

姓名：童锦

学号：211300097

邮箱：211300097@smail.nju.edu.cn

实验任务

1. 实现格式化输入函数
2. 实现信号量相关系统调用
3. 基于信号量解决进程同步问题

实验情况

实验进度

代码修改处

Note

接下来为实验的主体内容，每一个任务由具体实现和实验思路两部分组成

实验内容

Task1

| 实现格式化输入函数

具体实现

keyboardHandle(partial):

```
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));

    pt->state = STATE_RUNNABLE;
    pt->sleepTime = 0;

    dev[STD_IN].pcb.prev = (dev[STD_IN].pcb.prev)->prev;
```

```

        (dev[STD_IN].pcb.prev)→next = &(dev[STD_IN].pcb);
    }

```

`syscallReadStdIn(partial) :`

```

if (dev[STD_IN].value == 0) { // no precess blocked
    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;
    pcb[current].sleepTime = -1; // blocked
    asm volatile("int $0x20");

    // Resume
    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){
            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++;
            }
        }else break;
    }
    asm volatile("movb $0x00, %%es:(%0)"::"r"(str+i)); // the end
of str

    pcb[current].regs.eax = i; // strlen have been read
    return;
}else if (dev[STD_IN].value < 0) { // with process blocked
    pcb[current].regs.eax = -1;
    return;
}

```

实验思路

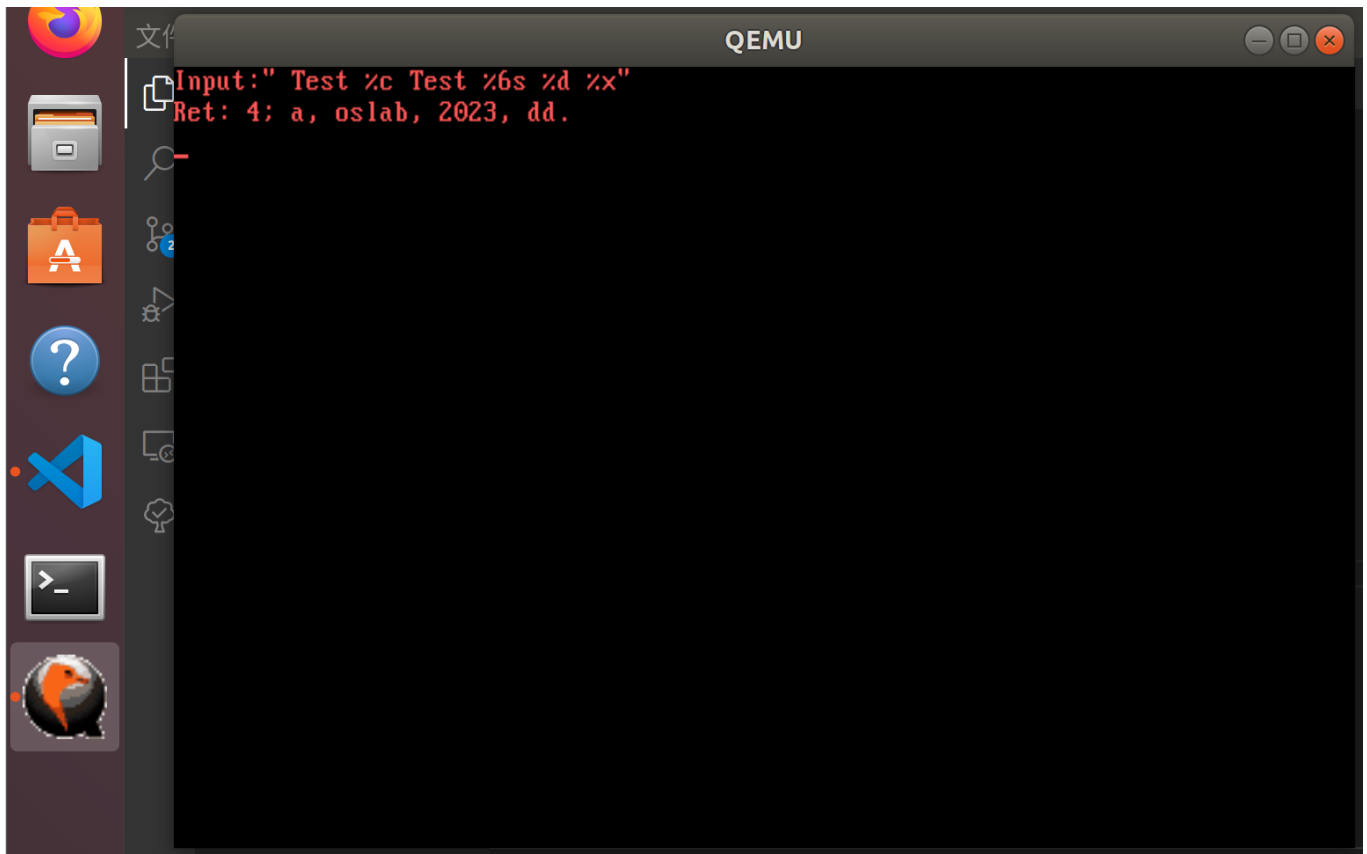
1. keyboardHandle

1. 这个函数的逻辑比较简单： 在从键盘中读到有效信息后从信号量队列中取出等待的进程(如果有的话)；
2. 这里比较有趣的是 `pt` 指针是如何指向队列中等待的进程的： 这里灵活地使用了c语言提供的自由地指定变量类型与进行地址计算的特性， 计算出了 `(uint32_t)&(((ProcessTable*)0)→blocked))` 这个 `blocked` 成员与进程变量首地址间的"偏移量"， 从而找到进程的首地址；

2. syscallReadStdIn

1. 这个函数的逻辑是首先判断当前信号量队列中是否有进程堵塞： 若有， 根据实验的要求， 读取失败并且返回 `-1`； 若无： 进行读取；
 1. 读取过程中， 首先是先把当前进程加入到该信号量的等待列表当中， 等待 `keyboardHandle` 的唤醒。 个人认为这里非常有趣的一个点是双向链表的结构给加入和取出元素带来的便利性， `dev[STD_IN].pcb.next` 指向的是队列的头， 我们在此写入往队列中加入， 而 `dev[STD_IN].pcb.prev` 指向的是队列的尾， 我们在此从队列里取出；
 2. 将 `sleepTime` 设置为 `-1`， 这意味着在无唤醒的情况下"永久睡眠"。 接下来切换到别的进程；
 3. 当处于等待的进程被唤醒之后， 读取并写至目标字符串的操作与lab2中 `syscallPrint` 的行为十分类似， 在此就不再赘述；
 4. `putChar(character)` 这行代码并无特殊含义， 只是输出到端口以便进行debug(当然也可以像lab2中一样在 `keyboardHandle` 中添加一段将键盘打印字符同步到屏幕上的代码)；
3. 进一步思考就会发现， 如果有两个进程同时在一个终端等待输入， 如果不做其他任何限制的话就有可能出现一段输入被两个进程"瓜分"的情况。 对这个问题我的想法是： 一方面我们可以增加一个缓冲区来接受字符， 但由于在此处只有一个终端， 这个处理就显得不太自然； 另一个做法是在 `scanf` 函数中引入互斥锁， 相同时间只允许一个进程进入读取的状态， 这样就可以避免两个进程同时尝试读取的情况！

结果展示



Task2

实现信号量相关系统调用

具体实现

`syscallSemInit:`

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

        pcb[current].regs.eax = i;
    }else{
```

```

        pcb[current].regs.eax = -1;
    }
    return;
}

```

`syscallSemDestroy` 实现逻辑类似，在此不占用篇幅

`syscallSemPost`:

```

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 = (ProcessTable*)((uint32_t)(sem[i].pcb.prev) -
            (uint32_t)&(((ProcessTable*)0)->blocked));
            pt->state = STATE_RUNNABLE;
            pt->sleepTime = 0;
            sem[i].pcb.prev = (sem[i].pcb.prev)->prev;
            (sem[i].pcb.prev)->next = &(sem[i].pcb);
        }
    } else {
        pcb[current].regs.eax = -1;
    }
    return;
}

```

`syscallSemWait` 实现逻辑类似，在此不占用篇幅

`syscallSem(partial)`:

```

void syscallSem(struct StackFrame *sf) {
    disableInterrupt();
    switch(sf->ecx) {
        ...
        ...
        ...
    }
}

```

```
enableInterrupt();  
}
```

实现思路

1. syscallSem

1. 由于系统调用时外部中断并未屏蔽，所以在进行信号量相关的系统调用时很容易会出现Race Condition的问题(例如 `syscallSemInit` 中 `sem[i].state` 的赋值并不是原子操作，所以又可能会出现多个信号量占用同一个位的情况，造成难以控制的后果)。所以统一在进入信号量相关调用之前关闭外部中断，在完成相关调用时再重新开启；

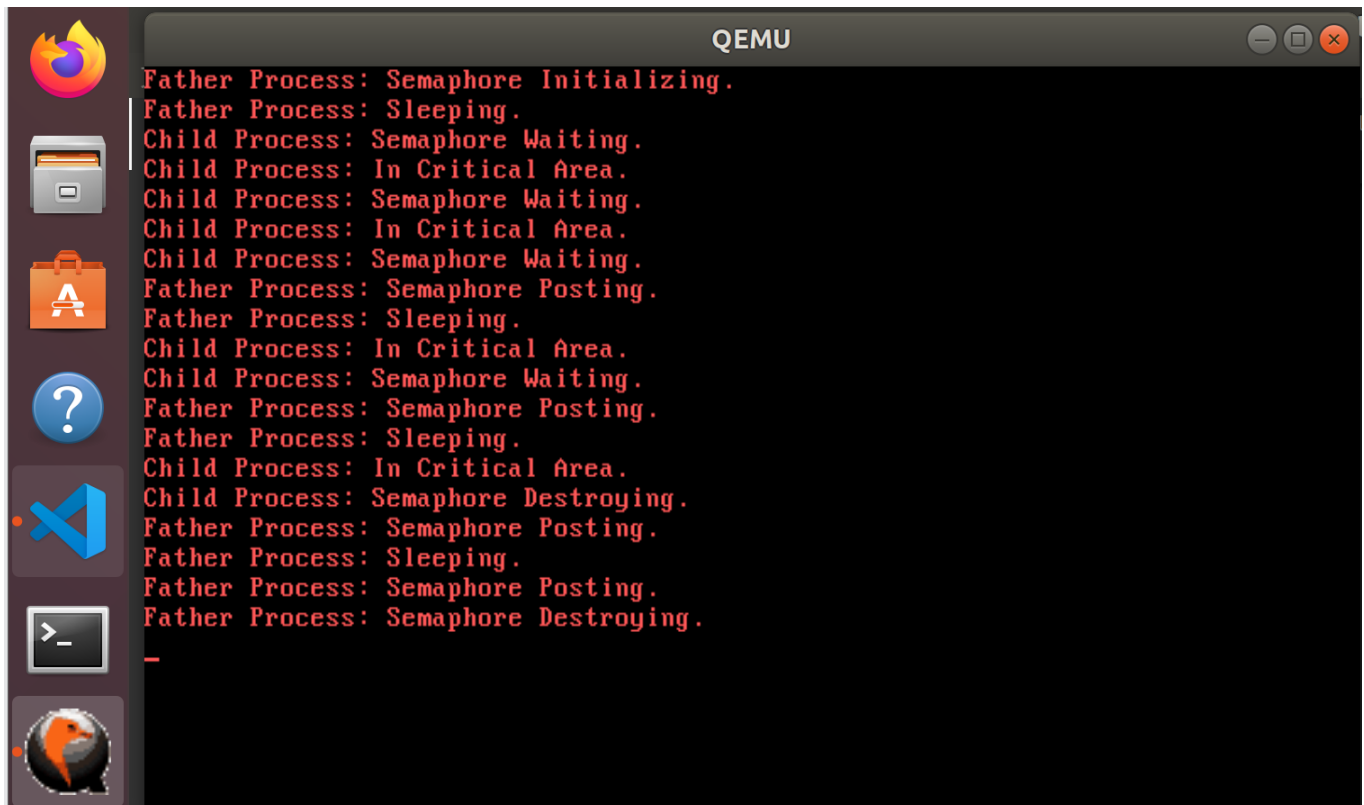
2. syscallSemPost/syscallSemWait

1. 这两个函数的对于信号量等待队列操作的实现分别与之前提到的 `keyboardHandle` 和 `syscallReadStdIn` 一致。除此之外，只需要添加一些条件判断进行检查即可；

3. syscallSemInit/syscallSemDestroy

1. 只需进行若干个条件判断即可，在此不多赘述。

结果展示



```
QEMU  
Father Process: Semaphore Initializing.  
Father Process: Sleeping.  
Child Process: Semaphore Waiting.  
Child Process: In Critical Area.  
Child Process: Semaphore Waiting.  
Child Process: In Critical Area.  
Child Process: Semaphore Waiting.  
Father Process: Semaphore Posting.  
Father Process: Sleeping.  
Child Process: In Critical Area.  
Child Process: Semaphore Waiting.  
Father Process: Semaphore Posting.  
Father Process: Sleeping.  
Child Process: In Critical Area.  
Child Process: Semaphore Destroying.  
Father Process: Semaphore Posting.  
Father Process: Sleeping.  
Father Process: Semaphore Posting.  
Father Process: Semaphore Destroying.  
-
```

Task3.1

哲学家就餐问题

具体实现

```

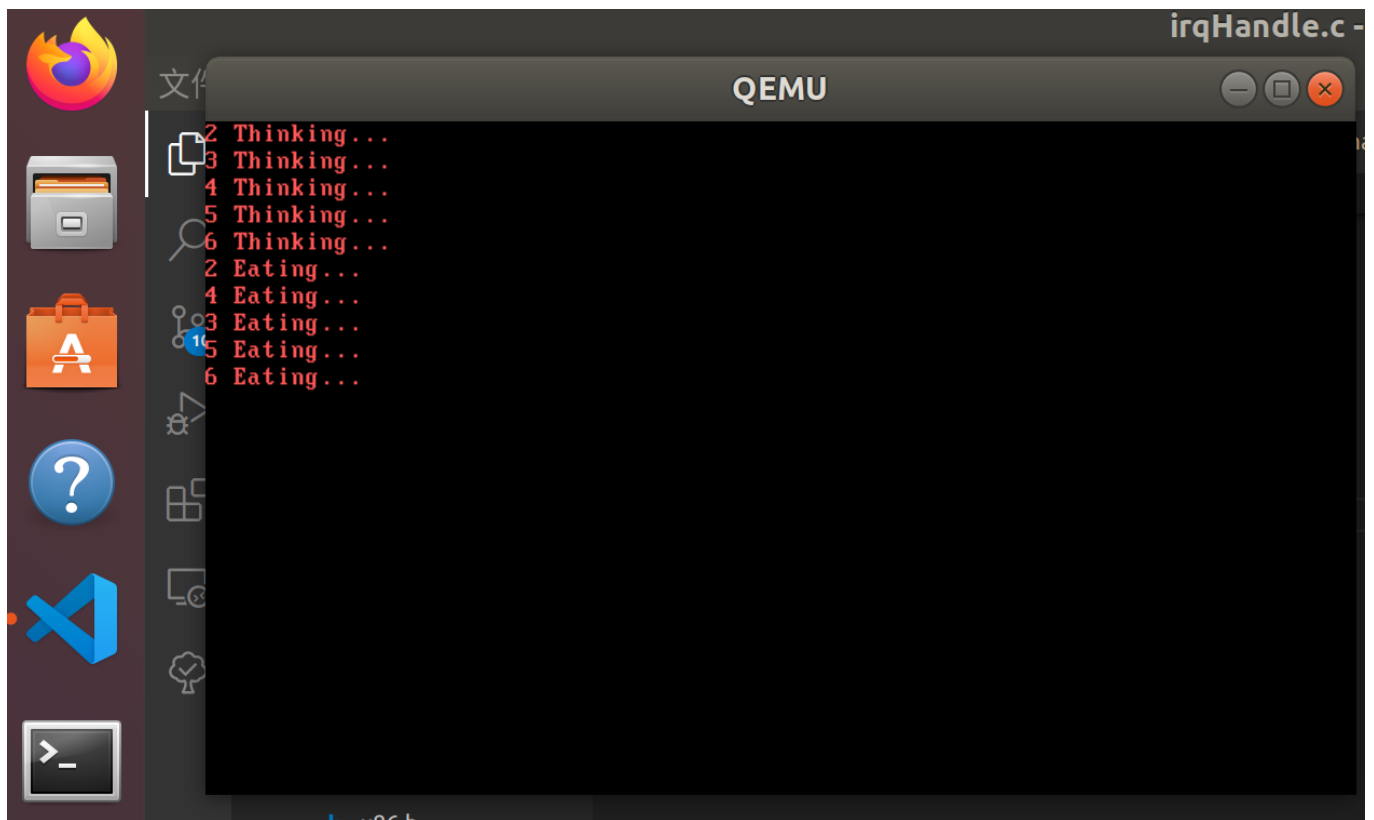
sem_t forks[5];
for(int i = 0; i < 5; i++) {
    sem_init(&forks[i], 1);
}
for(int i = 0; i < 5; i++) {
    if(fork() == 0) {
        volatile pid_t id = getpid();
        // printf("%d", id);
        think(id);
        sleep(128);
        if(id % 2 == 0) {
            sem_wait(&forks[id]);
            sem_wait(&forks[(id+1)%5]);
        } else {
            sem_wait(&forks[(id+1)%5]);
            sem_wait(&forks[id]);
        }
        eat(id);
        sleep(128);
        sem_post(&forks[id]);
        sem_post(&forks[(id+1)%5]);
        sleep(128);
        break;
    }
}
}

```

实现思路

1. 首先申请一个大小为5的信号量数组并一一初始化 `value = 1`;
2. 由进程1顺序创建出进程2/3/4/5/6，分别代表五个"哲学家";
3. 采用讲义中提到的方案3来实现没有死锁的就餐。

结果展示



Task3.2

生产者-消费者问题

具体实现

```
int buffer_size = 3;
sem_t full;
sem_init(&full, 0);
sem_t empty;
sem_init(&empty, buffer_size);
sem_t mutex;
sem_init(&mutex, 1);

for(int i = 0; i < 5; i++) {
    if(fork() == 0) {
        if(i == 4) { // Consumer
            while(1) {
                sem_wait(&full);
                sleep(128);
                sem_wait(&mutex);
                sleep(128);
                printf("Consumer consumes ... \n");
                sleep(128);
                sem_post(&mutex);
            }
        }
    }
}
```



```

        sleep(128);
        sem_post(&empty);
    }
} else {
    while(1) {
        pid_t id = getpid();
        sem_wait(&empty);
        sleep(128);
        sem_wait(&mutex);
        sleep(128);
        printf("Producer %d produces ... \n",
id);

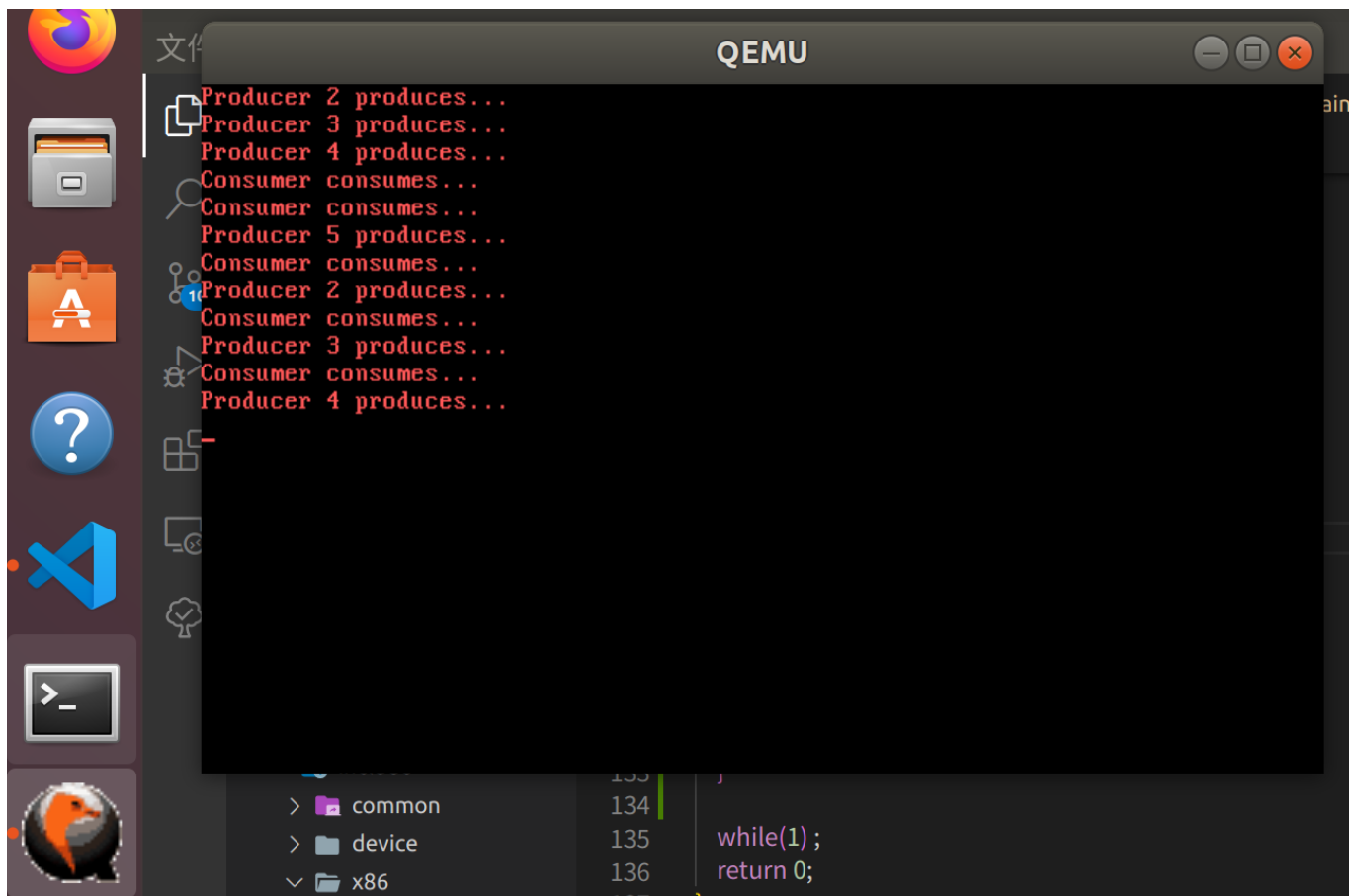
        sleep(128);
        sem_post(&mutex);
        sleep(128);
        sem_post(&full);
    }
}
}
}
}

```

实现思路

按照讲义中的要求实现，并且使用了讲义提供的方案。

结果展示



Task3.3

读者-写者问题

具体实现

```
#define ADD 1
#define SUB 0
var_init(0);
sem_t WriteMutex;
sem_init(&WriteMutex, 1);
sem_t CountMutex;
sem_init(&CountMutex, 1);
for(int i = 0; i < 6; i++) {
    if(fork() == 0) {
        if(i > 2) { // Reader
            while(1) {
                pid_t id = getpid();
                sem_wait(&CountMutex);
                sleep(128);
                if(var_change(ADD, 0) == 0) {
                    sem_wait(&WriteMutex);
                }
            }
        }
    }
}
```

```

    reader\n", id-4, var_change(ADD, 0));

    var_change(ADD, 1);
    sem_post(&CountMutex);
    printf("Reader %d: read, total %d\n", id-4, total);

    sleep(128);
    sem_wait(&CountMutex);
    var_change(SUB, 1);
    if(var_change(ADD, 0) == 0) {
        sem_post(&WriteMutex);
    }
    sleep(128);
    sem_post(&CountMutex);
}

}else {
    while(1) {
        pid_t id = getpid();
        sem_wait(&WriteMutex);
        sleep(128);
        printf("Writer %d: write\n", id-1);
        sem_post(&WriteMutex);
        sleep(128);
    }
}

}

}

```

syscallVar/syscallVarInit/syscallVarChange :

```

        var = sf->edx;
        pcb[current].regs.eax = 0;
        return;
    }

    void syscallVarChange(struct StackFrame *sf) {
        int num = sf->ebx;
        switch(sf->edx) {
            case 1: // add
                var += num;
                break;
            case 0:
                var -= num;
                break;
            default:
                break;
        }
        pcb[current].regs.eax = var;
        return;
    }
}

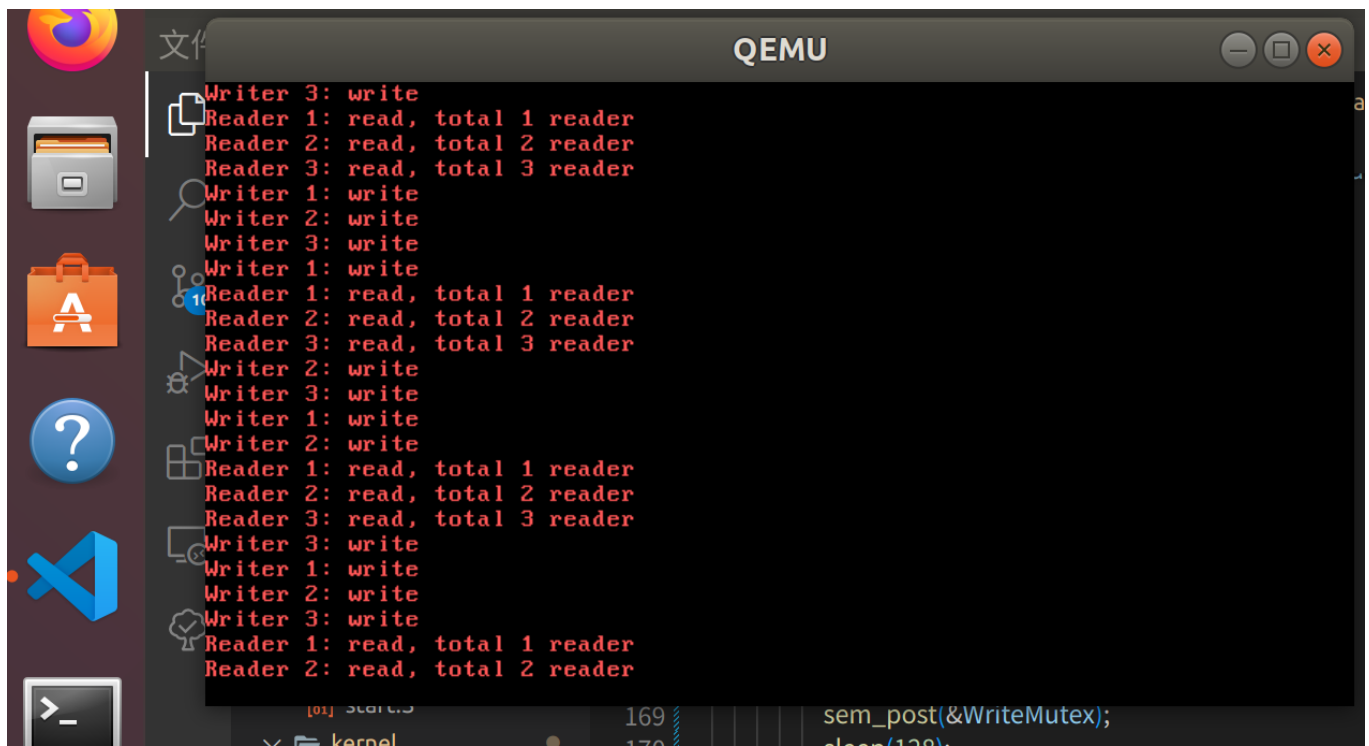
```

实现思路

这个进程同步任务在实现上要比前两个更为麻烦，主要在于如何实现共享 `Rcount` 变量：

1. 最开始并没有注意到这个问题，仅仅通过 `int Rcount = 0` 申请了一个变量。但一运行就发现了死锁的问题，也就是除了 `idle` 外所有进程都阻塞了！最开始以为是同步的代码编写的有问题，后来通过输出的方式发现了 `Rcount++` 后仍旧 `Rcount=0`，进一步探究发现读者二进程中才会发生这个问题而读者一进程中不会——这下就立马发现问题所在了，也即 `Rcount` 并不是共享的；
2. 那么如何使其共享呢？
 1. 由于没有实现分页/分段或虚拟存储的支持，所以让多个进程的 `Rcount` 同属于一块内存区域不太现实，所以放弃了这个方案；
 2. 那么还有什么位置可以实现共享的呢，一个自然的想法是内核区域：最开始的想法是为了省事拓展一下信号量系统调用的定义，加上几个系统调用来解决这个问题，但仔细想想这会破坏原本的框架并且说不定还有暴露内部信息的风险；所以干脆就定义了新的系统调用 `syscallVar/syscallVarInit/syscallVarChange` 来初始化和修改共享变量。只不过目前这些调用的实现还是十分简单，例如只有一个共享变量并且要是整型的，只能对这个变量进行加减...
3. 暂时没有想到更精巧的解决方式，有更好的想法之后再进行改进。

结果展示



```
Writer 3: write
Reader 1: read, total 1 reader
Reader 2: read, total 2 reader
Reader 3: read, total 3 reader
Writer 1: write
Writer 2: write
Writer 3: write
Writer 1: write
Reader 1: read, total 1 reader
Reader 2: read, total 2 reader
Reader 3: read, total 3 reader
Writer 2: write
Writer 3: write
Writer 1: write
Writer 2: write
Reader 1: read, total 1 reader
Reader 2: read, total 2 reader
Reader 3: read, total 3 reader
Writer 3: write
Writer 1: write
Writer 2: write
Writer 3: write
Reader 1: read, total 1 reader
Reader 2: read, total 2 reader
```

总结

这次实验算是深入体验了一把进程同步的机制，并且浅尝了一下编写同步程序的快乐。

在开始这次实验之前总是很难理解理论课中讲解的各种概念，对于各种经典进程同步问题还停留在“记忆”的层面，难以触及其中。但是在完成这次实验，写下这个总结的时刻，我发现自己对于同步问题再也不会望而生畏了，甚至产生了兴趣，或许这就是“get your hands dirty”的魅力！

除此之外，还要感谢实验提供的讲义和完善的框架，在实验的很多时刻都解答了我的疑惑，并且通过阅读这些收获了不少的增量知识。