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 4 月 1 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	线程的调度	指导教师	XX

教师评	教师签名： 2023 年　月　日

一、 实验目的

• 掌握线程的调度

-优先级调度

基本概念：

- 优先级
  - 静态优先级(nice)：内核不会修改它，不随时间而变化，除非用户通过系统调用setpriority进行修改
  - 动态优先级(priority)：内核根据线程使用CPU的状况、静态优先级nice和系统负荷计算出来，会随时间而变化
    - 最终的调度依据，即调度器只根据动态优先级进行调度

二、实验内容

实验内容：

- 静态优先级代码编写
- 动态优先级代码编写
- 测试静态优先级

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“struct tcb”中增加线程的静态优先级 nice;

注意：

①一定要加在 kstack 字段之后、signature 字段之前！！！

②在函数 sys_task_create 中初始化 nice=0

③nice 是整数，取值范围[-NZERO, NZERO-1]，值越小优先级越高

#define NZERO 20

```
#define NZERO 20
#define PRI_USER_MIN 0
#define PRI_USER_MAX 127
```

```
struct tcb {
    /*hardcoded*/
    uint32_t    kstack;      /*saved top of the kernel stack for this task*/

    int         tid;         /* task id */
    int         state;       /* -1:waiting,0:running,1:ready,2:zombie */
#define TASK_STATE_WAITING  -1
#define TASK_STATE_READY     1
#define TASK_STATE_ZOMBIE    2

    int         timeslice;   //时间片
#define TASK_TIMESLICE_DEFAULT 4

    int         code_exit;   //保存该线程的退出代码
    struct wait_queue *wq_exit; //等待该线程退出的队列

    struct tcb  *next;
    struct fpu   fpu;        //数学协处理器的寄存器

    int nice;               //EX3新增的静态优先级

    uint32_t     signature;  //必须是最后一个字段
#define TASK_SIGNATURE 0x20160201
};
```

```
    //EX3初始化
    new->nice = 0;
    new->estcpu = 0;
    new->priority = 0;
```

Step2：增加系统调用

注意：

①如果参数 tid=0，表示获取/设置当前线程的 nice 值，而不是 task0！

②函数“struct tcb *get_task(int tid)”用于根据 tid 获取 struct tcb 的指针。调用时一定要用 save_flags_cli/restore_flags 保护起来

```
//EX3
int sys_getpriority(int tid);
int sys_setpriority(int tid, int prio);
```

```
//EX3添加优先级函数
int sys_getpriority(int tid)
{
    uint32_t flags;
    struct tcb *tsk;
    save_flags_cli(flags);    //保存flags
    if (tid == 0)           //如果当前tid为0，则获取当前线程
        tsk = g_task_running;
    else                    //若tid不为0，则获取指定线程
        tsk = get_task(tid);
    restore_flags(flags);    //还原flags
    if (tsk != NULL)        //成功返回线程tid的(nice+NZERO)
        return tsk->nice + NZERO;
    else                    //失败则返回-1
        return -1;
}

int sys_setpriority(int tid, int prio)
{
    uint32_t flags;
    struct tcb* tsk;
    if (prio < 0|| prio >2 * NZERO - 1)    // 若prio超出范围则设置失败
            return -1;
    save_flags_cli(flags);                  //保存flags
    if(tid == 0)                            //如果当前tid为0，则表示设置当前线程的nice值
        tsk = g_task_running;
    else
        tsk = get_task(tid);
    restore_flags(flags);                   //还原flags
    if (tsk != NULL)                        //若获取目标线程成功返回0
    {
        tsk->nice = prio - NZERO;
        return 0;
    }
    else                                    //若获取目标线程失败返回-1
        return -1;
}
```

```
//EX3
#define SYSCALL_getpriority     11
#define SYSCALL_setpriority     12
```

```
//EX3
case SYSCALL_getpriority:
{
    int tid = *(int*)(ctx->esp + 4);
    ctx->eax = sys_getpriority(tid);
}break;

case SYSCALL_setpriority:
{
    int tid = *(int*)(ctx->esp + 4);
    int prio = *(int*)(ctx->esp + 8);
    ctx->eax = sys_setpriority(tid, prio);
}break;
```

```
25  WRAPPER(send)
26  WRAPPER(ioctl)
27  WRAPPER(getpriority)
28  WRAPPER(setpriority)
```

```c
#ifndef _SYSCALL_H
#define _SYSCALL_H

#include <sys/types.h>
#include <inttypes.h>
#include <time.h>
#include <ioctl.h>

//EX3
int sys_getpriority(int tid);
int sys_setpriority(int tid, int prio);
```

Step3：在“struct tcb”中，再增加两个属性;

注意：

1.estcpu：表示线程最近使用了多少 CPU 时间

①在函数 sys_task_create 中初始化 estcpu=0

②每次定时器中断：g_task_running->estcpu++，task0 除外！

③每秒钟为所有线程（运行、就绪和等待）更新一次

2. priority：表示线程的动态优先级

①priority = PRI_USER_MAX-(estcpu/4)-(nice*2)

截断到 PRI_USER_MIN 到 PRI_USER_MAX 范围内

#define PRI_USER_MIN 0

#define PRI_USER_MAX 127

②值越大优先级越高

```c
struct tcb {
    /*hardcoded*/
    uint32_t    kstack;      /*saved top of the kernel stack for this task*/

    int         tid;         /* task id */
    int         state;       /* -1:waiting,0:running,1:ready,2:zombie */
#define TASK_STATE_WAITING  -1
#define TASK_STATE_READY     1
#define TASK_STATE_ZOMBIE    2

    int         timeslice;   //时间片
#define TASK_TIMESLICE_DEFAULT 4

    int         code_exit;   //保存该线程的退出代码
    struct wait_queue *wq_exit; //等待该线程退出的队列

    //EX3新增
    fixedpt estcpu          //表示线程最近使用了多少CPU时间
    fixedpt priority        //表示线程的动态优先级

    struct tcb  *next;
    struct fpu   fpu;        //数学协处理器的寄存器

    int nice;               //EX3新增的静态优先级

    uint32_t     signature;  //必须是最后一个字段
#define TASK_SIGNATURE 0x20160201
};
```

```
    //EX3初始化
    new->nice = 0;
    new->estcpu = 0;
    new->priority = 0;
```

```
#define NZERO 20
#define PRI_USER_MIN 0
#define PRI_USER_MAX 127
```

Step4：增加一个全局变量；

注意：

1.g_load_avg：表示系统的平均负荷

①初值为 0

②每秒钟更新一次

- g_load_avg=(59/60) ×g_load_avg+(1/60) × nready

>>nready 表示处于就绪状态的线程个数，task0 除外！

```
/*记录系统启动以来，定时器中断的次数*/
unsigned volatile g_timer_ticks = 0;
fixedpt g_load_avg = 0;
```

Step5：属性计算；

1.g_load_avg 和线程的 estcpu

①在定时器的中断处理函数（ISR）中计算

文件 timer.c 中的函数 isr_timer

①如何每隔一秒计算一次？

(g_timer_ticks % HZ)

>>=0，表示 1 秒钟已经过去

>>否则，还不到 1 秒钟

```c
void isr_timer(uint32_t irq, struct context* ctx)
{
    g_timer_ticks++;
    //sys_putchar('.');
    if (g_task_running != NULL) {
        //如果是task0在运行，则强制调度
        if (g_task_running->tid == 0)
        {
            g_resched = 1;
        }
        else {
            //否则，把当前线程的时间片减一
            --g_task_running->timeslice;

            //如果当前线程用完了时间片，也要强制调度
            if (g_task_running->timeslice <= 0)
            {
                g_resched = 1;
                g_task_running->timeslice = TASK_TIMESLICE_DEFAULT;
            }
            g_task_running->estcpu = fixedpt_add(g_task_running->estcpu, FIXEDPT_ONE);

            //每秒更新一次
            if ((g_timer_ticks % HZ) == 0) {
                int nready = 0;//表述处于就绪态的线程个数
                struct tcb* tsk = g_task_head;//得到线程起始位置
                //遍历线程，已得到线程个数
                while (tsk != NULL)
                {
                    //如果线程处于就绪态那么记录此线程的个数
                    if (tsk->state == TASK_STATE_READY)
                        nready = nready + 1;
                    tsk = tsk->next;
                }

                //计算定点数59/60
                fixedpt r59_60 = fixedpt_div(fixedpt_fromint(59), fixedpt_fromint(60));
                //计算定点数1/60
                fixedpt r01_60 = fixedpt_div(FIXEDPT_ONE, fixedpt_fromint(60));
                //计算系统平均符合前一次负荷数与处于就绪态的线程数以59:1的权重相加取平均
                g_load_avg = fixedpt_add(fixedpt_mul(r59_60, g_load_avg),
                    fixedpt_mul(r01_60, fixedpt_fromint(nready)));
                tsk = g_task_head;//再次回到起始位置

                //更新线程使用时间
                while (tsk != NULL)
                {
                    fixedpt ratio;
                    //将g_load_avg乘以2
                    ratio = fixedpt_mul(FIXEDPT_TWO, g_load_avg);
                    // (2 * G - load_avg) / ((2 * G - load_avg) + 1)
                    ratio = fixedpt_div(ratio, fixedpt_add(ratio, FIXEDPT_ONE));
                    // ((2 * G - load_avg) / ((2 * G - load_avg) + 1)) * estcpu + nice
                    tsk->estcpu = fixedpt_add(fixedpt_mul(ratio, tsk->estcpu),
                        fixedpt_fromint(tsk->nice));
                    tsk = tsk->next;

                }
            }
        }
    }
}
```

Step6：修改 task.c 中的函数 schedule，实现优先级调度算法；

注意：

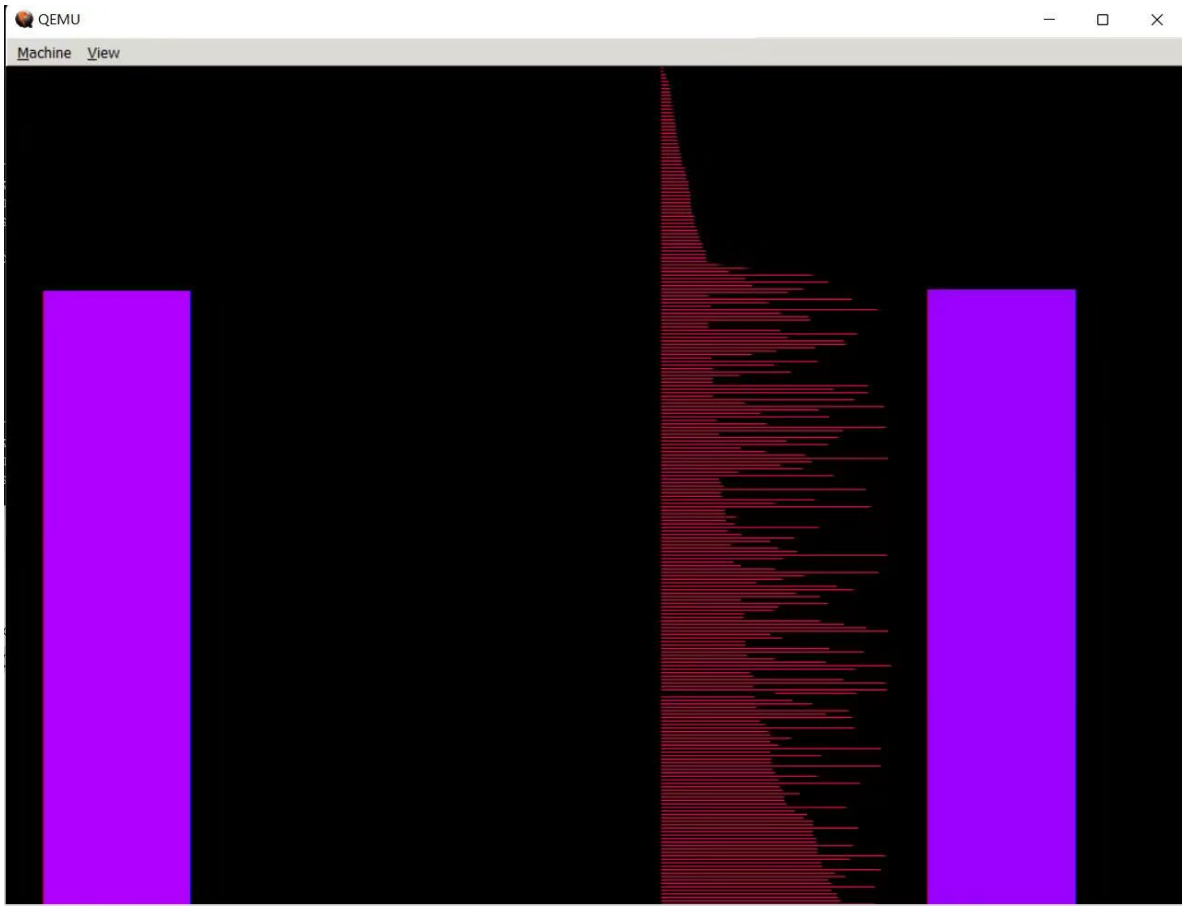
①先计算各个线程的动态优先级 priority（task0 除外！），然后进行调度；

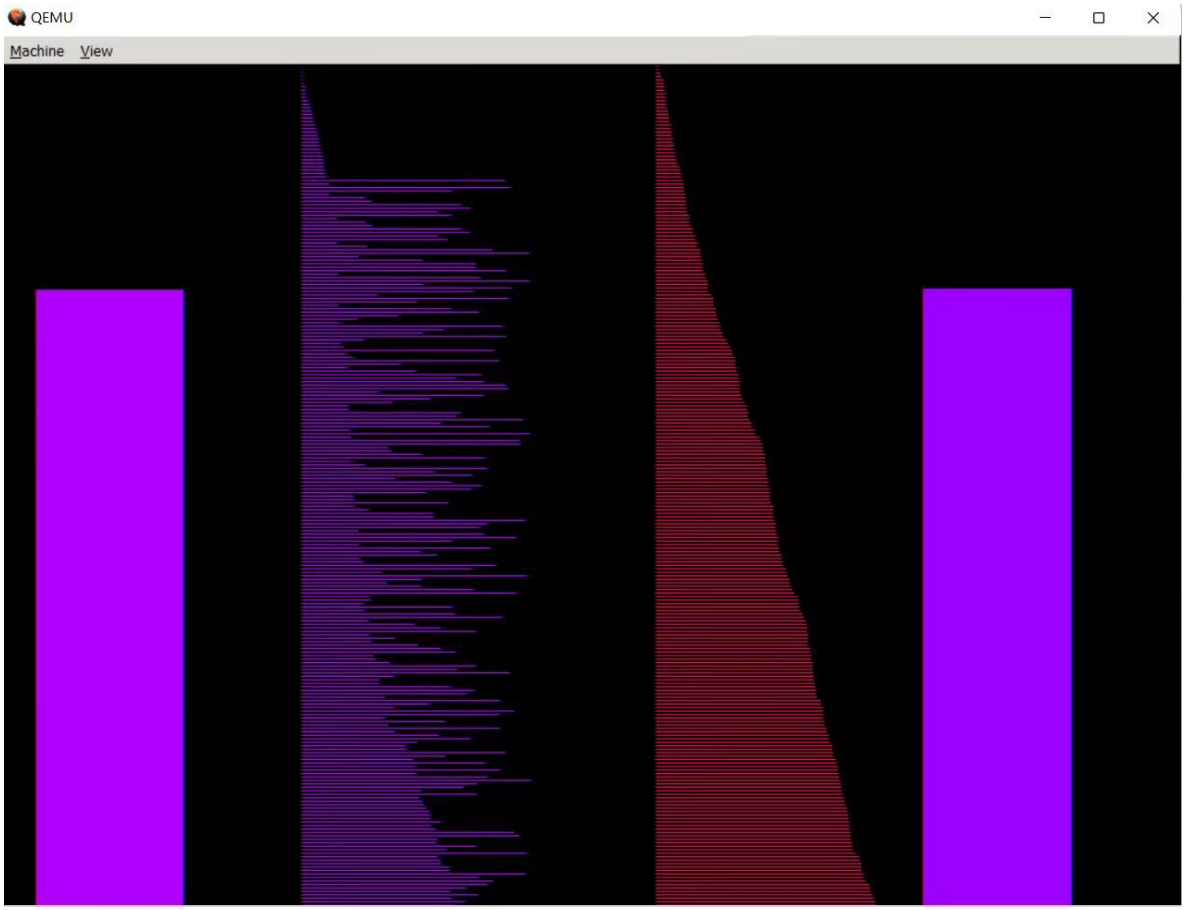
②线程 0（task0，它的 ID=0）是一个特殊的线程，仅当没有其他可运行的线程时，才能调度 task0 运行！

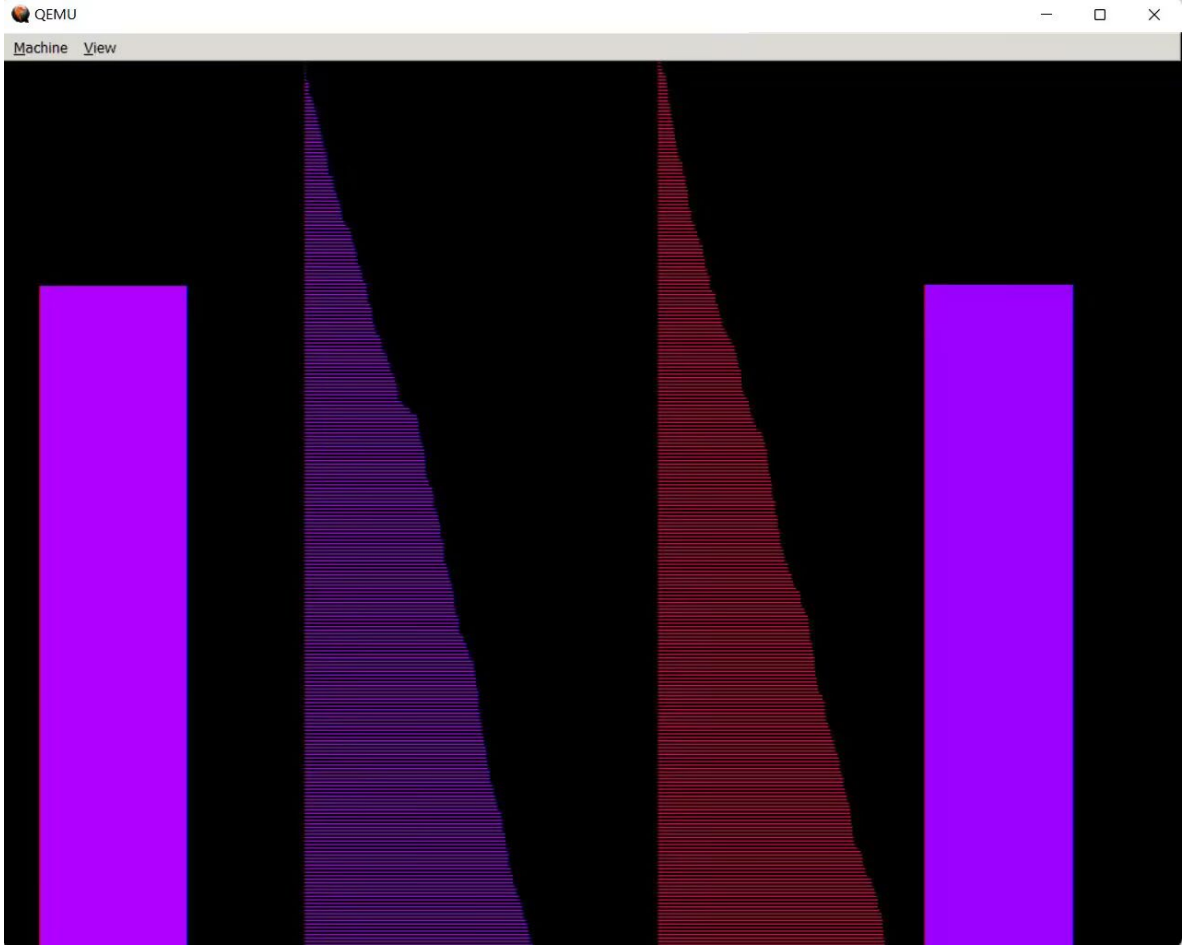
③函数 schedule 被执行时，CPU 的中断已经被关闭。

```
void schedule()
{
    struct tcb* select = g_task_head;
    struct tcb* select_curr = g_task_head;

    // 动态调度
    while (select_curr != NULL) {
        // 计算各个线程的优先级
        select_curr->priority = PRI_USER_MAX -
            fixedpt_toint(fixedpt_div(select_curr->estcpu, fixedpt_fromint(4))) -
            select_curr->nice * 2;
        // 确保线程优先级在允许范围内
        select_curr->priority = max(PRI_USER_MIN, min(PRI_USER_MAX, select_curr->priority));
        select_curr = select_curr->next;
}

        // 选择可调度的线程
    while (select_curr != NULL) {
        if (select_curr->tid != 0 && select_curr->state == TASK_STATE_READY &&
        (select_curr->priority >= select->priority || select->tid == 0)) {
        select = select_curr;
        }
        select_curr = select_curr->next;
    }

    // 回到 task0
    if (select == g_task_running && select->state == TASK_STATE_READY) {
        return;
    } else if (select == g_task_running) {
        select = task0;
    }

    g_resched = 0;
    switch_to(select);
}
```

运行结果：







核心代码：

```
#define NZERO 20
#define PRI_USER_MIN 0
#define PRI_USER_MAX 127

struct tcb {
    /*hardcoded*/
    uint32_t    kstack;      /*saved top of the kernel stack for this task*/

    int         tid;         /* task id */
    int         state;       /* -1:waiting,0:running,1:ready,2:zombie */
#define TASK_STATE_WAITING  -1
#define TASK_STATE_READY     1
#define TASK_STATE_ZOMBIE    2

    int         timeslice;   //时间片
#define TASK_TIMESLICE_DEFAULT 4

    int         code_exit;   //保存该线程的退出代码
    struct wait_queue *wq_exit; //等待该线程退出的队列

    //EX3新增
    fixedpt estcpu          //表示线程最近使用了多少CPU时间
    fixedpt priority        //表示线程的动态优先级

    struct tcb  *next;
    struct fpu   fpu;        //数学协处理器的寄存器

    int nice;               //EX3新增的静态优先级

    uint32_t     signature;  //必须是最后一个字段
#define TASK_SIGNATURE 0x20160201
};
```

```c
/**
 * 系统调用task_create的执行函数
 *
 * 创建一个新的线程，该线程执行func函数，并向新线程传递参数pv
 */
struct tcb *sys_task_create(void *tos,
                            void (*func)(void *pv), void *pv)
{
    static int tid = 0;
    struct tcb *new;
    char *p;
    uint32_t flags;
    uint32_t ustack=(uint32_t)tos;

    if(ustack & 3)
        return NULL;

    p = (char *)kmemalign(PAGE_SIZE, PAGE_SIZE);
    if(p == NULL)
        return NULL;

    new = (struct tcb *)p;

    memset(new, 0, sizeof(struct tcb));

    new->kstack = (uint32_t)(p+PAGE_SIZE);
    new->tid = tid++;
    new->state = TASK_STATE_READY;
    new->timeslice = TASK_TIMESLICE_DEFAULT;
    new->wq_exit = NULL;
    new->next = NULL;
    new->signature = TASK_SIGNATURE;

    //EX3初始化
    new->nice = 0;
    new->estcpu = 0;
    new->priority = 0;


    /*XXX - should be elsewhere*/
    new->fpu.cwd = 0x37f;
    new->fpu.twd = 0xffff;

    INIT_TASK_CONTEXT(ustack, new->kstack, func, pv);

    save_flags_cli(flags);
    add_task(new);
    restore_flags(flags);

    return new;
}
```

```c
//EX3添加优先级函数
int sys_getpriority(int tid)
{
    uint32_t flags;
    struct tcb *tsk;
    save_flags_cli(flags);    //保存flags
    if (tid == 0)           //如果当前tid为0，则获取当前线程
        tsk = g_task_running;
    else                    //若tid不为0，则获取指定线程
        tsk = get_task(tid);
    restore_flags(flags);    //还原flags
    if (tsk != NULL)        //成功返回线程tid的(nice+NZERO)
        return tsk->nice + NZERO;
    else                    //失败则返回-1
        return -1;
}

int sys_setpriority(int tid, int prio)
{
    uint32_t flags;
    struct tcb* tsk;
    if (prio < 0|| prio >2 * NZERO - 1)    // 若prio超出范围则设置失败
        return -1;
    save_flags_cli(flags);                  //保存flags
    if(tid == 0)                            //如果当前tid为0，则表示设置当前线程的nice值
        tsk = g_task_running;
    else
        tsk = get_task(tid);
    restore_flags(flags);                   //还原flags
    if (tsk != NULL)                        //若获取目标线程成功返回0
    {
        tsk->nice = prio - NZERO;
        return 0;
    }
    else                                    //若获取目标线程失败返回-1
        return -1;
}
```

```c
//EX3
#define SYSCALL_getpriority     11
#define SYSCALL_setpriority     12
```

```c
#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(getpriority)
WRAPPER(setpriority)
```

```c
void isr_timer(uint32_t irq, struct context* ctx)
{
    g_timer_ticks++;
    //sys_putchar('.');
    if (g_task_running != NULL) {
        //如果是task0在运行，则强制调度
        if (g_task_running->tid == 0)
        {
            g_resched = 1;
        }
        else {
            //否则，把当前线程的时间片减一
            --g_task_running->timeslice;

            //如果当前线程用完了时间片，也要强制调度
            if (g_task_running->timeslice <= 0)
            {
                g_resched = 1;
                g_task_running->timeslice = TASK_TIMESLICE_DEFAULT;
            }
            g_task_running->estcpu = fixedpt_add(g_task_running->estcpu, FIXEDPT_ONE);

            //每秒更新一次
            if ((g_timer_ticks % HZ) == 0) {
                int nready = 0;//表述处于就绪态的线程个数
                struct tcb* tsk = g_task_head;//得到线程起始位置
                //遍历线程，已得到线程个数
                while (tsk != NULL)
                {
                    //如果线程处于就绪态那么记录此线程的个数
                    if (tsk->state == TASK_STATE_READY)
                        nready = nready + 1;
                    tsk = tsk->next;
                }

                //计算定点数59/60
                fixedpt r59_60 = fixedpt_div(fixedpt_fromint(59), fixedpt_fromint(60));
                //计算定点数1/60
                fixedpt r01_60 = fixedpt_div(FIXEDPT_ONE, fixedpt_fromint(60));
                //计算系统平均符合前一次负荷数与处于就绪态的线程数以59:1的权重相加取平均
                g_load_avg = fixedpt_add(fixedpt_mul(r59_60, g_load_avg),
                    fixedpt_mul(r01_60, fixedpt_fromint(nready)));
                tsk = g_task_head;//再次回到起始位置

                //更新线程使用时间
                while (tsk != NULL)
                {
                    fixedpt ratio;
                    //将g_load_avg乘以2
                    ratio = fixedpt_mul(FIXEDPT_TWO, g_load_avg);
                    // (2 * G - load_avg) / ((2 * G - load_avg) + 1)
                    ratio = fixedpt_div(ratio, fixedpt_add(ratio, FIXEDPT_ONE));
                    // ((2 * G - load_avg) / ((2 * G - load_avg) + 1)) * estcpu + nice
                    tsk->estcpu = fixedpt_add(fixedpt_mul(ratio, tsk->estcpu),
                        fixedpt_fromint(tsk->nice));
                    tsk = tsk->next;

                }
            }
        }
    }
}
```

```
void schedule()
{
    struct tcb* select = g_task_head;
    struct tcb* select_curr = g_task_head;

    // 动态调度
    while (select_curr != NULL) {
        // 计算各个线程的优先级
        select_curr->priority = PRI_USER_MAX -
            fixedpt_toint(fixedpt_div(select_curr->estcpu, fixedpt_fromint(4))) -
            select_curr->nice * 2;
        // 确保线程优先级在允许范围内
        select_curr->priority = max(PRI_USER_MIN, min(PRI_USER_MAX, select_curr->priority));
        select_curr = select_curr->next;
}

        // 选择可调度的线程
    while (select_curr != NULL) {
        if (select_curr->tid != 0 && select_curr->state == TASK_STATE_READY &&
        (select_curr->priority >= select->priority || select->tid == 0)) {
        select = select_curr;
        }
        select_curr = select_curr->next;
    }

    // 回到 task0
    if (select == g_task_running && select->state == TASK_STATE_READY) {
        return;
    } else if (select == g_task_running) {
        select = task0;
    }

    g_resched = 0;
    switch_to(select);
}
```

```
void main(void* pv)
{
//进入图形模式
init_graphic(0x144);

//获取屏幕分辨率
int X = g_graphic_dev.XResolution;
int Y = g_graphic_dev.YResolution;

//申请线程栈并创建线程
const unsigned int STACK_SIZE = 1024 * 1024;
unsigned char* stack_foo1 = (unsigned char*)malloc(STACK_SIZE);
unsigned char* stack_foo2 = (unsigned char*)malloc(STACK_SIZE);
unsigned char* stack_foo3 = (unsigned char*)malloc(STACK_SIZE);

task_create(stack_foo1 + STACK_SIZE, &task1_Bubble_Sort, (void*)0);
task_create(stack_foo2 + STACK_SIZE, &task2_Bubble_Sort, (void*)0);
task_create(stack_foo3 + STACK_SIZE, &task_control, (void*)0);

//设置线程优先级
setpriority(2, 0); // control thread
setpriority(3, 10); // bubble sort thread 1
setpriority(4, 10); // bubble sort thread 2

//释放线程栈
free(stack_foo1);
free(stack_foo2);
free(stack_foo3);

//保持线程运行
while (1) {}

task_exit(0);
}
```

```
//显示优先级
void showPriority(int tid, int id)
{
    int x_start = X / 8;
    int x_end = X / 8;
    int length = ((getpriority(tid))* Y)/ 40;
    //对应展示优先级
    if (id == 1) {
        //设置起始位置以及结束位置
        x_start*= 5;
        x_end *= 6;//擦出
        for (int i = Y; i > 0; i--)
            line(x_start, i, x_end, i, RGB(0, 0, 0));//显示
        for (int i = Y; i > length; i--)
            line(x_start, i, x_end, i, RGB(255, 160, 155));
    }
    if (id == 2) {
        x_start *= 7;
        x_end *= 8;
        for (int i = Y; i > 0; i--)
            line(x_start, i, x_end, i, RGB(0, 0, 0));
        for (int i = Y; i > length; i--)
            line(x_start, i, x_end, i, RGB(255, 205, 175));
    }
    return;
}
```

```
void task_control(void* pv)
{
int key;
while (1)
{
if (getchar_nb(&key) == 1) // 非阻塞获取键盘输入
{
switch (key)
{
case 0x4800: // UP
thread_modify_priority(tid_foo_bubsort1, -2); // 减小排序线程1的优先级
break;
case 0x4d00: // RIGHT
thread_modify_priority(tid_foo_bubsort2, -2); // 减小排序线程2的优先级
break;
case 0x5000: // DOWN
thread_modify_priority(tid_foo_bubsort1, 2); // 增加排序线程1的优先级
break;
case 0x4b00: // LEFT
thread_modify_priority(tid_foo_bubsort2, 2); // 增加排序线程2的优先级
break;
default:
break;
}

        showPriority(tid_foo_bubsort1, 1);    // 显示线程1的优先级
        showPriority(tid_foo_bubsort2, 2);    // 显示线程2的优先级
    }
}
}

void thread_modify_priority(UBaseType_t tid, int priority_delta)
{
int old_priority = uxTaskPriorityGet(tid); // 获取线程原来的优先级
int new_priority = old_priority + priority_delta; // 计算新的优先级
new_priority = MAX(new_priority, configMAX_PRIORITIES - 1); // 限定优先级的最大值
new_priority = MIN(new_priority, 0); // 限定优先级的最小值
vTaskPrioritySet(tid, new_priority); // 设置线程的新优先级
}
```

```
#define MAX_NUM 200 // 宏定义，数组中随机数的最大值

void task1_Bubble_Sort(void* pv)
{
int data[N] = { 0 }; // 存储随机数的数组
int i, j, temp; // 循环计数器和中间变量

// 生成随机数并绘制线条
srand(time(NULL));
for (i = 0; i < N; i++)
{
    data[i] = rand() % MAX_NUM;
    line(X / 10, i * 3, X / 10 + data[i], i * 3, RGB(255, 105, 180));
}

// 冒泡排序
for (i = 0, j = N - 1; i < j; i++, j--)
{
    int k, flag = 0;
    for (k = i; k < j; k++)
    {
        if (data[k] > data[k + 1])
        {
            flag = 1;
            temp = data[k];
            data[k] = data[k + 1];
            data[k + 1] = temp;
        }
    }

    // 对每次交换的线条进行绘制
    for (k = i; k < j; k++)
    {
        line(X / 10, k * 3, X / 10 + data[k], k * 3, RGB(0, 0, 0));
        line(X / 10, k * 3, X / 10 + data[k], k * 3, RGB(255, 105, 180));
    }

    if (!flag)
    {
        break;  // 如果没有发生交换，说明已经排好序，直接退出循环
    }

    msleep(49);  // 暂停一段时间，动画效果更明显
}

cleardevice(); // 清屏，清除窗口上绘制的所有内容

task_exit(0);  // 任务退出
}
```

```
#define DATA_SIZE N
#define LINE_OFFSET 3 // 每个数据项占用的线条数量

void task2_Bubble_Sort(void* pv)
{
//printf("This is task selsort with tid=%d\r\n", task_getid());
//srand(time(NULL));

// 设置随机数并且画线
int data[DATA_SIZE];
for (int i = 0; i < DATA_SIZE; i++)
{
    data[i] = rand() % 200;
    int line_start_x = 2 * X / 5;
    int line_start_y = i * LINE_OFFSET;
    int line_end_x = line_start_x + data[i];
    int line_end_y = line_start_y;
    line(line_start_x, line_start_y, line_end_x, line_end_y, RGB(87, 250, 255));
}

// 开始冒泡排序
for (int i = 0; i < DATA_SIZE - 1; i++)
{
    for (int j = DATA_SIZE - 1; j > i; j--)
    {
        if (data[j] < data[j - 1])
        {
            // 交换数据
            int temp = data[j - 1];
            data[j - 1] = data[j];
            data[j] = temp;

            // 擦除之前的线并画出新的线
            int erased_line_start_x = 2 * X / 5;
            int erased_line_start_y = (j - 1) * LINE_OFFSET;
            int erased_line_end_x = erased_line_start_x + data[j - 1];
            int erased_line_end_y = erased_line_start_y;
            line(erased_line_start_x, erased_line_start_y, erased_line_end_x, erased_line_end_y, RGB(0, 0, 0));

            int new_line_start_x = 2 * X / 5;
            int new_line_start_y = j * LINE_OFFSET;
            int new_line_end_x = new_line_start_x + data[j];
            int new_line_end_y = new_line_start_y;
            line(new_line_start_x, new_line_start_y, new_line_end_x, new_line_end_y, RGB(87, 250, 255));

            msleep(49);
        }
    }
}

task_exit(0);
}
```

总结：

本次实验中，我学习了线程的调度和优先级调度的相关知识，并进行了实践操作。通过本次实验，我对于线程调度有了更深入的理解。

在实验过程中，我了解到线程调度是操作系统对于线程并发执行顺序的控制。在多线程程序中，由于多个线程同时运行，调度的顺序以及每个线程的执行时间都是不确定的。因此，需要使用线程的调度技术来保证程序的正确性和性能优化。

同时，在本次实验中，我也学习了优先级调度技术。优先级调度会为不同的线程分配不同的优先级，根据优先级高低来决定线程的调度。这种方式可以有效地提高程序的响应速度和吞吐量。

根据实验的结果，我发现线程的调度和优先级调度对于程序的性能和正确性都有很大的影响。合理的线程调度可以提高程序的效率和并发性，优先级调度可以让程序更快速地响应用户操作。同时，优先级调度也需要考虑到实际情况，不能只让一个线程一直占据最高优先级，导致其他线程无法执行。

综上，通过这次实验，我对于线程调度和优先级调度技术有了更深入的了解和掌握。这将对于我未来在开发多线程程序时有很大的帮助。