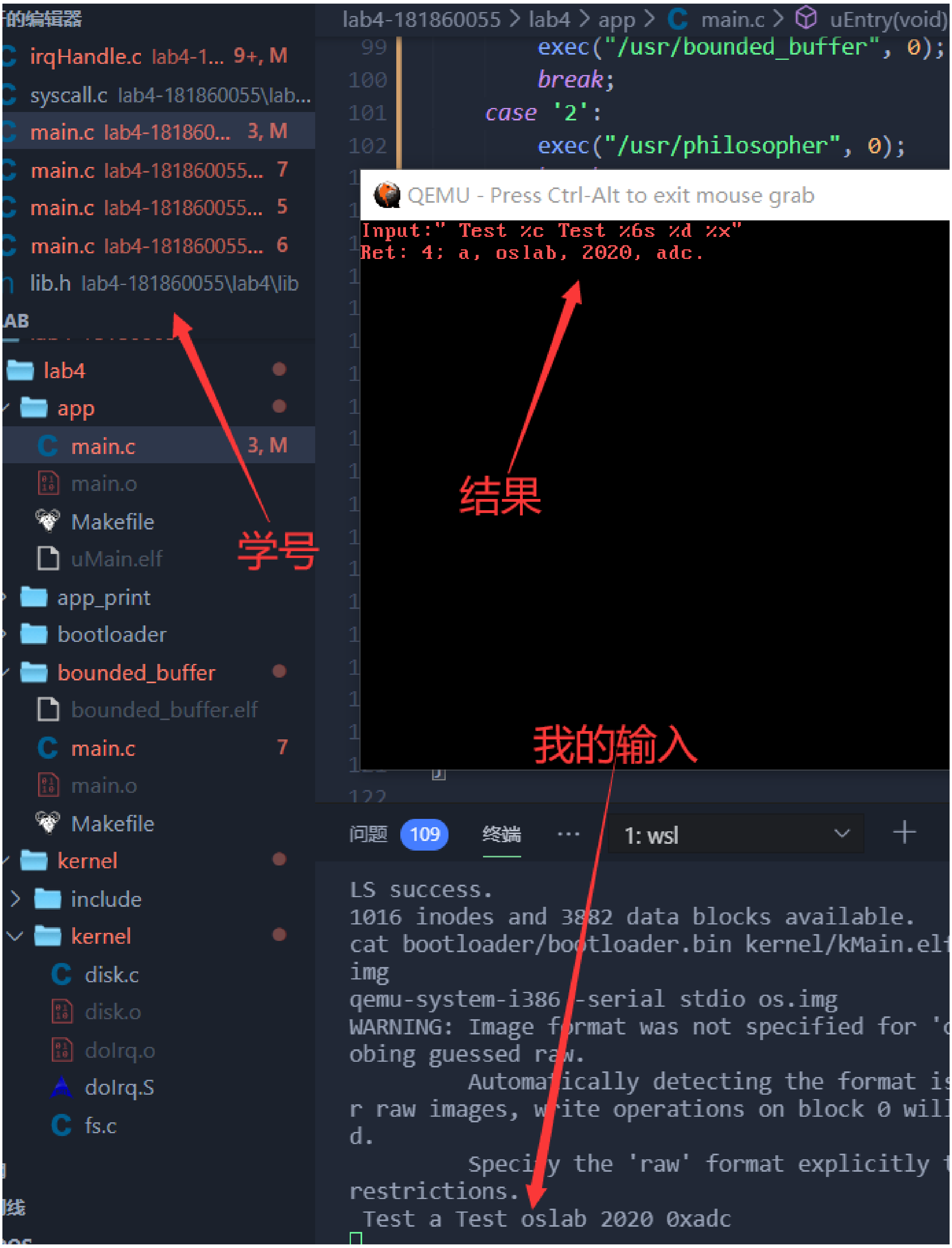
181860055@smail.nju.edu.cn

实验描述

实验进度:完成了所有必做的实验内容和随机数的选做内容

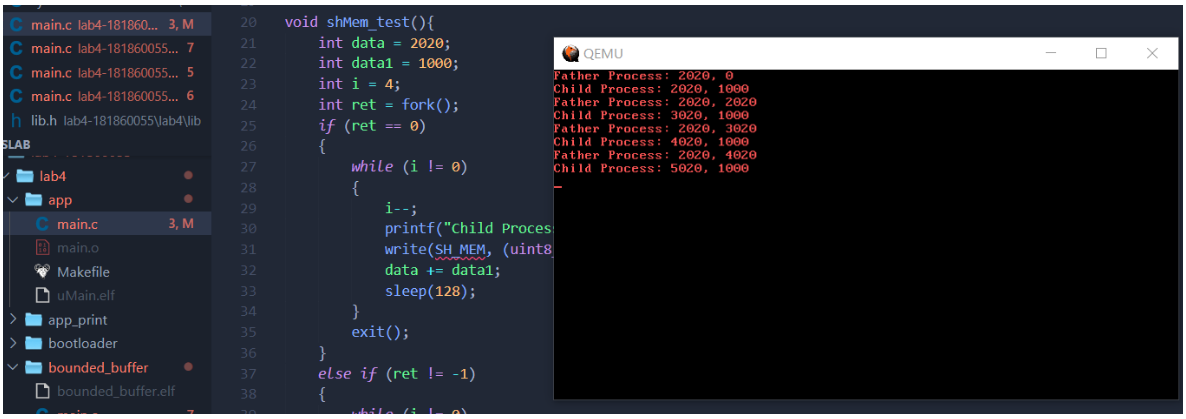
实验结果:

1、实现格式化输入



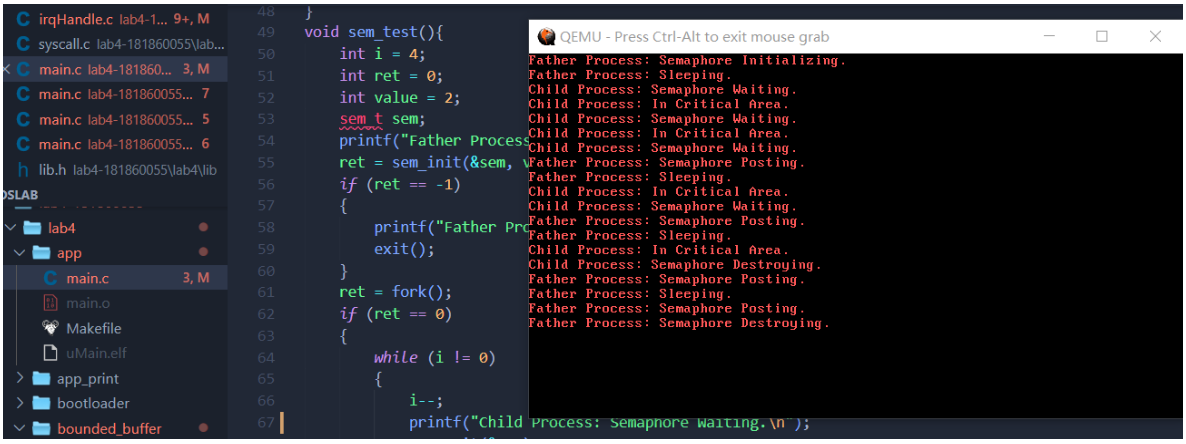
exec("/usr/bounded_buffer", 0);
break;
case '2':
exec("/usr/philosopher", 0);
QEMU - Press Ctrl-Alt to exit mouse grab
Input:" Test %c Test %6s %d %x"
Ret: 4; a, oslab, 2020, adc.
结果
学号
我的输入
LS success.
1016 inodes and 3882 data blocks available.
qemu-system-i386 -serial stdio os.img
Test a Test oslab 2020 0xadc

2、实现进程通信



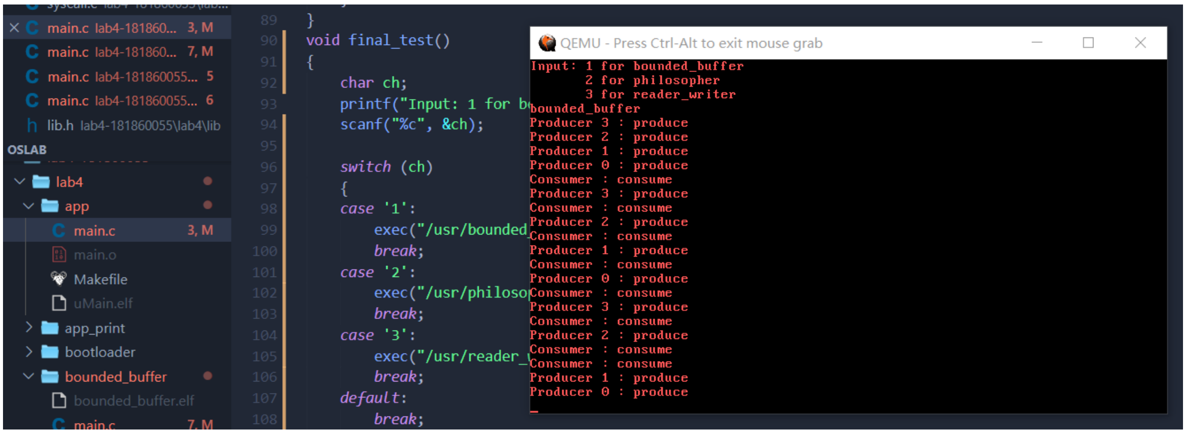
void shMem_test(){
int data = 2020;
int data1 = 1000;
int i = 4;
int ret = fork();
if (ret == 0)
QEMU
Father Process: 2020, 0
Child Process: 2020, 1000
Father Process: 2020, 2020
Child Process: 3020, 1000
Father Process: 2020, 3020
Child Process: 4020, 1000
Father Process: 2020, 4020
Child Process: 5020, 1000

3、实现信号量

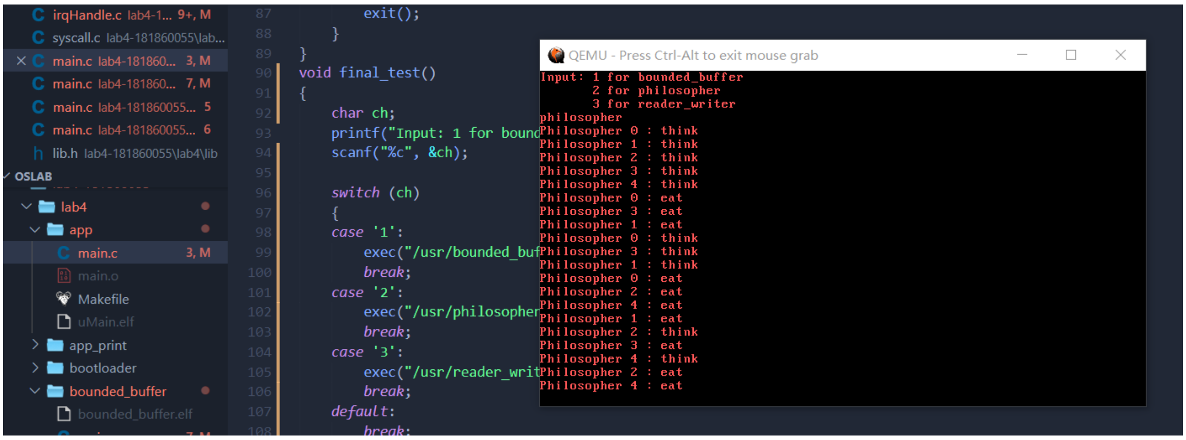


4、解决进程同步问题

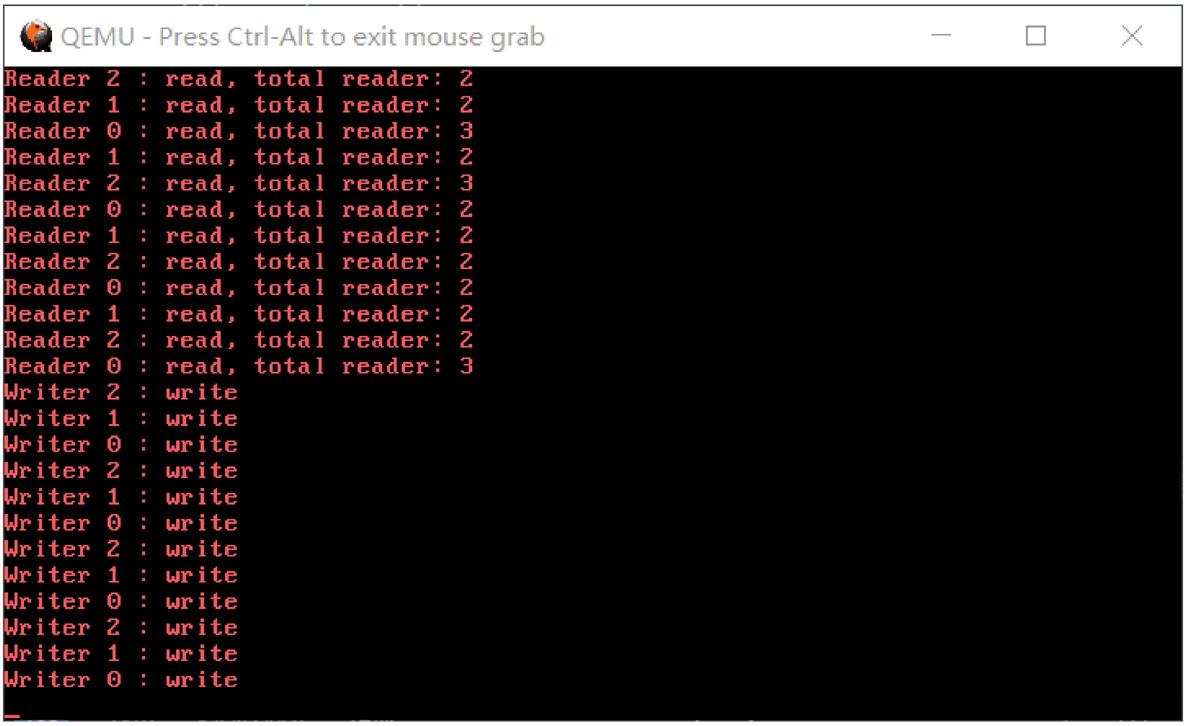
生产者消费者问题



哲学家就餐问题



读者写者问题



实验过程

1、实现格式化输入

在`keyboardHandle`中实现获取键盘输入并装入keyBuffer中：

```
    uint32_t keyCode = getKeyCode();
    if (keyCode == 0)
        return;
    keyBuffer[bufferTail] = keyCode;
    bufferTail = (bufferTail + 1) % MAX_KEYBUFFER_SIZE;
```

然后唤醒阻塞在dev[STD_IN]上的进程：

```
    if(dev[STD_IN].value < 0){
        ProcessTable* pt = (ProcessTable*)((uint32_t)(dev[STD_IN].pcb.prev)-
                            (uint32_t)&(((ProcessTable*)0)->blocked));
        dev[STD_IN].pcb.prev = (dev[STD_IN].pcb.prev)->prev;
        (dev[STD_IN].pcb.prev)->next = &(dev[STD_IN].pcb);
        pt->state = STATE_RUNNABLE;
        dev[STD_IN].value = 0;
    }
```

接着在 `syscallReadStdin` 中实现进程阻塞：

> 如果dev[STD_IN]中已有阻塞进程，则返回-1，否则阻塞当前进程

```
    if(dev[STD_IN].value == 0){
        pcb[current].blocked.next = dev[STD_IN].pcb.next;
        pcb[current].blocked.prev = &(dev[STD_IN].pcb);
        dev[STD_IN].pcb.next = &(pcb[current].blocked);
        (pcb[current].blocked.next)->prev = &(pcb[current].blocked);
        pcb[current].state = STATE_BLOCKED;
        pcb[current].sleepTime = 0xFFFFFFF;
        dev[STD_IN].value = -1;

    }
    else if(dev[STD_IN].value < 0){
        tf->eax = -1;
        return;
    }
```

成功阻塞后中断 `asm volatile("int $0x20");`，切换进程同时监听键盘输入

当再次进程再次被唤醒后，将keyBuffer中数据转换后输出到用户程序中

```
    int sel = tf->ds;
    char *str = (char*)tf->edx;
    int i = 0;
```

```
    char character = 0;
    int count = 0;
    int size = (bufferTail + MAX_KEYBUFFER_SIZE -
bufferHead) % MAX_KEYBUFFER_SIZE;
    asm volatile("movw %0, %%es"::"m"(sel));
    for(;i<size;++i){
        character = getChar(keyBuffer[bufferHead+i]);

        if(character>0){
            putChar(character);
            asm volatile("movb %0, %%es:(%1)"::"r"
(character),"r"(str+count));
            ++count;
        }

    }
    character = 0;
    asm volatile("movb %0, %%es:(%1)"::"r"(character),"r"
(str+count));
    bufferTail = bufferHead;
    tf->eax = count;
```

2、实现进程通信

实现读共享内存的核心代码：

```
    asm volatile("movw %0, %%es"::"m"(sel));
    for(int i=0;i<size;++i){
        character = *(shMem+index+i);
        asm volatile("movb %0, %%es:(%1)"::"r"
(character),"r"(str+i));
    }
```

实现写共享内存的核心代码：

```
    asm volatile("movw %0, %%es"::"m"(sel));
    for(int i=0;i<size;++i){
        asm volatile("movb %%es:(%1),%0":"=r"(character):"r"(str + i));
        *(shMem+index+i) = character;
    }
```

3、实现信号量

init

`init` 需要在信号量数组中找到一个空闲信号量，初始化该信号量后，然后将该信号量的地址返回

wait

`wait` 将信号量的值减一，如果值小于0，则阻塞当前进程，并中断切换进程

成功阻塞返回0，否则返回-1

post

`post` 将信号量的值加一，如果值不大于0，则释放阻塞在信号量上的一个进程，并将其唤醒

成功唤醒返回0，否则返回-1

destroy

`destroy` 将信号量还原到初始状态

如果信号量上仍有阻塞的进程，则返回-1

否则返回0

4、解决进程同步问题

(1)生产者-消费者问题

首先分别实现生产者进程和消费者进程的处理函数：

```
/*** 生产者处理函数 ***/
```

```c
void deposit(sem_t* mutex, sem_t* fullBuffers, sem_t* emptyBuffers,int index)
{
    //生产者
    int i = 4;
    while (i-- > 0)
    {
        sem_wait(emptyBuffers);
        sleep(sleepTime);
        sem_wait(mutex);
        sleep(sleepTime);
        produce(index);
        sem_post(mutex);
        sleep(sleepTime);
        sem_post(fullBuffers);
        sleep(sleepTime);
    }
}
/*** 消费者处理函数 ***/
void remove(sem_t* mutex, sem_t* fullBuffers, sem_t* emptyBuffers,int index)
{
    int i = 8;
    while (i-- > 0)
    {
        sem_wait(fullBuffers);
        sleep(sleepTime);
        sem_wait(mutex);
        sleep(sleepTime);
        consume(index);
        sem_post(mutex);
        sleep(sleepTime);
        sem_post(emptyBuffers);
        sleep(sleepTime);

    }
}
```

在主进程中实现生产者消费者进程的创建以及信号量的初始化

```c
    int n = 4;       // buffer size
    int producer = 4;
```

```c
    int consumer = 1;
    sem_t mutex,fullBuffers,emptyBuffers;
    sem_init(&mutex,1);
    sem_init(&fullBuffers,0);
    sem_init(&emptyBuffers,n);
    int ret;
    while(producer-->0){
        ret = fork();

        if(ret == 0){
            //printf("fork producer %d\n",producer);
            deposit(&mutex,&fullBuffers,&emptyBuffers,producer);
            exit();
        }
    }
    while(consumer-->0){
        ret = fork();

        if(ret == 0){
            //printf("fork consumer %d\n",consumer);
            remove(&mutex,&fullBuffers,&emptyBuffers,consumer);
            exit();
        }
    }
```

> PV的操作顺序的影响：
>
> 如果资源信号量与mutex的P、V顺序互换，则条件同步时可能出现死锁。例如消费者进入了缓冲区，发现缓冲区空，则需要等待生产者。此时由于消费者在缓冲区中等待，生产者无法进入缓冲区生成，因此，死锁产生

(2)哲学家就餐问题

首先实现哲学家的处理函数

```c
void philosopher(sem_t *forks,int index){
    int i=2;
    sem_t* left_fork = forks+index;
```

```
    sem_t* right_fork= forks+(index+1)%N;
    while(i-->0){
        think(index);
        if(index%2==0){
            sem_wait(left_fork);
            sleep(sleepTime);
            sem_wait(right_fork);
            sleep(sleepTime);
        }
        else{
            sem_wait(right_fork);
            sleep(sleepTime);
            sem_wait(left_fork);
            sleep(sleepTime);
        }
        eat(index);
        sem_post(left_fork);
        sleep(sleepTime);
        sem_post(right_fork);
        sleep(sleepTime);
    }
}
```

然后在主进程中实现信号量初始化和子进程的创建：

```
    sem_t forks[5];
    for(int i=0;i<5;++i)
        sem_init(forks+i,1);
    for(int i=0,ret=0;i<N;++i){
        ret = fork();
        if(ret == 0){
            philosopher(forks,i);
            exit();
        }

    }
```

(3)读者-写者问题

首先实现读者和写者的处理函数

> 这里Rcount需要通过共享内存实现

```
void writer(int index,sem_t* writemutex){
    int round = 5;
    while(round-->0){
        sem_wait(writemutex);
        sleep(sleepTime);
        W(index);
        sem_post(writemutex);
        sleep(sleepTime);
    }
}

void reader(int index,sem_t* writemutex,sem_t* countmutex){
    int round = 5;
    int Rcount = 0;
    while(round-->0){
        sem_wait(countmutex);
        sleep(sleepTime);
        read(SH_MEM, (uint8_t *)&Rcount, 4, 0);
        if(Rcount == 0){
            sem_wait(writemutex);
            sleep(sleepTime);
        }
        ++Rcount;
        write(SH_MEM, (uint8_t *)&Rcount, 4, 0);
        sem_post(countmutex);
        sleep(sleepTime);

        R(index,Rcount);

        sem_wait(countmutex);
        sleep(sleepTime);
        read(SH_MEM, (uint8_t *)&Rcount, 4, 0);
        --Rcount;
        write(SH_MEM, (uint8_t *)&Rcount, 4, 0);

        if(Rcount == 0){
            sem_post(writemutex);
            sleep(sleepTime);
```

```
            }
            sem_post(countmutex);
            sleep(sleepTime);
        }
    }
```

然后在主进程中实现信号量的初始化和子进程的创建

```
        int ret;
        sem_t writemutex,countmutex;
        int reader_num = 3;
        int writer_num = 3;
        sem_init(&writemutex,1);
        sem_init(&countmutex,1);
        int rcount = 0;
        write(SH_MEM,(uint8_t*)&rcount,4,0);

        while(writer_num-->0){
            ret = fork();
            if(ret == 0){
                //printf("writer %d setup\n",writer_num);
                writer(writer_num,&writemutex);
                exit();
            }
        }

        while(reader_num-->0){
            ret = fork();
            if(ret == 0){
                //printf("reader %d setup\n",reader_num);
                reader(reader_num,&writemutex,&countmutex);
                exit();
            }
        }
```

(4)随机函数

考虑到不能调用c的库函数，因此打算采用基于线性反馈移位寄存器的梅森旋转算法实现随机函数

首先在 `lib.h` 中进行声明:

```c
// add rand method
int index;
int MT[624];

void srand(int seed);
void generate();
int rand();
```

然后再 `syscall.c` 中实现相关算法:

```c
void srand(int seed)
{
    index = 0;
    MT[0] = seed;
    for (int i = 1; i < 624; i++)
    {
        int t = 1812433253 * (MT[i - 1] ^ (MT[i - 1] >> 30)) + i;
        MT[i] = t & 0xffffffff;
    }
}

void generate()
{
    for (int i = 0; i < 624; i++)
    {
        int y = (MT[i] & 0x80000000) + (MT[(i + 1) % 624] & 0x7fffffff);
        MT[i] = MT[(i + 397) % 624] ^ (y >> 1);
        if (y & 1)
            MT[i] ^= 2567483615;
    }
}

int rand()
{
    if (index == 0)
        generate();
    int y = MT[index];
    y = y ^ (y >> 11);
```

```
        y = y ^ ((y << 7) & 2636928640);
        y = y ^ ((y << 15) & 4022730752);
        y = y ^ (y >> 18);
        index = (index + 1) % 624;
        return y;
    }
```

接着在用户程序中引入随机函数，以 `philosopher.c` 为例

首先调用 `srand(0)` ，传入seed = 0进行随机函数初始化

然后将sleepTime改为随机数:

```
#define MAX_SLEEP_TIME 128
//#define sleepTime MAX_SLEEP_TIME
#define sleepTime rand()%MAX_SLEEP_TIME
```

至此必做部分和中断嵌套的选做部分都已完成

实验感受

本次实验让我进一步理解了进程之间的同步机制，对多进程编程有了进一步的掌握，对信号量的实现有了深刻的理解