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 DS1502

学 院 大数据与软件学院 年级 22 专业班 软件工程 2 班

学 生 姓 名 潘铷葳 学 号 20221982

开 课 时 间 2023 至 2024 学年第 二 学期

总 成 绩	
教师签名	

软件学院制

《操作系统》实验报告

开课实验室：DS1502 **2024 年 月 日**

学院	大数据与软件学院	年级、专业、班	2022 级软件工程 2 班	姓名	潘铷葳	成绩	
课程名称	操作系统	实验项目名 称		指导教师	操作系统		
教师评语	教师签名： 年 月 日						

一、实验目的

了解线程同步的概念：通过实验，学习什么是线程同步，为什么需要线程同步以及线程同步的作用和原理。

掌握线程同步的方法：通过实验，了解常用的线程同步方法，包括互斥锁、信号量、条件变量等，并学会如何在程序中使用这些方法来保证线程的安全性和正确性。

理解线程同步的应用场景：通过实验，掌握线程同步在实际编程中的应用场景，如生产者-消费者问题、读者-写者问题等，并学会如何利用线程同步机制解决这些问题。

二、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、实验过程原始记录(数据、图表、计算等)

Step1：定义信号量结构体

（1） 在 kernel/kernel.h 中定义信号量结构体

```
struct Semaphore{
int value;//信号量初值
struct Semaphore *next;
int semid; //信号量id
struct wait_queue *wq; //
};
```

Step2：增加系统调用函数

①int sys_sem_create(int value)，value 是信号量的初值，成功返回信号量 ID，否则返回-1。

②int sys_sem_destroy(int semid)，成功返回 0，否则返回-1。

③int sys_sem_wait(int semid)，P 操作，成功返回 0，否则返回-1。

④ int sys_sem_signal(int semid)，V 操作，成功返回 0，否则返回-1

（2）在 kernel/kernel.h 中声明，在 sem.c 中实现。

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

int sys_sem_create(int value)：这个函数用于创建一个新的信号量。它首先分配一个新的 Semaphore 结构体，并初始化其成员变量，包括信号量的值(value)、信号量的唯一标识符(semid)、以及等待队列(wq)等。最后，返回创建的信号量的标识符。

```
int sys_sem_create(int value)
{
static int semid = 0;
struct Semaphore *new_sem=(struct Semaphore*)kmalloc(sizeof(struct Semaphore));
new_sem->value=value;
new_sem->next=NULL;//很重要
new_sem->semid=semid++;
new_sem->wq=NULL;
```

int sys_sem_destroy(int semid)：这个函数用于销毁一个已经创建的信号量。首先通过 semid 找到对应的信号量结构体，然后从系统的信号量链表中移除该信号量。在移除信号量时，需要先关中断，以防止并发访问导致数据不一致。最后，释放信号量结构体所占用的内存，并返回成功或失败的结果。

```
int sys_sem_destroy(int semid)
{
struct Semaphore*sem;
sem=get_sem(semid);
if(sem==NULL) return -1;
uint32_t flags;
save_flags_cli(flags);
remove_sem(sem);
restore_flags(flags);
kfree(sem);
return 0;

}
```

int sys_sem_wait(int semid)：这个函数用于等待一个信号量，即将信号量的值减 1。首先通过 semid 找到对应的信号量结构体，然后判断当前信号量的值是否大于 0，如果大于 0，则表示可以立即获取信号量；如果小于等于 0，则需要将当前线程加入到信号量的等待队列中，并将当前线程置为睡眠状态。需要注意的是，在操作信号量前后需要关闭中断，以防止并发修改导致数据不一致。

```
int sys_sem_wait(int semid)
{
struct Semaphore*sem;
sem=get_sem(semid);
if(sem==NULL)return -1;
uint32_t flags;
save_flags_cli(flags);//关中断
if((--sem->value)<0)
sleep_on(&sem->wq);
restore_flags(flags);
return 0;
}
```

int sys_sem_signal(int semid)：这个函数用于释放一个信号量，即将信号量的值加 1。首先通过 semid 找到对应的信号量结构体，然后判断当前信号量的值是否小于等于 0，如果小于等于 0，则需要唤醒一个等待该信号量的线程；如果大于 0，则表示没有线程在等待该信号量，直接将信号量的值加 1 即可。和前面的函数一样，在操作信号量前后需要关闭中断，以防止并发修改导致数据不一致。

```
int sys_sem_signal(int semid)
{
struct Semaphore*sem;
sem=get_sem(semid);
if(sem==NULL)return -1;
uint32_t flags;
save_flags_cli(flags);//关中断
if((++sem->value)<=0)
wake_up(&sem->wq,1);//唤醒1个线
restore_flags(flags);
return 0;
```

（3）在 include/syscall-nr.h 中，定义系统调用的号码

```
#define SYSCALL_sem_create 3000
#define SYSCALL_sem_destroy 3001
#define SYSCALL_sem_wait 3002
#define SYSCALL_sem_signal 3003
```

（4） 在 kernel/machdep.c 的函数 syscall 中，增加 case 分支，调用相关函数。

```
    case
SYSCALL_sem_create:
{
int value= *(int *)(ctx->esp+4);
ctx->eax=sys_sem_create(value);
}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;
```

SYSCALL_sem_create：这个 case 处理了创建信号量的系统调用。它从系统调用参数中提取信号量的初值，并调用 sys_sem_create 函数来创建信号量。最后，将函数的返回值存储在 ctx->eax 中，以便返回给用户程序。

SYSCALL_sem_destroy：这个 case 处理了销毁信号量的系统调用。它从系统调用参数中提取信号量的 ID，并调用 sys_sem_destroy 函数来销毁该信号量。最后，将函数的返回值存储在 ctx->eax 中，以便返回给用户程序。

SYSCALL_sem_wait：这个 case 处理了等待信号量的系统调用。它从系统调用参数中提取信号量的 ID，并调用 sys_sem_wait 函数来等待该信号量。最后，将函数的返回值存储在 ctx->eax 中，以便返回给用户程序。

SYSCALL_sem_signal：这个 case 处理了发信号量的系统调用。它从系统调用参数中提取信号量的 ID，并调用 sys_sem_signal 函数来发送信号量。最后，将函数的返回值存储在 ctx->eax 中，以便返回给用户程序

（5） 在 userapp/lib/syscall-wrapper.S 末尾，加入汇编语言接口“WRAPPER(函数名)、

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

（6）在 userapp/include/syscall.h 中，加入 C 语言声明，调用时按照这个格式调用

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

Step3：Main.c 进行测试

（1） 定义全局信号量

```
#define step 2
#define buffer_num 10 //缓冲区数量
#define   x g_graphic_dev.XResolution/buffer_num //利用水平分辨率求出数组元素最大范围
#define   y g_graphic_dev.YResolution/step-50//利用垂直分辨率求出数组元素个数
```

```
    //定义三个信号量id
int mutex[buffer_num];
int full;
int empty;
```

（2） 在 main 函数中初始化

```
// 初始化信号量
int k;
for (k = 0; k < buffer_num; k++) {
    mutex[k] = sem_create(1);
}

full = sem_create(0);
empty = sem_create(buffer_num);
```

在 main 函数中，通过循环为每个缓冲区初始化一个互斥信号量，并为 full 和 empty 分别创建计数信号量。

（3）最后在 main 函数中销毁信号量

```
// 销毁信号量
for (k = 0; k < buffer_num; k++) {
    sem_destroy(mutex[k]);
}

sem_destroy(full);
sem_destroy(empty);
```

在 main 函数的最后，通过循环依次销毁每个缓冲区的互斥信号量，并销毁 full 和 empty 计数信号量。

（4）生产者函数：每次生产一组随机数填满缓冲区，填充之前先清空屏幕。

```
// 生产者线程函数
void tsk_producer(void* p) {
    printf("This is producer_task with tid=%d,priority=%d\r\n", task_getid(), getpriority(task_getid()));
    int (*arr)[buffer_num * y] = (int(*)[buffer_num * y])p;
    srand(time(NULL));

    int producer_buffer = 0;
    int i;

    do {
        sem_wait(empty);
        sem_wait(mutex[producer_buffer]);
        clearbuff(producer_buffer);

        for (i = 0; i < y; i++) {
            arr[producer_buffer][i] = rand() % x;
            line(x * producer_buffer, y * i, x * producer_buffer + arr[producer_buffer][i], y * i, RGB(255, 100, 255 - i));
            if (i % 5 == 0) nanosleep((const struct timespec[]){{0, 10000000L}}, NULL);
        }

        sem_signal(mutex[producer_buffer]);
        sem_signal(full);

        if ((++producer_buffer) == buffer_num) producer_buffer = 0;
    } while (1);

    task_exit(0);
}

// 消费者线程函数
void tsk_consumer(void* p) {
    printf("This is consumer_task with tid=%d,priority=%d\r\n", task_getid(), getpriority(task_getid()));
    int (*arr)[buffer_num * y] = (int(*)[buffer_num * y])p;
    int consumer_buffer = 0;
```

生产者函数通过不断地向缓冲区写入随机数来模拟生产过程。它首先等待空闲缓冲区（sem_wait(empty)),然后获取对应缓冲区的互斥信号量(sem_wait(mutex[producer_buffer]))。在写入数据之前，通过 clearbuff 函数清空缓冲区。然后，随机生成一组数据写入缓冲区，并通过 line 函数画线。写入完成后，释放对应缓冲区的互斥信号量（sem_signal(mutex[producer_buffer])），并增加 full 计数信号量以通知消费者数据已经写入。最后，更新 producer_buffer 变量以循环使用缓冲区。

（5）消费者函数：每次对一个缓冲区进行排序

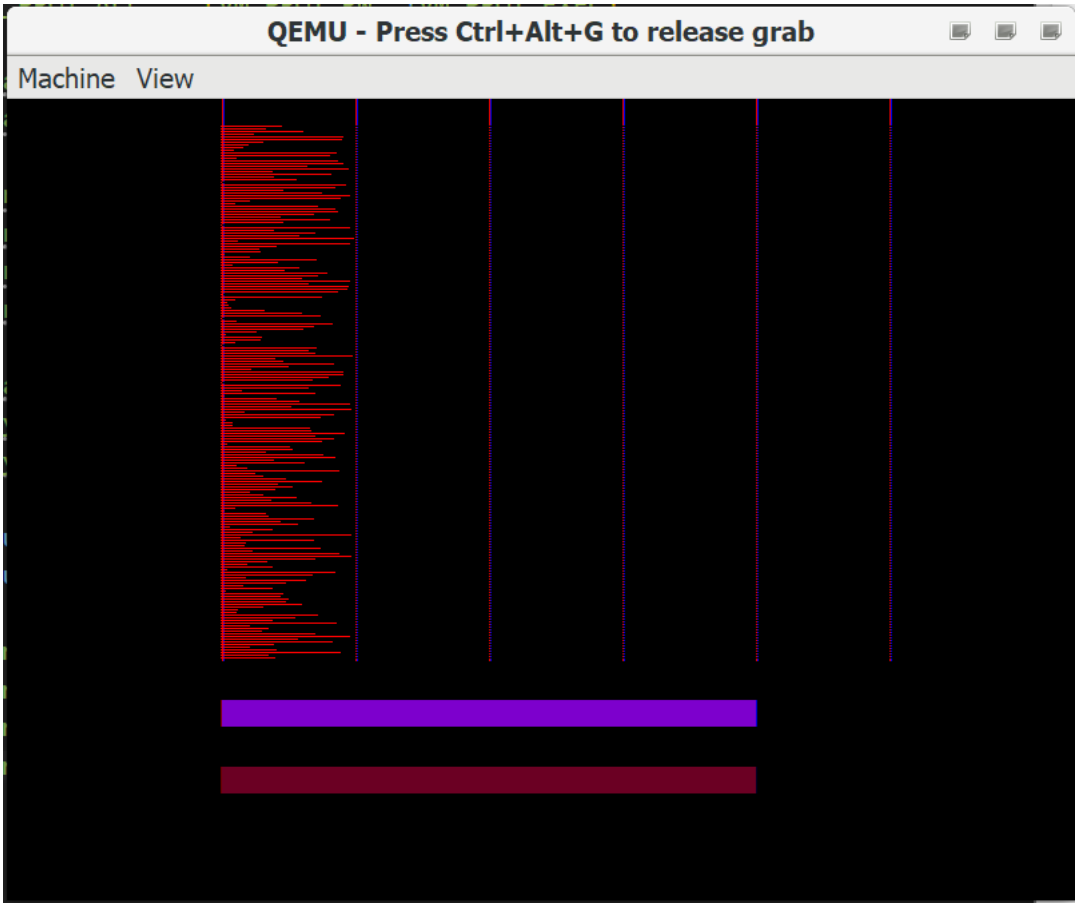
```
// 消费者线程函数
void tsk_consumer(void* p) {
    printf("This is consumer_task with tid=%d,priority=%d\r\n", task_getid(), getpriority(task_getid()));
    int (*arr)[buffer_num * y] = (int(*)[buffer_num * y])p;
    int consumer_buffer = 0;

    do {
        sem_wait(full);
        sem_wait(mutex[consumer_buffer]);
        bubble_sort(arr, consumer_buffer);
        sem_signal(mutex[consumer_buffer]);
        sem_signal(empty);

        if ((++consumer_buffer) == buffer_num) consumer_buffer = 0;
    } while (1);

    task_exit(0);
}
```

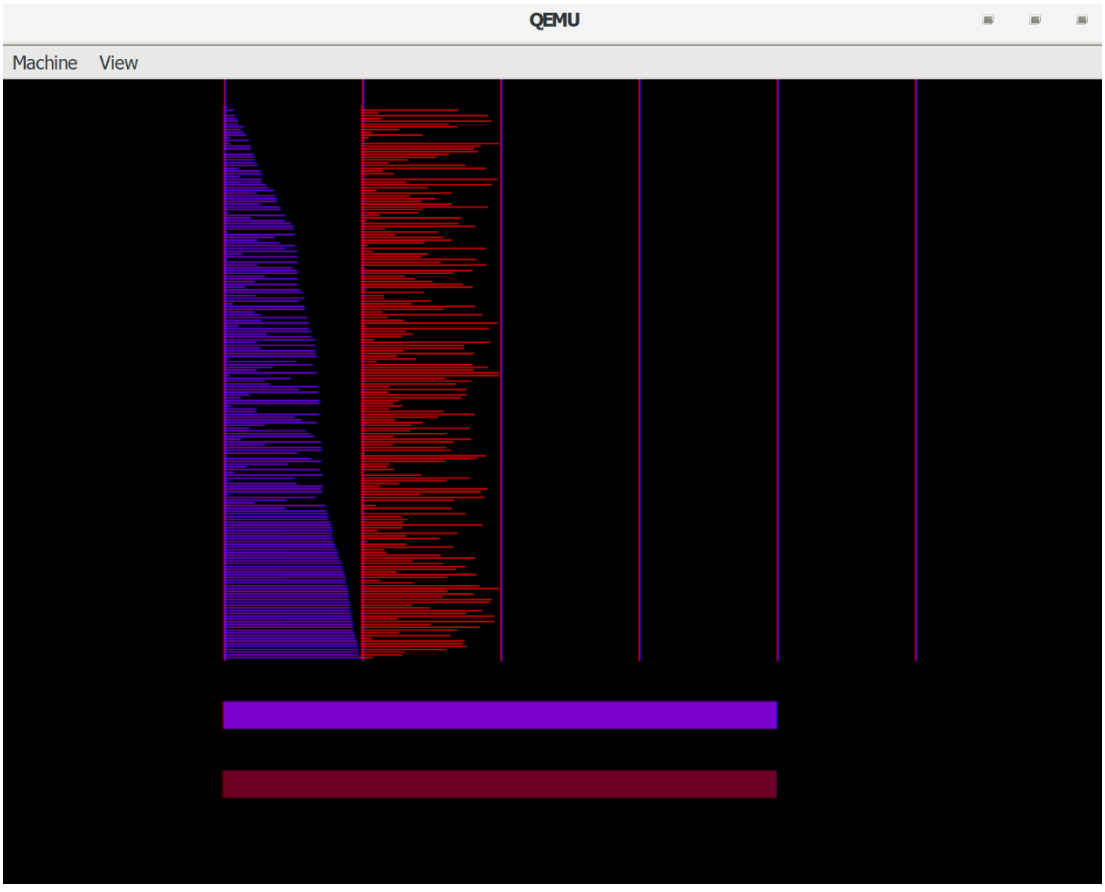
clearbuff 函数用于清空指定缓冲区的内容，将其填充为黑色，以准备写入新的数据。

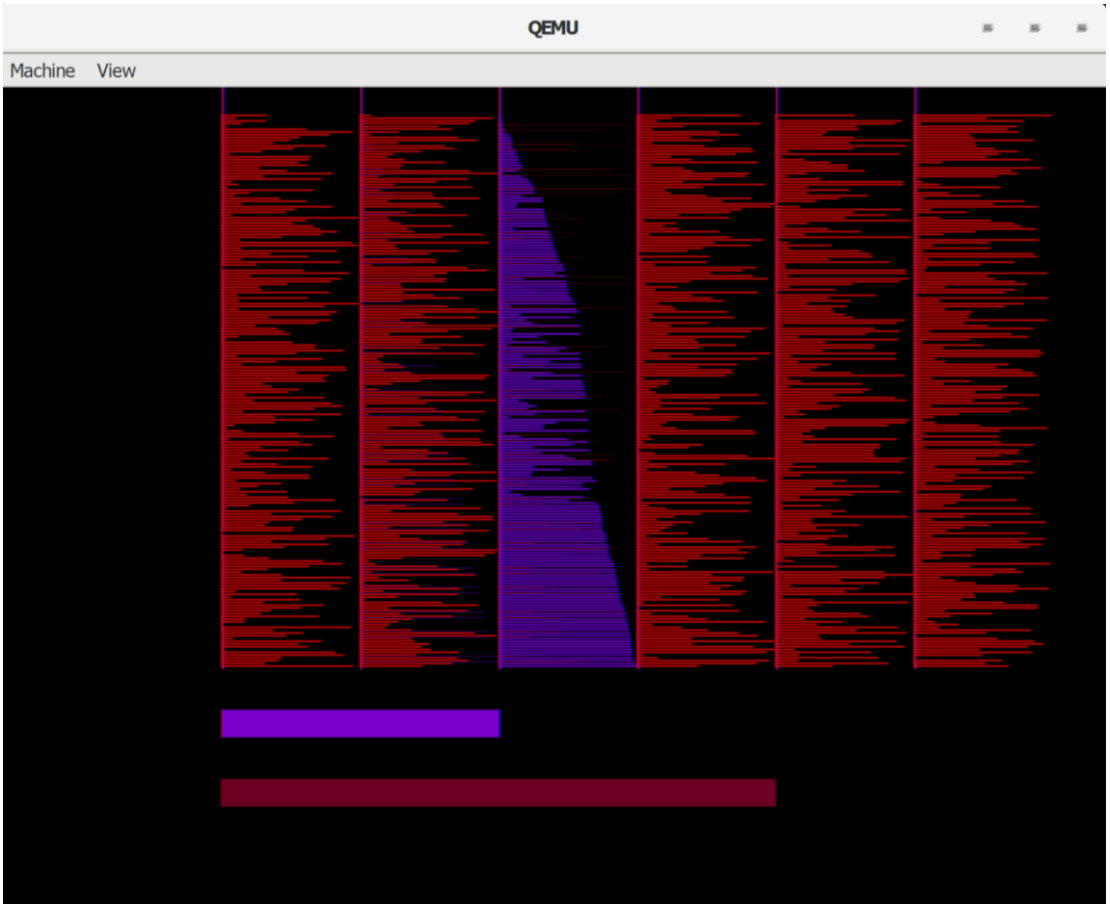
（6） 控制线程函数与实验三相同

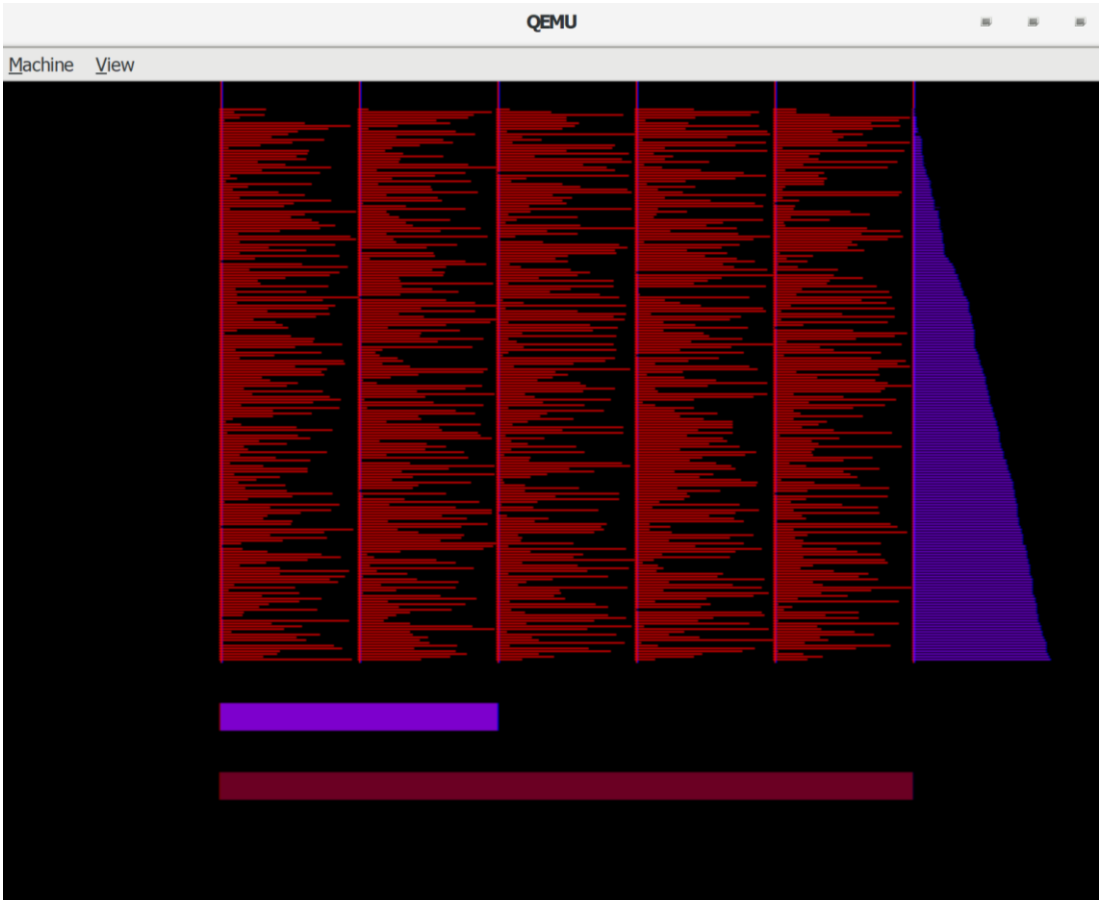
```
//控制线程
void mytask_control(void* pv) {
    show_priority(tid_foo1,1);
    show_priority(tid_foo2, 2);
    int mykeypress;
    while (1) {
        mykeypress = getchar();
        switch (mykeypress)
        {
        case 0x4800://(up)
        {
            setpriority(tid_foo1, getpriority(tid_foo1) + 2);
            show_priority(tid_foo1, 1);
        }
        break;
        case 0x5000://(down)
        {
            setpriority(tid_foo1, getpriority(tid_foo1)-2);
            show_priority(tid_foo1, 1);
        }
        break;
//0x4d00(right)/0x4b00(left)
        case 0x4d00:
        {
            setpriority(tid_foo2, getpriority(tid_foo2) + 2);
            show_priority(tid_foo2, 2);
        }
        break;
        case 0x4b00:
        {
            setpriority(tid_foo2, getpriority(tid_foo2) - 2);
            show_priority(tid_foo2, 2);
        }
        break;
        default:
            break;
        }
    }
}
```

四、实验结果及分析


QEMU - Press Ctrl+Alt+G to release grab
Machine View


QEMU
Machine View


QEMU
Machine View


QEMU
Machine View