重 庆 大 学

学 生 实 验 报 告

实验课程名称 操作系统

开课实验室 DS1501

学 院 大数据与软件学院 年级 2021 专业班 软件工程 X 班

学 生 姓 名 XXX 学 号 2021XXXX

开 课 时 间 2022 至 2023 学年第 二 学期

成 绩	
教师签名	

大数据与软件学院制

《操作系统》实验报告

开课实验室：DS1501 **2023 年 5 月 20 日**

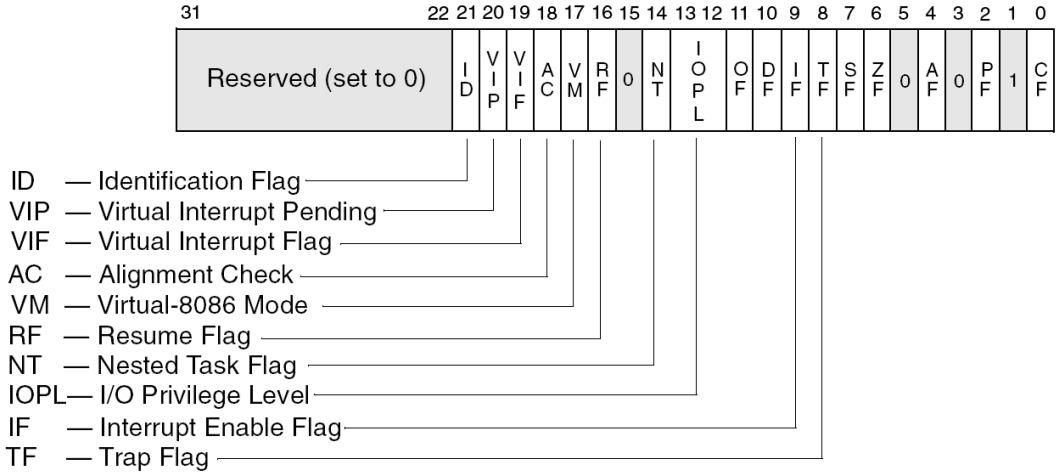
学院	大数据与软件学院	年级、专业、班	2021 级软件工程 X 班	姓名	XXX	成绩	
课程名称	操作系统	实验项目名　称	线程同步	指导教师	XX		
教师评	教师签名： 2023 年　月　日						

- **实验目的**

  • 掌握信号量的实现与应用

基本概念：

- 互斥
  - EPOS运行于单CPU的计算机上
  - 内核可以用开关中断实现互斥
  - 中断的开关由EFLAGS中的IF位决定
  - IF=1，中断打开
  - IF=0，中断关闭
  - 指令
  - sti，IF=1
  - cli，IF=0
  - 一般不直接用sti/cli，而是用
  - save_flags_cli(flags)
  - 保存EFLAGS的值到一个变量flags中，然后IF=0
  - restore_flags(flags)
  - 把变量flags的值恢复到EFLAGS中



- 例子

```
uint32_t flags;
save_flags_cli(flags);
临界区
restore_flags(flags)
```

- 睡眠
  - void sleep_on(struct wait_queue **head)
    - 参数head是睡眠队列的头指针的指针
- 唤醒
  - void wake_up(struct wait_queue **head, int n)
    - 参数n表示要唤醒的线程个数
      - n小于0表示唤醒该队列中的所有线程
- sleep_on和wake_up必须在关中断环境中运行
  - 用save_flags_cli/restore_flags保护
- 一个例子

```
struct wait_queue *wq_kbd = NULL;//!!!非常重要!!!
uint16_t     buf_kbd;
//从键盘读按键信息
int sys_getchar()
{
 uint32_t flags;
 save_flags_cli(flags);

 //睡眠等待用户按键
 sleep_on(&wq_kbd);

 restore_flags(flags);
 return buf_kbd;
}
//键盘中断处理函数
void isr_keyboard(uint32_t irq, struct context *ctx)
{
    buf_kbd = read_data_from_kbd();

    //注意：在ISR中，中断已经被关闭
    //唤醒1个等待用户按键的线程
    wake_up(&wq_kbd, 1);
}
```

二、实验内容

实验内容：

- 实现信号量
  - 编辑文件kernel/sem.c，实现如下四个函数
    - int sys_sem_create(int value)
      - value是信号量的初值
      - 分配内存要用kmalloc，不能用malloc！
      - 成功返回信号量ID，否则返回-1
    - int sys_sem_destroy(int semid)
      - 释放内存要用kfree，不能用free！
      - 成功返回0，否则返回-1
    - int sys_sem_wait(int semid)
      - P操作，要用save_flags_cli/restore_flags和函数sleep_on
      - 成功返回0，否则返回-1
    - int sys_sem_signal(int semid)
      - V操作，要用save_flags_cli/restore_flags和函数wake_up
      - 成功返回0，否则返回-1
  - 把这四个函数做成系统调用，分别是sem_create/destroy/wait/signal
- 生产者/消费者
  - Step1：首先，在图形模式下将屏幕沿垂直方向分成N份，作为N个缓冲区。其次，创建两个线程，其中一个是生产者，负责生成随机数并填到缓冲区中；另一个线程是消费者，负责把缓冲区中的随机数进行排序
    - 生产者生成随机数后，要画到缓冲区
    - 消费者完成排序之后，要清除缓冲区
  - Step2：创建一个控制线程
    - 用键up/down控制生产者的优先级，用键right/left控制消费者的优先级
    - 把控制线程的静态优先级设置到最高，以保证控制效果
    - 在屏幕上用进度条动态显示生产者和消费者的静态优先级

三、使用仪器、材料

Lenovo Legion R9000P2021H, Windows 11

四、实验过程原始记录(数据、图表、计算等)：

实验步骤:

Step1：在“Kernel.h”中，定义信号量结构体 semaphore；

```
struct semaphore//信号量结构体
{
int sem_id;//信号量id
int sem_val;
//信号量的初值
struct wait_queue* list; // 等待队列
    struct semaphore * next;//下一个信号量
};
extern struct semaphore* g_sem_head;//信号量链表的表头
extern struct semaphore*g_sem_select;  //当前信号量
extern int g_sem_id;  //信号量id
```

Step2：在“Kernel.h”和“syscall.h”中，声明 sem_create ,sem_destroy ,sem_wait ,sem_singal 函数；

```
//EX4
int sys_sem_create(int value);
int sys_sem_destroy(int semid);
int sys_sem_wait(int semid);
int sys_sem_signal(int semid);
```

```
//EX4
int sem_create(int value);
int sem_destroy(int semid);
int sem_wait(int semid);
int sem_signal(int semid);
```

Step3：在“syscall-wrapper.S”中，加入汇编语言接口和 C 语言声明 ；

```
WRAPPER(sem_create)
WRAPPER(sem_destroy)
WRAPPER(sem_wait)
WRAPPER(sem_signal)
```

Step4：在“syscall-nr.h”中，定义系统调用的号码;

```
//EX4
#define SYSCALL_sem_create 1002
#define SYSCALL_sem_destroy 1003
#define SYSCALL_sem_wait 1004
#define SYSCALL_sem_signal 1005
```

Step5：在“sem.c”中，实现函数；

(1) add_sem；

```
void add_sem(struct semaphore* sem)  //将新创建的信号量加入到信号量链表中
{
    if (g_sem_head == NULL)    //信号量列表为空
    {
        g_sem_head = sem;
        sem->next = NULL;
        return;
    }
    else          //信号量列表不为空
    {
        struct semaphore* select = g_sem_head;
        while (select->next != NULL)
        {
            select = select->next;   //找到链表尾部信号量
        }
        select->next = sem;        //将新信号量加入至信号量链表
        sem->next = NULL;
        return;
    }
}
```

(2) sem 结构体；

```
struct semaphore* get_sem(int semid) //获取该id对应的信号量指针,成功返回指针
{
    struct semaphore* select = g_sem_head;
    while (select != NULL)
    {
        if (select->sem_id == semid)
            return select;
        else
            select = select->next;
    }
    return NULL;
}
```

(3)sys_sem_create；

```
int sys_sem_create(int value)
{
    struct semaphore* sem;
    sem = (struct semaphore*)kmalloc(sizeof(struct semaphore));
    if (sem == NULL)
        return -1;//信号量创建失败,返回-1
    else
    {
        sem->sem_id = g_sem_id++;       //信号量唯一id
        sem->sem_val = value;          //信号量赋初值
        sem->list = NULL;          //初始等待队列为空
        add_sem(sem);      //将新建信号量加入至信号量链表
        return sem->sem_id;     //创建成功,返回信号量id

    }
}
```

(4) sys_sem_destroy；

```
int sys_sem_destroy(int semid)
{
    if (g_sem_head == NULL)       // 信号量列表为空
        return -1;
    else
    {
        struct  semaphore* select = get_sem(semid); //获取对应信号量
        struct semaphore* prev = get_sem(semid - 1);// 链表前一个结点
        if (select == NULL)//不存在该信号量
            return -1;
        else if (prev == NULL) {
            g_sem_head = select->next;
            kfree(select);
            return 0;
        }
        else
        {
            prev->next = select->next;
            kfree(select);
            return 0;
        }
    }
}
```

(5) sys_sem_wait；

```
//P操作
int sys_sem_wait(int semid)
{
    struct semaphore* select = get_sem(semid); //获取对应信号量
    if (select == NULL)//不存在该信号量
        return -1;
    else
    {
        select->sem_val--;
        if (select->sem_val < 0)
        {
            uint32_t flags;
            save_flags_cli(flags);
            sleep_on(&(select->list));  //加入等待列表
                restore_flags(flags);
        }
            return 0;
    }
}
```

(6) sys_sem_signal；

```
//V操作
int sys_sem_signal(int semid)
{
    struct semaphore* select = get_sem(semid); //获取对应信号量
    if (select == NULL) //不存在该信号量
        return -1;
    else
    {
        select->sem_val++;
        if (select->sem_val <= 0) {
            uint32_t flags;
            save_flags_cli(flags);
            wake_up(&(select->list), 1); //从等待列表中唤醒一个线程
            restore_flags(flags);
        }
        return 0;
    }
}
```

Step6：在“machdep.c”中，加入 case 语句；

```
//EX4
case SYSCALL_sem_create:
{
    int val = *(int*)(ctx->esp + 4);
    ctx->eax = sys_sem_create(val);
}
break;
case SYSCALL_sem_destroy:
{
    int semid = *(int*)(ctx->esp + 4);
    ctx->eax = sys_sem_destroy(semid);
}
break;
case SYSCALL_sem_wait:
{
    int semid = *(int*)(ctx->esp + 4);
    ctx->eax = sys_sem_wait(semid);
}
break;
case SYSCALL_sem_signal:
{
    int semid = *(int*)(ctx->esp + 4);
    ctx->eax = sys_sem_signal(semid);
}
break;
```

Step7：在“main.c”中，定义缓冲区；

```
//EX4
int step=2;
#define buffer_size 10 //缓冲区数量
#define x g_graphic_dev.XResolution/buffer_size
#define y g_graphic_dev.YResolution/step-30
```

Step8：在“main.c”中，声明函数；

```
//EX4
void thread_bubbleSort(void *p);
void thread_insertSort(void *p);
void thread_quickSort(void *p);
void thread_shellSort(void *p);
void thread_bubbleSort_A(void *p);
void thread_bubbleSort_B(void *p);
void thread_control(void *p);
void thread_producer(void *p);
void thread_consumer(void *p);
```

Step9：在“main.c”中，实现生产者线程函数；

```
//EX4生产者线程函数
void thread_producer(void p)
{
//将传入的void指针转换成二维数组指针
int (arr)[buffer_sizey]=(int()[buffer_sizey])p;

//定义变量和参数
int buffer_p=0,i,j;

//对产生随机数的种子进行初始化，以使得每次程序运行时，所产生的随机数序列不同
srand(time(NULL));

while(1){
    //等待空闲缓冲区
    sem_wait(space);
    //获得互斥量
    sem_wait(mutex[buffer_p]);

    //清除缓冲区
    for(j=0;j<y;j++)
        //使用指定颜色清除一行
        line(x*buffer_p,step*j,x*(buffer_p+1),step*j,RGB(0,0,0));

    //产生随机数并绘制线条
    for (i=0;i<y;i++){
        //在二维数组相应位置赋予随机值
        arr[buffer_p][i]=rand()%x;
        //绘制线条
        line(x*buffer_p,step*i,x*buffer_p+arr[buffer_p][i],step*i, RGB(255,120,120));
        //线程休眠一段时间
        nanosleep((const struct timespec[]){{0,5000000L}}, NULL);
    }
    //释放互斥量
    sem_signal(mutex[buffer_p]);

    //释放信号量，表示当前缓冲区已装满
    sem_signal(items);

    //调整缓冲区指针
    buffer_p++;
    if(buffer_p==buffer_size)
        buffer_p=0;
    }
}
```

Step10：在“main.c”中，实现消费者线程函数；

```
//EX4消费者线程函数
void thread_consumer(void p)
{
//将传入的void指针转换成二维数组指针
int (arr)[buffer_sizey]=(int()[buffer_sizey])p;

//定义变量和参数
int buffer_c=0;

while(1)
{
    //等待已填满缓冲区
    sem_wait(items);
    //获得互斥量
    sem_wait(mutex[buffer_c]);

    //插入排序，根据数组值绘制线条
    int i,j,temp,n=buffer_c;
    for (i = 1; i <y; i++){
        temp = arr[n][i];
        for (j = i - 1; j >= 0 && arr[n][j] > temp;j--){
            //使用指定颜色清除一行
            line(x*n,step * (j + 1), x*n+arr[n][j + 1], step * (j + 1), RGB(0, 0, 0));
            //将j+1位置值与j位置值交换，并绘制对应线条
            arr[n][j + 1] = arr[n][j];
            line(x*n,step * (j + 1), x*n+arr[n][j + 1], step * (j + 1), RGB(0,255, 0));
        }
        //将temp值插入到正确位置，并绘制对应线条
        line(x*n, step * (j + 1),x*n+arr[n][j + 1],step * (j + 1), RGB(0, 0, 0));
        arr[n][j + 1] = temp;
        line(x*n, step* (j + 1),x*n+arr[n][j + 1], step * (j + 1), RGB(0, 255, 0));
        //线程休眠一段时间
        nanosleep((const struct timespec[]){{0, 5000000L}}, NULL);
    }

    //释放互斥量
    sem_signal(mutex[buffer_c]);

    //释放信号量，表示当前缓冲区已空
    sem_signal(space);

    //调整缓冲区指针
    buffer_c++;
    if(buffer_c==buffer_size)
        buffer_c=0;
}
}
```

Step11：在“main.c”中，实现冒泡排序线程函数；

```
//EX4冒泡排序线程函数
void thread_bubbleSort(void *p){
    // 定义变量
    int i, j, temp;
    // 将传入的参数p强制转化为整型数组
    int arr = (int )p;
    //y = g_graphic_dev.YResolution/step;
    // 根据数组长度循环画出每个元素对应的线段
    for(i=0; i<y; i++){
    line(0, stepi, arr[i], stepi, RGB(255, 0, 0)); // 画出线段
    }

    //进行冒泡排序
    for(i=0; i<y-1; i++){
        for(j=0; j<y-1-i; j++){
            if(arr[j] > arr[j+1]){ // 比较大小，判断是否需要交换位置
                temp = arr[j];  // 交换位置
                line(0, step*j, arr[j], step*j, RGB(0, 0, 0)); // 将原来的线段删除
                arr[j] = arr[j+ 1]; // 更新数组
                line(0, step*j, arr[j], step*j, RGB(255,0,0));  // 将更新后的数组重新画出
                line(0, step*(j + 1), arr[j + 1],step*(j + 1), RGB(0, 0, 0)); // 将原来的线段删除并将数组重新画出
                arr[j+ 1] = temp; // 更新数组
                line(0,step*(j + 1), arr[j + 1], step*(j + 1), RGB(255, 0, 0));  // 将更新后的数组重新画出
            }
        }
        //线程睡眠1秒
        nanosleep((const struct timespec[]){{0,10000000L}}, NULL);
    }
    //线程结束
    task_exit(0);
}
```

Step12：在“main.c”中，实现控制线程函数；

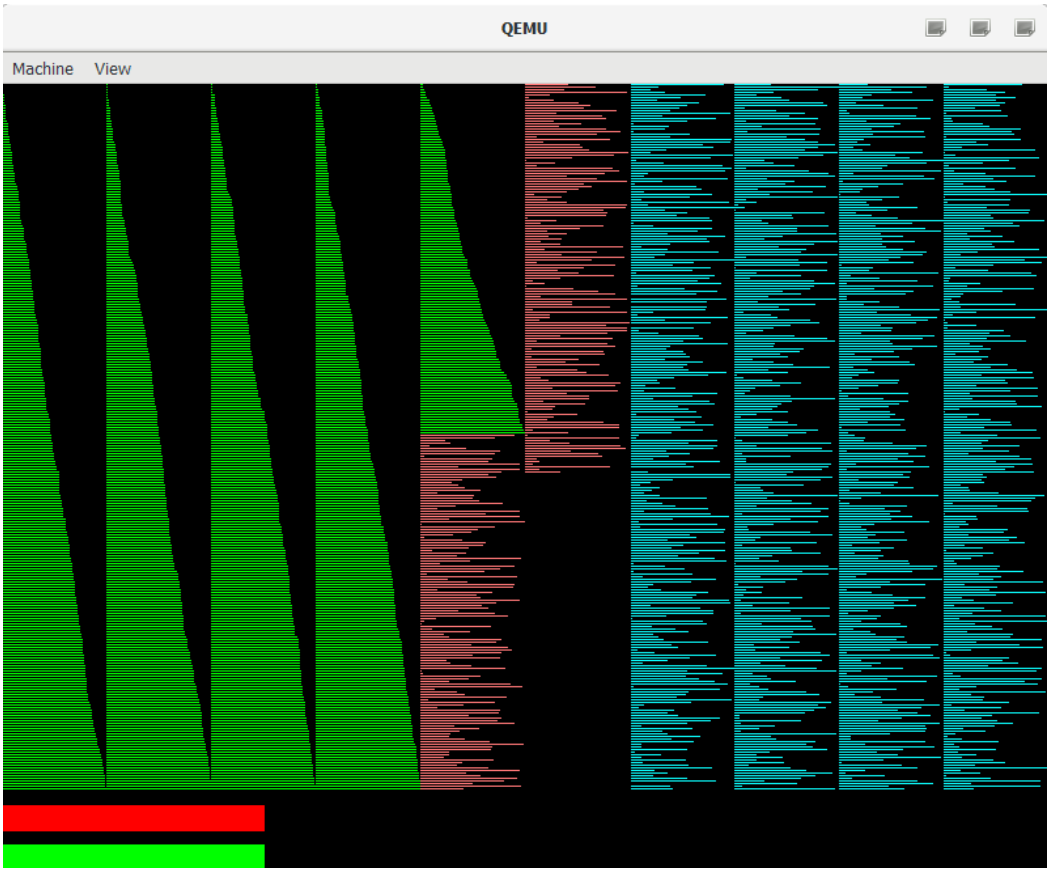
```c
//EX4控制线程函数
void thread_control(void p){
//将传入的void指针转换成Control结构体指针
struct Control ctl= (struct Control)p;
//定义参数和变量
int a = 20;
int b = 50;
int c = 20;
//在屏幕上显示A和B的优先级条形图
for (int i = a; i > 0; i--)  {
    //使用指定颜色在指定位置绘制线条
    line(0, step * (y + 30) -b+i, 10 * x, step * (y + 30) - b + i, RGB(0, 0, 0));
    line(0, step * (y + 30) - b + i, 20 * getpriority(ctl->tid_A), step * (y + 30) - b + i, RGB(255, 0, 0));
}
for (int i = a; i > 0; i--)  {
    //使用指定颜色在指定位置绘制线条
    line(0, step * (y + 30) -c+i, 10 * x, step * (y + 30) - c + i, RGB(0, 0, 0));
    line(0, step * (y + 30) - c + i, 20 * getpriority(ctl->tid_B), step * (y + 30) - c + i, RGB(0, 255, 0));
}
while(1){//循环键入
    int keyboard= getchar();
    switch(keyboard){
        case 0x4d00:{//RIGHT
            //增加B的优先级
            for (int i = a; i > 0; i--)
            {
                line(0, step * (y + 30) - c + i, 10 * x, step * (y + 30) - c + i, RGB(0, 0, 0));
                setpriority(ctl->tid_B, getpriority(ctl->tid_B) - 2);
                line(0, step * (y + 30) - c + i, 20 * getpriority(ctl->tid_B), step * (y + 30) - c + i, RGB(0, 255, 0));
            }
        }break;
        case 0x4b00:{//LEFT
            //降低B的优先级
            for (int i = a; i > 0; i--)
            {
                line(0, step * (y + 30) - c + i, 10 * x, step * (y + 30) - c + i, RGB(0, 0, 0));
                setpriority(ctl->tid_B, getpriority(ctl->tid_B) + 2);
                line(0, step * (y + 30) - c + i, 20 * getpriority(ctl->tid_B), step * (y + 30) - c + i, RGB(0, 255, 0));
            }
        }break;
        case 0x4800:{//UP
            //增加A的优先级
            for (int i = a; i > 0; i--)
            {
                line(0, step * (y + 30) - b + i, 10 * x, step * (y + 30) - b + i, RGB(0, 0, 0));
                setpriority(ctl->tid_A, getpriority(ctl->tid_A) - 2);
                line(0, step * (y + 30) - b + i, 20 * getpriority(ctl->tid_A), step * (y + 30) - b + i, RGB(255, 0, 0));
            }
        }break;
        case 0x5000:{//DOWN
            //降低A的优先级
            for (int i = a; i > 0; i--)
            {
                line(0, step * (y + 30) - b + i, 10 * x, step * (y + 30) - b + i, RGB(0, 0, 0));
                setpriority(ctl->tid_A, getpriority(ctl->tid_A) + 2);
                line(0, step * (y + 30) - b + i, 20 * getpriority(ctl->tid_A), step * (y + 30) - b + i, RGB(255, 0, 0));
            }
        }break;

        default:break;
    }
}
task_exit(0);
}
```

Step13：在“main.c”中，实现程序入口函数；

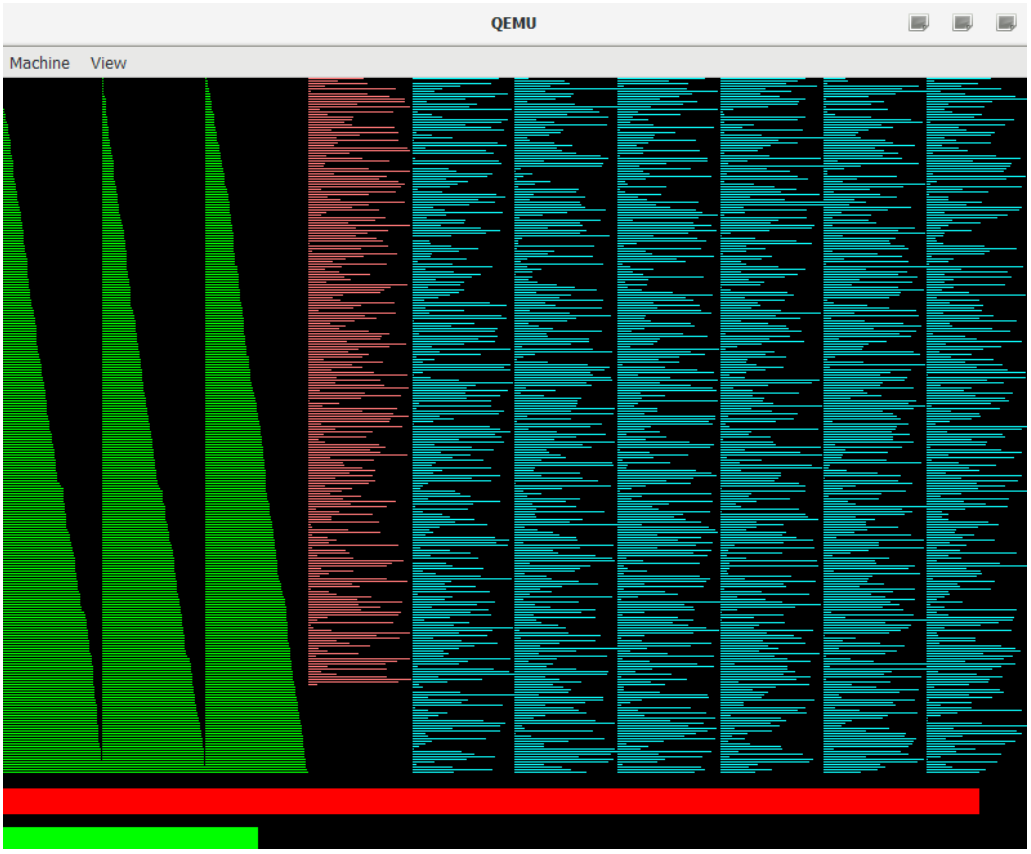
```
// EX4程序入口函数
void main(void *pv)
{
    // 打印任务id和参数
    printf("task #%d: I'm the first user task(pv=0x%08x)!\r\n", task_getid(), pv);

    //TODO: Your code goes here
    // 进入图形页面
    init_graphic(0x143);
    int i, j;
    // 创建信号量
    for(i=0; i<buffer_size; i++){
        mutex[i] = sem_create(1);
    }
    items = sem_create(0);
    space = sem_create(buffer_size);

    // 申请三个线程的栈空间
    unsigned int stack_size = 1024*1024;
    unsigned char* stack_producer = (unsigned char *)malloc(stack_size);
    unsigned char* stack_consumer = (unsigned char *)malloc(stack_size);
    unsigned char* stack_control = (unsigned char *)malloc(stack_size);

    // 申请数组空间并初始化
    int(*arr)[buffer_size*y] = (int(*)[buffer_size*y])malloc(buffer_size*y*sizeof(int));
    srand(time(NULL));
    for (i=0; i<buffer_size; i++){
        for(j=0; j<y; j++){
            arr[i][j] = rand()%x;// 随机生成数值进行初始化
            line(i*x, j*step, x*i+arr[i][j], j*step, RGB(10, 255,250));// 画出线段
        }
    }
    // 创建三个线程：生产者、消费者和控制线程
    int tid_producer = task_create(stack_producer+stack_size, &thread_producer, (void*)arr);
    int tid_consumer = task_create(stack_consumer+stack_size, &thread_consumer, (void*)arr);
    struct Control *ctl = (struct Control*)malloc(sizeof(struct Control));
    ctl->tid_B = tid_consumer;
    ctl->tid_A = tid_producer;
    int tid_control = task_create(stack_control+stack_size, &thread_control,(void*)ctl);
    setpriority(tid_producer,10);
    setpriority(tid_consumer,10);
    setpriority(tid_control,0);

    while(1)
        ;
    // 销毁信号量
    sem_destroy(space);
    sem_destroy(items);
    for(i=0;i<buffer_size;i++){
        sem_destroy(mutex[i]);
    }
    exit_graphic();
    task_exit(0);
}
```

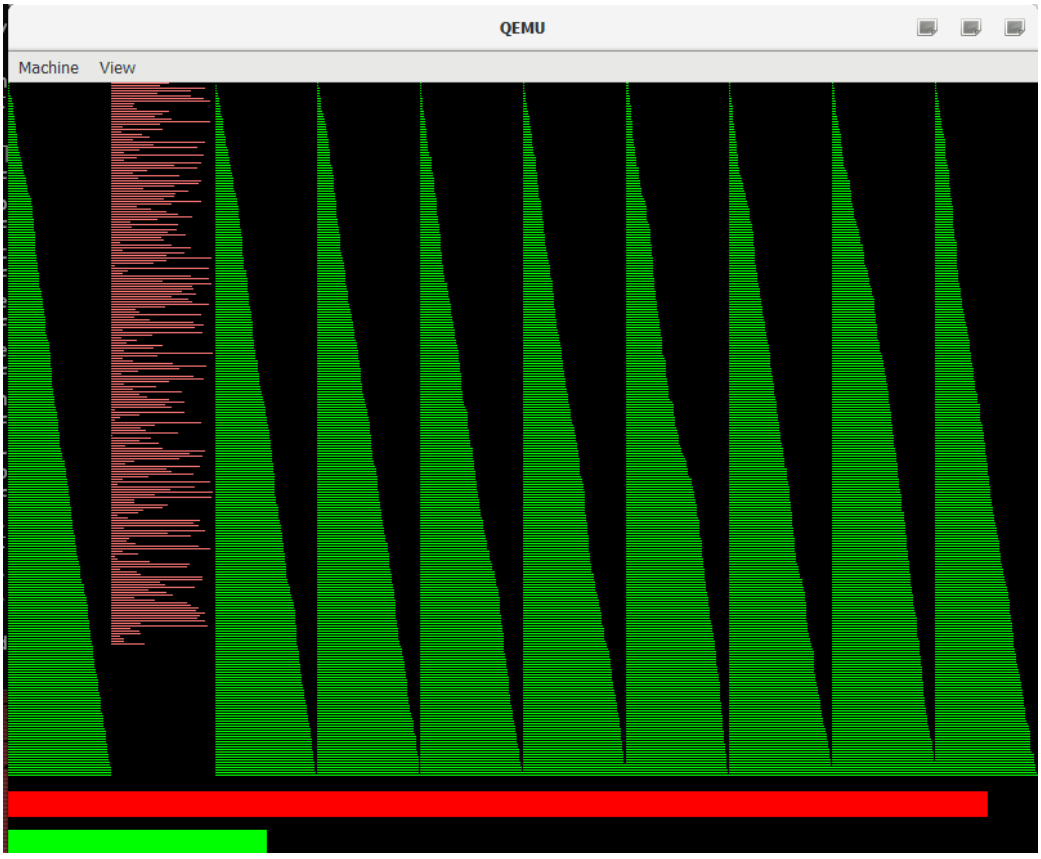
运行结果：

(1)生产者消费者优先级相同；

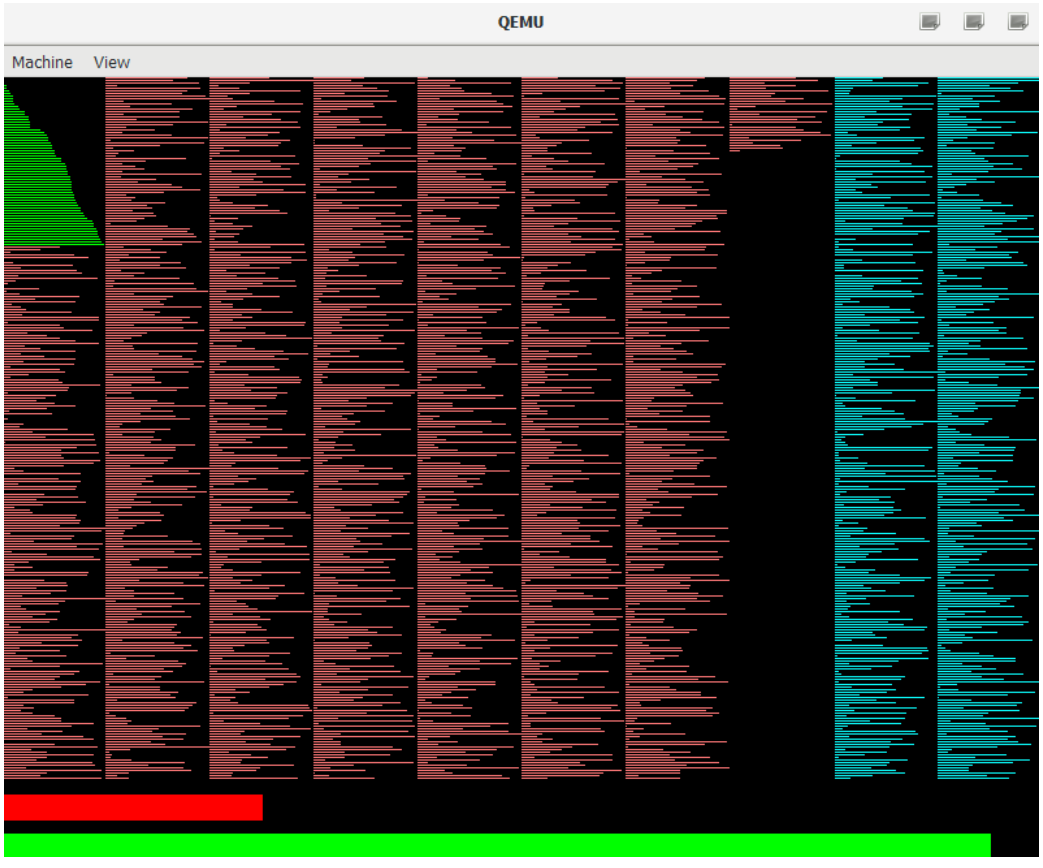


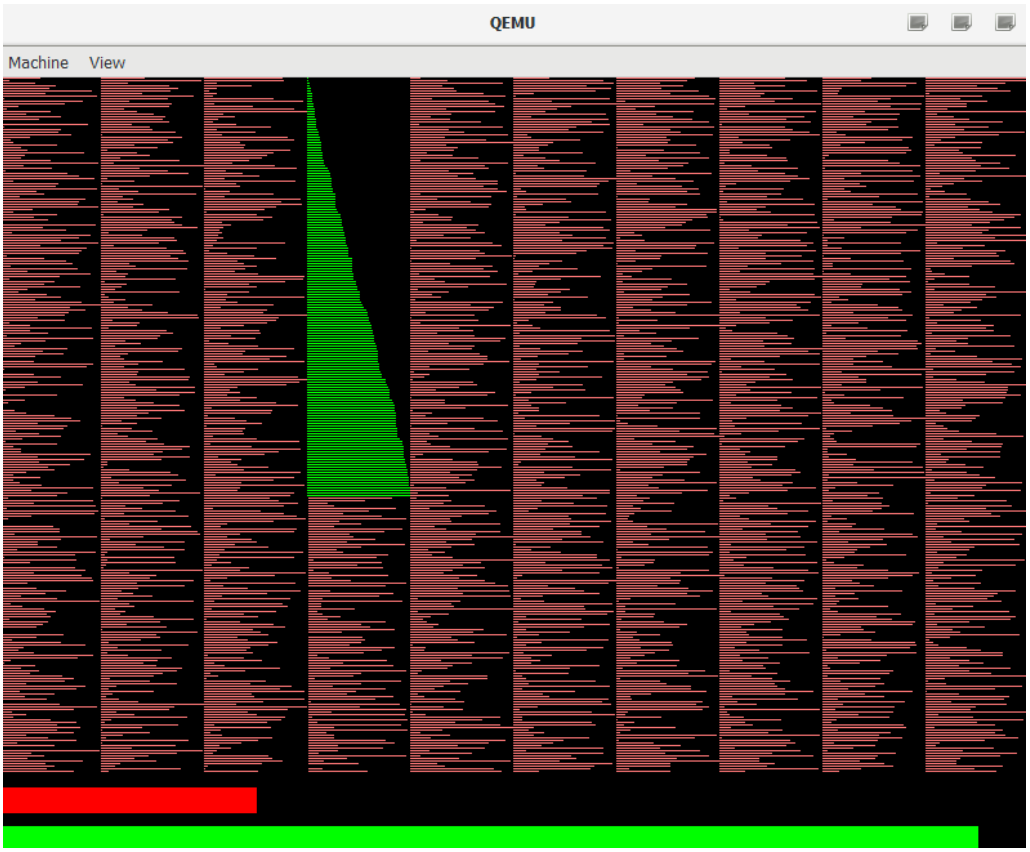
(2)生产者优先级低于消费者



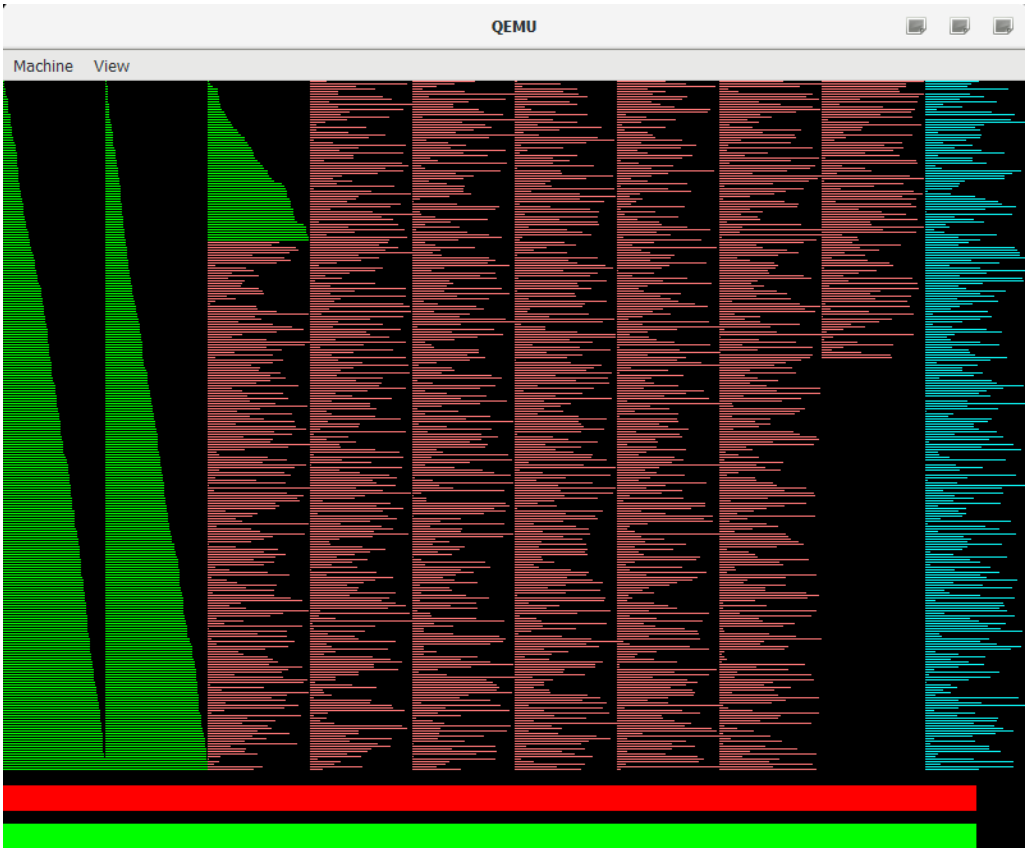


(3)生产者优先级高于消费者；





(4)生产者优先级再度等于消费者；



核心代码：

```
//EX4
struct semaphore//信号量结构体
{
int sem_id;//信号量id
int sem_val;
//信号量的初值
struct wait_queue* list; // 等待队列
    struct semaphore * next;//下一个信号量
};
extern struct semaphore* g_sem_head;//信号量链表的表头
extern struct semaphore*g_sem_select;  //当前信号量
extern int g_sem_id;  //信号量id

//EX4
int sys_sem_create(int value);
int sys_sem_destroy(int semid);
int sys_sem_wait(int semid);
int sys_sem_signal(int semid);
```

```
//EX4
int sem_create(int value);
int sem_destroy(int semid);
int sem_wait(int semid);
int sem_signal(int semid);
```

```
#include "syscall-nr.h"

#define WRAPPER(name) \
  .globl _ ## name; \
_ ## name: \
    movl $SYSCALL_ ## name, %eax; \
    int $0x82; \
    ret

WRAPPER(task_exit)
WRAPPER(task_create)
WRAPPER(task_getid)
WRAPPER(task_yield)
WRAPPER(task_wait)
WRAPPER(reboot)
WRAPPER(mmap)
WRAPPER(munmap)
WRAPPER(sleep)
WRAPPER(nanosleep)
WRAPPER(beep)
WRAPPER(vm86)
WRAPPER(putchar)
WRAPPER(getchar)
WRAPPER(recv)
WRAPPER(send)
WRAPPER(ioctl)


WRAPPER(sem_create)
WRAPPER(sem_destroy)
WRAPPER(sem_wait)
WRAPPER(sem_signal)
```

```c
#ifndef _SYSCALLNR_H
#define _SYSCALLNR_H

#define SYSCALL_task_exit     1
#define SYSCALL_task_create   2
#define SYSCALL_task_getid    3
#define SYSCALL_task_yield    4
#define SYSCALL_task_wait     5
#define SYSCALL_reboot        6
#define SYSCALL_mmap          7
#define SYSCALL_munmap        8
#define SYSCALL_sleep         9
#define SYSCALL_nanosleep     10

#define SYSCALL_beep          181
#define SYSCALL_vm86          182
#define SYSCALL_recv          183
#define SYSCALL_send          184
#define SYSCALL_ioctl         185

#define SYSCALL_putchar       1000
#define SYSCALL_getchar       1001

//EX4
#define SYSCALL_sem_create 1002
#define SYSCALL_sem_destroy 1003
#define SYSCALL_sem_wait 1004
#define SYSCALL_sem_signal 1005


#endif /*_SYSCALLNR_H*/
```

```c
//EX4
case SYSCALL_sem_create:
{
    int val = *(int*)(ctx->esp + 4);
    ctx->eax = sys_sem_create(val);
}
break;
case SYSCALL_sem_destroy:
{
    int semid = *(int*)(ctx->esp + 4);
    ctx->eax = sys_sem_destroy(semid);
}
break;
case SYSCALL_sem_wait:
{
    int semid = *(int*)(ctx->esp + 4);
    ctx->eax = sys_sem_wait(semid);
}
break;
case SYSCALL_sem_signal:
{
    int semid = *(int*)(ctx->esp + 4);
    ctx->eax = sys_sem_signal(semid);
}
break;

default:
    printk("syscall #%d not implemented.\r\n", ctx->eax);
    ctx->eax = -ctx->eax;
    break;
}
```

```c
/**
 * vim: filetype=c:fenc=utf-8:ts=4:et:sw=4:sts=4
 */
#include <stddef.h>
#include "kernel.h"

void add_sem(struct semaphore* sem)  //将新创建的信号量加入到信号量链表中
{
    if (g_sem_head == NULL)    //信号量列表为空
    {
        g_sem_head = sem;
        sem->next = NULL;
        return;
    }
    else          //信号量列表不为空
    {
        struct semaphore* select = g_sem_head;
        while (select->next != NULL)
        {
            select = select->next;   //找到链表尾部信号量
        }
        select->next = sem;        //将新信号量加入至信号量链表
        sem->next = NULL;
        return;
    }
}

struct semaphore* get_sem(int semid) //获取该id对应的信号量指针,成功返回指针
{
    struct semaphore* select = g_sem_head;
    while (select != NULL)
    {
        if (select->sem_id == semid)
            return select;
        else
            select = select->next;
    }
    return NULL;
}


int sys_sem_create(int value)
{
    struct semaphore* sem;
    sem = (struct semaphore*)kmalloc(sizeof(struct semaphore));
    if (sem == NULL)
        return -1;//信号量创建失败,返回-1
    else
    {
        sem->sem_id = g_sem_id++;       //信号量唯一id
        sem->sem_val = value;          //信号量赋初值
        sem->list = NULL;           //初始等待队列为空
        add_sem(sem);      //将新建信号量加入至信号量链表
        return sem->sem_id;     //创建成功,返回信号量id

    }
}
```

```c
int sys_sem_destroy(int semid)
{
    if (g_sem_head == NULL)       // 信号量列表为空
        return -1;
    else
    {
        struct  semaphore* select = get_sem(semid); //获取对应信号量
        struct semaphore* prev = get_sem(semid - 1);// 链表前一个结点
        if (select == NULL)//不存在该信号量
            return -1;
        else if (prev == NULL) {
            g_sem_head = select->next;
            kfree(select);
            return 0;
        }
        else
        {
            prev->next = select->next;
            kfree(select);
            return 0;
        }
    }
}

//P操作
int sys_sem_wait(int semid)
{
    struct semaphore* select = get_sem(semid); //获取对应信号量
    if (select == NULL)//不存在该信号量
        return -1;
    else
    {
        select->sem_val--;
        if (select->sem_val < 0)
        {
            uint32_t flags;
            save_flags_cli(flags);
            sleep_on(&(select->list));  //加入等待列表
                restore_flags(flags);
        }
            return 0;
    }
}

//V操作
int sys_sem_signal(int semid)
{
    struct semaphore* select = get_sem(semid); //获取对应信号量
    if (select == NULL) //不存在该信号量
        return -1;
    else
   {
        select->sem_val++;
        if (select->sem_val <= 0) {
            uint32_t flags;
            save_flags_cli(flags);
            wake_up(&(select->list), 1); //从等待列表中唤醒一个线程
            restore_flags(flags);
        }
        return 0;
    }
}
```

```c
/*
 * vim: filetype=c:fenc=utf-8:ts=4:et:sw=4:sts=4
 */
#include <inttypes.h>
#include <stddef.h>
#include <math.h>
#include <stdio.h>
#include <sys/mman.h>
#include <syscall.h>
#include <netinet/in.h>
#include <stdlib.h>
#include "graphics.h"

//EX4
int step=2;
#define buffer_size 10 //缓冲区数量
#define x g_graphic_dev.XResolution/buffer_size
#define y g_graphic_dev.YResolution/step-30

extern void *tlsf_create_with_pool(void* mem, size_t bytes);
extern void *g_heap;

//EX4
void thread_bubbleSort(void *p);
void thread_insertSort(void *p);
void thread_quickSort(void *p);
void thread_shellSort(void *p);
void thread_bubbleSort_A(void *p);
void thread_bubbleSort_B(void *p);
void thread_control(void *p);
void thread_producer(void *p);
void thread_consumer(void *p);

/**
 * GCC insists on __main
 *    http://gcc.gnu.org/onlinedocs/gccint/Collect2.html
 */
void __main()
{
    size_t heap_size = 32*1024*1024;
    void  *heap_base = mmap(NULL, heap_size, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, -1, 0);
    g_heap = tlsf_create_with_pool(heap_base, heap_size);
}

/**
 * 第一个运行在用户模式的线程所执行的函数
 */

//多参数结构体
struct Control{
    int tid_A,tid_B;
};
int mutex[buffer_size];
int items;
int space;
```

```c
// EX4程序入口函数
void main(void *pv)
{
    // 打印任务id和参数
    printf("task #%d: I'm the first user task(pv=0x%08x)!\r\n", task_getid(), pv);

    //TODO: Your code goes here
    // 进入图形页面
    init_graphic(0x143);
    int i, j;
    // 创建信号量
    for(i=0; i<buffer_size; i++){
        mutex[i] = sem_create(1);
    }
    items = sem_create(0);
    space = sem_create(buffer_size);

    // 申请三个线程的栈空间
    unsigned int stack_size = 1024*1024;
    unsigned char* stack_producer = (unsigned char *)malloc(stack_size);
    unsigned char* stack_consumer = (unsigned char *)malloc(stack_size);
    unsigned char* stack_control = (unsigned char *)malloc(stack_size);

    // 申请数组空间并初始化
    int(*arr)[buffer_size*y] = (int(*)[buffer_size*y])malloc(buffer_size*y*sizeof(int));
    srand(time(NULL));
    for (i=0; i<buffer_size; i++){
        for(j=0; j<y; j++){
            arr[i][j] = rand()%x;// 随机生成数值进行初始化
            line(i*x, j*step, x*i+arr[i][j], j*step, RGB(10, 255,250));// 画出线段
        }
    }
    // 创建三个线程：生产者、消费者和控制线程
    int tid_producer = task_create(stack_producer+stack_size, &thread_producer, (void*)arr);
    int tid_consumer = task_create(stack_consumer+stack_size, &thread_consumer, (void*)arr);
    struct Control *ctl = (struct Control*)malloc(sizeof(struct Control));
    ctl->tid_B = tid_consumer;
    ctl->tid_A = tid_producer;
    int tid_control = task_create(stack_control+stack_size, &thread_control,(void*)ctl);
    setpriority(tid_producer,10);
    setpriority(tid_consumer,10);
    setpriority(tid_control,0);

    while(1)
        ;
    // 销毁信号量
    sem_destroy(space);
    sem_destroy(items);
    for(i=0;i<buffer_size;i++){
        sem_destroy(mutex[i]);
    }
    exit_graphic();
    task_exit(0);
}
```

```c
//EX4生产者线程函数
void thread_producer(void p)
{
//将传入的void指针转换成二维数组指针
int (arr)[buffer_sizey]=(int()[buffer_sizey])p;

//定义变量和参数
int buffer_p=0,i,j;

//对产生随机数的种子进行初始化，以使得每次程序运行时，所产生的随机数序列不同
srand(time(NULL));

while(1){
    //等待空闲缓冲区
    sem_wait(space);
    //获得互斥量
    sem_wait(mutex[buffer_p]);

    //清除缓冲区
    for(j=0;j<y;j++)
        //使用指定颜色清除一行
        line(x*buffer_p,step*j,x*(buffer_p+1),step*j,RGB(0,0,0));

    //产生随机数并绘制线条
    for (i=0;i<y;i++){
        //在二维数组相应位置赋予随机值
        arr[buffer_p][i]=rand()%x;
        //绘制线条
        line(x*buffer_p,step*i,x*buffer_p+arr[buffer_p][i],step*i, RGB(255,120,120));
        //线程休眠一段时间
        nanosleep((const struct timespec[]){{0,5000000L}}, NULL);
    }
    //释放互斥量
    sem_signal(mutex[buffer_p]);

    //释放信号量，表示当前缓冲区已装满
    sem_signal(items);

    //调整缓冲区指针
    buffer_p++;
    if(buffer_p==buffer_size)
        buffer_p=0;
    }
}
```

```c
//EX4消费者线程函数
void thread_consumer(void p)
{
//将传入的void指针转换成二维数组指针
int (arr)[buffer_sizey]=(int()[buffer_sizey])p;

//定义变量和参数
int buffer_c=0;

while(1)
{
    //等待已填满缓冲区
    sem_wait(items);
    //获得互斥量
    sem_wait(mutex[buffer_c]);

    //插入排序，根据数组值绘制线条
    int i,j,temp,n=buffer_c;
    for (i = 1; i <y; i++){
        temp = arr[n][i];
        for (j = i - 1; j >= 0 && arr[n][j] > temp;j--){
            //使用指定颜色清除一行
            line(x*n,step * (j + 1), x*n+arr[n][j + 1], step * (j + 1), RGB(0, 0, 0));
            //将j+1位置值与j位置值交换，并绘制对应线条
            arr[n][j + 1] = arr[n][j];
            line(x*n,step * (j + 1), x*n+arr[n][j + 1], step * (j + 1), RGB(0,255, 0));
        }
        //将temp值插入到正确位置，并绘制对应线条
        line(x*n, step * (j + 1),x*n+arr[n][j + 1],step * (j + 1), RGB(0, 0, 0));
        arr[n][j + 1] = temp;
        line(x*n, step* (j + 1),x*n+arr[n][j + 1], step * (j + 1), RGB(0, 255, 0));
        //线程休眠一段时间
        nanosleep((const struct timespec[]){{0, 5000000L}}, NULL);
    }

    //释放互斥量
    sem_signal(mutex[buffer_c]);

    //释放信号量，表示当前缓冲区已空
    sem_signal(space);

    //调整缓冲区指针
    buffer_c++;
    if(buffer_c==buffer_size)
        buffer_c=0;
}
}
```

```c
//EX4冒泡排序线程函数
void thread_bubbleSort(void *p){
    // 定义变量
    int i, j, temp;
    // 将传入的参数p强制转化为整型数组
    int arr = (int )p;
    //y = g_graphic_dev.YResolution/step;
    // 根据数组长度循环画出每个元素对应的线段
    for(i=0; i<y; i++){
    line(0, stepi, arr[i], stepi, RGB(255, 0, 0)); // 画出线段
    }

    //进行冒泡排序
    for(i=0; i<y-1; i++){
        for(j=0; j<y-1-i; j++){
            if(arr[j] > arr[j+1]){ // 比较大小，判断是否需要交换位置
                temp = arr[j];  // 交换位置
                line(0, step*j, arr[j], step*j, RGB(0, 0, 0)); // 将原来的线段删除
                arr[j] = arr[j+ 1]; // 更新数组
                line(0, step*j, arr[j], step*j, RGB(255,0,0));  // 将更新后的数组重新画出
                line(0, step*(j + 1), arr[j + 1],step*(j + 1), RGB(0, 0, 0)); // 将原来的线段删除并将数组重新画出
                arr[j+ 1] = temp; // 更新数组
                line(0,step*(j + 1), arr[j + 1], step*(j + 1), RGB(255, 0, 0));  // 将更新后的数组重新画出
            }
        }
        //线程睡眠1秒
        nanosleep((const struct timespec[]){{0,10000000L}}, NULL);
    }
    //线程结束
    task_exit(0);
}
```

```c
//EX4控制线程函数
void thread_control(void p){
//将传入的void指针转换成Control结构体指针
struct Control ctl= (struct Control)p;
//定义参数和变量
int a = 20;
int b = 50;
int c = 20;
//在屏幕上显示A和B的优先级条形图
for (int i = a; i > 0; i--)  {
    //使用指定颜色在指定位置绘制线条
    line(0, step * (y + 30) -b+i, 10 * x, step * (y + 30) - b + i, RGB(0, 0, 0));
    line(0, step * (y + 30) - b + i, 20 * getpriority(ctl->tid_A), step * (y + 30) - b + i, RGB(255, 0, 0));
}
for (int i = a; i > 0; i--)  {
    //使用指定颜色在指定位置绘制线条
    line(0, step * (y + 30) -c+i, 10 * x, step * (y + 30) - c + i, RGB(0, 0, 0));
    line(0, step * (y + 30) - c + i, 20 * getpriority(ctl->tid_B), step * (y + 30) - c + i, RGB(0, 255, 0));
}
while(1){//循环键入
    int keyboard= getchar();
    switch(keyboard){
        case 0x4d00:{//RIGHT
            //增加B的优先级
            for (int i = a; i > 0; i--)
            {
                line(0, step * (y + 30) - c + i, 10 * x, step * (y + 30) - c + i, RGB(0, 0, 0));
                setpriority(ctl->tid_B, getpriority(ctl->tid_B) - 2);
                line(0, step * (y + 30) - c + i, 20 * getpriority(ctl->tid_B), step * (y + 30) - c + i, RGB(0, 255, 0));
            }
        }break;
        case 0x4b00:{//LEFT
            //降低B的优先级
            for (int i = a; i > 0; i--)
            {
                line(0, step * (y + 30) - c + i, 10 * x, step * (y + 30) - c + i, RGB(0, 0, 0));
                setpriority(ctl->tid_B, getpriority(ctl->tid_B) + 2);
                line(0, step * (y + 30) - c + i, 20 * getpriority(ctl->tid_B), step * (y + 30) - c + i, RGB(0, 255, 0));
            }
        }break;
        case 0x4800:{//UP
            //增加A的优先级
            for (int i = a; i > 0; i--)
            {
                line(0, step * (y + 30) - b + i, 10 * x, step * (y + 30) - b + i, RGB(0, 0, 0));
                setpriority(ctl->tid_A, getpriority(ctl->tid_A) - 2);
                line(0, step * (y + 30) - b + i, 20 * getpriority(ctl->tid_A), step * (y + 30) - b + i, RGB(255, 0, 0));
            }
        }break;
        case 0x5000:{//DOWN
            //降低A的优先级
            for (int i = a; i > 0; i--)
            {
                line(0, step * (y + 30) - b + i, 10 * x, step * (y + 30) - b + i, RGB(0, 0, 0));
                setpriority(ctl->tid_A, getpriority(ctl->tid_A) + 2);
                line(0, step * (y + 30) - b + i, 20 * getpriority(ctl->tid_A), step * (y + 30) - b + i, RGB(255, 0, 0));
            }
        }break;

        default:break;
    }
}
task_exit(0);
}
```

总结：

在本次实验中，我学到了如何在内核中实现信号量以及相关的 P/V 操作。这包括如何使用 kmalloc 和 kfree 来分配和释放内存，如何使用 save_flags_cli/restore_flags 和 sleep_on/wake_up 函数来实现 P/V 操作，以及如何将这四个函数变成系统调用等。

此外，我还学到了如何在图形模式下分割屏幕以用作缓冲区，对如何创建多个线程并控制它们的优先级这一方面有了更深刻的理解。同时，我也学习了如何在屏幕上显示进度条来监视线程的优先级和运行状态，并且掌握了如何使用键盘输入来控制线程的优先级的相关知识。

综上所述，本次实验使我更加熟练地掌握了内核编程和多线程编程的基本知识，并且对操作系统的运行原理和实现有了更深入的理解。