

# OS\_Lab6\_Experimental report

---

湖南大学信息科学与工程学院

计科 210X 甘晴void （学号 202108010XXX）

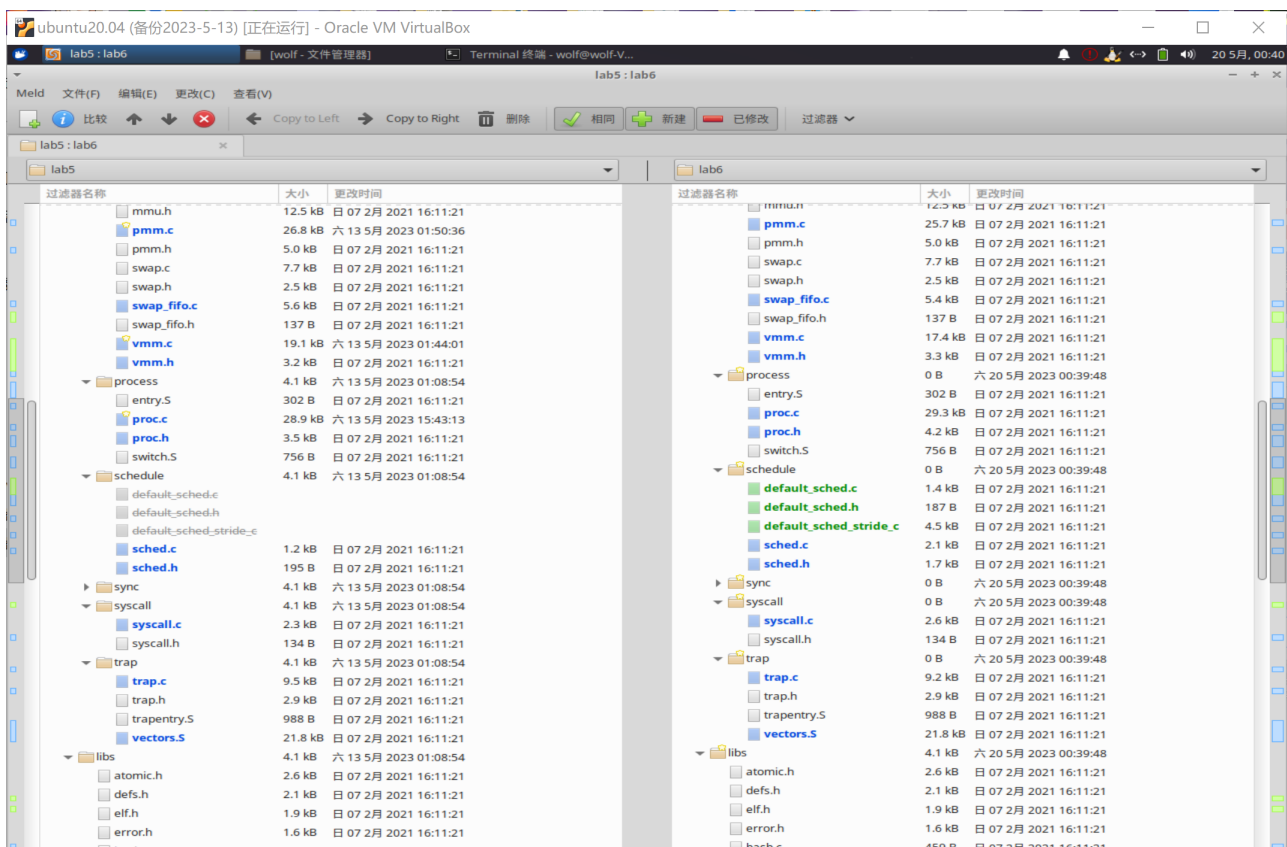
## 实验目的

- 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

## 实验内容

### 练习0：填写已有实验

lab6 会依赖 lab1~lab5，我们需要把做的 lab1~lab5 的代码填到 lab6 中缺失的位置上面。练习 0 就是一个工具的利用。这里我使用的是meld 工具。和 lab5 操作流程一样，我们只需要将已经完成的 lab1~lab5 与待完成的 lab6（由于 lab6 是基于 lab1~lab5 基础上完成的，所以这里只需要导入 lab5）分别导入进来，然后点击 compare 就行了。



对比并进行删改即可。

其他需要修改的地方主要有以下六个文件，通过对比复制完成即可：

- `proc.c`
- `default_pmm.c`
- `pmm.c`
- `swap_fifo.c`
- `vmm.c`
- `trap.c`

根据试验要求，我们需要对部分代码进行改进，这里讲需要改进的地方的代码和说明罗列如下：

- PCT 中增加了三个与 `stride` 调度算法相关的成员变量，以及增加了对应的初始化过程；
- 新增了斜堆数据结构的实现；
- 新增了默认的调度算法 `Round Robin` 的实现，具体为调用 `sched_class_*` 等一系列函数之后，进一步调用调度器 `sched_class` 中封装的函数，默认该调度器为 `Round Robin` 调度器，这是在 `default_sched.*` 中定义的；
- 新增了 `set_priority`, `get_time` 的系统调用；

## proc\_struct 结构体

我们在原来的实验基础上，新增了 9 行代码：

所以改进后的 `proc_struct` 结构体如下：

```
//在os_kernel_lab-master/labcodes/lab6/kern/process/proc.h
struct proc_struct {                                //进程控制块
    enum proc_state state;                          //进程状态
    int pid;                                         //进程ID
    int runs;                                        //运行时间
    uintptr_t kstack;                               //内核栈位置
    volatile bool need_resched;                    //是否需要调度，只对当前进程有效
    struct proc_struct *parent;                    //父进程
    struct mm_struct *mm;                          //进程的虚拟内存
    struct context context;                        //进程上下文
    struct trapframe *tf;                          //当前中断帧的指针
    uintptr_t cr3;                                  //当前页表地址
    uint32_t flags;                                 //进程
    char name[PROC_NAME_LEN + 1];                 //进程名字
    list_entry_t list_link;                        //进程链表
    list_entry_t hash_link;                        //进程哈希表
    //以下为新增的部分
    int exit_code;                                 //退出码(发送到父进程)
    uint32_t wait_state;                           //等待状态
    struct proc_struct *cptr, *yptr, *optr;       //进程间的一些关系
    struct run_queue *rq;                          //运行队列中包含进程
    list_entry_t run_link;                         //该进程的调度链表结构，该结构内部的连接组成了 运行队列 列表
    int time_slice;                                //该进程剩余的时间片，只对当前进程有效
    skew_heap_entry_t lab6_run_pool;               //该进程在优先队列中的节点，仅在 LAB6 使用
    uint32_t lab6_stride;                          //该进程的调度步进值，仅在 LAB6 使用
    uint32_t lab6_priority;                        //该进程的调度优先级，仅在 LAB6 使用
};
```

## alloc\_proc() 函数

我们在原来的实验基础上，新增了 6 行代码：

所以改进后的 `alloc_proc` 函数如下：

```
//在os_kernel_lab-master/labcodes/lab6/kern/process/proc.c下
static struct proc_struct *alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
        proc->state = PROC_UNINIT; //设置进程为未初始化状态
        proc->pid = -1; //未初始化的的进程id为-1
        proc->runs = 0; //初始化时间片
        proc->kstack = 0; //内存栈的地址
        proc->need_resched = 0; //是否需要调度设为不需要
        proc->parent = NULL; //父节点设为空
        proc->mm = NULL; //虚拟内存设为空
        memset(&(proc->context), 0, sizeof(struct context)); //上下文
        //的初始化
        proc->tf = NULL; //中断帧指针置为空
        proc->cr3 = boot_cr3; //页目录设为内核页目录表的基址
        proc->flags = 0; //标志位
        memset(proc->name, 0, PROC_NAME_LEN); //进程名
        proc->wait_state = 0; //PCB 进程控制块中新增的条目，初始化进程等待状
        //态
        proc->cptr = proc->optr = proc->yptr = NULL; //进程相关指针初始
        //化
        //以下为新增部分
        proc->rq = NULL; //初始化运行队列为空
        list_init(&(proc->run_link));
        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; //初始化指针为空
        proc->lab6_stride = 0; //设置步长为 0
        proc->lab6_priority = 0; //设置优先级为 0
    }
    return proc;
}
```

## trap\_dispatch() 函数

我们在原来的实验基础上，新增了 1 行代码：

对进程调度要对进程的时间片情况进行记录，在进程控制块中新定义的time\_slice用来记录剩余时间片长度。而每次发生时钟中断时，都将时间片长度-1。因此修改trap\_dispatch，时钟中断时调用sched\_class\_proc\_tick将进程剩余时间片-1。

所以改进后的 `trap_dispatch` 函数如下：

```
//在os_kernel_lab-master/labcodes/lab6/kern/trap/trap.c下
static void trap_dispatch(struct trapframe *tf) {
    .....
    .....
    ticks ++;
    assert(current != NULL);
    //以下一句为新增句
    sched_class_proc_tick(current); //该函数会调用sched_class中的
    proc_tick
    break;
    .....
    .....
}
```

上面的函数会调用sched\_class中的proc\_tick，该函数会将进程剩余时间片-1。如果时间片已用完，会将进程设置为需要调度。

```
void
sched_class_proc_tick(struct proc_struct *proc) {
    if (proc != idleproc) {
        sched_class->proc_tick(rq, proc);
    }
    else {
        proc->need_resched = 1; //当前进
        程为idle_proc，需要调度运行其他进程
    }
}

//默认轮转调度sched_class中的proc_tick
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; //时间片
    }
}

```

用完，需要进行调度

## 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。

请在实验报告中完成：

- 请理解并分析sched\_class中各个函数指针的用法，并结合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

## 1 算法概述

**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。

## 2 调度器框架

ucore中使用运行队列（**run\_queue**）管理需要调度的进程。在该队列中所有进程都是就绪态，即可以准备运行的状态。当需要选择一个进程进行调度时，从队首拎一个出来运行。运行队列使用一个结构体来表示，其中有一个链表，这个链表链接了所有处于就绪态等待调度的进程，还有一些其他队列相关的信息，如队列进程总数和进程一次调度占用最多的时间片长度。

## run\_queue

```
struct run_queue {
    //运行队列链表
    list_entry_t run_list;
    //优先队列形式的进程容器，步长调度使用
    skew_heap_entry_t *lab6_run_pool;
    //表示队列进程总数
    unsigned int proc_num;
    //每个进程一轮占用的最多时间片
    int max_time_slice;
};
```

为了支持调度，同时在进程控制块内增加了一些变量，保存相关信息。其中就包括运行队列的指针，运行队列链表结点以及剩余时间片长度等，并且在创建进程控制块时初始化，已在练习0中补充。

## skew\_heap\_entry

而 run\_queue 结构体中的 skew\_heap\_entry 结构体如下：

```
struct skew_heap_entry {
    struct skew_heap_entry *parent, *left, *right; //树形结构的进程容器
};
typedef struct skew_heap_entry skew_heap_entry_t;
```

## sched\_class

调度器框架结构 sched\_class 与调度算法无关。

这样可以保证调度算法的通用性以及调度器框架结构对不同调度算法的兼容性。调度器框架中定义了一些调度器接口，通过调度器框架就可以使用调度器的功能。

```
struct sched_class {
    // 调度器名
    const char *name;
    // 初始化运行队列
    void (*init) (struct run_queue *rq);
```

```

// 将进程 p 插入队列 rq
void (*enqueue) (struct run_queue *rq, struct proc_struct
*p);

// 将进程 p 从队列 rq 中删除
void (*dequeue) (struct run_queue *rq, struct proc_struct
*p);

// 返回运行队列中下一个可执行的进程
struct proc_struct* (*pick_next) (struct run_queue *rq);
// 时钟中断时调用，减少进程可用时间，检查是否需要调度
void (*proc_tick)(struct run_queue* rq, struct
proc_struct* p);
};

```

具体使用的调度器框架是在sched\_init这个函数中指定的，在内核初始化时，即在kern\_init中会调用sched\_init，确定一个具体的调度器框架并进行运行队列初始化。

```

static list_entry_t timer_list;
struct run_queue;
void
sched_init(void) {
    list_init(&timer_list); //这个链表并
    //没有被使用，可能是旧的ucore版本使用的
    sched_class = &default_sched_class; //默认为RR调
    //度
    rq = &__rq; //运行队列
    rq->max_time_slice = MAX_TIME_SLICE; //sched.h中
    //定义: #define MAX_TIME_SLICE 5
    sched_class->init(rq); //调用调度器
    //的init函数进行初始化
    cprintf("sched class: %s\n", sched_class->name);
}

```

### 3 算法详解

RR调度主要是通过维护运行队列实现的。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。运行队列run\_queue用一个双向链表将进程连接，并记录了进程运行队列中的最大执行时间片。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长度。通过时间片是否用完决定是否要进行进程调度工作。RR调度算法的各个函数实现如下。

```

//下列函数都在在kern/schedule/default_sched.c下

```



## ①RR\_init

该函数被用于调度算法的初始化。

该函数被封装为 sched\_init 函数。

该函数仅在 ucore 入口的 init.c 里面被调用进行初始化。

```
static void RR_init(struct run_queue *rq) { //初始化进程队列
    list_init(&(rq->run_list)); //初始化运行队列
    rq->proc_num = 0; //初始化进程数为 0
}
```

## ②RR\_enqueue

该函数的功能为将指定的进程的状态置成 RUNNABLE，并且放入调用算法中的可执行队列中，还要设置进程的可用时间片为最大时间片。

具体来说，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。

该函数被封装成 sched\_class\_enqueue 函数。

可以发现这个函数仅在 wakeup\_proc 和 schedule 函数中被调用，前者为将某个不是 RUNNABLE 的进程加入可执行队列，而后者是将正在执行的进程换出到可执行队列中去。

```
static void RR_enqueue(struct run_queue *rq, struct proc_struct
*proc) { //将进程加入就绪队列
    assert(list_empty(&(proc->run_link))); //进程控制块指针非空
    list_add_before(&(rq->run_list), &(proc->run_link)); //把进程的进
程控制块指针放入到 rq 队列末尾
    if (proc->time_slice == 0 || proc->time_slice > rq-
>max_time_slice) { //进程控制块的时间片为 0 或者进程的时间片大于分配给进程的最
大时间片
        proc->time_slice = rq->max_time_slice; //修改时间片
    }
    proc->rq = rq; //加入进程池
    rq->proc_num ++; //就绪进程数加一
}
```

### ③RR\_dequeue

该函数的功能为将某个在队列中的进程取出。

具体来说，它简单地把就绪进程队列 `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。

该函数被封装成函数 `sched_class_dequeue`。

该函数仅在 `schedule` 中被调用，表示将调度算法选择的进程从等待的可执行的进程队列中取出进行执行。

```
static void RR_dequeue(struct run_queue *rq, struct proc_struct
*proc) { //将进程从就绪队列中移除
    assert(!list_empty(&(proc->run_link)) && proc->rq == rq); //进程
    控制块指针非空并且进程在就绪队列中
    list_del_init(&(proc->run_link)); //将进程控制块指针从就绪队列中删除
    rq->proc_num --; //就绪进程数减一
}
```

### ④RR\_pick\_next

该函数功能为选择要执行的下个进程，也叫选取函数。

具体来说，它选取就绪进程队列 `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。

该函数的封装函数仅在 `schedule` 中被调用。

```
static struct proc_struct *RR_pick_next(struct run_queue *rq) { //选
    择下一调度进程
    list_entry_t *le = list_next(&(rq->run_list)); //选取就绪进程队列
    rq 中的队头队列元素
    if (le != &(rq->run_list)) { //取得就绪进程
        return le2proc(le, run_link); //返回进程控制块指针
    }
    return NULL;
}
```

## ⑤RR\_proc\_tick

该函数的功能为减少进程可用时间片。当可用时间片为0时，会设置进程控制块的 `need_resched` 为1，后续在中断处理中会根据这个值决定进行进程调度。

具体来说，它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，这样在下一次中断来后执行 `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。

该函数仅在时间中断的 ISR 中调用。

```
static void RR_proc_tick(struct run_queue *rq, struct proc_struct
*proc) { //时间片
    if (proc->time_slice > 0) { //到达时间片
        proc->time_slice --; //执行进程的时间片 time_slice 减一
    }
    if (proc->time_slice == 0) { //时间片为 0
        proc->need_resched = 1; //设置此进程成员变量 need_resched 标识为
1, 进程需要调度
    }
}
```

## ⑥sched\_class default\_sched\_class（切换接口）

`sched_class` 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 = "RR_scheduler",
    .init = RR_init,
    .enqueue = RR_enqueue,
    .dequeue = RR_dequeue,
    .pick_next = RR_pick_next,
    .proc_tick = RR_proc_tick,
};
```

## 问题1 分析 sched\_class 中各个函数指针的用法

请理解并分析 *sched\_class* 中各个函数指针的用法，并结合 *Round Robin* 调度算法描述 *ucore* 的调度执行过程；

首先我们可以查看一下 *sched\_class* 类中的内容：

```
struct sched_class {
    const char *name; // 调度器的名字
    void (*init) (struct run_queue *rq); // 初始化运行队列
    void (*enqueue) (struct run_queue *rq, struct proc_struct *p); //
    // 将进程 p 插入队列 rq
    void (*dequeue) (struct run_queue *rq, struct proc_struct *p); //
    // 将进程 p 从队列 rq 中删除
    struct proc_struct* (*pick_next) (struct run_queue *rq); // 返回运行
    // 队列中下一个可执行的进程
    void (*proc_tick) (struct run_queue *rq, struct proc_struct *p); //
    // timetick 处理函数
};
```

接下来我们结合具体算法来描述一下 *ucore* 调度执行过程：

- 在 *ucore* 中调用调度器的主体函数（不包括 *init*, *proc\_tick*）的代码仅存在在 *wakeup\_proc* 和 *schedule*，前者的作用在于将某一个指定进程放入可执行进程队列中，后者在于将当前执行的进程放入可执行队列中，然后将队列中选择的下一个执行的进程取出执行；
- 当需要将某一个进程加入就绪进程队列中，则需要将这个进程的能够使用的时间片进行初始化，然后将其插入到使用链表组织的队列的对尾；这就是具体的 *Round-Robin enqueue* 函数的实现；
- 当需要将某一个进程从就绪队列中取出的时候，只需要将其直接删除即可；
- 当需要取出执行的下一个进程的时候，只需要将就绪队列的队头取出即可；
- 每当出现一个时钟中断，则会将当前执行的进程的剩余可执行时间减 1，一旦减到了 0，则将其标记为可以被调度的，这样在 *ISR* 中的后续部分就会调用 *schedule* 函数将这个进程切换出去；

## 问题2 实现“多级反馈队列调度算法”

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；

## 1 算法概述

多级反馈调度队列算法采用多优先级队列，根据进程反馈信息决定进程优先级，根据优先级进行调度。具体的规则如下：

- 如果A的优先级>B的优先级，先运行A
- 如果A的优先级=B的优先级，轮转运行A和B
- 工作进入系统时，放在最高优先级
- 每当工作用完其在某一层的时间配额，就降低其优先级（避免交互型进程独占CPU）
- 每经过一段时间，就将系统中所有工作重新加入最高优先级（避免饥饿）

设计如下：

- 在 `proc_struct` 中添加总共 `N` 个多级反馈队列的入口，每个队列都有着各自的优先级，编号越大的队列优先级越低，并且优先级越低的队列上时间片的长度越大，为其上一个优先级队列的两倍；并且在 `PCB` 中记录当前进程所处的队列的优先级；
- 处理调度算法初始化的时候需要同时对 `N` 个队列进行初始化；
- 在处理将进程加入到就绪进程集合的时候，观察这个进程的时间片有没有使用完，如果使用完了，就将所在队列的优先级调低，加入到优先级低 1 级的队列中去，如果没有使用完时间片，则加入到当前优先级的队列中去；
- 在同一个优先级的队列内使用时间片轮转算法；
- 在选择下一个执行的进程的时候，有限考虑高优先级的队列中是否存在任务，如果不存在才转而寻找较低优先级的队列；（有可能导致饥饿）
- 从就绪进程集合中删除某一个进程就只需要在对应队列中删除即可；
- 处理时间中断的函数不需要改变；

至此完成了多级反馈队列调度算法的具体设计。

下面给出代码实现。

直接对 `default_sched` 进行修改。首先是在 `proc` 结构中增加时间配额 `time_quantum`，并在 `alloc_proc` 中进行初始化。用完时间片会进行进程调度，如果同时用完时间配额，则需要降低优先级。将这个值宏定义在 `default_sched.c` 中，这个值此处设置为 20。每次时间片用完这个值也 -1。

## 2 调度器框架

改动**proc\_struct**结构体和**alloc\_proc**函数

```
//添加时间配额
struct proc_struct {
    .....
    uint32_t time_quantum;
};
//初始化为0
static struct proc_struct *
alloc_proc(void) {
    .....
    proc->time_quantum = 0;
}
return proc;
}
//default_sched.c, 定义时间配额
#define TIME_QUANTUM 20
```

改动**run\_queue**

修改运行队列，设置三个运行列表，数字小的优先级高。

```
struct run_queue {
    list_entry_t run_list_1;
    list_entry_t run_list_2;
    list_entry_t run_list_3;
    unsigned int proc_num;
    int max_time_slice;
    // For LAB6 ONLY
    skew_heap_entry_t *lab6_run_pool;
};
```

其中，相关结构定义与函数如下

```

// 优先队列节点的结构
typedef struct skew_heap_entry skew_heap_entry_t;
// 初始化一个队列节点
void skew_heap_init(skew_heap_entry_t *a);
// 将节点 b 插入至以节点 a 为队列头的队列中去，返回插入后的队列
skew_heap_entry_t *skew_heap_insert(skew_heap_entry_t *a,
                                     skew_heap_entry_t *b,
                                     compare_f comp);
// 将节点 b 插入从以节点 a 为队列头的队列中去，返回删除后的队列
skew_heap_entry_t *skew_heap_remove(skew_heap_entry_t *a,
                                     skew_heap_entry_t *b,
                                     compare_f comp);

```

### 3 算法详解

#### ①RR\_init替换为MLFQ\_init

接下来对调度器函数进行修改，首先修改init函数，对三个队列进行初始化。

```

static void
MLFQ_init(struct run_queue *rq) {
    list_init(&(rq->run_list_1));
    list_init(&(rq->run_list_2));
    list_init(&(rq->run_list_3));
    rq->proc_num = 0;
}

```

#### ②RR\_enqueue替换为MLFQ\_enqueue

对于加入队列的函数，需要根据优先级进行判断。如果为0则为新进程，加入最高优先级队列，其他则判断时间配额是否用完，如果用完，放入低一级的优先队列，并重设配额。

```

static void
MLFQ_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    assert(list_empty(&(proc->run_link)));
    if(proc->lab6_priority == 0){
        proc->lab6_priority = 1;
        proc->time_quantum = TIME_QUANTUM;
        list_add_before(&(rq->run_list_1), &(proc->run_link));
    }
}

```

```

else{
    if(proc->time_quantum == 0){ //配额用
        proc->time_quantum = TIME_QUANTUM; //重设配
        switch(proc->lab6_priority){ //加入低
            case 1:
                proc->lab6_priority = 2;
                list_add_before(&(rq->run_list_2), &(proc-
>run_link));
                break;
            case 3: //3为最
                case 2:
                    proc->lab6_priority = 3;
                    list_add_before(&(rq->run_list_3), &(proc-
>run_link));
                    break;
        }
    }
    else{
        switch(proc->lab6_priority){ //按照优
            case 1:
                list_add_before(&(rq->run_list_1), &(proc-
>run_link));
                break;
            case 2:
                list_add_before(&(rq->run_list_2), &(proc-
>run_link));
                break;
            case 3:
                list_add_before(&(rq->run_list_3), &(proc-
>run_link));
                break;
        }
    }
}
if (proc->time_slice == 0 || proc->time_slice > rq-
>max_time_slice) {
    proc->time_slice = rq->max_time_slice;
}

```

完

额

优先级

低优先级，无法再降低

优先级加入队列



```

    }
    proc->rq = rq;
    rq->proc_num ++;
}

```

### ③RR\_dequeue可以沿用为MLFQ\_dequeue

移出队列不需要改动，直接沿用RR调度的实现。

```

static void
MLFQ_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 --;
}

```

### ④RR\_pick\_next替换为MLFQ\_pick\_next

选择下一个进程需要从优先级高的列表寻找，即从列表1开始寻找，如果为空则寻找低一级中是否存在进程。

```

static struct proc_struct *
MLFQ_pick_next(struct run_queue *rq) {
    list_entry_t *le1 = list_next(&(rq->run_list_1));
    list_entry_t *le2 = list_next(&(rq->run_list_2));
    list_entry_t *le3 = list_next(&(rq->run_list_3));
    if (le1 != &(rq->run_list_1)) {
        return le2proc(le1, run_link);
    }
    else if (le2 != &(rq->run_list_2)){
        return le2proc(le2, run_link);
    }
    else if (le3 != &(rq->run_list_3)){
        return le2proc(le3, run_link);
    }
    return NULL;
}

```

## ⑤RR\_proc\_tick替换为MLFQ\_proc\_tick

时间片减少时需要将配额时间也减小，同时引入trap.c中定义的时钟中断计数，如果时钟中断达到1000，将所有进程放到最高优先级列表。此处重新调整所有进程至最高优先级列表的操作在实现的过程中遇到了问题。暂时不添加，会有饥饿问题产生。

```
#include <trap.h>
extern int ticks;           //引入ticks
static void
MLFQ_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;
    }
    if (proc->time_quantum > 0) {
        proc->time_quantum --; //可用配额-1
    }
    /* 未完成：重新将所有进程加入最高优先级列表 */
}
```

## 实验结果

最后的测试，由于不清楚具体怎样衡量调度算法的效率，此处只验证调度算法可以正常运行。在上述实现中添加了一些输出提示，然后make run-matrix运行用户程序matrix，这个程序会创建比较多的子进程，可以简单测试调度器是否能正常工作。输出大致如下，调度器可以工作。

```
.....
从队列1中选择进程运行
从队列1中选择进程运行
pid 3 is running (1000 times)!.
pid 3 done!.
从队列1中选择进程运行
pid 4 is running (1000 times)!.
.....
配额用完，调整队列
从队列1中选择进程运行
配额用完，调整队列
从队列1中选择进程运行
```

配额用完，调整队列  
从队列2中选择进程运行  
.....

//队列1中的都运行结束或配额用完

## 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。

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 run-priority` 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。

## 覆盖

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。

## 1 算法概述

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：

- 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。

- 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）。
- 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。

## 2 算法详解

接下来针对代码我们逐步分析。

### 新增 **proc\_stride\_comp\_f**

相比于 RR 调度，Stride Scheduling 函数定义了一个比较器 **proc\_stride\_comp\_f**。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小关系。

```
//proc_stride_comp_f: 优先队列的比较函数，主要思路就是通过步数相减，然后根据其
//正负比较大小关系
static int proc_stride_comp_f(void *a, void *b)
{
    struct proc_struct *p = le2proc(a, lab6_run_pool); //通过进程控制
    //块指针取得进程 a
    struct proc_struct *q = le2proc(b, lab6_run_pool); //通过进程控制
    //块指针取得进程 b
    int32_t c = p->lab6_stride - q->lab6_stride; //步数相减，通过正负比
    //较大小关系
    if (c > 0) return 1;
    else if (c == 0) return 0;
    else return -1;
}
```

★如何避免stride溢出的问题：

由于stride是不断增长的，当溢出的时候，便又是从小值开始，如果是这样的话，这意味着更容易被调度，从而会导致错误的调度结果。于是ucore巧妙的利用了整数在机器中的表示，将stride设置成无符号的类型，并且将BIG\_STRIDE设置成有符号整数的最大值，通过指定priority大于1，可以将两个进程的stride之差锁定小于BIG\_STRIDE。然后利用：

```
(int32_t)(p->lab6_stride - q->lab6_stride) > 0
```

便可正确的判断出两进程的stride真实大小关系。

## ①stride\_init

首先是 `stride_init` 函数，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。

```
//stride_init: 进行调度算法初始化的函数，在本 stride 调度算法的实现中使用了斜堆来实现优先队列，因此需要对相应的成员变量进行初始化
static void stride_init(struct run_queue *rq) {
    /* LAB6: YOUR CODE */
    list_init(&(rq->run_list)); //初始化调度器类
    rq->lab6_run_pool = NULL; //对斜堆进行初始化，表示有限队列为空
    rq->proc_num = 0; //设置运行队列为空
}
```

## ②stride\_enqueue

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。

```
//stride_enqueue: 在将指定进程加入就绪队列的时候，需要调用斜堆的插入函数将其插入到斜堆中，然后对时间片等信息进行更新
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); //将新的进程插入到表示就绪队列的斜堆中，该函数的返回结果是斜堆的新的根
    #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; //将该进程剩余时间置为时间片大小
    }
    proc->rq = rq; //更新进程的就绪队列
    rq->proc_num++; //维护就绪队列中进程的数量加一
}
```

```
}
```

里面有一个条件编译：

```
#if USE_SKEW_HEAP
    rq->lab6_run_pool = skew_heap_insert(rq->lab6_run_pool, &
(proc->lab6_run_pool), proc_stride_comp_f); //将新的进程插入到表示就绪队
列的斜堆中，该函数的返回结果是斜堆的新的根
#else
    assert(list_empty(&(proc->run_link)));
    list_add_before(&(rq->run_list), &(proc->run_link));
#endif
```

在 ucore 中 USE\_SKEW\_HEAP 定义为 1，因此 #else 与 #endif 之间的代码将会被忽略。

其中的 skew\_heap\_insert 函数如下：

```
static inline skew_heap_entry_t *skew_heap_insert(skew_heap_entry_t
*a, skew_heap_entry_t *b, compare_f comp)
{
    skew_heap_init(b); //初始化进程b
    return skew_heap_merge(a, b, comp); //返回a与b进程结合的结果
}
```

函数中的 skew\_heap\_init 函数如下：

```
static inline void skew_heap_init(skew_heap_entry_t *a)
{
    a->left = a->right = a->parent = NULL; //初始化相关指针
}
```

函数中的 skew\_heap\_merge 函数如下：

```
static inline skew_heap_entry_t *skew_heap_merge(skew_heap_entry_t
*a, skew_heap_entry_t *b, compare_f comp)
{
    if (a == NULL) return b;
    else if (b == NULL) return a;
```

```

skew_heap_entry_t *l, *r;
if (comp(a, b) == -1) //a进程的步长小于b进程
{
    r = a->left; //a的左指针为r
    l = skew_heap_merge(a->right, b, comp);

    a->left = l;
    a->right = r;
    if (l) l->parent = a;

    return a;
}
else
{
    r = b->left;
    l = skew_heap_merge(a, b->right, comp);

    b->left = l;
    b->right = r;
    if (l) l->parent = b;

    return b;
}
}

```

### ③stride\_dequeue

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。

```

//stride_dequeue: 将指定进程从就绪队列中删除，只需要将该进程从斜堆中删除掉即可
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); //删除斜堆中的指定进程
#else
    assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
    list_del_init(&(proc->run_link));
#endif
    rq->proc_num --; //维护就绪队列中的进程总数
}

```

里面的代码比较简单，只有一个主要函数：skew\_heap\_remove。该函数实现过程如下：

```

static inline skew_heap_entry_t *skew_heap_remove(skew_heap_entry_t
*a, skew_heap_entry_t *b, compare_f comp)
{
    skew_heap_entry_t *p = b->parent;
    skew_heap_entry_t *rep = skew_heap_merge(b->left, b->right,
comp);
    if (rep) rep->parent = p;

    if (p)
    {
        if (p->left == b)
            p->left = rep;
        else p->right = rep;
        return a;
    }
    else return rep;
}

```

#### ④stride\_pick\_next

接下来就是进程的选择调度函数 `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。



```

//stride_pick_next: 选择下一个要执行的进程，根据stride算法，只需要选择stride
值最小的进程，即斜堆的根节点对应的进程即可
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);//选择 stride 值最小的进程
#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)//优先级为 0
        p->lab6_stride += BIG_STRIDE;//步长设置为最大值
    else p->lab6_stride += BIG_STRIDE / p->lab6_priority;//步长设置
为优先级的倒数，更新该进程的 stride 值
    return p;
}

```

## ⑤stride\_proc\_tick

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，不需要改动。

```

//stride_proc_tick: 每次时钟中断需要调用的函数，仅在进行时间中断的ISR中调用
static void stride_proc_tick(struct run_queue *rq, struct
proc_struct *proc) {
    /* LAB6: YOUR CODE */
    if (proc->time_slice > 0) { //到达时间片
        proc->time_slice --; //执行进程的时间片 time_slice 减一
    }
    if (proc->time_slice == 0) { //时间片为 0
        proc->need_resched = 1; //设置此进程成员变量 need_resched 标识
        为 1，进程需要调度
    }
}
}

```

## ⑥ sched\_class default\_sched\_class（切换接口）

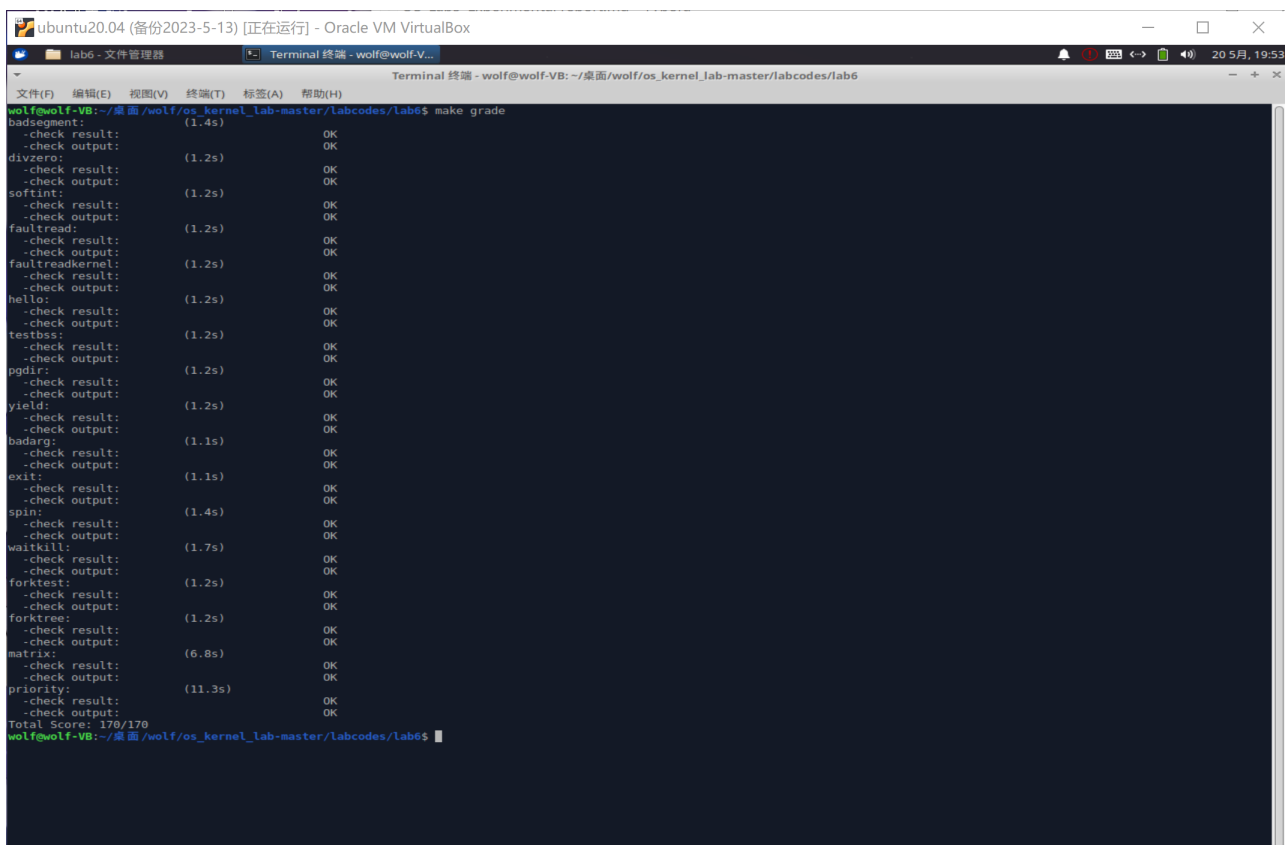
`sched_class` 定义一个 c 语言类的实现，提供调度算法的切换接口。

```

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,
};

```

## 实验结果



```
wolf@wolf-VB: ~/桌面/wolf/os_kernel_lab-master/labcodes/lab6$ make grade
badsegment: (1.4s) OK
-check result: OK
-check output: OK
divzero: (1.2s) OK
-check result: OK
-check output: OK
softint: (1.2s) OK
-check result: OK
-check output: OK
faultread: (1.2s) OK
-check result: OK
-check output: OK
faultreadkernel: (1.2s) OK
-check result: OK
-check output: OK
hello: (1.2s) OK
-check result: OK
-check output: OK
testbss: (1.2s) OK
-check result: OK
-check output: OK
pgdir: (1.2s) OK
-check result: OK
-check output: OK
yield: (1.2s) OK
-check result: OK
-check output: OK
badarg: (1.1s) OK
-check result: OK
-check output: OK
exit: (1.1s) OK
-check result: OK
-check output: OK
spin: (1.4s) OK
-check result: OK
-check output: OK
waitkill: (1.7s) OK
-check result: OK
-check output: OK
forktest: (1.2s) OK
-check result: OK
-check output: OK
forktree: (1.2s) OK
-check result: OK
-check output: OK
matrix: (6.8s) OK
-check result: OK
-check output: OK
priority: (11.3s) OK
-check result: OK
-check output: OK
Total Score: 170/170
wolf@wolf-VB: ~/桌面/wolf/os_kernel_lab-master/labcodes/lab6$
```

可知正确。

## Challenge 1：实现 Linux 的 CFS 调度算法

在ucore的调度器框架下实现下Linux的CFS调度算法。可阅读相关Linux内核书籍或查询网上资料，可了解CFS的细节，然后大致实现在ucore中。

### 1 算法概述

CFS能在真实硬件上模拟出一种“公平的、精确的任务多处理CPU”。

1. 公平，即对于n个正在运行的任务，当这些任务同时不断地运行时，CPU会尽可能分配给他们1/n的处理时间。CFS是一种基于加权公平排队思想的调度算法。
2. 精确，指的是它采用红黑树作为调度的任务队列的数据结构。

CFS 算法的基本思路就是尽量使得每个进程的运行时间相同，所以需要记录每个进程已经运行的时间：

## 红黑树

红黑树是一种特殊的二叉搜索树，也就是左边节点都小于根节点都小于右边节点，递归整个树都满足这一点。也就是说最左边的叶子节点是最小的，最右边的叶子节点是最大的。红黑树相比二叉搜索树多了红色黑色两个颜色的宏定义[1]，红黑树有以下5个性质：

- 每个结点要么是红的要么是黑的。
- 根结点是黑的。
- 每个叶结点都是黑的。
- 如果一个结点是红的，那么它的两个儿子都是黑的。
- 对于任意结点而言，其到叶结点的每条路径都包含相同数目的黑结点。

## CFS主要实现

- CFS使用红黑树结构，来存储要调度的任务队列。
- 每个节点代表了一个要调度的任务，节点的key即为虚拟时间（vruntime），虚拟时间由这个人物的运行时间计算而来。
- key越小，也就是vruntime越小的话，红黑树对应的节点就越靠左。
- CFS scheduler每次都挑选最左边的节点作为下一个要运行的任务，\*\*这个节点是“缓存的”——由一个特殊的指针指向；不需要进行 $O(\log n)$ 遍历来查找。也因此，CFS搜索的时间是 $O(1)$ 。

## 2 调度器框架

### proc\_struct

每次调度的时候，选择已经运行时间最少的进程。所以，也就需要一个数据结构来快速获得最少运行时间的进程，CFS 算法选择的是红黑树，但是项目中的斜堆也可以实现，只是性能不及红黑树。CFS是对于优先级的实现方法就是让优先级低的进程的时间过得很快。

### 运行队列

首先需要在 run\_queue 增加一个斜堆：

```
struct run_queue {  
    ...  
    skew_heap_entry_t *fair_run_pool;  
};
```

## 调度器框架

在 `proc_struct` 中增加三个成员：

- 虚拟运行时间
- 优先级系数：从 1 开始，数值越大，时间过得越快
- 斜堆

```
struct proc_struct {  
    ...  
    int fair_run_time;           // FOR CFS ONLY:  
run time  
    int fair_priority;          // FOR CFS ONLY:  
priority  
    skew_heap_entry_t fair_run_pool; // FOR CFS ONLY:  
run pool  
};
```

将 `proc` 初始化时将增添的三个变量一并初始化：

```
skew_heap_init(&(proc->fair_run_pool));  
proc->fair_run_time = 0;  
proc->fair_priority = 1;
```

## 3 算法实现

### `proc_fair_comp_f`

首先需要一个比较函数，同样根据

$MAX_{RUNTIME} - MIN_{RUNTIME} < MAX_{PRIORITY}$  完全不需要考虑虚拟运行时溢出的问题。

```
static int proc_fair_comp_f(void *a, void *b)
{
    struct proc_struct *p = 1e2proc(a, fair_run_pool);
    struct proc_struct *q = 1e2proc(b, fair_run_pool);
    int32_t c = p->fair_run_time - q->fair_run_time;
    if (c > 0) return 1;
    else if (c == 0) return 0;
    else return -1;
}
```

### ① fair\_init

```
static void fair_init(struct run_queue *rq) {
    rq->fair_run_pool = NULL;
    rq->proc_num = 0;
}
```

### ② fair\_enqueue

和 Stride Scheduling 类似，但是不需要更新 stride。

```
static void fair_enqueue(struct run_queue *rq, struct proc_struct
*proc) {
    rq->fair_run_pool = skew_heap_insert(rq->fair_run_pool, &(proc-
>fair_run_pool), proc_fair_comp_f);
    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 ++;
}
```

### ③ fair\_dequeue

```
static void fair_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    rq->fair_run_pool = skew_heap_remove(rq->fair_run_pool, &(proc->fair_run_pool), proc_fair_comp_f);
    rq->proc_num --;
}
```

#### ④fair\_pick\_next

```
static struct proc_struct * fair_pick_next(struct run_queue *rq) {
    if (rq->fair_run_pool == NULL)
        return NULL;
    skew_heap_entry_t *le = rq->fair_run_pool;
    struct proc_struct * p = le2proc(le, fair_run_pool); //取得进程控制块指针并返回
    return p;
}
```

#### ⑤fair\_proc\_tick

需要更新虚拟运行时，增加量为优先级系数。

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

兼容调整

为了保证测试可以通过，需要将 Stride Scheduling 的优先级对应到 CFS 的优先级：

```

void lab6_set_priority(uint32_t priority)
{
    ...
    // FOR CFS ONLY
    current->fair_priority = 60 / current->lab6_priority + 1;
    if (current->fair_priority < 1)
        current->fair_priority = 1;
}

```

由于调度器需要通过虚拟运行时间确定下一个进程，如果虚拟运行时间最小的进程需要 `yield`，那么必须增加虚拟运行时间，例如可以增加一个时间片的运行时。

```

int do_yield(void) {
    ...
    // FOR CFS ONLY
    current->fair_run_time += current->rq->max_time_slice *
current->fair_priority;
    return 0;
}

```

遇到的问题：为什么 **CFS** 调度算法使用红黑树而不使用堆来获取最小运行时进程？

查阅了网上的资料以及自己分析，得到如下结论：

- 堆基于数组，但是对于调度器来说进程数量不确定，无法使用定长数组实现的堆；
- `ucore` 中的 *Stride Scheduling* 调度算法使用了斜堆，但是斜堆没有维护平衡的要求，可能导致斜堆退化成为有序链表，影响性能。

综上所述，红黑树因为平衡性以及非连续所以是 **CFS** 算法最佳选择。

- 堆基于数组，但是对于调度器来说进程数量不确定，无法使用定长数组实现的堆；
- `ucore` 中的 **Stride Scheduling** 调度算法使用了斜堆，但是斜堆没有维护平衡的要求，可能导致斜堆退化成为有序链表，影响性能。

综上所述，红黑树因为平衡性以及非连续所以是 **CFS** 算法最佳选择。



## Challenge 2：实现尽可能多的各种基本调度算法

在ucore上实现尽可能多的各种基本调度算法(FIFO, SJF,...)，并设计各种测试用例，能够定量地分析出各种调度算法在各种指标上的差异，说明调度算法的适用范围。

### 实验总结

通过本次实验对RR调度和Stride调度有了更深入的学习与理解，通过验收以及助教老师的提问对这部分内容掌握的更加牢固，这两个调度算法的实现都基于调度类五元组：初始化、入队、出队、选择下一个、中断处理。区别就在于Stride基于比较步长和进程执行进度的思想，要求频繁比较Stride值，因此选用了适应斜堆的函数，就代码而言，差别不大。相信本次实验的内容与收获会对今后的学习起到很好的帮助。

### 实验扩展

#### 1.CPU资源的时分复用

- 进程切换：CPU资源的当前占用者切换
  - 保存当前进程在PCB中的执行上下文（CPU状态）
  - 恢复下一个进程的执行上下文
- 处理机调度
  - 从就绪队列中挑选下一个占用CPU运行的进程。
  - 从多个可用CPU中挑选就绪进程可使用的CPU资源。
- 调度程序：挑选就绪进程的内核函数
  - 调度策略：依据什么原理挑选进程/线程
  - 调度时机：什么时候进行调度
    - 内核运行调度程序的条件
      - 进程从运行状态切换到等待状态
      - 进程被终结了
    - 非抢占系统：当前进程主动放弃CPU时
    - 可抢占系统
      - 中断请求被服务例程响应完成时

- 当前进程被抢占
  - 进程的时间片耗尽
  - 进程从等待状态切换到就绪状态

## 2.调度准则

- 比较调度算法的准则
  - CPU使用率：CPU处于忙状态的时间百分比
  - 吞吐量：单位时间内完成的进程数量
  - 周转时间：进程从初始化到结束（包括等待）的总时间
  - 等待时间：进程在就绪队列中的总时间
  - 响应时间：从提交请求到产生响应所花费的总时间
- 调度策略的目标
  - 减少响应时间：及时处理用户的输入，尽快将输出反馈给用户
  - 减少平均响应时间的波动：在交互系统中，可预测性比高差异低平均更重要。
    - 低延迟调度改善用户的交互体验。
    - 响应时间是操作系统的计算延迟。
- 调度策略的吞吐量目标
  - 增加吞吐量
    - 减小开销（例如上下文切换的开销）
    - 系统资源的高效利用（例如CPU和IO设备的并行使用）
  - 减少每个进程的等待时间
  - 保证吞吐量不受用户交互的影响
  - 吞吐量是操作系统的计算带宽。
- 调度的公平性目标
  - 保证每个进程占用相同的CPU时间
  - 保证每个进程的等待时间相同
  - 公平通常会增加平均响应时间

## 3.调度算法

## a. 先来先服务算法（First Come First Served, FCFS）

依据进程进入就绪状态的先后顺序排序

- 优点：简单
- 缺点：
  - 平均等待时间波动较大（短进程可能排在长进程后面）
  - IO资源和CPU资源的利用效率可能较低
  - CPU密集型进程会导致IO设备闲置时，IO密集型进程也在等待。（CPU和IO设备可并行执行）

## b. 短进程优先算法（SPN）

选择就绪队列中执行时间最短进程占用的CPU进入运行状态。就绪队列按预期的执行时间来排序。

- 优点：短进程优先算法具有最优平均周转时间。
- 缺点：
  - 可能导致饥饿。例如连续的短进程流会使长进程无法获得CPU资源。
  - 需要预估下一个CPU计算的持续时间
  - 一种方法是，用历史执行时间预估未来执行时间

短剩余时间优先算法（SRT）：SPN算法的可抢占改进

## c. 最高响应比优先算法（HRRN）

选择就绪队列中响应比 $R$ 值最高的进程

其中 $R=(w+s)/s$ ， $s$ ：执行时间； $w$ ：等待时间

- 在短进程优先算法基础上的改进
- 不可抢占
- 关注进程的等待时间
- 防止无限期推迟

#### d. 时间片轮转算法（RR, Round-Robin）

- 时间片：分配处理机资源的基本时间单位
- 算法思路：
  - 时间片结束时，按FCFS算法切换到下一个就绪进程。
  - 每隔 $n-1$ 个时间片，进程执行一个时间片。
- 时间片长度选择
  - 时间片长度过长，则等待时间太长，极端情况下退化成FCFS。
  - 时间片长度过短，则反应较为迅速，但产生大量进程上下文切换，影响系统吞吐量。
  - 需要选择一个合适的时间片长度，以维持上下文切换开销处于1%状态。

#### e. 多级队列调度算法（MQ）

- 就绪队列被划分为多个独立的子队列，每个队列拥有自己的调度策略
- 队列间的调度
  - 固定优先级。例如先处理前台，后处理后台。但可能会导致饥饿。
  - 时间片轮转。每个队列都得到一个确定的能够调度其进程的CPU总时间。
  - 例如80%CPU时间用于前台，20%CPU时间用于后台。

#### f. 多级反馈队列算法（MLFQ）

- 进程可在不同队列间移动的多级队列算法。
- - 时间片大小随优先级级别的增加而增加。
  - 例如进程在当前时间片内没有完成，则降到下一个优先级。
- 特征：CPU密集型进程优先级下降的很快，IO密集型进程停留在高优先级。

#### g. 公平共享调度（FSS, Fair Share Scheduling）

FSS控制用户对系统资源的访问

- 一些用户组比其他用户组更重要。
- 保证不重要的组无法垄断资源
  - 未使用的资源按比例分配
  - 没有达到资源使用率目标的组获得更高的优先级。

## 4.实时操作系统

- 实时操作系统的定义：正确性依赖于其时间和功能两方面的操作系统
- 实时操作系统的性能指标：
  - 时间约束的及时性（**deadline**）
  - 速度和平均性能相对不重要
- 实时操作系统的特性：时间约束的可预测性
- 实时任务：
  - 任务：一次计算/文件读取/信息传递等等。
  - 任务属性：完成任务所需的资源以及定时参数。
- 周期实时任务：一系列相似的任务
  - 任务有规律的重复
  - 周期 $p$  = 任务请求间隔( $0 < p$ )
  - 执行时间 $e$  = 最大执行时间( $0 < e < p$ )
  - 使用率 $U = e/p$
- 软时限和硬时限
  - 硬时限（**hard deadline**）
    - 错过任务时限将会导致灾难性或非常严重的后果
    - 必须验证，在最坏的情况下能够满足时限
  - 软时限（**soft deadline**）
    - 通常能满足任务时限。如有时不能满足，则降低要求
    - 尽力保证满足任务时限。
- 实时调度
  - 速率单调调度算法（**RM, Rate Monotonic**）
    - 通过周期安排优先级
    - 周期越短优先级越高
    - 执行周期越短的任务。
  - 最早截止时间优先算法（**EDF, Earliest Deadline First**）
    - 截止时间越早优先级越高
    - 执行截止时间最早的任务

## 5.多处理器调度

- 多处理器调度的特征
  - 多个处理机组成一个多处理系统
  - 处理机间可负载共享
- 对称多处理器（**SMP, Symmetric multiprocessing**）调度
  - 每个处理器运行自己的调度程序

- 调度程序对共享资源的访问需要进行同步
- 对称多处理器的进程分配
  - 静态进程分配
    - 进程从开始到结束都被分配到一个固定的处理机上执行
    - 每个处理机都有自己的就绪队列
    - 调度开销小
    - 各处理机可能忙闲不均（例如一核工作，七核在看 XD）
  - 动态进程分配
    - 进程在执行中可分配到任意空闲处理机执行
    - 所有处理机共享一个公共的就绪队列
    - 调度开销大
    - 各处理机的负载是均衡的

## 6. 优先级反置

优先级反置（*Priority Inversion*），是操作系统中出现的高优先级进程长时间等待低优先级进程所占用的资源的现象。

基于优先级的可抢占调度算法存在优先级反置。

- 优先级继承（*Priority Inheritance*）
  - 占用资源的低优先级进程继承申请资源的高优先级进程的优先级。
  - 只在占有资源的低优先级进程被阻塞时，才提高占有资源进程的优先级。
- 优先级天花板协议（*Priority ceiling protocol*）
  - 占用资源进程的优先级和所有可能的申请该资源的进程的最高优先级相同。
  - 不管是否发生等待，都提升占用资源进程的优先级。
  - 优先级高于系统中所有被锁定的资源的优先级上限，任务执行临界区时就不会被阻塞。

## 参考资料

- [Stride Scheduling: Deterministic Proportional-Share Resource Management \(1995\)](#)
- [G53OPS : Process Scheduling](#)
- [Linux 2.6 Completely Fair Scheduler 内幕](#)
- [斜堆 - 维基百科，自由的百科全书](#)
- [https://github.com/AngelKitty/review\\_the\\_national\\_post-graduate\\_entrance\\_examination/tree/master/books\\_and\\_notes/professional\\_courses/operating\\_system/s](https://github.com/AngelKitty/review_the_national_post-graduate_entrance_examination/tree/master/books_and_notes/professional_courses/operating_system/s)

ources/ucore\_os\_lab/docs/lab\_report

- 调度算法部分<https://blog.csdn.net/Aaron503/article/details/125039019>
- 有关CFS的实现<https://zhuanlan.zhihu.com/p/372441187>
- 实验总结部分[https://blog.csdn.net/qq\\_51684393/article/details/124992574](https://blog.csdn.net/qq_51684393/article/details/124992574)
- 实验总结部分<https://blog.csdn.net/JustinAustin/article/details/123429205>