实验六: 调度器

计31 张正 2013011418

一、实验目的

- 理解操作系统的调度管理机制
- 熟悉 ucore 的系统调度器框架，以及缺省的Round-Robin 调度算法
- 基于调度器框架实现一个(Stride Scheduling)调度算法来替换缺省的调度算法

二、实验内容

实验五完成了用户进程的管理，可在用户态运行多个进程。但到目前为止，采用的调度策略是很简单的FIFO调度策略。本次实验，主要是熟悉ucore的系统调度器框架，以及基于此框架的Round-Robin（RR） 调度算法。然后参考RR调度算法的实现，完成Stride Scheduling调度算法。

练习0：填写已有实验

本实验依赖实验1/2/3/4/5。请把你做的实验2/3/4/5的代码填入本实验中代码中有“LAB1”/“LAB2”/“LAB3”/“LAB4”“LAB5”的注释相应部分。并确保编译通过。注意：为了能够正确执行lab6的测试应用程序，可能需对已完成的实验1/2/3/4/5的代码进行进一步改进。

在填充之前实验的代码，根据提示，对之前的代码进行更新

对lab4练习一的代码进行更新

```
proc->state = PROC_UNINIT;
proc->pid = -1;
proc->runs = 0;
proc->kstack = 0;
proc->need_resched = 0;
proc->parent = NULL;
proc->mm = NULL;
memset(&(proc->context), 0, sizeof(struct context));
proc->tf = NULL;
proc->cr3 = boot_cr3;
proc->flags = 0;
memset(proc->name, 0, PROC_NAME_LEN);
proc->wait_state = 0;
proc->cptr = proc->optr = proc->yptr = NULL;
proc->rq=NULL;
list_init(&(proc->run_link));
skew_heap_init(&(proc->lab6_run_pool));
proc->time_slice=0;
```

```
        proc->lab6_stride=0;
        proc->lab6_priority=1;
```

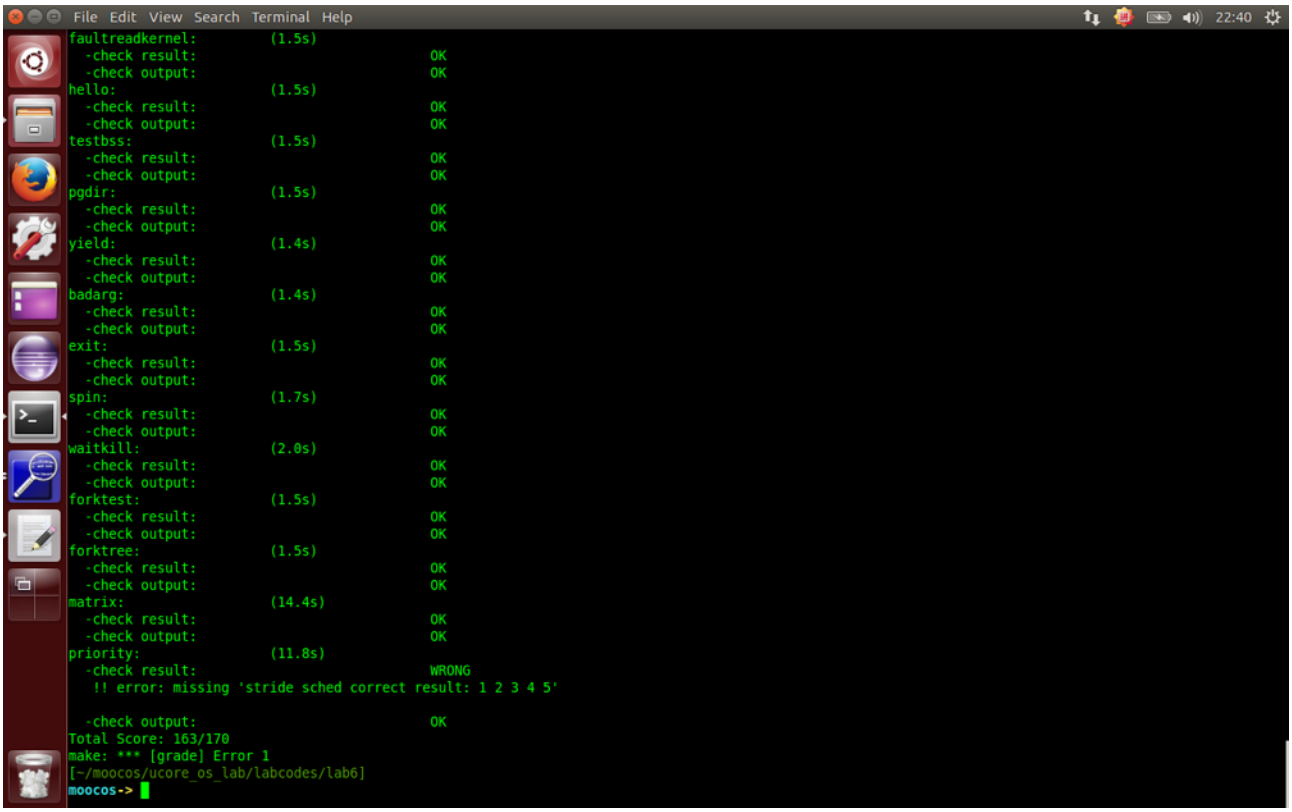
对lab1中的时钟中断部分更新

```
        ticks ++;
        if (ticks % TICK_NUM == 0) {
            assert(current != NULL);
            current->need_resched = 1;
        }
        sched_class_proc_tick(current);
```

练习1: 使用 Round Robin 调度算法（不需要编码）

完成练习0后，建议大家比较一下（可用kdiff3等文件比较软件）个人完成的lab5和练习0完成后的刚修改的lab6之间的区别，分析了解lab6采用RR调度算法后的执行过程。执行make grade，大部分测试用例应该通过。但执行priority.c应该过不去。

执行如下



请在实验报告中完成：

• 请理解并分析sched_calss中各个函数指针的用法，并接合Round Robin 调度算法描ucore的调度执行过程

让所有runnable态的进程分时轮流使用CPU时间。RR调度器维护当前runnable进程的有序运行队列。当前进程的时间片用完之后，调度器将当前进程放置到运行队

列的尾部，再从其头部取出进程进行调度。RR调度算法的就绪队列在组织结构上也是一个双向链表，只是增加了一个成员变量，表明在此就绪进程队列中的最大执行时间片。而且在进程控制块proc_struct中增加了一个成员变量time_slice，用来记录进程当前的可运行时间片段。这是由于RR调度算法需要考虑执行进程的运行时间不能太长。在每个timer到时的时候，操作系统会递减当前执行进程的time_slice，当time_slice为0时，就意味着这个进程运行了一段时间，需要把CPU让给其他进程执行，于是操作系统就需要让此进程重新回到rq的队列尾，且重置此进程的时间片为就绪队列的成员变量最大时间片max_time_slice值，然后再从rq的队列头取出一个新的进程执行。

• 请在实验报告中简要说明如何设计实现”多级反馈队列调度算法“，给出概要设计，鼓励给出详细设计

练习2: 实现 Stride Scheduling 调度算法（需要编码）

首先需要换掉RR调度器的实现，即用default_sched_stride_c覆盖default_sched.c。然后根据此文件和后续文档对Stride度器的相关描述，完成Stride调度算法的实现。

1、比较器定义

```
/* You should define the BigStride constant here*/
/* LAB6: 2013011418 */
#define BIG_STRIDE    0x7FFFFFFF /* ??? */

/* The compare function for two skew_heap_node_t's and the
 * corresponding procs*/
static int
proc_stride_comp_f(void *a, void *b)
{
    struct proc_struct *p = le2proc(a, lab6_run_pool);
    struct proc_struct *q = le2proc(b, lab6_run_pool);
    int32_t c = p->lab6_stride - q->lab6_stride;
    if (c > 0) return 1;
    else if (c == 0) return 0;
    else return -1;
}
```

2、运行队列初始化

```
static void
stride_init(struct run_queue *rq) {
    /* LAB6: 2013011418 */
```

```
        list_init(&(rq->run_list));
        rq->lab6_run_pool = NULL;
        rq->proc_num = 0;
    }
```

3、入队操作

```
    static void
    stride_enqueue(struct run_queue *rq, struct proc_struct *proc) {
        /* LAB6: 2013011418 */
    #if USE_SKEW_HEAP
        rq->lab6_run_pool =
            skew_heap_insert(rq->lab6_run_pool, &(proc->lab6_run_pool),
proc_stride_comp_f);
    #else
        assert(list_empty(&(proc->run_link)));
        list_add_before(&(rq->run_list), &(proc->run_link));
    #endif
        if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {
            proc->time_slice = rq->max_time_slice;
        }
        proc->rq = rq;
        rq->proc_num ++;
    }
```

4、出队操作

```
    static void
    stride_dequeue(struct run_queue *rq, struct proc_struct *proc) {
        /* LAB6: 2013011418 */
    #if USE_SKEW_HEAP
        rq->lab6_run_pool =skew_heap_remove(rq->lab6_run_pool, &(proc-
>lab6_run_pool), proc_stride_comp_f);
    #else
        assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
        list_del_init(&(proc->run_link));
    #endif
        rq->proc_num --;
    }
```

5、选择进程调度

```
static struct proc_struct *
stride_pick_next(struct run_queue *rq) {
    /* LAB6: 2013011418 */
#if USE_SKEW_HEAP
    if (rq->lab6_run_pool == NULL) return NULL;
    struct proc_struct *p = le2proc(rq->lab6_run_pool, lab6_run_pool);
#else
    list_entry_t *le = list_next(&(rq->run_list));
    if (le == &rq->run_list)
        return NULL;
    struct proc_struct *p = le2proc(le, run_link);
    le = list_next(le);
    while (le != &rq->run_list)
    {
        struct proc_struct *q = le2proc(le, run_link);
        if ((int32_t)(p->lab6_stride - q->lab6_stride) > 0)
            p = q;
        le = list_next(le);
    }
#endif
    if (p->lab6_priority == 0)
        p->lab6_stride += BIG_STRIDE;
    else p->lab6_stride += BIG_STRIDE / p->lab6_priority;
    return p;
}
```

6.时间片部分同RR算法思想。

```
static void
stride_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    /* LAB6: 2013011418 */
    if (proc->time_slice > 0) {
        proc->time_slice --;
    }
    if (proc->time_slice == 0) {
        proc->need_resched = 1;
    }
}
```

7、定义一个c语言类的实现，提供调度算法的切换接口

```
struct sched_class default_sched_class = {
    .name = "stride_scheduler",
    .init = stride_init,
    .enqueue = stride_enqueue,
    .dequeue = stride_dequeue,
    .pick_next = stride_pick_next,
    .proc_tick = stride_proc_tick,
};
```

运行如下