

# uCore实验六报告

黄郭斌 计科1502班 201507010206

## 一. 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。

## 二. 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

## 三. 实验分析及结果

### 0. 练习零(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

这里简单将我们需要修改的地方罗列如下：

- proc.c
- default\_pmm.c
- pmm.c
- swap\_fifo.c
- vmm.c
- trap.c

然后是一些需要简单修改的部分，根据注释的提示，主要是一下两个函数需要额外加以修改。

### alloc\_proc函数

这里 `alloc_proc` 还需要修改一下，完整的代码如下：

```
static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
        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));
    proc->time_slice = 0;
    proc->lab6_run_pool.left = proc->lab6_run_pool.right = proc->lab6_run_pool.parent =
NULL;
    proc->lab6_stride = 0;
    proc->lab6_priority = 0;
}
return proc;
}123456789101112131415161718192021222324252627

```

相比于lab5，lab6对 `proc_struct` 结构体再次做了扩展，这里主要是多出了以下部分

```

proc->rq = NULL; //初始化运行队列为空
list_init(&(proc->run_link)); //初始化运行队列的指针
proc->time_slice = 0; //初始化时间片
proc->lab6_run_pool.left = proc->lab6_run_pool.right proc->lab6_run_pool.parent = NULL; //初
始化各类指针为空，包括父进程等待
proc->lab6_stride = 0; //步数初始化
proc->lab6_priority = 0; //初始化优先级123456

```

具体的解释见注释。

## trap\_dispatch函数

这里在时钟产生的地方需要对定时器做初始化，修改的部分如下：

```

static void
trap_dispatch(struct trapframe *tf) {
    .....
    .....
    ticks ++;
    assert(current != NULL);
    run_timer_list(); //更新定时器，并根据参数调用调度算法
    break;
    .....
    .....
}

```

### 1. 练习一(使用Round Robin调度算法)

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。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 调度算法描述ucore的调度执行过程 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

Round Robin 调度算法的调度思想是让所有 runnable 态的进程分时轮流使用 CPU 时间。Round Robin 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。

在这个理解的基础上，我们来分析算法的具体实现。这里 Round Robin 调度算法的主要实现在 default\_sched.c 之中，源码如下：

```
static void
RR_init(struct run_queue *rq) {
    list_init(&(rq->run_list));
    rq->proc_num = 0;
}

static void
RR_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    assert(list_empty(&(proc->run_link)));
    list_add_before(&(rq->run_list), &(proc->run_link));
    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 ++;
}

static void
RR_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
    list_del_init(&(proc->run_link));
    rq->proc_num --;
}

static struct proc_struct *
RR_pick_next(struct run_queue *rq) {
    list_entry_t *le = list_next(&(rq->run_list));
    if (le != &(rq->run_list)) {
        return le2proc(le, run_link);
    }
    return NULL;
}

static void
RR_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    if (proc->time_slice > 0) {
        proc->time_slice --;
    }
    if (proc->time_slice == 0) {
        proc->need_resched = 1;
    }
}
```

```

    }
}

struct sched_class default_sched_class = {
    .name = "RR_scheduler",
    .init = RR_init,
    .enqueue = RR_enqueue,
    .dequeue = RR_dequeue,
    .pick_next = RR_pick_next,
    .proc_tick = RR_proc_tick,
};123456789101112131415161718192021222324252627282930313233343536373839404142434445464748495051

```

现在我们来逐个函数的分析，从而了解 Round Robin 调度算法的原理。首先是 `RR_init` 函数，函数比较简单，不再罗列，完成了对进程队列的初始化。

然后是 `RR_enqueue` 函数，

```

static void RR_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    assert(list_empty(&(proc->run_link)));
    list_add_before(&(rq->run_list), &(proc->run_link));
    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 ++;
}123456789

```

看代码，首先，它把进程的进程控制块指针放入到rq队列末尾，且如果进程控制块的时间片为0，则需要把它重置为 `max_time_slice`。这表示如果进程在当前的执行时间片已经用完，需要等到下一次有机会运行时，才能再执行一段时间。然后在依次调整rq和rq的进程数目加一。

然后是 `RR_dequeue` 函数

```

static void
RR_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
    list_del_init(&(proc->run_link));
    rq->proc_num --;
}123456

```

即简单的把就绪进程队列rq的进程控制块指针的队列元素删除，然后使就绪进程个数的proc\_num减一。

接下来是 `RR_pick_next` 函数。

```

static struct proc_struct *RR_pick_next(struct run_queue *rq) {
    list_entry_t *le = list_next(&(rq->run_list));
    if (le != &(rq->run_list)) {
        return le2proc(le, run_link);
    }
    return NULL;
}1234567

```

选取函数，即选取就绪进程队列rq中的队头队列元素，并把队列元素转换成进程控制块指针，即置为当前占用CPU的程序。

最后是

```
static void RR_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    if (proc->time_slice > 0) {
        proc->time_slice --;
    }
    if (proc->time_slice == 0) {
        proc->need_resched = 1;
    }
} 12345678
```

观察代码，即每一次时间片到时的時候，当前执行进程的时间片 `time_slice` 便减一。如果 `time_slice` 降到零，则设置此进程成员变量 `need_resched` 标识为1，这样在下次中断来后执行trap函数时，会由于当前进程成员变量 `need_resched` 标识为1而执行schedule函数，从而把当前执行进程放回就绪队列末尾，而从就绪队列头取出在就绪队列上等待时间最久的那个就绪进程执行。之后是一个对象化，提供一个类的实现，不是重点，这里不做赘述。

## 2. 练习2(实现Stride Scheduling调度算法)

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调度器的相关描述，完成Stride调度算法的实现。后面的实验文档部分给出了Stride调度算法的大体描述。这里给出Stride调度算法的一些相关的资料（目前网上中文的资料比较欠缺）。[strid-shed paper location1](#) [strid-shed paper location2](#) 也可GOOGLE “Stride Scheduling” 来查找相关资料 执行：`make grade`。如果所显示的应用程序检测都输出ok，则基本正确。如果只是priority.c过不去，可执行 `make runpriority` 命令来单独调试它。大致执行结果可看附录。（使用的是 `qemu-1.0.1`）。请在实验报告中简要说明你的设计实现过程

首先，根据实验指导书的要求，先用 `default_sched_stride_c` 覆盖 `default_sched.c`，即覆盖掉 `Round Robin` 调度算法的实现。覆盖掉之后需要在该框架上实现 `Stride Scheduling` 调度算法。关于 `Stride Scheduling` 调度算法，经过查阅资料和实验指导书，我们可以简单的把思想归结如下：

- 1、为每个 `runnable` 的进程设置一个当前状态stride，表示该进程当前的调度权。另外定义其对应的pass值，表示对应进程在调度后，stride 需要进行的累加值。
- 2、每次需要调度时，从当前 `runnable` 态的进程中选择 stride最小的进程调度。对于获得调度的进程P，将对应的stride加上其对应的步长pass（只与进程的优先权有关系）。
- 3、在一段固定的时间之后，回到步骤2，重新调度当前stride最小的进程

参照实验指导书所给的伪代码能够更好的理解该调度算法：



接下来针对代码我们逐步分析，首先完整代码如下：

```
#include <defs.h>
#include <list.h>
#include <proc.h>
#include <assert.h>
#include <default_sched.h>

#define USE_SKEW_HEAP 1
```

```

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;
}

static void
stride_init(struct run_queue *rq) {
    /* LAB6: YOUR CODE */
    list_init(&(rq->run_list));
    rq->lab6_run_pool = NULL;
    rq->proc_num = 0;
}

static void
stride_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    /* LAB6: YOUR CODE */
#ifdef 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 ++;
}

static void
stride_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    /* LAB6: YOUR CODE */
#ifdef 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 --;
}

static struct proc_struct *
stride_pick_next(struct run_queue *rq) {
    /* LAB6: YOUR CODE */

#ifdef 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;
}

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

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,
};

```

首先是初始化函数 `stride_init`。开始初始化运行队列，并初始化当前的运行队，然后设置当前运行队列内进程数目为0。

```
static void
stride_init(struct run_queue *rq) {
    /* LAB6: YOUR CODE */
    list_init(&(rq->run_list));
    rq->lab6_run_pool = NULL;
    rq->proc_num = 0;
} 1234567
```

然后是入队函数 `stride_enqueue`，根据之前对该调度算法的分析，这里函数主要是初始化刚进入运行队列的进程 `proc` 的 `stride` 属性，然后比较队头元素与当前进程的步数大小，选择步数最小的运行，即将其插入放入运行队列中去，这里并未放置在队列头部。最后初始化时间片，然后将运行队列进程数目加一。

```
static void
stride_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    /* LAB6: YOUR CODE */
    #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 ++;
}
```

然后是出队函数 `stride_dequeue`，即完成将一个进程从队列中移除的功能，这里使用了优先队列。最后运行队列数目减一。

```
static void
stride_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    /* LAB6: YOUR CODE */
    #if USE_SKEW_HEAP
        rq->lab6_run_pool = skew_heap_remove(rq->lab6_run_pool, &(proc->lab6_run_pool),
        proc_stride_comp_f); //从优先队列中移除
    #else
        assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
        list_del_init(&(proc->run_link));
    #endif
    rq->proc_num --;
}
```

接下来就是进程的调度函数 `stride_pick_next`，观察代码，它的核心是先扫描整个运行队列，返回其中 `stride` 值最小的对应进程，然后更新对应进程的 `stride` 值，将步长设置为优先级的倒数，如果为0则设置为最大的步长。

```
static struct proc_struct *
stride_pick_next(struct run_queue *rq) {
    /* LAB6: YOUR CODE */
```



```

#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
    //更新对应进程的stride值
    if (p->lab6_priority == 0)
        p->lab6_stride += BIG_STRIDE;
    else p->lab6_stride += BIG_STRIDE / p->lab6_priority;
    return p;
}

```

最后是时间片函数 `stride_proc_tick`，主要工作是检测当前进程是否已用完分配的时间片。如果时间片用完,应该正确设置进程结构的相关标记来引起进程切换。这里和之前实现的 `Round Robin` 调度算法一样，所以不赘述。

还有一个优先队列的比较函数 `proc_stride_comp_f` 的实现，主要思路就是通过步数相减，然后根据其正负比较大大小关系。

```

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; //步数相减，通过正负比较大大小关系
    if (c > 0) return 1;
    else if (c == 0) return 0;
    else return -1;
}

```

另外还有就是一个封装类的实现，也不详细解释了。

### 3. 运行结果

```

mooocos-> make grade
badsegment:          (4.9s)
-check result:          OK
-check output:         WRONG
!! error: missing 'check_slab() succeeded!'

divzero:             (1.5s)
-check result:          OK
-check output:         WRONG
!! error: missing 'check_slab() succeeded!'

softint:             (1.4s)
-check result:          OK
-check output:         WRONG
!! error: missing 'check_slab() succeeded!'

faultread:           (1.5s)
-check result:          OK
-check output:         WRONG
!! error: missing 'check_slab() succeeded!'

faultreadkernel:     (1.5s)
-check result:          OK
-check output:         WRONG
!! error: missing 'check_slab() succeeded!'

hello:               (1.5s)
-check result:          OK
-check output:         WRONG
!! error: missing 'check_slab() succeeded!'

testbss:             (1.7s)
-check result:          OK
-check output:         WRONG
!! error: missing 'check_slab() succeeded!'

pgdir:               (1.5s)
-check result:          OK
-check output:         WRONG
!! error: missing 'check_slab() succeeded!'

yield:               (1.5s)
-check result:          OK
-check output:         WRONG
!! error: missing 'check_slab() succeeded!'

badarg:              (1.4s)
-check result:          OK
-check output:         WRONG
!! error: missing 'check_slab() succeeded!'

exit:                (1.5s)
-check result:          OK
-check output:         WRONG
!! error: missing 'check_slab() succeeded!'

spin:                (2.4s)
-check result:          OK
-check output:         WRONG
!! error: missing 'check_slab() succeeded!'

waitkill:            (3.7s)
-check result:          OK
-check output:         WRONG
!! error: missing 'check_slab() succeeded!'

forktest:            (1.5s)
-check result:          WRONG
!! error: missing 'init check memory pass.'

-check output:         WRONG
!! error: missing 'check_slab() succeeded!'

```

```

forktree:                (1.5s)
-check result:                WRONG
!! error: missing 'init check memory pass.'

-check output:                WRONG
!! error: missing 'check_slab() succeeded!'

matrix:                   (12.1s)
-check result:                WRONG
!! error: missing 'init check memory pass.'

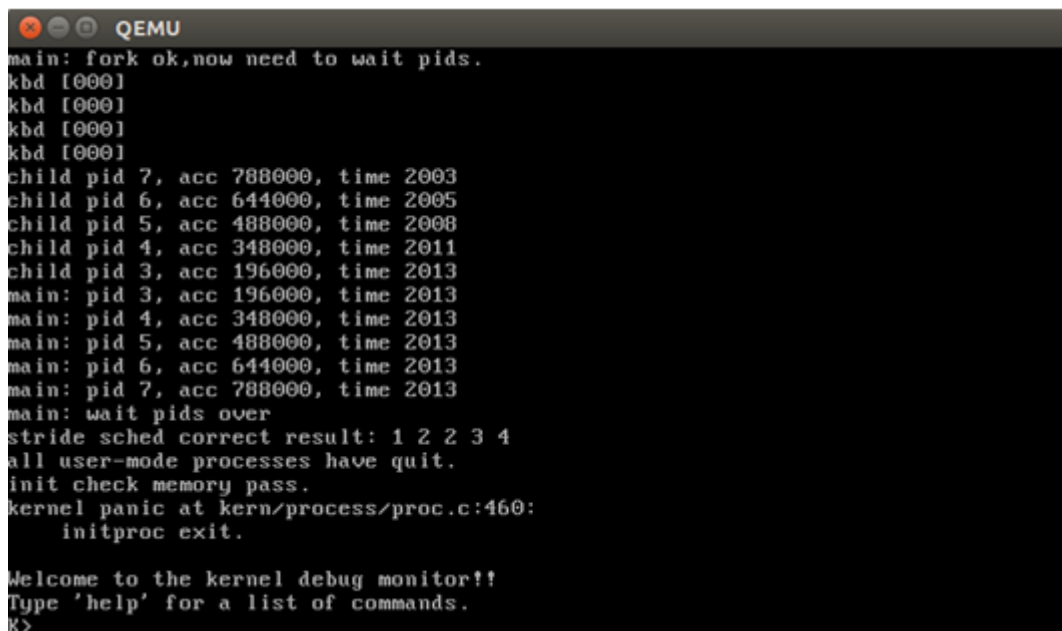
-check output:                WRONG
!! error: missing 'check_slab() succeeded!'

priority:                  (21.5s)
-check result:                WRONG
!! error: missing 'stride sched correct result: 1 2 3 4 5'

-check output:                WRONG
!! error: missing 'check_slab() succeeded!'

Total Score: 91/170
make: *** [grade] Error 1

```



```

QEMU
main: fork ok,now need to wait pids.
kbd [000]
kbd [000]
kbd [000]
kbd [000]
child pid 7, acc 788000, time 2003
child pid 6, acc 644000, time 2005
child pid 5, acc 488000, time 2008
child pid 4, acc 348000, time 2011
child pid 3, acc 196000, time 2013
main: pid 3, acc 196000, time 2013
main: pid 4, acc 348000, time 2013
main: pid 5, acc 488000, time 2013
main: pid 6, acc 644000, time 2013
main: pid 7, acc 788000, time 2013
main: wait pids over
stride sched correct result: 1 2 2 3 4
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:460:
initproc exit.

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>

```

## 四. 实验体会

通过本次实验，让我了解了操作系统底层是如何完成的CPU调度，并通过阅读Round Robin调度算法，亲自实现一个Stride Scheduling，将其安装到UCore操作系统上，从而提高了对CPU调度算法实现层的理解和掌握。