

## lab3 实验报告

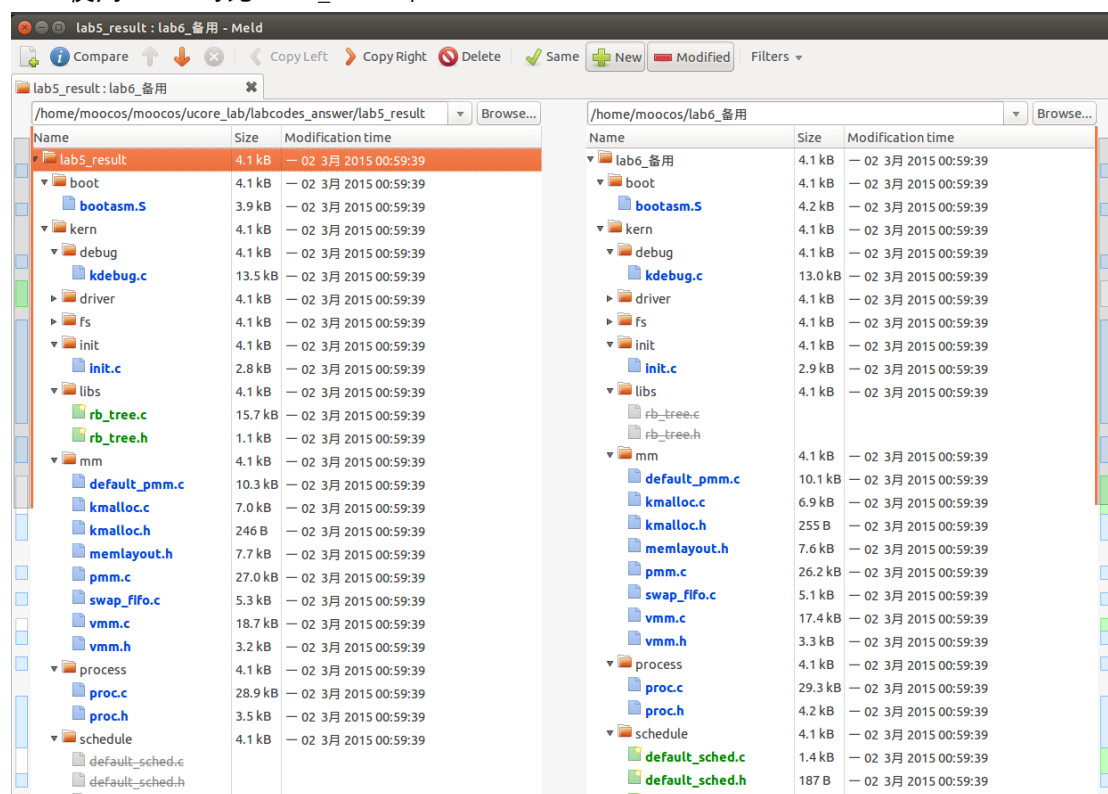
### 一、实验目的

- 1) 理解操作系统的调度管理机制
- 2) 熟悉 ucore 系统调度器框架，以及基于此框架的 RR 调度算法
- 3) 参考 RR 算法实现，完成 Stride Scheduling 调度算法

### 二、实验内容

#### 0) 填写已有实验

使用 meld 对比 Lab5\_result 和 Lab6



替换修改如下文件：

kdebug.c, init.c, rb\_tree.h, rb\_tree.c, default\_pmm.c, pmm.c, vmm.c, trap.c, grade.sh

根据注释，需要更新一些代码

- proc.c 下的 **alloc\_proc()**函数

初始化 proc.h 中 lab6 新定义的成员

```
137     proc->rq = NULL; //初始化运行队列为空
138     proc->run_link.prev = proc->run_link.next = NULL; //初始化运行队列指针
139     proc->time_slice = 0; //初始化时间片为0
140     proc->lab6_run_pool.left = proc->lab6_run_pool.right = proc->lab6_run_pool.parent = NULL; //各类指针
141     proc->lab6_stride = 0; //初始化步数为0
142     proc->lab6_priority = 0; //初始化优先级为0
```

- trap.c 下的 **trap\_dispatch()**函数

在每次 tick 后更新定时器

```
242     ticks ++;
243     assert(current != NULL);
244     run_timer_list(); //更新定时器
245     break;
```

## 1) Lab4, Lab5 的调度算法

lab4 和 lab5 中实现的进程调度:

0 号进程 `idleproc` 在创建后, 不断遍历进程池, 直到找到第一个 `runnable` 状态的进程, 调用并通过进程切换来执行新进程。首先通过调度, 切换到 1 号进程 `initproc`, 然后再通过 `do_fork()` 新建其它进程插入到 `proc_list`。某一进程开始执行后一直到执行结束不会被打扰, 完成后再选择下一进程执行, 这种方法实现简单, 但是 CPU 利用率较低

进程的正常生命周期如下:



- 进程首先在 `cpu` 初始化或者 `sys_fork` 的时候被创建, 当为该进程分配了一个进程控制块之后, 该进程进入 `uninit` 态(在 `proc.c` 中 `alloc_proc`)
  - 当进程完全完成初始化之后, 该进程转为 `runnable` 态
  - 当到达调度点时, 由调度器 `sched_class` 根据运行队列 `rq` 的内容来判断一个进程是否应该被运行, 即把处于 `runnable` 态的进程转换成 `running` 状态, 从而占用 CPU 执行。
  - `running` 态的进程通过 `wait` 等系统调用被阻塞, 进入 `sleeping` 态
  - `sleeping` 态的进程被 `wakeup` 变成 `runnable` 态的进程
  - `running` 态的进程主动 `exit` 变成 `zombie` 态, 然后由其父进程完成对其资源的最后释放, 子进程的进程控制块成为 `unused`
  - 所有从 `runnable` 态变成其他状态的进程都要出运行队列, 反之被放入某个运行队列中
- 进程调度的核心函数是 **`schedule`**, 讲义中给出了调用该函数的位置和原因, 也即调度点

## 2) 调度框架及 RR 调度算法

### ● timer 时间事件感知操作:

在进程的执行过程中, 就绪进程的等待时间和执行进程的执行时间, 这两个因素随着时间的流逝和各种事件的发生在不停地变化, 比如处于就绪态的进程等待调度的时间在增长, 处于运行态的进程所消耗的时间片在减少等。这些进程状态变化需要及时让进程调度器知道, 便于选择更合适的进程执行。这种进程变化的情况就形成了调度器相关的一个变化感知操作: `timer` 时间事件感知操作。

sched.h 中 **timer\_t** 的定义如下

结构成员包括 int 型的时间计数变量，挂在此时间上的进程，和用于连接各时间事件的指针

```
12 typedef struct {
13     unsigned int expires;           //the expire time
14     struct proc_struct *proc;       //the proc wait in this timer. If the expire time is end, then this proc
15     list_entry_t timer_link;        //the timer list
16 } timer_t;
17
18 #define le2timer(le, member) \
19 to_struct((le), timer_t, member)
20
21 // init a timer
22 static inline timer_t *
23 timer_init(timer_t *timer, struct proc_struct *proc, int expires) {
24     timer->expires = expires;
25     timer->proc = proc;
26     list_init(&(timer->timer_link));
27     return timer;
28 }
29
```

### ● 运行队列 rq

调度器引入运行队列 run-queue (rq)，通过链表的形式组织管理进程。链表的每一个节点是一个 **list\_entry\_t**，每个 **list\_entry\_t** 又对应到了 **struct proc\_struct \***，这期间的转换是通过宏 **le2proc** 来完成的。具体来说，我们知道在 **struct proc\_struct** 中有一个叫 **run\_link** 的 **list\_entry\_t**，因此可以通过偏移量逆向找到对应某个 **run\_list** 的 **struct proc\_struct**。即进程结构指针 **proc = le2proc(链表节点指针, run\_link)**

通过数据结构 **struct run\_queue** 来描述完整的 **run\_queue**（运行队列）。它的主要结构如下：

```
57 struct run_queue {
58     list_entry_t run_list;
59     unsigned int proc_num;
60     int max_time_slice;
61     // For LAB6 ONLY
62     skew_heap_entry_t *lab6_run_pool;
63 };
--
```

### ● 接口

与实验五相比，实验六需要针对处理器调度框架和各种算法进行设计与实现，为此对 ucore 的调度部分进行了适当的修改，使得 **kern/schedule/sched.c** 只实现调度器框架，而不再涉及具体的调度算法实现。而调度算法在单独的文件（**default\_sched.[ch]**）中实现。为了保证调度器接口的通用性，ucore 调度框架在 **sched.h** 中定义了 **sched\_class** 接口类：

```
35 struct sched_class {
36     // the name of sched_class
37     const char *name;
38     // Init the run queue
39     void (*init)(struct run_queue *rq);
40     // put the proc into runqueue, and this function must be called with rq_lock
41     void (*enqueue)(struct run_queue *rq, struct proc_struct *proc);
42     // get the proc out runqueue, and this function must be called with rq_lock
43     void (*dequeue)(struct run_queue *rq, struct proc_struct *proc);
44     // choose the next runnable task
45     struct proc_struct *(*pick_next)(struct run_queue *rq);
46     // dealer of the time-tick
47     void (*proc_tick)(struct run_queue *rq, struct proc_struct *proc);
48     /* for SMP support in the future
49     * load balance
50     * void (*load_balance)(struct rq* rq);
51     * get some proc from this rq, used in load_balance,
52     * return value is the num of gotten proc
53     * int (*get_proc)(struct rq* rq, struct proc* procs_moved[]);
54     */
55 };
--
```

在 sched.c 中首先定义了一个 **sched\_class** 类

```
12 static struct sched_class *sched_class;
```

然后初始化成 default\_sched\_class

```
45 void
46 sched_init(void) {
47     list_init(&timer_list);
48
49     sched_class = &default_sched_class;
```

而对应 RR 算法，default\_sched.c 中定义了 **default\_sched\_class** 调度策略类

```
155 struct sched_class default_sched_class = {
156     .name = "stride_scheduler",
157     .init = stride_init,
158     .enqueue = stride_enqueue,
159     .dequeue = stride_dequeue,
160     .pick_next = stride_pick_next,
161     .proc_tick = stride_proc_tick,
162 };
---
```

接下来定义四个调度类接口函数：

sched\_class\_enqueue

sched\_class\_dequeue

sched\_class\_pick\_next

sched\_class\_proc\_tick

这 4 个函数的实现其实就是调用某基于 sched\_class 数据结构的特定调度算法实现的 4 个指针函数。采用这样的调度类框架后，如果我们需要实现一个新的调度算法，则我们需要定义一个针对此算法的调度类的实例，一个就绪进程队列的组织结构描述就行，其他的事情都可交给调度类框架来完成

```
16 static inline void
17 sched_class_enqueue(struct proc_struct *proc) {
18     if (proc != idleproc) {
19         sched_class->enqueue(rq, proc);
20     }
21 }
22
23 static inline void
24 sched_class_dequeue(struct proc_struct *proc) {
25     sched_class->dequeue(rq, proc);
26 }
27
28 static inline struct proc_struct *
29 sched_class_pick_next(void) {
30     return sched_class->pick_next(rq);
31 }
32
33 static void
34 sched_class_proc_tick(struct proc_struct *proc) {
35     if (proc != idleproc) {
36         sched_class->proc_tick(rq, proc);
37     }
38     else {
39         proc->need_resched = 1;
40     }
41 }
```

最顶层 sched.c 中调度相关函数主要有三个：

- **wakeup\_proc 函数：**调用一个调度类接口函数 sched\_class\_enqueue 把一个进程放入到就绪进程队列中，这使得 wakeup\_proc 的实现与具体调度算法无关。
- **schedule 函数：**把当前继续占用 CPU 执行的运行进程放入到就绪进程队列中；从就

就绪进程队列中选择一个“合适”就绪进程；把该进程从就绪进程队列中摘除。通过调用三个调度类接口函数 `sched_class_enqueue()`, `sched_class_pick_next()`, `sched_class_enqueue()`来使得完成这三件事情与具体的调度算法无关。

- **run\_timer\_list 函数**: 在每次 timer 中断处理过程中被调用, 从而可用来调用调度算法所需的 timer 时间事件感知操作, 调整相关进程的进程调度相关的属性值。通过调用调度类接口函数 `sched_class_proc_tick` 使得此操作与具体调度算法无关。

## ● RR 调度算法

RR 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。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 (这个时间片段称为进程的时间片), 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。下面来分析一下其调度算法的实现

`default_sched.c` 中实现了相关函数:

**RR\_init()**: 初始化函数, 把 `rq` 队列清空, 等待进程数设为 0

```
7 static void
8 RR_init(struct run_queue *rq) {
9     list_init(&(rq->run_list));
10    rq->proc_num = 0;
11 }
```

**RR\_enqueue()**: 把某进程的进程控制块指针放入到 `rq` 队列末尾。如果进程控制块的时间片为 0, 则需要把它重置为 `rq` 成员变量 `max_time_slice`, 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。然后就绪进程数+1

```
13 static void
14 RR_enqueue(struct run_queue *rq, struct proc_struct *proc) {
15     //assert(list_empty(&(proc->run_link)));
16     list_add_before(&(rq->run_list), &(proc->run_link));
17     if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {
18         proc->time_slice = rq->max_time_slice;
19     }
20     proc->rq = rq;
21     rq->proc_num ++;
22     proc->enqueue_times++;
23 }
```

**RR\_dequeue()**: 把就绪进程队列 `rq` 的进程控制块指针的队列元素删除, 并把 `proc_num-1`

```
25 static void
26 RR_dequeue(struct run_queue *rq, struct proc_struct *proc) {
27     assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
28     list_del_init(&(proc->run_link));
29     rq->proc_num --;
30 }
```

**RR\_pick\_next()**: 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。



```

32 static struct proc_struct *
33 RR_pick_next(struct run_queue *rq) {
34     list_entry_t *le = list_next(&(rq->run_list));
35     if (le != &(rq->run_list)) {
36         return le2proc(le, run_link);
37     }
38     return NULL;
39 }

```

**RR\_proc\_tick():** 每次 timer 到时后, trap 函数将会间接调用此函数来把当前执行进程的时间片 time\_slice 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

```

41 static void
42 RR_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
43     if (proc->time_slice > 0) {
44         proc->time_slice --;
45     }
46     if (proc->time_slice == 0) {
47         proc->need_resched = 1;
48     }
49 }

```

最后是前面提到过的接口定义:

```

51 struct sched_class default_sched_class = {
52     .name = "RR_scheduler",
53     .init = RR_init,
54     .enqueue = RR_enqueue,
55     .dequeue = RR_dequeue,
56     .pick_next = RR_pick_next,
57     .proc_tick = RR_proc_tick,
58 };

```

**[TODO1]** 简述计时器 timer 触发中断后, 从中断到 RR\_proc\_tick 函数, 代码中函数的调用过程 (中断在 trap.c 文件中)

trap.c:

```

234 /* LAB6 YOUR CODE */
235 /* IMPORTANT FUNCTIONS:
236  * run_timer_list
237  * -----
238  * you should update your lab5 code (just add ONE or TWO lines of code):
239  * Every tick, you should update the system time, iterate the timers, and trigger the timers wh
240  * You can use one functions to finish all these things.
241  */
242 ticks ++;
243 assert(current != NULL);
244 run_timer_list();//更新定时器
245 break;

```

调用了 run\_timer\_list() 函数, 该函数先唤醒等待时间片用完的进程, 然后调用 sched\_class\_proc\_tick() 检查当前正在运行的进程

```

146 // call scheduler to update tick related info, and check the timer is expired? If expired, then wakeup proc
147 void
148 run_timer_list(void) {
149     bool intr_flag;
150     local_intr_save(intr_flag);
151     {
152         list_entry_t *le = list_next(&timer_list);
153         if (le != &timer_list) {
154             timer_t *timer = le2timer(le, timer_link);
155             assert(timer->expires != 0);
156             timer->expires --;
157             while (timer->expires == 0) {
158                 le = list_next(le);
159                 struct proc_struct *proc = timer->proc;
160                 if (proc->wait_state != 0) {
161                     assert(proc->wait_state & WT_INTERRUPTED);
162                 }

```

```

163         else {
164             warn("process %d's wait_state == 0.\n", proc->pid);
165         }
166         wakeup_proc(proc);
167         del_timer(timer);
168         if (le == &timer_list) {
169             break;
170         }
171         timer = le2timer(le, timer_link);
172     }
173 }
174 sched_class_proc_tick(current);
175 }
176 local_intr_restore(intr_flag);
177 }

```

### sched\_class\_proc\_tick()

```

34 static void
35 sched_class_proc_tick(struct proc_struct *proc) {
36     if (proc != idleproc) {
37         sched_class->proc_tick(rq, proc);
38     }
39     else {
40         proc->need_resched = 1;
41     }
42 }

```

sched\_class 即为前面所说的 default\_sched\_class 类型结构

```

50 sched_class = &default_sched_class;

```

proc\_tick 成员是对应 RR 算法中 **RR\_proc\_tick()** 函数，也就是更新该进程的时间片，如果减到 0 就中断执行，调度器选择下一个进程执行；否则 time\_slice--

```

41 static void
42 RR_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
43     if (proc->time_slice > 0) {
44         proc->time_slice--;
45     }
46     if (proc->time_slice == 0) {
47         proc->need_resched = 1;
48     }
49 }
50
51 struct sched_class default_sched_class = {
52     .name = "RR_scheduler",
53     .init = RR_init,
54     .enqueue = RR_enqueue,
55     .dequeue = RR_dequeue,
56     .pick_next = RR_pick_next,
57     .proc_tick = RR_proc_tick,
58 };

```

## TODO2

- 补全 lab6 使得 print.c 文件可以正常运行
- 在不修改 default\_sched.c 的前提下，用 RR 调度运行 print.c，分析输出结果
- 命令：将 print.c 放在 lab6/user 文件夹下，在终端中输入 make run-print

注：为保证 print 中 fork() 正常运行，需要屏蔽 RR\_enqueue 函数中的 assert

### print.c

```

#include<stdio.h>
#include<ulib.h>

int main(void) {
    fork();
    fork();
    int i;
    for (i = 0; i < 3; i++) {
        cprintf("%d %d\n", getpid(), i);
        yield();
    }
    return 0;
}

```

```

++ setup timer interrupts
kernel_execve: pid = 2, name = "print".
2 0
3 0
4 0
2 1
5 0
3 1
4 1
2 2
5 1
3 2
4 2
5 2
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:476:
  initproc exit.

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K> qemu: terminating on signal 2

```

根据运行结果分析 RR 运行机制：

使用 Understand 逐层分析调用可知 fork() 对应 lab4 中的 do\_fork() 函数

由前面 lab4&5 的分析可知初始在执行的是 pid=2 的**进程 2**，经过两个 fork() 调用，产生了进程 3 和进程 4，依次放入队列，然后 2 进入输出循环，i=0 时输出“2 0”，调用 yield() 自己中断，插入到队列末尾，队列排序为 3，4，2，调度进程 3；

**进程 3** 拷贝了进程 2 的后续操作，经过一个 fork() 调用，产生了进程 5，进入输出循环后输出“3 0”，然后 yield() 中断，插入到队列末尾，队列排序为 4，2，5，3，调度进程 4；

**进程 4** 不再产生新的进程，直接输出“4 0”，然后中断，队列排序为 2，5，3，4；

进程 2 继续之前的执行，进入第二轮循环，i=2 时输出“2 1”

进程 5 第一轮循环输出“5 0”

进程 3 输出“3 1”

进程 4 输出“4 1”

下一轮 2、5、3、4 依次输出“2 2”，“5 1”，“3 2”，“4 2”

此时队列中只剩进程 5，其它进程都已结束退出，进程 5 输出“5 2”后也退出，任务完成

### 3) Stride Scheduling 算法

#### 基本思路

我们希望调度器能够更智能地为每个进程分配合理的 CPU 资源。为不同的进程分配不同的优先级，则可能使每个进程得到的时间资源与他们的优先级成正比关系。Stride 调度即基于这种想法，基本思想可以考虑如下：

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
3. 对于获得调度的进程 P，将对应的 stride 加上其对应的步长 pass（只与优先权有关）
4. 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。可以证明，如



果令  $P.\text{pass} = \text{BigStride} / P.\text{priority}$ ，其中  $P.\text{priority}$  表示进程的优先权（大于 1），而  $\text{BigStride}$  表示一个预先定义的大常数，则该调度方案为每个进程分配的时间将与其优先级成正比。

**提问：**在 ucore 中，目前 Stride 是采用无符号的 32 位整数表示。则 **BigStride** 应该取多少才能保证比较的正确性？

```
9 /* You should define the BigStride constant here*/
10 /* LAB6: YOUR CODE */
11 #define BIG_STRIDE 0x7FFFFFFF /* you should give a value, and is ??? */
```

**优先队列**是这样一种数据结构：使用者可以快速的插入和删除队列中的元素，并且在预先指定的顺序下快速取得当前在队列中的最小（或者最大）值及其对应元素。这样的 数据结构非常符合 Stride 调度器的实现，ucore 中的优先队列利用了 libs/skew\_heap.h 中的斜堆，在实现 SS 算法时可以调用其中的 skew\_heap\_insert()和 skew\_heap\_remove()等函数

**TODO3:** 实现 Stride Scheduling 调度算法对应的代码

- 对应文件为 kern/schedule/default\_sched\_stride\_c，在报告中写出以下函数的作用
- stride\_init(...)
- stride\_enqueue(...)
- stride\_dequeue(...)
- stride\_pick\_next(...)
- stride\_proc\_tick(...)
- default\_sched\_class {...}

**default\_sched\_stride\_c** 中首先是一个优先队列的比较函数 **proc\_stride\_comp\_f()**，将步数相减，根据其正负比较大小关系。

```
15 static int
16 proc_stride_comp_f(void *a, void *b)
17 {
18     struct proc_struct *p = le2proc(a, lab6_run_pool);
19     struct proc_struct *q = le2proc(b, lab6_run_pool);
20     int32_t c = p->lab6_stride - q->lab6_stride;
21     if (c > 0) return 1;
22     else if (c == 0) return 0;
23     else return -1;
24 }
```

初始化函数 **stride\_init()**：初始化运行队列，并初始化 run\_pool（即进程控制块新定义的成员变量，用于堆操作），然后设置当前运行队列内进程数目为 0。

```
37 static void
38 stride_init(struct run_queue *rq) {
39     /* LAB6: YOUR CODE
40      * (1) init the ready process list: rq->run_list
41      * (2) init the run pool: rq->lab6_run_pool
42      * (3) set number of process: rq->proc_num to 0
43      */
44     list_init(&(rq->run_list)); //初始化运行队列
45     rq->lab6_run_pool = NULL; //初始化当前队列为空
46     rq->proc_num = 0; //当前等待进程数为0
47 }
```

入队函数 **stride\_enqueue()**，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。最后初始化时间片，将运行队列进程数目加一

```

73 #if USE_SKEW_HEAP
74     rq->lab6_run_pool=skew_heap_insert(rq->lab6_run_pool, &(proc->lab6_run_pool),
    proc_stride_comp_f); //比较队头元素与当前进程的步数大小, 选择步数小的运行
75 #else
76     assert(list_empty(&(proc->run_link)));
77     list_add_before(&(rq->run_list), &(proc->run_link)); //将proc插入到运行队列中
78 #endif
79     if(proc->time_slice==0 || proc->time_slice>rq->max_time_slice) //初始化时间片
80         proc->time_slice=rq->max_time_slice;
81     proc->rq=rq;
82     rq->proc_num++;
83 }

```

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-1。

```

93 static void
94 stride_dequeue(struct run_queue *rq, struct proc_struct *proc) {
95     /* LAB6: YOUR CODE
96     * (1) remove the proc from rq correctly
97     * NOTICE: you can use skew_heap or list. Important functions
98     *         skew_heap_remove: remove a entry from skew_heap
99     *         list_del_init: remove a entry from the list
100    */
101 #if USE_SKEW_HEAP
102     rq->lab6_run_pool=skew_heap_remove(rq->lab6_run_pool, &(proc->lab6_run_pool),
    proc_stride_comp_f); //比较队头元素与当前进程的步数大小, 选择步数小的删除
103 #else
104     assert(!list_empty(&(proc->run_link)) && proc->rq==rq);
105     list_del_init(&(proc->run_link)); //从运行队列中删除相应元素
106 #endif
107     rq->proc_num--; //运行队列进程数-1
108 }

```

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。

```

122 static struct proc_struct *
123 stride_pick_next(struct run_queue *rq) {
124     /* LAB6: YOUR CODE
125     * (1) get a proc_struct pointer p with the minimum value of stride
126     *     (1.1) If using skew_heap, we can use le2proc get the p from rq->lab6_run_poll
127     *     (1.2) If using list, we have to search list to find the p with minimum stride value
128     * (2) update p's stride value: p->lab6_stride
129     * (3) return p
130    */
131 #if USE_SKEW_HEAP
132     if(rq->lab6_run_pool==NULL)
133         return NULL;
134     struct proc_struct *p=le2proc(rq->lab6_run_pool, lab6_run_pool);
135 #else
136     list_entry_t *le=list_next(&(rq->run_list)); //列表上下一个进程
137     if(le==&rq->run_list) //如果是头节点 (当前的进程是最后一个)
138         return NULL;
139
140     struct proc_struct *p=le2proc(le, run_link);
141     le=list_next(le);
142     while(le!=&rq->run_list); //循环找最小的步长
143     struct proc_struct *q=le2proc(le, run_link);
144     if(int32_t)(p->lab6_stride-q->lab6_stride>0)
145         p=q;
146     le=list_next(le);
147 }
148 #endif
149 if(p->lab6_priority==0)
150     p->lab6_stride+=BIG_STRIDE;
151 else p->lab6_stride+=BIG_STRIDE/p->lab6_priority; //更新进程的stride
152 return p;
153 }

```

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

```

141 static void
142 stride_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
143     /* LAB6: YOUR CODE */
144     if (proc->time_slice > 0) {
145         proc->time_slice--;
146     }
147     if (proc->time_slice == 0) {
148         proc->need_resched = 1;
149     }
150 }

```

最后是同 RR 算法的 `default_sched_class` 调度策略类，方便框架调用

```

173 /*
174 struct sched_class default_sched_class = {
175     .name = "stride_scheduler",
176     .init = stride_init,
177     .enqueue = stride_enqueue,
178     .dequeue = stride_dequeue,
179     .pick_next = stride_pick_next,
180     .proc_tick = stride_proc_tick,
181 };
182 */

```

make 结果

```

check_swap() succeeded!
++ setup timer interrupts
kernel_execve: pid = 2, name = "exit".
I am the parent. Forking the child...
I am parent, fork a child pid 3
I am the parent, waiting now..
I am the child.
waitpid 3 ok.
exit pass.
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:477:
    initproc exit.

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K> qemu: terminating on signal 2

```

make grade

```

priority: (21.8s)
-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

```

#### 4) Challenge

### Challenge: TODO4:

- 在进程控制块中加入新的成员 `enqueue_times`，并保证 `printother.c` 能够正常运行。
- `printother.c` 中调用了 `getenqueue_times()` 函数，这个函数输出当前进程的入队次数。修改代码保证此函数被实现。

观察 `printother.c` 代码

```
Getting Started x printother.c x ulib.c x syscall.c x syscall.c x proc.c x proc.h x sched.c x
1 #include<stdio.h>
2 #include<ulib.h>
3
4 int main(void){
5     fork();
6     fork();
7     int i;
8     for(i=0;i<10;i++){
9         cprintf("%d %d\n",getpid(),i);
10        yield();
11    }
12    cprintf("Process: %d, enqueue_times: %d\n", getpid(), getenqueue_times());
13    return 0;
14 }
```

可以看到 `getenqueue_times()` 函数和 `getpid()` 调用方式类似，所以参考 `getpid()` 的定义

`getpid()` 函数在 `user/libs/ulib.c` 中定义

```
Getting Started x printother.c x ulib.c x syscall.c x syscall.c x proc.c x proc.h x sched.c x
38 int
39 getpid(void) {
40     return sys_getpid();
41 }
```

`sys_getpid()` 函数在 `user/libs/syscall.c` 中定义

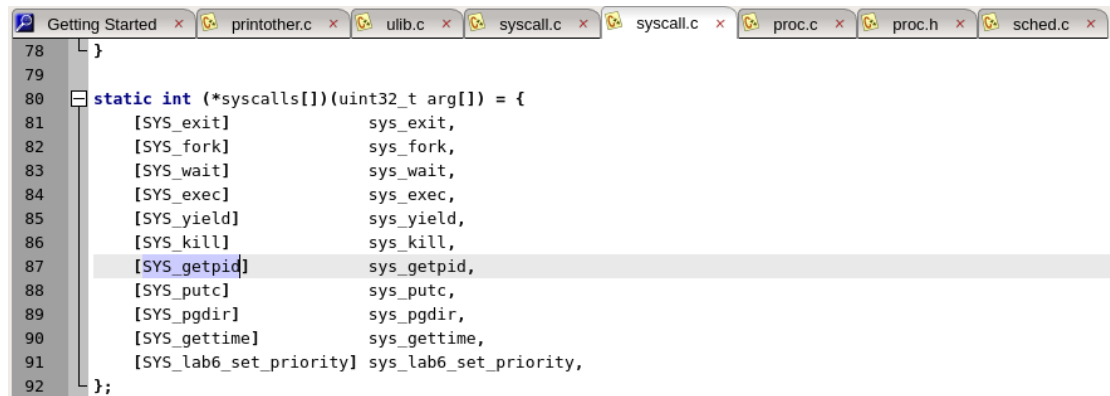
```
Getting Started x printother.c x ulib.c x syscall.c x syscall.c x proc.c x proc.h x sched.c x
58 int
59 sys_getpid(void) {
60     return syscall(SYS_getpid);
61 }
```

`syscall()` 是定义在该文件中的一个内联函数，是对汇编/机器语言的一个封装

```
Getting Started x printother.c x ulib.c x syscall.c x syscall.c x proc.c x proc.h x sched.c x
7
8 static inline int
9 syscall(int num, ...) {
10     va_list ap;
11     va_start(ap, num);
12     uint32_t a[MAX_ARGS];
13     int i, ret;
14     for (i = 0; i < MAX_ARGS; i++) {
15         a[i] = va_arg(ap, uint32_t);
16     }
17     va_end(ap);
18
19     asm volatile (
20         "int %1;"
21         : "=a" (ret)
22         : "i" (T_SYSCALL),
23           "a" (num),
24           "d" (a[0]),
25           "c" (a[1]),
26           "b" (a[2]),
27           "D" (a[3]),
28           "S" (a[4])
29         : "cc", "memory");
30     return ret;
31 }
```

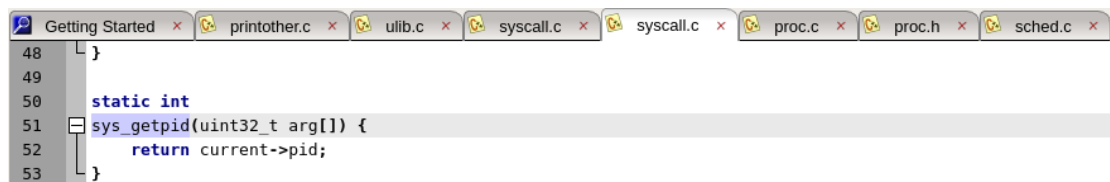
而括号里的参数是 **SYS\_getpid**

在 understand 中右键 view information 看到在 kern/syscall/syscall.c 中的使用  
应该是对 sys\_getpid 的重命名（或者说是类型转换）



```
78 }
79
80 static int (*syscalls[])(uint32_t arg[]) = {
81     [SYS_exit]      sys_exit,
82     [SYS_fork]      sys_fork,
83     [SYS_wait]      sys_wait,
84     [SYS_exec]      sys_exec,
85     [SYS_yield]     sys_yield,
86     [SYS_kill]      sys_kill,
87     [SYS_getpid]    sys_getpid,
88     [SYS_putc]      sys_putc,
89     [SYS_pgidir]    sys_pgidir,
90     [SYS_gettime]   sys_gettime,
91     [SYS_lab6_set_priority] sys_lab6_set_priority,
92 };
```

在本文件中找到 **sys\_getpid()**函数



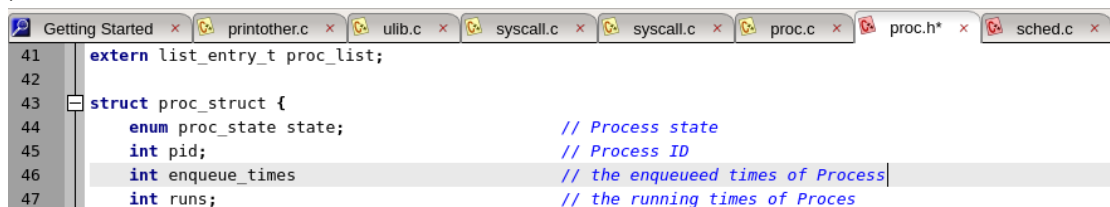
```
48 }
49
50 static int
51 sys_getpid(uint32_t arg[]) {
52     return current->pid;
53 }
```

current 是在 proc.c 中定义的当前进程，进程控制块 proc\_struct 中有成员 pid，即进程的 ID。  
这应该是最底层的函数，返回当前进程的 ID

通过对于 getpid()函数的分析，不难得出 getenqueue\_times()的写法，区别就是后者稍微多了一点儿，就是在每次入队之后要+1

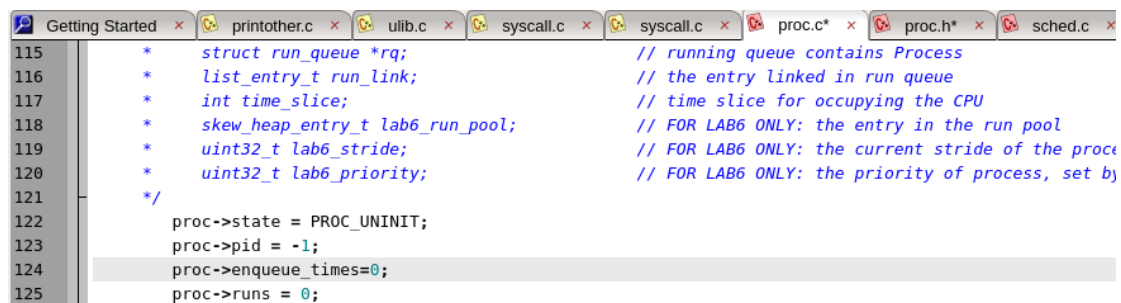
首先在进程控制块 **proc\_struct** 的定义中添加成员 enqueue\_times

proc.h



```
41 extern list_entry_t proc_list;
42
43 struct proc_struct {
44     enum proc_state state;           // Process state
45     int pid;                         // Process ID
46     int enqueue_times;               // the enqueued times of Process
47     int runs;                       // the running times of Proces
```

proc.c 中 **alloc\_proc()**函数进行初始化，初始化为 0



```
115 * struct run_queue *rq;           // running queue contains Process
116 * list_entry_t run_link;          // the entry linked in run queue
117 * int time_slice;                 // time slice for occupying the CPU
118 * skew_heap_entry_t lab6_run_pool; // FOR LAB6 ONLY: the entry in the run pool
119 * uint32_t lab6_stride;           // FOR LAB6 ONLY: the current stride of the process
120 * uint32_t lab6_priority;         // FOR LAB6 ONLY: the priority of process, set by user
121 */
122 proc->state = PROC_UNINIT;
123 proc->pid = -1;
124 proc->enqueue_times=0;
125 proc->runs = 0;
```

接下来就是在每次 enqueue 的时候让 enqueue\_times 递增

回想之前对入队函数的封装，sched.c 中是最顶层的调用，实际实现是在 default\_sched.c (RR 调度算法) 和 default\_sched\_stride.c (Stride 调度算法)

这里选择 RR 算法，需要把 stride.c 中的 default\_sched\_class 注释掉

然后在 default\_sched.c 中找到 RR\_enqueue()函数

添加语句: proc->enqueue\_times++;

```
Getting Started x printother.c x ulib.c x syscall.c x syscall.c x default_sched.c x proc.c x
13 static void
14 RR_enqueue(struct run_queue *rq, struct proc_struct *proc) {
15     //assert(list_empty(&(proc->run_link)));
16     list_add_before(&(rq->run_list), &(proc->run_link));
17     if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {
18         proc->time_slice = rq->max_time_slice;
19     }
20     proc->rq = rq;
21     rq->proc_num ++;
22     proc->enqueue_times++;
23 }
```

接下来就跟 gettid()实现思路一样了, 倒着回去

最底层函数是在 kern/syscall/syscall.c 中添加 sys\_getenqueue\_times()

```
Getting Started x printother.c x ulib.c x syscall.c x syscall.c x default_sched.c x proc.c x
48 }
49
50 static int
51 sys_getpid(uint32_t arg[]) {
52     return current->pid;
53 }
54
55 static int
56 sys_getenqueue_times(uint32_t arg[]) {
57     return current->enqueue_times;
58 }
```

类型转换

```
Getting Started x printother.c x ulib.c x syscall.c x syscall.c x default_sched.c x proc.c x
84
85 static int (*syscalls[])(uint32_t arg[]) = {
86     [SYS_exit] sys_exit,
87     [SYS_fork] sys_fork,
88     [SYS_wait] sys_wait,
89     [SYS_exec] sys_exec,
90     [SYS_yield] sys_yield,
91     [SYS_kill] sys_kill,
92     [SYS_getpid] sys_getpid,
93     [SYS_getenqueue_times] sys_getenqueue_times,
94     [SYS_putc] sys_putc,
95     [SYS_pgdir] sys_pgdir,
96     [SYS_gettime] sys_gettime,
97     [SYS_lab6_set_priority] sys_lab6_set_priority,
98 };
```

user/libs/syscall.c 中添加 sys\_getenqueue\_times()函数调用上面的函数

```
Getting Started x printother.c x ulib.c x syscall.c x syscall.c x default_sched.c x proc.c x
58 int
59 sys_getpid(void) {
60     return syscall(SYS_getpid);
61 }
62
63 int
64 sys_getenqueue_times(void) {
65     return syscall(SYS_getenqueue_times);
66 }
```

接下来是 ulib.c 中的 getenqueue\_times()函数



```
Getting Started x printother.c x ulib.c x syscall.c x syscall.c x default_sched.c x proc.c x
37
38 int
39 getpid(void) {
40     return sys_getpid();
41 }
42
43 int
44 getenquee_times(void){
45     return sys_getenquee_times();
46 }
```

最后要注意在各个.h 文件中声明

## ulib.h

```
Getting Started x printother.c x ulib.h* x ulib.c x syscall.c x syscall.c x default_sched.c x proc.c
13 _panic(__FILE__, __LINE__, __VA_ARGS__)
14
15 #define assert(x) \
16 do { \
17     if (!(x)) { \
18         panic("assertion failed: %s", #x); \
19     } \
20 } while (0)
21
22 // static_assert(x) will generate a compile-time error if 'x' is false.
23 #define static_assert(x) \
24 switch (x) { case 0: case (x): ; }
25
26 void __noreturn exit(int error_code);
27 int fork(void);
28 int wait(void);
29 int waitpid(int pid, int *store);
30 void yield(void);
31 int kill(int pid);
32 int getpid(void);
33 int getenquee_times(void);
34 void print_pgid(void);
35 unsigned int gettime_msec(void);
36 void lab6_set_priority(uint32_t priority);
37
38 #endif /* !_USER_LIBS_ULIB_H_ */
```

## syscall.h

```
Getting Started x printother.c x ulib.h x ulib.c x syscall.c x syscall.h* x syscall.c x default
1 #ifndef _USER_LIBS_SYSCALL_H_
2 #define _USER_LIBS_SYSCALL_H_
3
4 int sys_exit(int error_code);
5 int sys_fork(void);
6 int sys_wait(int pid, int *store);
7 int sys_yield(void);
8 int sys_kill(int pid);
9 int sys_getpid(void);
10 int sys_getenquee_times(void);
11 int sys_putc(int c);
12 int sys_pgid(void);
13 int sys_gettime(void);
14 /* FOR LAB6 ONLY */
15 void sys_lab6_set_priority(uint32_t priority);
16
17 #endif /* !_USER_LIBS_SYSCALL_H_ */
```

运行时出现错误

```
kern/syscall/syscall.c:93:3: error: 'SYS_getenquee_times' undeclared here (not in a function)
[SYS_getenquee_times] sys_getenquee_times,
^
kern/syscall/syscall.c:93:2: error: array index in initializer not of integer type
[SYS_getenquee_times] sys_getenquee_times,
^
kern/syscall/syscall.c:93:2: error: (near initialization for 'syscalls')
make: *** [obj/kern/syscall/syscall.o] Error 1
```

错误原因是没有定义系统调用号! 在 unistd.h 中选择 0-255 范围内没有被用过的数字定义

```
Getting Started x printother.c x ulib.h x ulib.c x syscall.c x syscall.h x unistd.h x syscall.c x
1 #ifndef _LIBS_UNISTD_H_
2 #define _LIBS_UNISTD_H_
3
4 #define T_SYSCALL 0x80
5
6 /* syscall number */
7 #define SYS_exit 1
8 #define SYS_fork 2
9 #define SYS_wait 3
10 #define SYS_exec 4
11 #define SYS_clone 5
12 #define SYS_yield 10
13 #define SYS_sleep 11
14 #define SYS_kill 12
15 #define SYS_gettime 17
16 #define SYS_getpid 18
17 #define SYS_getenqueue_times 7
18 #define SYS_brk 19
19 #define SYS_mmap 20
20 #define SYS_munmap 21
21 #define SYS_shmem 22
22 #define SYS_putc 30
23 #define SYS_pgdir 31
24 /* ONLY FOR LAB6 */
25 #define SYS_lab6_set_priority 255
```

make 结果

```
'obj/bootblock.out' size: 434 bytes
build 512 bytes boot sector: 'bin/bootblock' success!
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB) copied, 0.0498647 s, 103 MB/s
1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.000108099 s, 4.7 MB/s
1605+1 records in
1605+1 records out
822239 bytes (822 kB) copied, 0.00363632 s, 226 MB/s
128+0 records in
128+0 records out
13421728 bytes (134 MB) copied, 0.331028 s, 405 MB/s
[~/moocos/ucores_lab/labcodes/lab6]
moocos->
```

然后 make run-printother

```
printother.c - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
#include<stdio.h>
#include<ulib.h>

int main(void) {
    fork();
    fork();
    int i;
    for (i = 0; i < 10; i++) {
        cprintf("%d %d\n", getpid(), i);
        yield();
    }
    cprintf("Process: %d, enqueue_times: %d\n", getpid(), getenqueue_times());
    return 0;
}
```

```
kernel_execve: pid = 2, name = "printother".
```

```
2 0
3 0
4 0
2 1
5 0
3 1
4 1
2 2
5 1
3 2
4 2
2 3
5 2
3 3
4 3
2 4
5 3
3 4
4 4
2 5
5 4
3 5
```

```
4 5
2 6
5 5
3 6
4 6
2 7
5 6
3 7
4 7
2 8
5 7
3 8
4 8
2 9
5 8
3 9
4 9
```

```
Process: 2, enqueue_times: 11
5 9
Process: 3, enqueue_times: 11
Process: 4, enqueue_times: 11
Process: 5, enqueue_times: 11
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:477:
    initproc exit.
```

```
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K> qemu: terminating on signal 2
```

输出结果分析和之前 RR 算法运行 print.c 相同，不同的是这里进行了 10 次循环，入队次数为 11 的原因是：在 10 次循环之前进程已经在队列中了，所以才会 printf( pid, i)，此后每次输出之前都有一次入队，所以一共有 11 次

使用 SS 调度算法实现同理（因为之前把 RR 算法的调度策略类注释掉，所以 default 调度算法目前为 Stride Scheduling 算法）

#### **[TODO5]**

分析认为是 initproc 内核线程创建完成后，do\_wait()退出 rq 队列进入 SLEEPING 状态；2 号进程被创建，然后按照之前分析 RR 算法执行结果进行。讲义中讲解调度点时提到：1、initproc 内核线程等待所有用户进程结束，如果没有结束，就主动放弃 CPU 控制权；2、initproc 内核线程在所有用户进程结束后，让 kswapd 内核线程执行 10 次，用于回收空闲内存资源，所以最后 initproc 又出现在 rq 队列中

### **三、实验感悟**

本次实验内容相对简单，了解了 ucore 的进程调度框架，了解了课本上学过的 RR 算法，学习并实现了之前没学到过的 Stride Scheduling 调度算法。因为主要是在仿照着写，SS 框架与 RR 相同，只是引入了 stride 和 priority；todo4 中 getenqueue\_times 与 get\_pid 的调用顺序相同，所以得到结果较为容易。