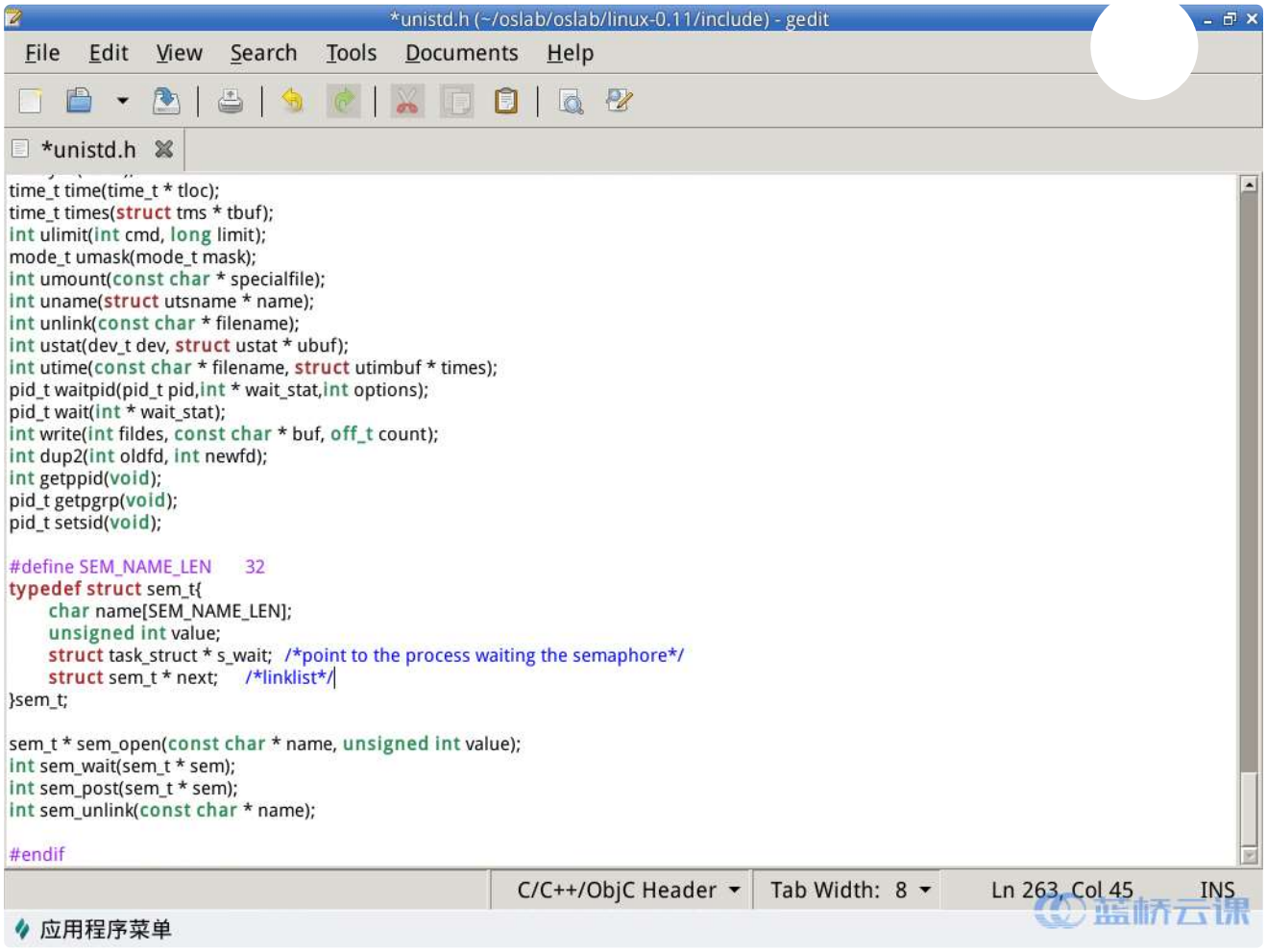
“操作系统原理与实践”实验报告

信号量的实现和应用

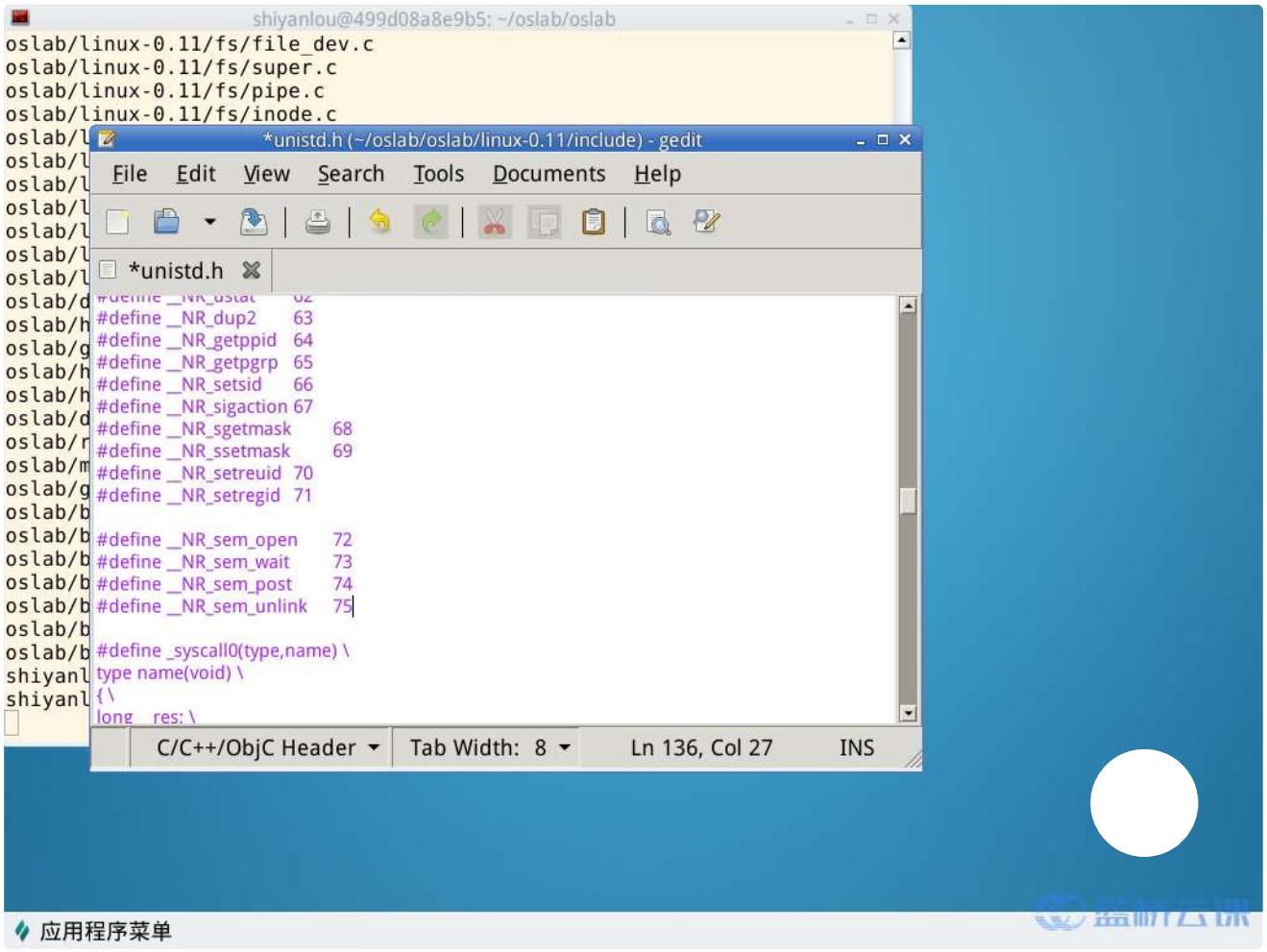
信号量的实现和应用

在linux 0.11上实现信号量

在实现的过程中将运用到系统调用的实验知识，通过此次实验也对前面做了一个比较系统的复习 ###在include/unistd.h文件中 定义如下数据结构，添加声明

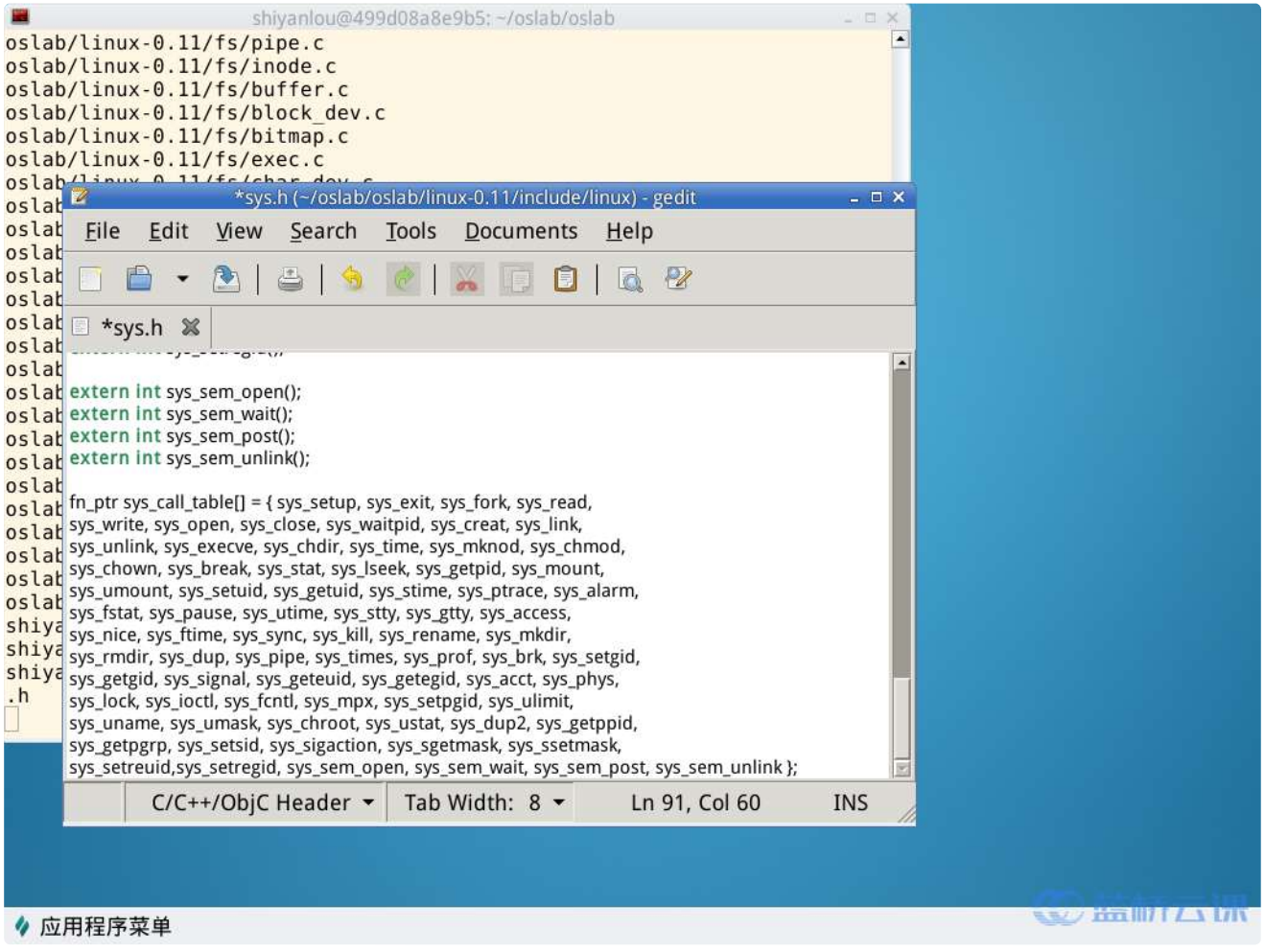


添加系统调用号



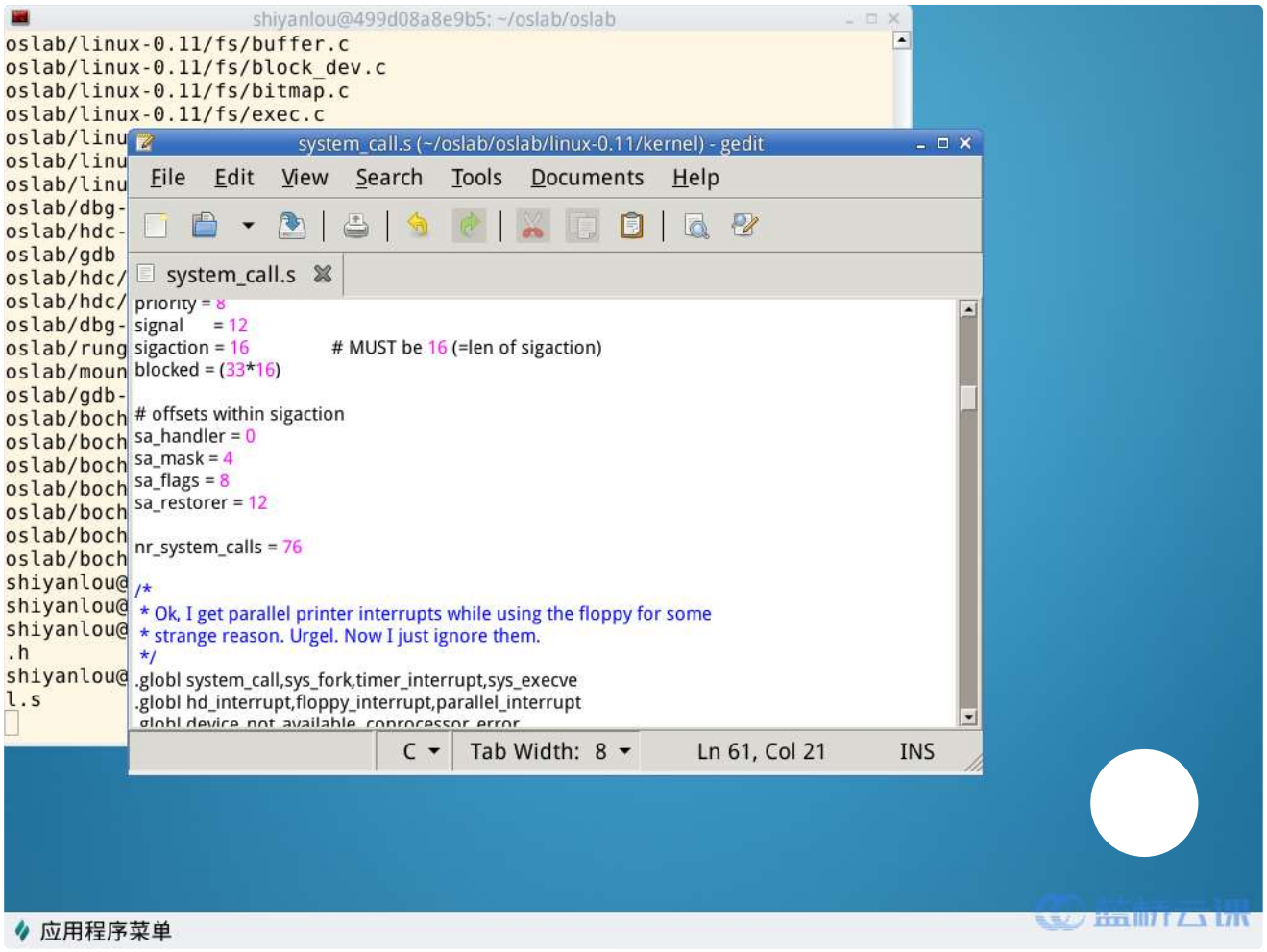
在include/linux/sys.h文件中

添加声明，sys_call_table数组添加两个元素



在kernel/system_call.s中

增加系统调用总数



在kernel/中实现sem.c

在此文件内实现4个山寨信号量函数。 首先包含了必要的头文件，然后定义信号量队列的头指针。

```
#include <linux/kernel.h>
#include <asm/system.h>
#include <linux/sched.h>
#include <asm/segment.h>
#include <unistd.h>

sem_t *sem_head = &((sem_t *){"", 0, NULL, NULL});    /* head of the linklist */
```

自己新增的一个函数，负责把信号量的名字从用户栈拷贝到内核栈

```
/* copy sem's name form user stack to kernel stack */
static inline int str_u2k(const char *ustr, char *kstr, unsigned int length)
{
    char c;
    int i;

    for(i=0; (c=get_fs_byte(ustr++))!='\0' && i<length; i++)
        *(kstr+i)=c;
    *(kstr+i)='\0';

    return i;
}
```

下面进入正题，实现四个山寨的信号量函数

```
sem_t *sys_sem_open(const char *name, unsigned int value)
{
    sem_t *sem_cur, *sem_pre;
    char pname[SEM_NAME_LEN];

    str_u2k(name, pname, SEM_NAME_LEN);

    /*whether the sem existes in the linklist */
    for(sem_pre=sem_head, sem_cur=sem_head->next; sem_cur && strcmp(pname, sem_cur->name);
            sem_pre=sem_cur, sem_cur=sem_cur->next);

    /*if not exists, new a sem */
    if(!sem_cur)
    {
        printk("semaphore %s no found. created a new one. \n", pname);
        sem_cur = (sem_t *)malloc(sizeof(sem_t));
        strcpy(sem_cur->name, pname);
        sem_cur->value = value;
        sem_cur->next = NULL;
        sem_pre->next = sem_cur;
    }
    printk("pid %d opens semaphore %s(value %u) OK. \n", current->pid, pname, sem_cur->value);
    return sem_cur;
}
```

```
int sys_sem_wait(sem_t *sem)
{
    cli();
    while(sem->value<=0)
        sleep_on(&(sem->s_wait));
    sem->value--;
    sti();
    return 0;
}
```

```
int sys_sem_post(sem_t *sem)
{
    cli();
    sem->value++;
    /* if still have process waiting for sem */
    if(sem->s_wait)
    {
        wake_up(&(sem->s_wait));
        sti();
        return 0;
    }
    sti();
    return -1;
}
```

```
int sys_sem_unlink(const char *name)
{
    sem_t *sem_cur, *sem_pre;
    char pname[SEM_NAME_LEN];
    int i;

    str_u2k(name, pname, SEM_NAME_LEN);

    for(sem_pre=sem_head, sem_cur=sem_head->next; sem_cur && strcmp(pname, sem_cur->name);
            sem_pre=sem_cur, sem_cur=sem_cur->next);

    /*if not found, return -1 */
    if(!sem_cur)
        return -1;

    /*if found, free it */
    sem_pre->next = sem_cur->next;
    free(sem_cur);
    printk("unlink semaphore %s OK. \n", pname);
    return 0;
}
```

编写在linux 0.11下运行的pc.c程序

18.2.4 : 之前大括号打错了，以至于fork了新的消费者而没参与到进程同步，现进行修正。此程序有时会出现死锁情况，原因不明。不过输出也是正确的，遇到死锁情况采取放置策略，重新启动系统运行程序即可

```
#define __LIBRARY__
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

_syscall2(sem_t *,sem_open,const char *,name,unsigned int,value)
_syscall1(int,sem_wait,sem_t *,sem)
_syscall1(int,sem_post,sem_t *,sem)
_syscall1(int,sem_unlink,const char*,name)

const char *FILENAME = "/usr/root/buffer_file";
const int NR_CONSUMERS = 5;                         /* number of consumers */
const int NR_ITEMS = 50;                          /* the maximum of item */
const int BUFFER_SIZE = 10;
sem_t *mutex, *full, *empty;
unsigned int item_pro, item_used;                /* the NO. of item which is produced or consumed just now */
int fi, fo;                                         /* the I/O pointer*/


int main(int argc, char *argv[])
{
    char *filename;
    int pid;
    int i, flag = 1;

    filename = argc > 1 ? argv[1] : FILENAME;

    /* O_TRUNC means open the file in read-only or write-only, if exists, wipe the file*/
    fi = open(filename, O_CREAT| O_TRUNC| O_WRONLY, 0222);    /* producer write-only */
    fo = open(filename, O_TRUNC| O_RDONLY, 0444);            /* consumer read-only */

    mutex = sem_open("MUTEX", 1);
    full = sem_open("FULL", 0);
    empty = sem_open("EMPTY", BUFFER_SIZE);

    item_pro = 0;

    if ((pid = fork()))    /* father process is producer */
    {
        printf("pid %d:\tproducer created....\n", pid);
        fflush(stdout);

        while (item_pro < NR_ITEMS)    /* while not finish produce the item  */
        {
            sem_wait(empty);
            sem_wait(mutex);

            /* if the buffer is full, reset the pointer to the head of file */
            if(!(item_pro % BUFFER_SIZE))
                lseek(fi, 0, 0);

            write(fi, (char *) &item_pro, sizeof(item_pro));        /* produce */
            printf("pid %d:\tproduces item %d\n", pid, item_pro);
            fflush(stdout);
            item_pro++;

            sem_post(mutex);
            sem_post(full);        /* wake up the consumer */
        }
    }
    else    /* child process is consumer */
    {
        i = NR_CONSUMERS;
        while(i--)
        {
            if(!(pid=fork()))    /* new the consumer */
            {
                pid = getpid();
                printf("pid %d:\tconsumer %d created....\n", pid, NR_CONSUMERS-i);
                fflush(stdout);

                while(1)
                {
                    sem_wait(full);
                    sem_wait(mutex);

                    /*when read() finish, return 0, reset the pointer to the head of file */
```

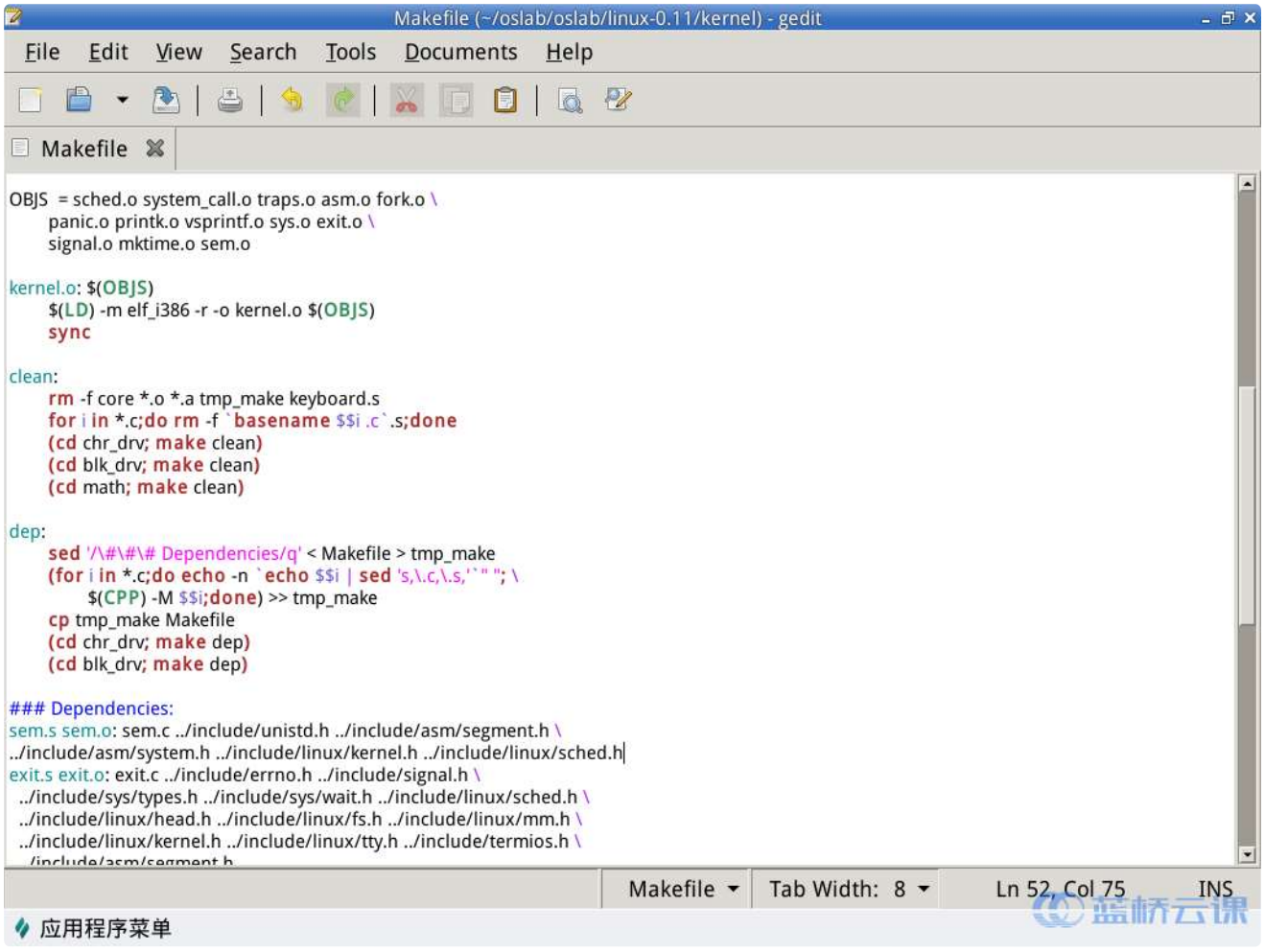
```
                    if(!read(fo, (char *)&item_used, sizeof(item_used)))
                    {
                        lseek(fo, 0, 0);
                        read(fo, (char *)&item_used, sizeof(item_used));
                    }

                    printf("pid %d:\tconsumer %d consumes item %d\n", pid, NR_CONSUMERS-i+1, item_used);
                    fflush(stdout);

                    sem_post(mutex);
                    sem_post(empty);    /* wake up the producer */

                    if(item_used == NR_ITEMS){    /* if all items have been consumed */
                        flag = 0;
                        goto OK;
                    }
                }
            }
        }
    }
OK:
    while(flag);
    sem_unlink("MUTEX");
    sem_unlink("FULL");
    sem_unlink("EMPTY");
    close(fi);
    close(fo);
    return 0;
}
```

修改kernel/Makefile



运行程序

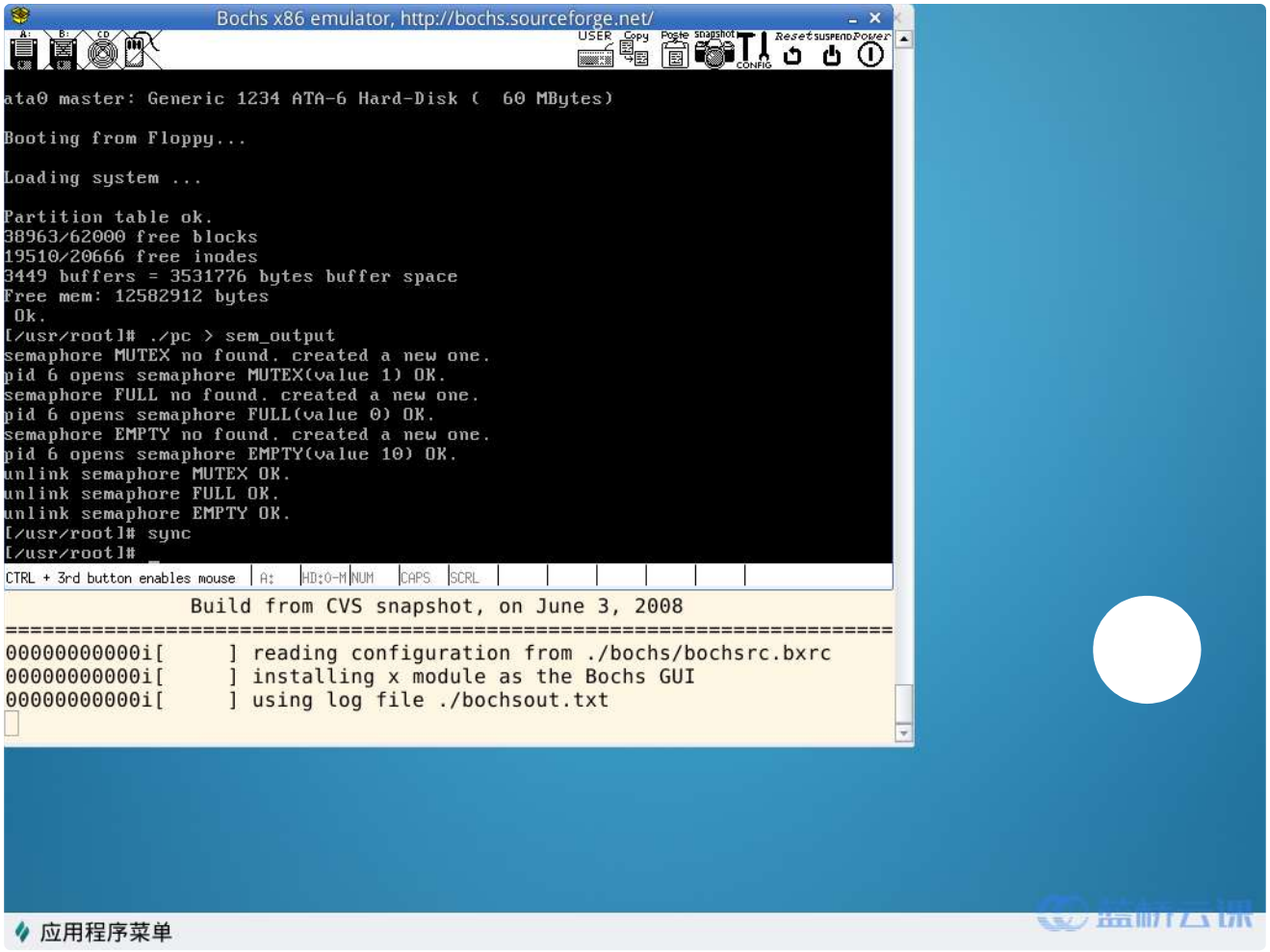
手动替换库文件

在oslab下输入sudo ./mount-hdc，将做过修改的库文件和pc.c拷贝进虚拟镜像之中，执行`sudo umount hdc`取消挂载

编译运行，查看结果

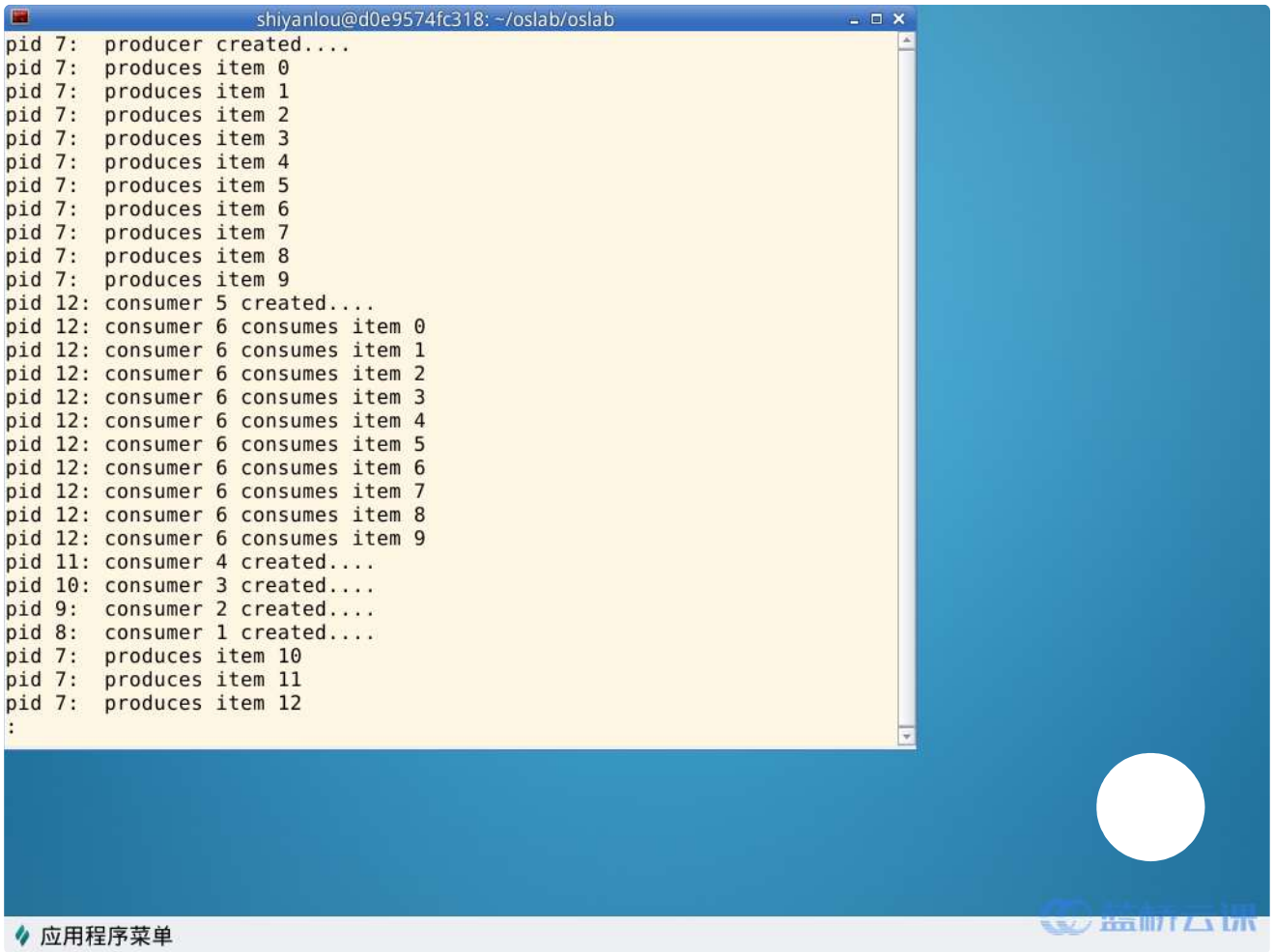
在linux 0.11下执行

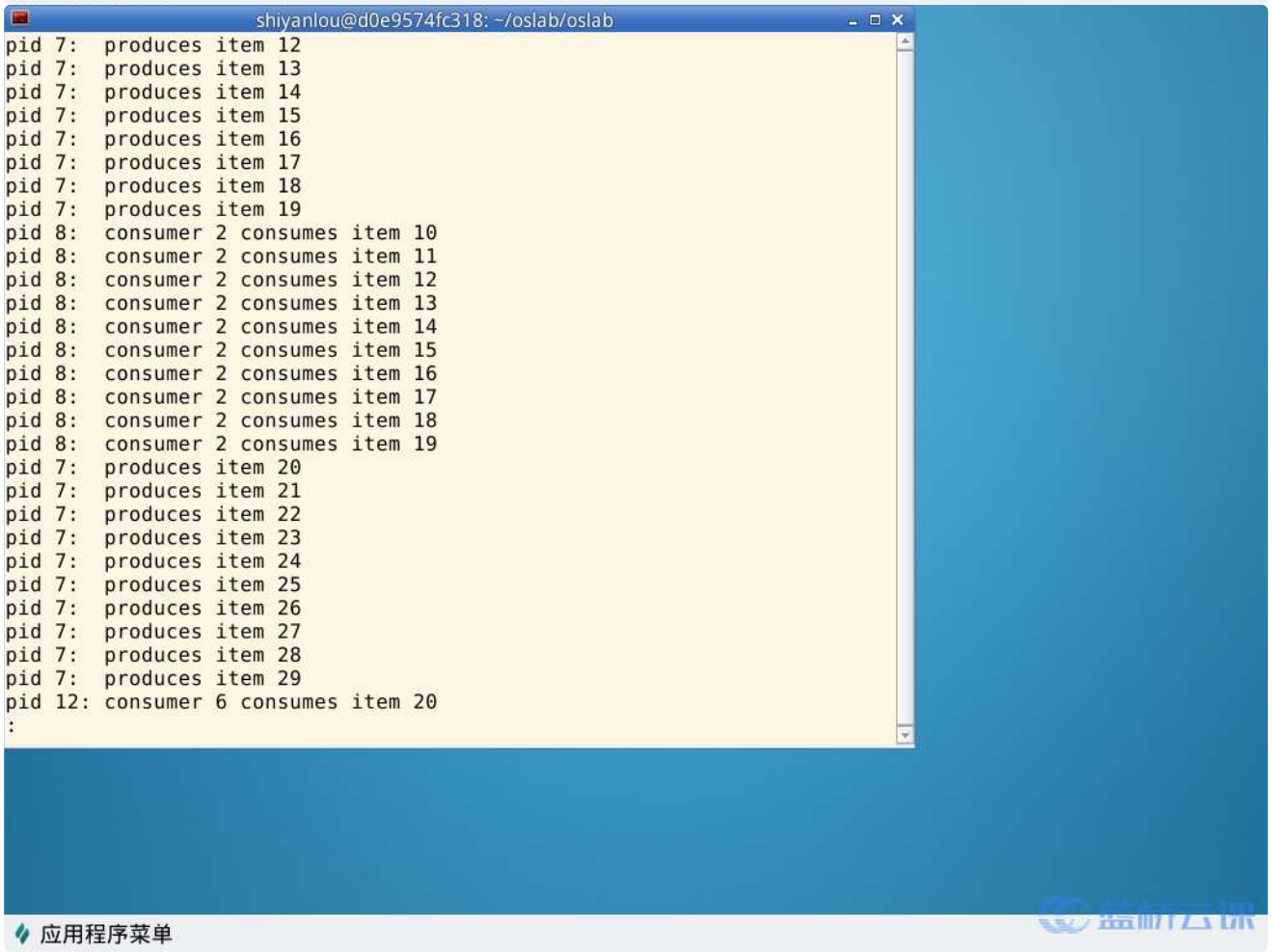
```
gcc -o pc pc.c -Wall
./pc > sem_output
sync
```

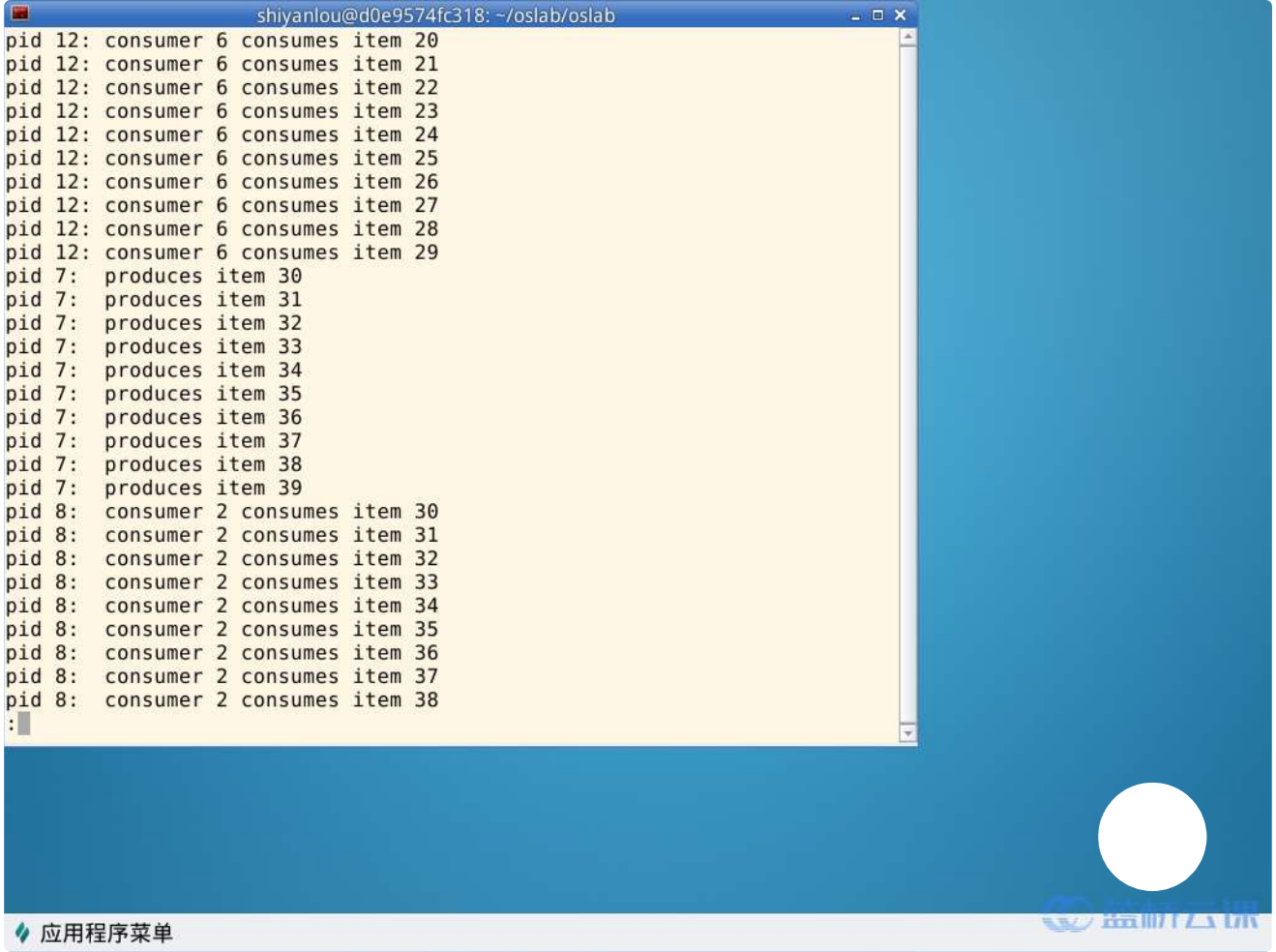


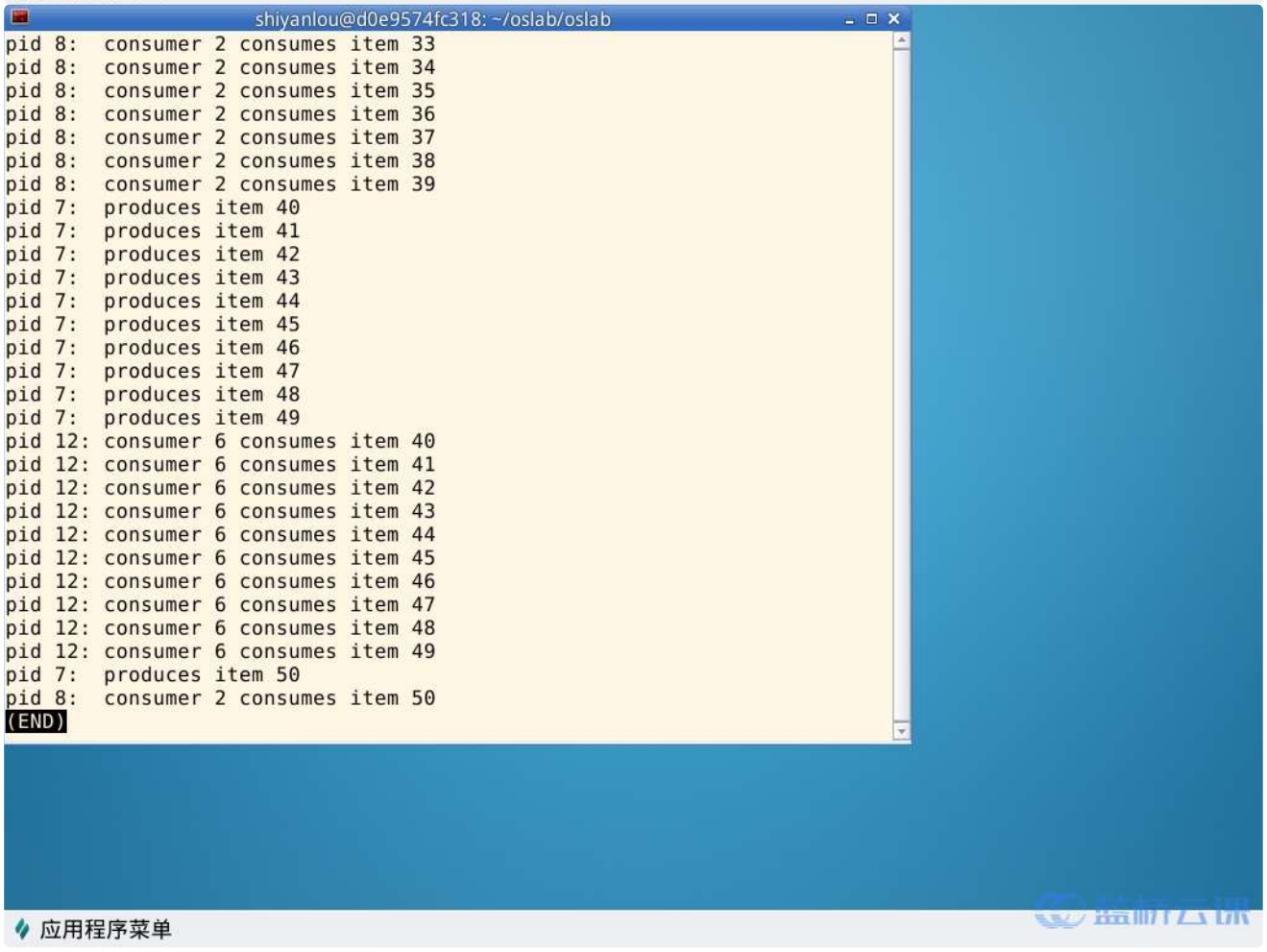
退出前务必执行sync

在Ubuntu下执行`sudo less hdc/usr/root/sem_output`，得到输出









与实验所要求的输出基本一致

实验报告问题

1.在pc.c中去掉所有与信号量有关的代码，再运行程序，执行效果有变化吗？为什么会这样？

执行效果变化很大，打印出的消费序列完全是乱的。因为没有信号量对临界区进行保护，加上进程调度是不确定的，导致生产者和消费者们的动作都无法做到互斥。也就是说,在临界区中完成一个动作之前，它们都有可能被另一个进程抢占，导致在缓冲区中的数据是混乱的。比如，消费者C正准备在缓冲区的起始处取出产品0时，就被生产者P抢占，而P已经完成了一轮的生产，也处于缓冲区的起始处，它往这里写入一个1。如果有轮到C执行。那么C取出的产品就是1，而不是0。

2.实验的设计者在第一次编写生产者——消费者程序的时候，是这么做的：

```
Producer()
{
    P(Mutex);  //互斥信号量
    生产一个产品item;
    P(Empty);  //空闲缓存资源
    将item放到空闲缓存中;
    V(Full);  //产品资源
    V(Mutex);
}

Consumer()
{
    P(Mutex);
    P(Full);
    从缓存区取出一个赋值给item;
    V(Empty);
    消费产品item;
    V(Mutex);
}
```

这样可行吗？如果可行，那么它和标准解法在执行效果上会有什么不同？如果不可行，那么它有什么问题使它不可行？

1. 假设Producer刚生产完一件商品，释放了Mutex，Mutex为1，此时缓存区满了，Empty为0；
2. 然后OS执行调度，若被Producer拿到CPU，它拿到Mutex，使Mutex为0，而Empty为0，Producer让出CPU，等待Consumer执行V(Empty)；
3. 而Consumer拿到CPU后，却要等待Producer执行V(Mutex)；
4. 两者相互持有对方需要的资源，造成死锁。

实验数据