

实验项目：调度器

姓名：张伟焜 学号：17343155 邮箱：zhangwk8@mail2.sysu.edu.cn
院系：数据科学与计算机学院 专业：17 级软件工程 指导教师：张永东

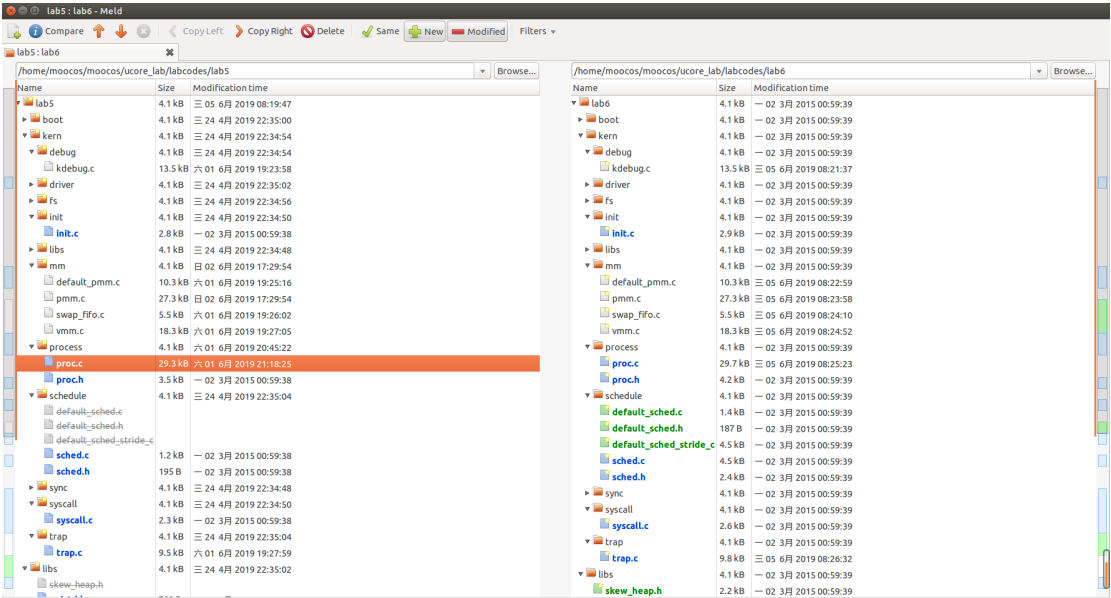
【实验题目】
调度器

【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

【实验要求】
根据指导，完成练习 0~3。

【实验方案】
实验环境：老师提供的虚拟机（Virtual box），无特殊硬件要求
实验思路：根据实验指导，先了解理论知识，再进行实验

【实验过程】
练习 0：填写已有实验。
使用 meld 软件将 ucore 启动实验的代码导入。



注意要点击标星文件进行对比，将上次实验完成的函数复制过来，不要将整个文件进行覆盖。之前实验修改的内容主要在 kdebug.c trap.c pmm.c default_pmm.c vmm.c swap_fifo.c proc.c 等

proc.c 中有一处对之前代码的更新

由于头文件中对 proc_struct 结构体进行了扩展，所以要在此对 static struct proc_struct
* alloc_proc(void) 进行修改，即补上对相关定义的初始化。

添加的初始化定义如下：

```
proc->rq = NULL; //将运行队列初始化为NULL
list_init(&(proc->run_link)); //将运行队列指针初始化
proc->time_slice = 0; //时间碎片初始化为0
proc->lab6_run_pool.left = proc->lab6_run_pool.right = proc->lab6_run_pool.parent =
NULL; //将优先队列的相关指针初始化为NULL
proc->lab6_stride = 0; //步数初始化为0
proc->lab6_priority = 0; //初始化优先级为0
```

trap.c 中也有一处代码更新

```
/* LAB6 YOUR CODE */
/* you should upate you lab5 code
 * IMPORTANT FUNCTIONS:
 * sched_class_proc_tick
 */
ticks++;
assert(current != NULL);
sched_class_proc_tick(current);
break;
```

练习 1：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。

(1)对比联系 0 后的 lab6 和 lab5，发现有一下主要的区别：

- a) 增加了调度算法 Round Robin。
- b) PCT 中增加了三个与 stride 调度算法相关的成员变量，并增加了对应的初始化过程
- c) 增加 set_priority, get_time 系统调用；
- d) 增加了斜堆数据结构的实现。

(2)执行 make grade 结果如下，未通过 priority 检查。

```
-check result: OK
-check output: OK
priority: (11.8s)
-check result: WRONG
-e !! error: missing 'sched class: stride_scheduler'
!! error: missing 'stride sched correct result: 1 2 3 4 5'

-check output: OK
Total Score: 163/170
make: *** [grade] Error 1
zhangweikun$>
```

请理解并分析 sched_class 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。

```
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,
};
```

Round Robin 调度算法的原理是让所有运行状态的进程分时轮流使用 CPU。每个时钟中断，操作系统会递减当前执行进程的时间片，当前进程的时间片 time_slice 减为 0 后，调度器将当前进程放置到运行队列队尾，重置此进程的时间片，再从队列头部取出进程开始执行。

结合上面的分析及代码，我们来看具体的函数实现。

首先是初始化函数：

该函数对运行队列进行初始化。

```
static void
RR_init(struct run_queue *rq) {
    list_init(&(rq->run_list));    //初始化运行队列
    rq->proc_num = 0;             //运行队列中进程个数初始化为0
}
```

入队函数。

首先保证该进程目前不在运行队列中；将进程加入运行队列开始执行，若时间片有误则重置时间片；更新队列。

```
Static void
RR_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    assert(list_empty(&(proc->run_link)));    //检查该进程目前没有在运行队列中
    list_add_before(&(rq->run_list), &(proc->run_link));    //将进程加入到运行队列
    if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {
        proc->time_slice = rq->max_time_slice;    //设置时间片
    }
    proc->rq = rq;    //更新运行队列
    rq->proc_num++;
}
```

出队函数。

保证该进程目前在运行队列中；从队列中移除该进程；更新运行队列中的进程数。

```
static void
RR_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    assert(!list_empty(&(proc->run_link)) && proc->rq == rq);    //确保要出队的进程在当前
```

的队列中

```
list_del_init(&(proc->run_link));    //将进程从运行队列中移除
rq->proc_num--;    //运行队列进程数减1
}
```

选择下一进程。

返回队列中第一个待运行的进程。

```
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);    //获取队列中第一个待运行的进程
    }
    return NULL;
}
```

时间片的递减。

每一个时钟中断，运行中的进程时间片-1；减为0，需要进行调度。

```
static void
RR_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    if (proc->time_slice > 0) {
        proc->time_slice--;    //每一个时钟中断进程的时间片减1
    }
    if (proc->time_slice == 0) {
        proc->need_resched = 1;    //标志需要进行调度
    }
}
```

接下来是对函数指针的调用分析

kern/schedule/sched.c 中的 schedule 函数实现对函数指针的调用。

```
void
schedule(void) {
    bool intr_flag;
    struct proc_struct *next;
    local_intr_save(intr_flag);
    {
        current->need_resched = 0;    //标志不需要调度
        if (current->state == PROC_RUNNABLE) {    //若是可运行状态，则加入到运行队列中
            sched_class_enqueue(current);
        }
        if ((next = sched_class_pick_next()) != NULL) {    //若运行队列中还有其他进程，则根据算法挑选出某一个出队
            sched_class_dequeue(next);
        }
    }
}
```

```

        if (next == NULL) {    //若没有其他运行进程，设置为idleproc进程
            next = idleproc;
        }
        next->runs++;    //下一个运行的进程运行次数加1
        if (next != current) {
            proc_run(next);    //执行下一个进程
        }
    }
    local_intr_restore(intr_flag);
}

```

Ucore 调度执行过程：

wakeup_proc 将某一个指定进程放入可执行进程队列中。schedule 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，将其直接删除。

当需要取出执行的下一个进程的时候，将就绪队列的队头取出。

每次时钟中断，将当前执行的进程的时间片减 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；

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

首先给出多级反馈队列调度算法的描述。

- a) 进程在进入待调度的队列等待时，首先进入优先级最高的 Q1 等待。
- b) 首先调度优先级高的队列中的进程。若高优先级中队列中已没有调度的进程，则调度次优先级队列中的进程。例如：Q1,Q2,Q3 三个队列，当且仅当在 Q1 中没有进程等待时才去调度 Q2，同理，只有 Q1,Q2 都为空时才会去调度 Q3。
- c) 对于同一个队列中的各个进程，按照 FCFS 分配时间片调度。比如 Q1 队列的时间片为 N，那么 Q1 中的作业在经历了 N 个时间片后若还没有完成，则进入 Q2 队列等待，若 Q2 的时间片用完后作业还不能完成，一直进入下一级队列，直至完成。
- d) 在最后一个队列 QN 中的各个进程，按照时间片轮转分配时间片调度。
- e) 在低优先级的队列中的进程在运行时，又有新到达的作业，此时须立即把正在运行的进程放回当前队列的队尾，然后把处理机分给高优先级进程。换言之，任何时刻，只有当第 1~i-1 队列全部为空时，才会去执行第 i 队列的进程（抢占式）。特别说明，当再度运行到当前队列的该进程时，仅分配上次还未完成的时间片，不再分配该队列对应的完整时间片。

接下来给出具体实现。

在 proc_struct 中设置 N 个多级反馈队列入口。队列编号越大优先级越低，优先级越低的队列上进程的时间片越大，具体可以设置为上一个队列的两倍。

为了记录进程所在的队列，我们在 PCB 中增加条目来进行记录。

首先，对所有优先级队列进行初始化。将进程加入到就绪进程集合时，观察该进程剩余的时间片，如果为 0，就降一级；如果不为 0，则不降级。

对同一个优先级的队列内的进程使用时间片轮转算法。

选择下一个执行进程时，优先看较高优先级的队列中是否存在任务，如果不存在才在较低优先级的队列中寻找进程去执行。

从就绪进程集合中删除某一个进程也要在对应队列中删除。

处理时间中断的函数仍然不需要改变。(与 RR 相同)

练习 2：实现 Stride Scheduling 调度算法。

Stride Scheduling 算法：希望每个进程得到的时间资源与他们的优先级成正比关系。

《操作系统实验指导(清华大学)陈渝、向勇编著》提供的具体的算法思路：

1) 为每个 runnable 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

4) 在一段固定的时间后，回到步骤 2

代码如下：

init

初始化调度器类的信息。初始化当前的运行队列为一个空的容器结构。

```
static void
stride_init(struct run_queue *rq) {
    /* LAB6: YOUR CODE
     * (1) init the ready process list: rq->run_list
     * (2) init the run pool: rq->lab6_run_pool
     * (3) set number of process: rq->proc_num to 0
     */
    list_init(&(rq->run_list));
    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;
}
```

enqueue

初始化刚进入运行队列的进程 proc 的 stride 属性。将进程插入运行队列中。

```
static void
stride_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    /* LAB6: YOUR CODE
     * (1) insert the proc into rq correctly
     * NOTICE: you can use skew_heap or list. Important functions
     *         skew_heap_insert: insert a entry into skew_heap
     *         list_add_before: insert a entry into the last of list
     * (2) recalculate proc->time_slice
     * (3) set proc->rq pointer to rq
     * (4) increase rq->proc_num
     */
    //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
}
```

```

    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;    //设置进程的时间片大小
    }
    proc->rq = rq;    //更新进程的就绪队列
    rq->proc_num++;
}

```

dequeue

从运行队列中删除相应的元素。

```

static void
stride_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    /* LAB6: YOUR CODE
    * (1) remove the proc from rq correctly
    * NOTICE: you can use skew_heap or list. Important functions
    *         skew_heap_remove: remove a entry from skew_heap
    *         list_del_init: remove a entry from the list
    */
    //删除斜堆中的指定进程
    rq->lab6_run_pool =skew_heap_remove(rq->lab6_run_pool, &(proc->lab6_run_pool),
proc_stride_comp_f);
    rq->proc_num--;
}

```

pick next

扫描整个运行队列，返回其中 stride 值最小的对应进程。

更新对应进程的 stride 值。

```

static struct proc_struct *
stride_pick_next(struct run_queue *rq) {
    /* LAB6: YOUR CODE
    * (1) get a  proc_struct pointer p  with the minimum value of stride
    *         (1.1) If using skew_heap, we can use le2proc get the p from
rq->lab6_run_poll
    *         (1.2) If using list, we have to search list to find the p with minimum
stride value
    * (2) update p;s stride value: p->lab6_stride
    * (3) return p
    */
    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
    //更新该进程的stride值

```

```

    if (p->lab6_priority == 0)    //考虑进程优先级为0的情况，否则会出现divide error
        p->lab6_stride += BIG_STRIDE;
    else
        p->lab6_stride += BIG_STRIDE / p->lab6_priority;
    return p;
}

```

proc tick

检测当前进程是否已经用完分配的时间片。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。

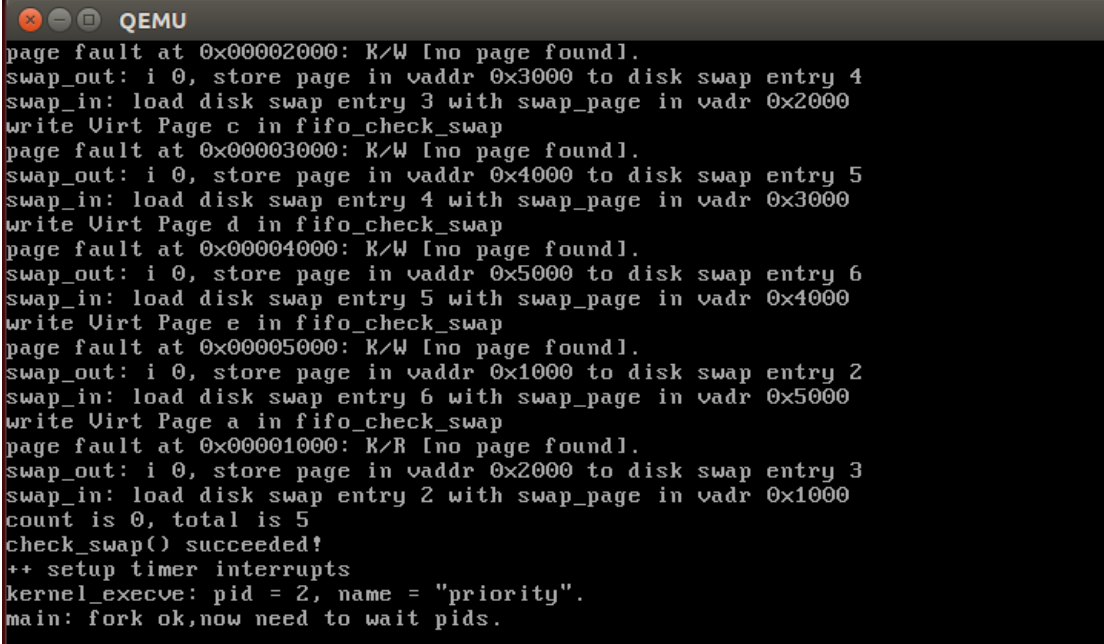
```

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;
    }
}

```

运行结果:

make qemu:



```

QEMU
page fault at 0x00002000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x3000 to disk swap entry 4
swap_in: load disk swap entry 3 with swap_page in vadr 0x2000
write Virt Page c in fifo_check_swap
page fault at 0x00003000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x4000 to disk swap entry 5
swap_in: load disk swap entry 4 with swap_page in vadr 0x3000
write Virt Page d in fifo_check_swap
page fault at 0x00004000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x5000 to disk swap entry 6
swap_in: load disk swap entry 5 with swap_page in vadr 0x4000
write Virt Page e in fifo_check_swap
page fault at 0x00005000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
swap_in: load disk swap entry 6 with swap_page in vadr 0x5000
write Virt Page a in fifo_check_swap
page fault at 0x00001000: K/R [no page found].
swap_out: i 0, store page in vaddr 0x2000 to disk swap entry 3
swap_in: load disk swap entry 2 with swap_page in vadr 0x1000
count is 0, total is 5
check_swap() succeeded!
++ setup timer interrupts
kernel_execve: pid = 2, name = "priority".
main: fork ok, now need to wait pids.

```

make grade:


```
matrix: (12.6s)
  -check result: OK
  -check output: OK
priority: (11.8s)
  -check result: OK
  -check output: OK
Total Score: 170/170
zhangweikun$>
```

练习 3：阅读分析源代码。

结合中断处理和调度程序，再次理解进程控制块中的 `trapframe` 和 `context` 在进程切换时作用。

线程控制块中的 `context` 是保存的线程运行的上下文信息。

结合 `proc_struct` 结构体注释“Trap frame for current interrupt.”,可以得出 `tf` 为中断帧的指针。`tf` 总是指向内核栈的某个位置。当进程从用户空间跳到内核空间时，中断帧记录了进程在被中断前的状态。当内核需要跳回用户空间时，需要调整中断帧以恢复让进程继续执行的各寄存器值。由于 `uCore` 内核允许嵌套中断，为了保证嵌套中断发生时 `tf` 总是能够指向当前的 `trapframe`，`uCore` 在内核栈上维护了 `tf` 链。

首先在执行某进程 A 的用户代码时，出现了一个 `trap`(例如，是一个外设产生的中断)，这时就会从进程 A 的用户态切换到内核态(过程(1))，并且保存好进程 A 的 `trapframe`;当内核态处理中断时发现需要进行进程切换时，`ucore` 要通过 `schedule` 函数选择下一个将占用 CPU 执行的进程(即进程 B)，然后会调用 `proc run` 函数，`proc run` 函数进步调用 `switch to` 函数，切换到进程 B 的内核态(过程(2))，继续进程 B 上一次在内核态的操作，并通过 `iret` 指令，最终将执行权转交给进程 B 的用户空间(过程(3))。

当进程 B 由于某种原因发生中断之后(过程(4))，会从进程 B 的用户态切换到内核态，并且保存好进程 B 的 `trapframe`;当内核态处理中断时发现需要进行进程切换时，即需要切换到进程 A,`ucore` 再次切换到进程 A(过程(5))，会执行进程 A 上一次在内核调用 `shedule`(具体还要跟踪到 `switch to` 函数)函数返回后的下一行代码，这行代码当然还是在进程 A 的上一次中断处理流程中。最后当进程 A 的中断处理完毕的时候执行权又会反交给进程 A 的用户代码(过程(6))。这就是在只有两个进程的情况下，进程切换间的大体流程。

【实验总结】

完成实验后，请分析 `ucore_lab` 中提供的参考答案，并请在实验报告中说明你的实现与参考答案的区别。

与参考答案相比，我只给出了使用堆的实现，而没有用链表实现，但就运行结果来看，我的代码也达到了实验的要求。

列出你认为本实验中重要的知识点，以及与对应的 OS 原理中的知识点，并简要说明你对二者的含义，关系，差异等方面的理解（也可能出现实验中的知识点没有对应的原理知识点）

1.实验涉及到对轮转法调度的分析。

- 2.对多级反馈队列调度方案的设计
- 3.涉及到 Stride Scheduling 调度算法的相关内容

对应了 OS 原理中的：

轮转法理论知识的应用

多级反馈队列调度方案的原理与概念

理论知识是基础；

实验知识是理论知识的实际应用与实践。

列出你认为 OS 原理中很重要，但在实验中没有对应上的知识点

- 1.没有涉及到多处理器调度
- 2.没有涉及多级队列调度，直接要求我们设计多级反馈队列调度方案
- 3.最短作业优先调度
- 4.先到先服务调度

心得体会

单看本次实验的内容还算简单，但是在 make grade 环节出现了很多奇奇怪怪的错误。主要原因是练习 0 中有很多代码没有修改完全，导致出现了很多莫名其妙的错误。

总的来说，本次实验让我更深入地了解了调度算法的相关知识。在 debug 的过程中也加深了对之前实验内容的印象。通过阅读 RR 调度算法，我理解了 ucore 进行调度的过程。在练习 2 中，自己动手编写调度算法很大程度上借鉴了练习 1 对调度过程的理解。最后的练习 3 回到了 trapframe 和 context 在进程切换时作用。体现出实验内容的层层递进，由宏观认识到具体实现再到细节的把握。

【参考文献】

《操作系统实验指导(清华大学)陈渝、向勇编著》