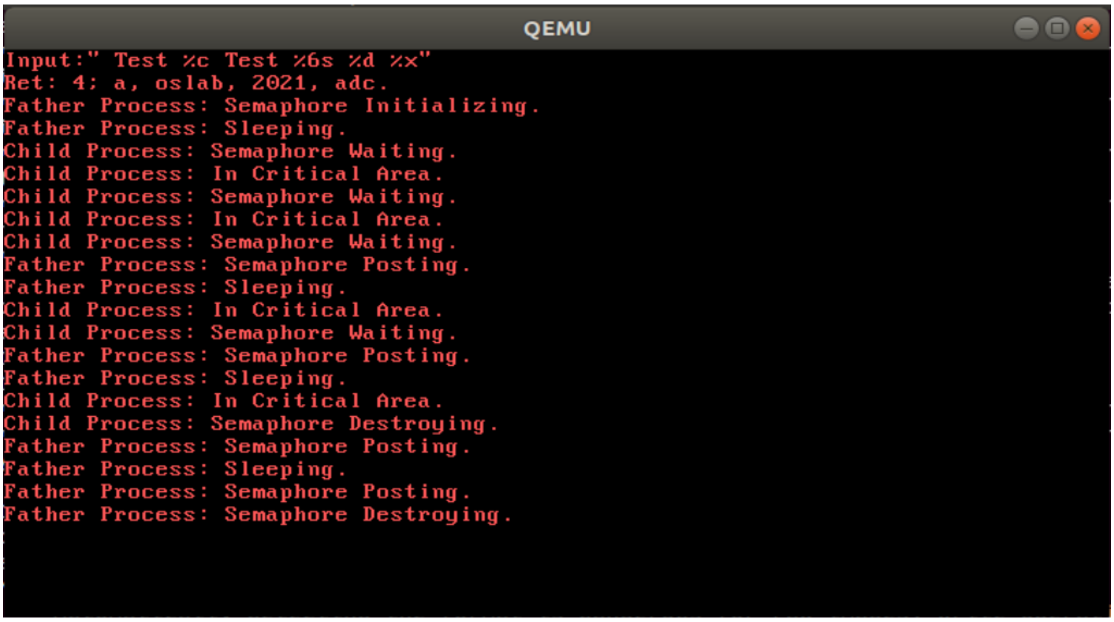
姓名：张涵之　　学号：191220154　　邮箱：1683762615@qq.com

实验进度：我完成了 3.1-3.3.1 的必做内容，选做内容只实现了生产者-消费者

实验结果：下图为 3.1-3.2 测试代码运行效果



解决进程同步问题（3.3）的结果将在思考和总结部分展示

实验修改的代码位置：

3.1. 实现格式化输入函数 lab4/kernel/kernel/irqHandle.c

keyboardHandle：框架代码已经实现了将读取到的 keyCode 放到 keyBuffer 中

根据手册中的示例代码，从 Device STD_IN 上阻塞的进程列表取出一个进程

将这个进程的 state 设为 RUNNABLE，sleepTime 设为 0，实现该进程的唤醒

```c
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 = (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;
        pt->sleepTime = 0;
    }

    return;
}
```

syscallReadStdIn：最多只有一个进程被阻塞在 dev[STD_IN]上，后来的返回-1

即如果 dev[STD_IN].value < 0，证明此时已有多个进程想读，直接设置 eax 为 1

如果 dev[STD_IN].value == 0，则需要将当前进程阻塞在 dev[STD_IN]上

根据手册中的示例代码，可以将 current 线程加到 Device STD_IN 的阻塞列表
将进程的 state 设为 BLOCKED，调用 timerHandle 唤醒进程，读 keyBuffer 中的数据
根据手册中的示例代码，可以依次取出字符 character 传到用户进程
用计数器 i 统计实际读取的字节数，并通过设置当前进程的 eax 进行返回

```
void syscallReadStdIn(struct StackFrame *sf) {
    // TODO: complete `stdin`
    if (dev[STD_IN].value < 0)
        pcb[current].regs.eax = -1;
    else if (dev[STD_IN].value == 0) {
        dev[STD_IN].value--;
        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;
        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 && bufferHead != bufferTail) {
            character = getChar(keyBuffer[bufferHead]);
            bufferHead = (bufferHead + 1) % MAX_KEYBUFFER_SIZE;
            if (character != 0) {
                putChar(character);
                asm volatile("movb %0, %%es:(%1)"::"r"(character),"r"(str+i));
                i++;
            }
        }
        asm volatile("movb $0x00, %%es:(%0)"::"r"(str+i));
        pcb[current].regs.eax = i;
    }
    return;
}
```

3.2. 实现信号量
syscallSemInit：在 sem 数组中寻找未用（state==0）的信号量
若找不到则初始化失败，通过 eax 设置返回值为-1
若找到了则初始化成功，将 state 设为 1，value 设为指定的初始值
仿照 initSem 完成信号量的初始化，通过 eax 设置返回值为 0

```
void syscallSemInit(struct StackFrame *sf) {
    // TODO: complete `SemInit`
    int index = -1;
    for (int i = 0; i < MAX_SEM_NUM; i++) {
        if (sem[i].state == 0) {
            index = i;
            break;
        }
    }
    if (index == -1)
        pcb[current].regs.eax = -1;
    else {
        sem[index].state = 1;
        sem[index].value = (int)sf->edx;
        sem[index].pcb.next = &(sem[index].pcb);
        sem[index].pcb.prev = &(sem[index].pcb);
        pcb[current].regs.eax = 0;
    }
    return;
}
```

syscall_wait：若下标 i 不合法（越界）或信号量未用则操作失败，返回-1

否则使对应 sem 的 value 减一，若 value 取值小于 0 则阻塞自身，返回 0

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

syscall_post：与 syscall_wait 类似，若操作失败则返回值为-1，否则返回值为 0
使对应 sem 的 value 增一，若 value 取值不大于 0 则释放一个阻塞进程

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

syscallSemDestroy：设置对应 state 为 0 即可，失败返回-1，成功返回 0

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

3.3. 解决进程同步问题

1. 哲学家就餐问题

```
#define N 5
sem_t forks[N];

void philosopher(int i, sem_t forks[]) {
    int id = getpid();
    while(1) {
        printf("Philosopher %d: think\n", id);
        sleep(128);
        if (i % 2 == 0) {
            sem_wait(&forks[i]);
            sleep(128);
            sem_wait(&forks[(i + 1) % N]);
            sleep(128);
        }
        else {
            sem_wait(&forks[(i + 1) % N]);
            sleep(128);
            sem_wait(&forks[i]);
            sleep(128);
        }
        printf("Philosopher %d: eat\n", id);
        sleep(128);
        sem_post(&forks[i]);
        sleep(128);
        sem_post(&forks[(i + 1) % N]);
        sleep(128);
    }
}
```

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

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

2. 生产者-消费者问题

```
sem_t mutex;
sem_t fullbuffer;
sem_t emptybuffer;

void deposit() {
    int id = getpid();
    while(1) {
        sem_wait(&emptybuffer);
        sleep(128);
        sem_wait(&mutex);
        sleep(128);
        printf("Producer %d: produce\n", id);
        sleep(128);
        sem_post(&mutex);
        sleep(128);
        sem_post(&fullbuffer);
        sleep(128);
    }
}

void remove() {
    while(1) {
        sem_wait(&fullbuffer);
        sleep(128);
        sem_wait(&mutex);
        sleep(128);
        printf("Consumer: consume\n");
        sleep(128);
        sem_post(&mutex);
        sleep(128);
        sem_post(&emptybuffer);
        sleep(128);
    }
}
```

```
    sem_init(&mutex, 1);
    sem_init(&fullbuffer, 0);
    sem_init(&emptybuffer, 4);
    for (int i = 0; i < 4; i++) {
        if (fork() == 0)
            deposit();
    }
    if (fork() == 0)
        remove();
    exit();
    sem_destroy(&mutex);
    sem_destroy(&fullbuffer);
    sem_destroy(&emptybuffer);
```

3. 读者-写者问题（失败了）

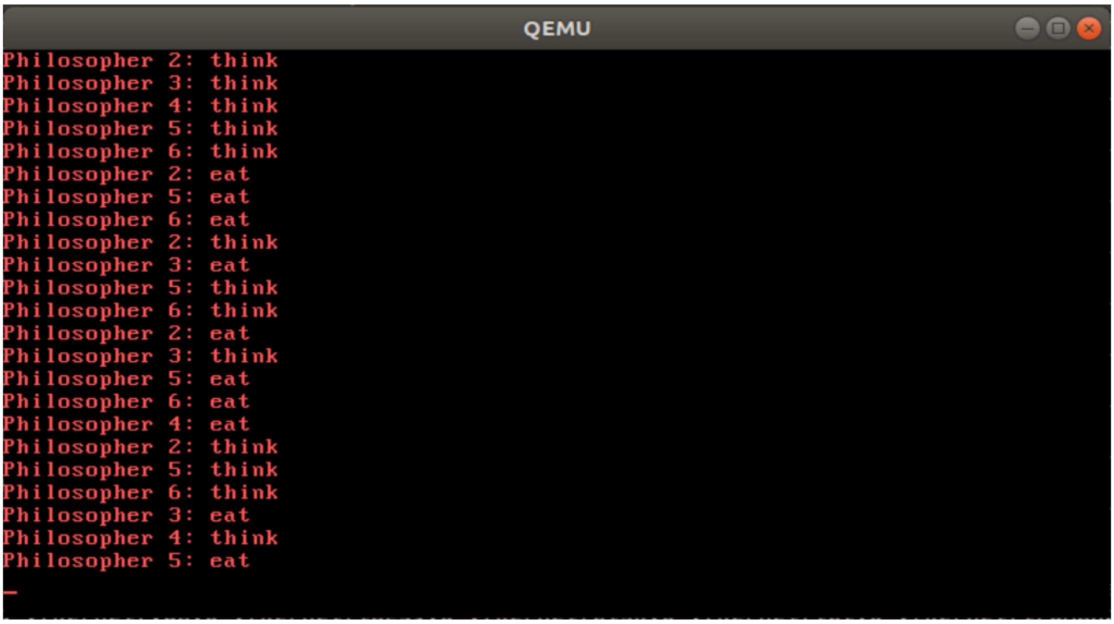
```
sem_t writemutex;
int rcount;
sem_t countmutex;

void write() {
    int id = getpid();
    while(1) {
        sem_wait(&writemutex);
        sleep(128);
        printf("Writer %d: write\n", id);
        sleep(128);
        sem_post(&writemutex);
        sleep(128);
    }
}
```

```
void read() {
    int id = getpid();
    while(1) {
        sem_wait(&countmutex);
        sleep(128);
        if (rcount == 0) {
            sem_wait(&writemutex);
            sleep(128);
        }
        rcount++;
        sleep(128);
        sem_post(&countmutex);
        sleep(128);
        printf("Reader %d: read, total %d reader\n", id, rcount);
        sleep(128);
        sem_wait(&countmutex);
        sleep(128);
        rcount--;
        sleep(128);
        if (rcount == 0) {
            sem_post(&writemutex);
            sleep(128);
        }
        sem_post(&countmutex);
        sleep(128);
    }
}
```

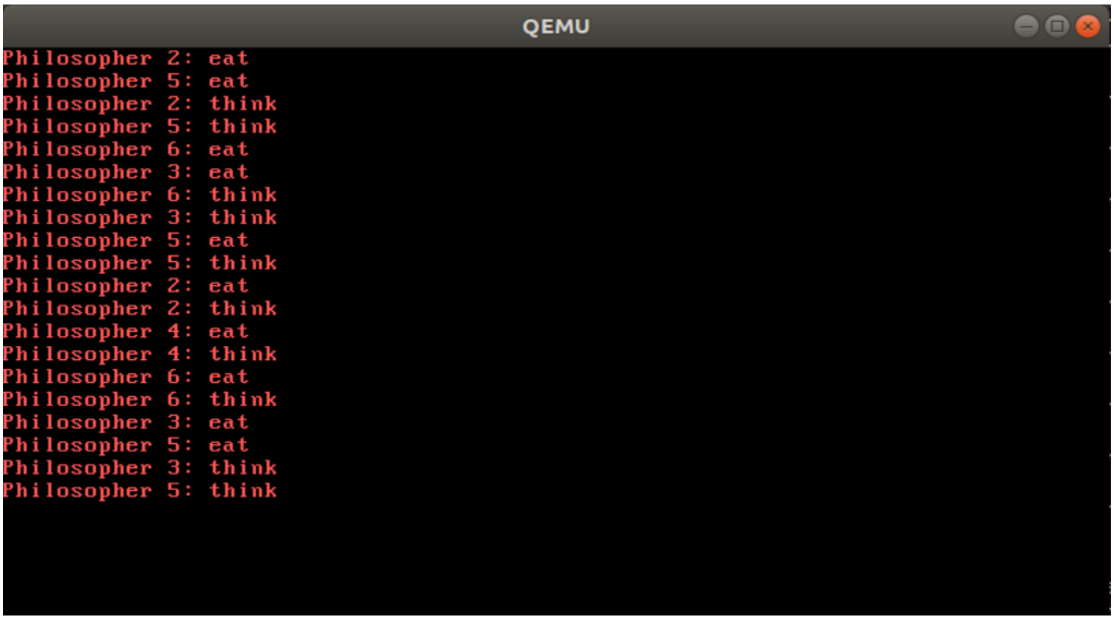
```
    sem_init(&writemutex, 1);
    rcount = 0;
    sem_init(&countmutex, 1);
    for (int i = 0; i < 3; i++) {
        if (fork() == 0)
            write();
    }
    for (int i = 0; i < 3; i++) {
        if (fork() == 0)
            read();
    }
    exit();
    sem_destroy(&writemutex);
    sem_destroy(&countmutex);
```

思考和总结：
分析信号量的测试代码，可见其原理大致是子进程申请信号量进入临界区，父进程归还。

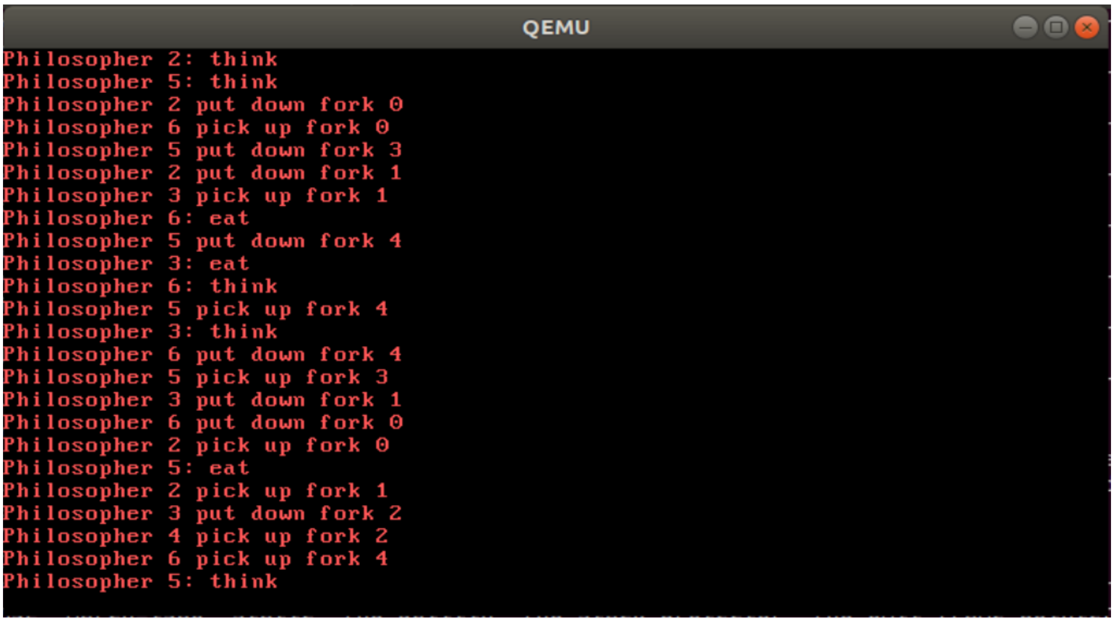
分析哲学家问题的伪代码和输出结果，可以得出哲学家和叉子的序号关系：
哲学家： 2 3 4 5 6
叉子： 0 1 2 3 4 0



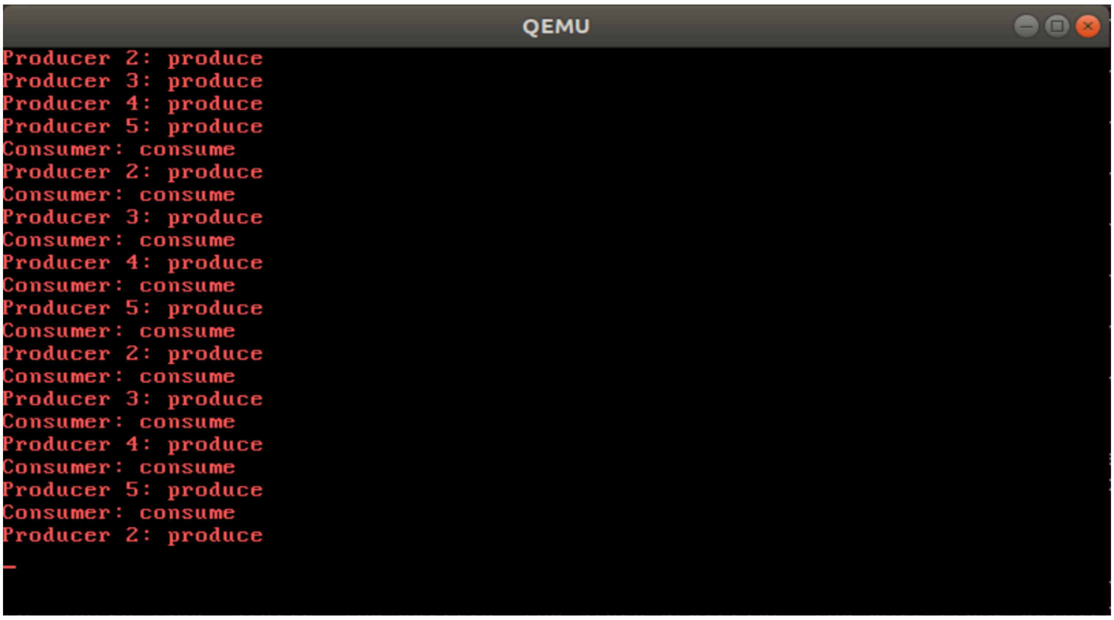
这个输出结果看起来有点奇怪，好像三个以上的哲学家可以同时吃。但 5 eat 和 6 eat 之间的时间间隔极长，而 6 eat 和 2 think 之间几乎没有时差，由于 2 和 5 的 think 在两个 fork 都已被释放，再经过一个 sleep 才输出，推测哲学家 2 和 5 已经分别释放了叉子 0 和 4，6 开始吃，此时由于叉子 1 已经释放，叉子 2 原本就未被占用，所以 3 也开始吃，然后才输出 5 think 的信息，我试着把 think 的输出语句放到 sem_post 两个 fork 前面，效果如下：



可见逻辑基本正常了，每一时刻有且只有两个人同吃，有人放下叉子开始思考，其他人才能开始吃。继续加入调试输出语句，每次执行 sem_wait、sem_post 与 sleep 之间输出是哪位哲学家拿起或放下了哪个叉子，输出的结果中拿起放下叉子的行为比较符合预期：



生产者-消费者的 buffer 大小被设为 4，可见 4 名生产者先轮流生产，将 buffer 填满，在这以后，consumer 每次取出一个产品，就有一个生产者放入新的产品：



在读者-写者程序中，注意到读写进程可以互斥，但读的进程不但不能同步，而且会产生死锁现象，输出调试信息发现全局变量 Rcount 无法被各子进程共享，即在每个进程中都是独立的，不能同步修改。怎么会这样呢？查阅资料发现 fork 操作和 pthread create 线程工作原理不同，fork 得到的子进程拥有独立的代码段和数据段，即使不同进程中全局变量的逻辑地址是相同的，它们实际在内存空间中的物理地址是不同的。要想在 fork 出的父子进程中正确更新 Rcount 需要共享内存、管道通信等方法，由于时间和技术有限这里不作实现。