OS_lab4 实验报告

多线程，信号量与同步机制

陈永恒 151220012

必答题

1. 什么地方是所有进程共享的呢？你可以据此实现你的进程匿名信号量么？

> 内核是所有进程共享的，可以让内核管理进程匿名信号量

2. 普通的全局变量是不是可以被用户修改？

> 是的，这是用户创建的且共享的

3. linux进程和线程下匿名信号量的实现有什么本质区别？

> 进程由于内存空间是独立的，不能简单地共享内存．这就得通过一些别的机制，如信号机制，管道机制来实现共享内存，而匿名信号量便可以建立这个基础上．

4. 结合你的os，你该怎样实现进程/线程的具名信号量？

> 具名信号量可以由内核管理，然后用户进程通过系统函数来访问修改它们即可

实验要求

1. 实现多线程
2. 实现匿名信号量，具体实现5个函数，分别如下：

> sem_init: 初始化一个信号量
> sem_wait：相当于P操作，将信号量减1，如果该信号量本来为0，则挂起当前进程/线程
> sem_post：相当于V操作，将信号量加1，如果该信号量本来为0，则唤醒因该信号量而挂起的进程/线程
> sem_trywait:非阻塞版的P操作
> sem_destroy: 当不需要使用某个信号量后，就可以销毁该信号量

3. 展示生产者和消费者问题

实验进度

所有要求全部完成

1. 多线程的实现

   > 基本思想和进程的创建差不多，线程的实验要更简单，因为它是浅拷贝．大致流程是：拷贝一级页表，拷贝内核栈，分配线程的独立栈，设置栈顶和eip.

```
int thread_create(uint32_t func){
    PCB* thread_pcb=pcb_create();
    if(thread_pcb==NULL){
        return -1;
    }
    int old_pid=thread_pcb->pid;
    pte_t* old_pte=thread_pcb->pgdir;
    memcpy((void*)thread_pcb,(void*)cur_pcb,sizeof(PCB));
    thread_pcb->pgdir=old_pte;
    thread_pcb->pid=old_pid;
    memcpy(thread_pcb->pgdir,cur_pcb->pgdir,PGSIZE);
    thread_pcb->ppid=cur_pcb->pid;
    mm_alloc(thread_pcb->pgdir,USER_STACK-2*STACKSIZ,2*STACKSIZ);
    thread_pcb->tf=(struct TrapFrame*)((uint32_t)thread_pcb->kern_stack+((uint32_t)cur_pcb->tf-(uint32_t)cur_pcb->kern_stack));
    thread_pcb->tf->esp=USER_STACK-0x80;
    thread_pcb->tf->eip=func;
    pcb_ready(thread_pcb);
    return 0;
}
```

2. 匿名信号量的实现

> 由于线程是共享全局变量的，这使得信号量的实现非常简单，只需记住把它们实现成原子操作，即在函数开始的时候关中断，结束时开中断．当然，还要把它们封装成系统函数．

3. 展示生产者和消费者问题

> 只要线程和信号量的实现正确了，展示生产者和消费者问题也是非常简单的．
> 我先创建了一个信号量，把 count设置成 1 ，即把它当作互斥信号量来使用：

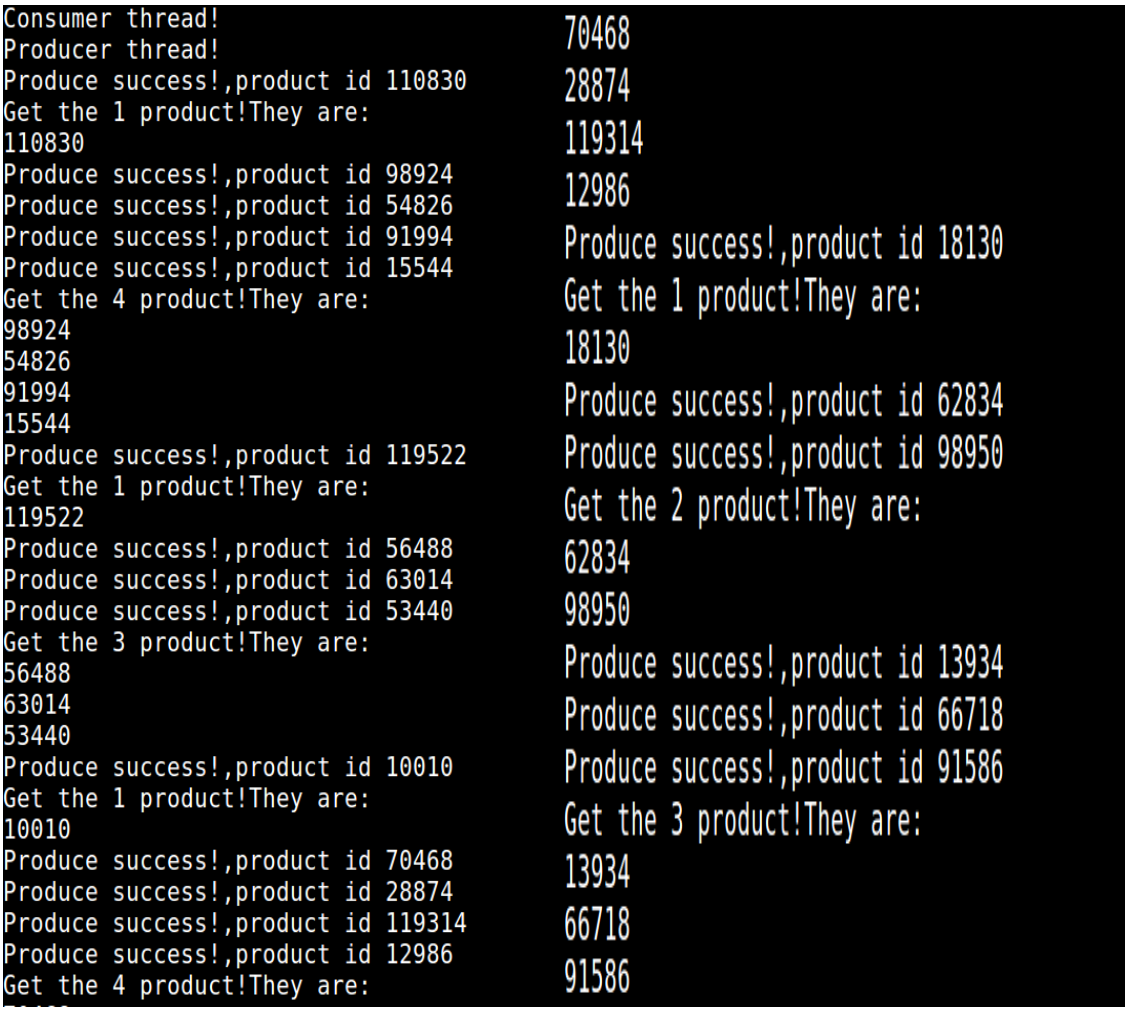
```
    system_sem_init(&mutex,1);
    system_thread_create((uint32_t)producer);
    system_thread_create((uint32_t)consumer);
```

我创建了两个线程，分别代表生产者和消费者：

```
void producer(){
    printk("Producer thread!\n");
    int c=1;
    while(1){
        system_sem_wait(&mutex);
        product_id=product_id*3179%123546;
        products[product_c++]=product_id;
        printk("Produce success!,product id %d\n",produ
ct_id);
        if(c++%(product_id%5+1)==0)system_yield();
        system_sem_post(&mutex);
    }
}

void consumer(){
    printk("Consumer thread!\n");
    int c=1;
    while(1){
        system_sem_wait(&mutex);
        if(product_c!=0){
        printk("Get the %d product!They are:\n",product
_c);
        for(int i=0;i<product_c;i++)printk("%d\n",produ
cts[i]);
        product_c=0;
        }
        if(c++%(product_id%5+1)==0)system_yield();
        system_sem_post(&mutex);
    }
}
```

可以看到，生产者随机生产1-5个产品，消费者在有产品的时候消费所有的产品．我为每个产品生成一个伪随机的id,下面是实际的运行效果（为了节省空空间，我把图片分成了两半）



调度是随机的，可以看到，这就像在串行执行．这是因为为了突出重点，我把调试信息关了，在助教实际运行我的lab的时候可以看到调度信息．生产者消费者问题解决！

实验中遇到的坑

在实现sem_wait()的时候，我在思考如何存放被阻塞的线程．一开始我是直接把线程放到阻塞队列中，然后这sem_wait里面直接调度．后来发现这样有问题，因为在重新调度这个线程的时候就不是在调度前的下一条语句中继续执行了．这样会导致信号量的混乱．

解决方案:在信号量中添加一个等待队列，每次把等待的线程加入里面，在sem_post()的时候在把它单独调度，而不是把它和别的线程一起调度．