Lab6实验报告

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：已有实验代码改进

1.proc_struct

为了支持Lab6中实现的调度算法，进程控制块中需要添加新的变量来记录相关信息。proc_struct中补充的变量如下：

```
struct proc_struct {
    ......
    struct run_queue *rq;                       // running queue contains Process
    list_entry_t run_link;                      // the entry linked in run queue
    int time_slice;                             // time slice for occupying the
CPU
    /* 以下仅在步长调度使用 */
    skew_heap_entry_t lab6_run_pool;            // FOR LAB6 ONLY: the entry in the
run pool
    uint32_t lab6_stride;                       // 步长
    uint32_t lab6_priority;                     // FOR LAB6 ONLY: the priority of
process, set by lab6_set_priority(uint32_t)
};
```

- rq：run_queue是定义在sche.h中的结构，用于维护运行队列，rq是运行队列的指针
- run_link：该进程在运行队列的链表中的结点
- time_slice：进程剩余时间片
- lab6_run_pool：skew_heap是为了实现步长调度实现的优先队列，该变量为进程在优先队列中的结点
- lab6_stride：进程行程值
- lab6_priority：进程优先级

2.alloc_proc

对应proc_struct新增的成员变量，分配进程控制块时要进行初始化。补充alloc_proc如下：

```
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
        .......
```

```
        proc->rq = NULL;//初始化运行队列为空
        proc->run_link.prev = proc->run_link.next = NULL;//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;//进程运行进度初始化（针对于stride调度算法，下同）
        proc->lab6_priority = 0;      //初始化优先级
    }
    return proc;
}
```

其中lab6_run_pool的初始化是根据skew_heap_entry的结构进行的：

```
//skew_heap.h
struct skew_heap_entry {
     struct skew_heap_entry *parent, *left, *right;
};
```

3.trap_dispatch

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。

```
......
case IRQ_OFFSET + IRQ_TIMER:
        ticks ++;
        assert(current != NULL);
        sched_class_proc_tick(current);
        break;
......
```

这个函数会调用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

1.调度器框架

运行队列

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，就从运行队列中进行选择。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。

```
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，进程控制块增加了一些变量，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。

调度器框架

ucore实现了一个与调度算法无关的调度器框架结构sched_class，以保证调度算法的通用性。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);
};
```

这些函数指针使用情况如下：

- init：只在操作系统启动，指定具体调度器框架时调用，完成对运行队列的初始化
- enqueue：将进程插入运行队列。一共有两种情况会将进程插入运行队列：
    - do_fork创建子进程结束时，调用wakeup_proc，将子进程设置为可运行状态（PROC_RUNNABLE），并将其插入运行队列。
    - 当一个进程用完了可用时间片，在调度器进行调度时（调用schedule），会调度其他进程运行，将该进程插入运行队列
- dequeue：进程被调度时，将进程从运行队列中删除（运行队列只保存就绪态的进程）
- pick_next：返回运行队列中下一个将要执行的进程，这个进程是根据调度策略进行选择的
- proc_tick：时钟中断时将进程的可用时间片-1，并判断时间片是否用完，是否需要调度器进行调度

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);                             //定时器队列，本实验没有使用
    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);
}
```

2.RR调度算法

RR调度算法是让所有状态为PROC_RUNNABLE的进程轮流使用CPU时间。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。

RR_init

是初始化环节，初始化rq的进程队列，并将其进程数量置零。

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

RR_enqueue

是一个进程入队的操作：进程队列是一个双向链表，一个进程加入队列的时候，会将其加入到队列的第一位，并给它初始数量的时间片；并更新队列的进程数量。

```
static void
RR_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    assert(list_empty(&(proc->run_link)));                                  //确定
进程不在链表中
    list_add_before(&(rq->run_list), &(proc->run_link));                    //加入
运行队列链表
    if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {
        proc->time_slice = rq->max_time_slice;                              //如果
进程时间片用完，将其重置为max_time_slice
    }
    proc->rq = rq;
    rq->proc_num ++;
}
```

RR_dequeue

从就绪队列中取出这个进程，并将其调用list_del_init删除。同时，进程数量减一。

```
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));//将proc删除
    rq->proc_num --;
}
```

RR_pick_next

通过list_next函数的调用，会从队尾选择一个进程，代表当前应该去执行的那个进程。如果选不出来有处在就绪状态的进程，那么返回NULL，并将执行权交给内核线程idle，idle的功能是不断调用schedule，直到整个系统出现下一个可以执行的进程。

```
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;
}
```

RR_proc_tick

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是否进行进程调度，调用schedule进行进程调度。

```
static void
RR_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    if (proc->time_slice > 0) {
        proc->time_slice --;//当前执行进程的时间片time_slice减一
    }
    if (proc->time_slice == 0) {
        proc->need_resched = 1;                                                    //
放弃对于CPU的占有，需要将别的进程调度进来执行，而当前进程需要等待
    }
}
```

在schedule中将当前进程的时间片重设，加入运行队列，然后调用pick_next选出要调度运行的进程，调用proc_run运行该进程。就完成了一次进程调度。

```
void
schedule(void) {
    bool intr_flag;
    struct proc_struct *next;
    local_intr_save(intr_flag);
    {
        current->need_resched = 0;
        if (current->state == PROC_RUNNABLE) {
            sched_class_enqueue(current);                    //当前进程加入运行队列
        }
        if ((next = sched_class_pick_next()) != NULL) {
            sched_class_dequeue(next);                       //选出调度进程并从运行队
列删除
        }
        if (next == NULL) {
            next = idleproc;
        }
        next->runs ++;
        if (next != current) {
            proc_run(next);                                  //调度运行新进程
        }
    }
```

```
    local_intr_restore(intr_flag);
}
```

3.调度过程

- ucore调用sched_init函数用于初始化相关的就绪队列。
- 在proc_init函数中，建立第一个内核进程，并将其添加至就绪队列中
- 当所有的初始化完成后，ucore执行cpu_idle函数，并在其内部的schedule函数中，调用sched_class_enqueue将当前进程添加进就绪队列中（因为当前进程要被切换出CPU了）
- 然后，调用sched_class_pick_next获取就绪队列中可被轮换至CPU的进程。如果存在可用的进程，则调用sched_class_dequeue函数，将该进程移出就绪队列，并在之后执行proc_run函数进行进程上下文切换。
- 需要注意的是，每次时间中断都会调用函数sched_class_proc_tick。该函数会减少当前运行进程的剩余时间片。如果时间片减小为0，则设置need_resched为1，并在时间中断例程完成后，在trap函数的剩余代码中进行进程切换。

4.多级反馈队列调度算法

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）

实现多级反馈队列调度的大致思路如下：

- 创建多个运行队列，并在进程控制块中记录进程所在队列的优先级，初始化为0
- 定义时间配额，在进程控制块中进行记录，用于调整优先级
- 将新的进程(根据优先级初始化为0判断是否为新进程)加入运行队列时，加入最高优先级的运行队列
- 将时间片用完的进程加入运行队列时，判断配额是否用完，若用完则加入低优先级队列
- 选择将要运行的进程时，先在高优先级队列寻找是否有进程等待运行，没有则在同级别优先队列按照轮转调度选择下一个要运行的进程

接下来给出具体的实现，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。

```
//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
```

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;
};
```

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;
}
```

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){                        //配额用完
            proc->time_quantum = TIME_QUANTUM;              //重设配额
            switch(proc->lab6_priority){                    //加入低优先级
```

```
                case 1:
                    proc->lab6_priority = 2;
                    list_add_before(&(rq->run_list_2), &(proc->run_link));
                    break;
                case 3:                                         //3为最低优先级，无法再
降低
                case 2:
                    proc->lab6_priority = 3;
                    list_add_before(&(rq->run_list_3), &(proc->run_link));
                    break;
            }
        }
        else{
            switch(proc->lab6_priority){                        //按照优先级加入队列
                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;
    }
    proc->rq = rq;
    rq->proc_num ++;
}
```

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 --;
}
```

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;
}
```

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。

```
#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 */
}
```

练习2: 实现 Stride Scheduling 调度算法

1.步长调度算法

RR调度下，进程轮流使用cpu资源，是一种公平的调度。为了提高工作效率，有时需要根据进程的优先级分配cpu资源的使用，使高优先级的进程占用更多的cpu资源。步长调度是基于这种需求实现的，调度规则大致如下：

- 为每个runnable的进程设置一个当前行程值，表示该进程当前的调度权。定义对应的步长值，表示对应进程在调度后，行程值需要进行的累加值。
- 每次需要调度时，从当前就绪态的进程中选择行程值最小的进程调度。
- 对于获得调度的进程，将对应的行程值加上步长。
- 在一段固定的时间之后，重新调度当前行程值最小的进程。

2.优先级队列

在步长调度选择将要运行的进程时，需要找到步长最小的进程，为了避免遍历队列找到这个进程，ucore提供了优先级队列，使用该队列结构来保存就绪态进程，就可以在选择时快速取出步长最小的进程。结构定义以及主要使用的函数如下：

```
// 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);
```

其中优先队列的顺序是由比较函数comp决定的，sched_stride.c中提供了proc_stride_comp_f比较器用来比较两个stride的大小，可以直接使用。

3.步长调度算法实现

在ucore中实现步长调度算法，只需要补全调度器框架中对应的函数，将初始化时指定的调度器框架修改为步长调度算法框架。进程控制块中已经包含了优先队列结点和行程值记录变量以及进程优先级，分配进程控制块时也已经做好了初始化工作。在步长调度算法中，是进程优先级决定了步长，从而影响cpu资源的分配，在本实验中增加了一个系统调用sys_lab6_set_priority，可以修改进程控制块中的priority变量，指定进程的优先级。

proc_stride_comp_f

通过comp函数，实现比较当前stride最小的进程，从而去调度它，具体实现如下：

```
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，通过正负比较大小关系
     if (c > 0) return 1;
     else if (c == 0) return 0;
     else return -1;
}
```

stride_init

初始化运行队列，对运行队列进行初始化，将进程数设置为0。

```
static void
stride_init(struct run_queue *rq) {
    list_init(&(rq->run_list));                    //初始化调度器类的信息
    rq->lab6_run_pool = NULL;                          //初始化当前的运行队列为一个
空的容器结构。
    rq->proc_num = 0;
    return;
}
```

stride_enqueue

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) {
    //将proc对应的堆插入到rq中
    rq->lab6_run_pool = skew_heap_insert(rq->lab6_run_pool, &(proc-
>lab6_run_pool), proc_stride_comp_f);
   if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {
        proc->time_slice = rq->max_time_slice;          //重设可用时间片
   }
   //将proc的rq指向总的rq，rq中的进程数+1
   proc->rq = rq;
   rq->proc_num++;
   return;
}
```

stride_dequeue

将进程移出运行队列，并将队列进程数-1。

```
static void
stride_dequeue(struct run_queue *rq, struct proc_struct *proc) {
      rq->lab6_run_pool = skew_heap_remove(rq->lab6_run_pool, &(proc-
>lab6_run_pool), proc_stride_comp_f);
      rq->proc_num--;
      return;
}
/*
```

stride_pick_next

选出行程值最小的进程进行调度运行，并将该进程的行程值加上步长(BIG_STRIDE/prority)。选择进程时首先要保证优先队列非空，如果非空，则直接取出队列首的进程。还需要特别注意优先级priority，因为初始化时设置为0，这里必须判断优先级是否被设置，如果还是初始化时的0，直接给行程值加上最大步长，避免除0错误。

```
static struct proc_struct *
stride_pick_next(struct run_queue *rq) {
    if(rq->lab6_run_pool != NULL){
        struct proc_struct proc = le2proc(rq->lab6_run_pool);           //选出进程
        if(proc->lab6_priority == 0){
            proc->lab6_pass += BIG_STRIDE;                              //没有
设置则直接加最大步长
        else
            proc->lab6_pass += BIG_STRIDE/proc->lab6_priority;          //行程
值 += 步长
        return proc;
    }
    return NULL;
}
```

stride_proc_tick

减小时间片，实现与RR调度的相同，不需要改动。

```
static void
stride_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;
    }
}
```

4.步长调度测试

将原来的RR调度算法在default_sched.c中替换掉就可以使用步长调度算法进行调度了。

```
     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,
};
```

make run-priority运行程序进行测试，结果符合预期。

```
check_swap() succeeded!
++ setup timer interrupts
kernel_execve: pid = 2, name = "priority".
main: fork ok,now need to wait pids.
child pid 7, acc 648000, time 2003
child pid 6, acc 540000, time 2007
child pid 5, acc 408000, time 2011
child pid 4, acc 288000, time 2014
child pid 3, acc 164000, time 2015
main: pid 3, acc 164000, time 2016
main: pid 4, acc 288000, time 2016
main: pid 5, acc 408000, time 2016
main: pid 6, acc 540000, time 2016
main: pid 7, acc 648000, time 2016
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.
```

实验总结

make grade

```
badarg:                 (1.3s)
  -check result:                             OK
  -check output:                             OK
exit:                   (1.3s)
  -check result:                             OK
  -check output:                             OK
spin:                   (2.1s)
  -check result:                             OK
  -check output:                             OK
waitkill:               (3.4s)
  -check result:                             OK
  -check output:                             OK
forktest:               (1.4s)
  -check result:                             OK
  -check output:                             OK
forktree:               (1.4s)
  -check result:                             OK
  -check output:                             OK
matrix:                 (6.7s)
  -check result:                             OK
  -check output:                             OK
priority:               (21.4s)
  -check result:                             OK
  -check output:                             OK
Total Score: 170/170
```

与参考答案的比较

练习1

不需要补全代码。主要是参考给出的RR调度算法，理解ucore实现的调度器框架。熟悉进程调度的具体实现。理解RR调度算法的具体实现。最后给出了MLFQ多级反馈队列调度算法的具体实现。

练习2

答案中给出了步长调度的列表实现与优先队列实现，自己只完成了优先队列实现的步长调度算法。两种算法都可以实现步长调度算法，优先队列效率更高。如果使用列表，需要在选择下一个进程的stride_pick_next中遍历整个列表，比较并找出行程值最小的进程进行调度。另外，使用stride表示步长，pass表示行程值，与答案中的表示不同，仅是命名不同，原理是完全相同的。

涉及知识点

- 进程调度
- RR轮转调度
- MLFQ多级反馈队列调度
- Stride Scheduling步长调度

未涉及的内容

- 虚拟内存

- 并发
- 文件系统