Lab6—调度器

一、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

二、实验内容：

练习0：填写已有实验

将lab1-5的代码填入lab6后，把lab5_result和lab6进行比较，可以发现以下几点差异：

1.proc.h中PCB结构体proc_struct新增6个成员变量：

- run_queue(ready queue)：ready队列；
- run_link：作为ready队列中连接的节点，可由run_link来找到相应的PCB；
- time_slice：当前进程剩余的时间片；
- lab6_run_pool：该进程在优先队列中的节点；
- lab6_stride：该进程的调度步进值；
- lab6_priority：该进程的调度优先级。

注：代码里面（包括实验指导书上）所说的run_queue实际上是指ready_queue

```
struct proc_struct {
    enum proc_state state;                      // Process state
    int pid;                                    // Process ID
    int runs;                                   // the running times of Proces
    uintptr_t kstack;                           // Process kernel stack
    volatile bool need_resched;                 // bool value: need to be
rescheduled to release CPU?
    struct proc_struct *parent;                 // the parent process
    struct mm_struct *mm;                       // Process's memory management
field
    struct context context;                     // Switch here to run process
    struct trapframe *tf;                       // Trap frame for current
interrupt
    uintptr_t cr3;                              // CR3 register: the base addr
of Page Directroy Table(PDT)
    uint32_t flags;                             // Process flag
    char name[PROC_NAME_LEN + 1];               // Process name
    list_entry_t list_link;                     // Process link list
    list_entry_t hash_link;                     // Process hash list
    int exit_code;                              // exit code (be sent to parent
proc)
    uint32_t wait_state;                        // waiting state
    struct proc_struct *cptr, *yptr, *optr;     // relations between processes
    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
    skew_heap_entry_t lab6_run_pool;            // FOR LAB6 ONLY: the entry in
the run pool
    uint32_t lab6_stride;                       // FOR LAB6 ONLY: the current
stride of the process
    uint32_t lab6_priority;                     // FOR LAB6 ONLY: the priority
of process, set by lab6_set_priority(uint32_t)
};
```

2.proc.c中alloc_proc新增6条初始化语句（也就是对上文中新增的6个成员变量的初始化）：

```
//LAB6 YOUR CODE : (update LAB5 steps)
    /*
     * below fields(add in LAB6) in proc_struct need to be initialized
     *     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
     *     skew_heap_entry_t lab6_run_pool;            // FOR LAB6 ONLY: the
entry in the run pool
     *     uint32_t lab6_stride;                       // FOR LAB6 ONLY: the
current stride of the process
     *     uint32_t lab6_priority;                     // FOR LAB6 ONLY: the
priority of process, set by lab6_set_priority(uint32_t)
     */
```

更新之后的alloc_proc是这样：

```
// alloc_proc - alloc a proc_struct and init all fields of proc_struct
static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
        proc->state = PROC_UNINIT;
        proc->pid = -1;
        proc->runs = 0;
        proc->kstack = 0;
        proc->need_resched = 0;
        proc->parent = NULL;
        proc->mm = NULL;
        memset(&(proc->context), 0, sizeof(struct context));
        proc->tf = NULL;
        proc->cr3 = boot_cr3;
        proc->flags = 0;
        memset(proc->name, 0, PROC_NAME_LEN);
        proc->wait_state = 0;
        proc->cptr = proc->optr = proc->yptr = NULL;

        //以下是新增的初始化语句（赋值0或NULL）
        proc->rq = NULL;
        list_init(&(proc->run_link));
        proc->time_slice = 0;
```

```
        proc->lab6_run_pool.left = proc->lab6_run_pool.right = proc-
>lab6_run_pool.parent = NULL;
        proc->lab6_stride = 0;
        proc->lab6_priority = 0;
    }
    return proc;
}
```

3.trap.c中的 trap_dispatch 函数需要更新：

```
/* LAB6 YOUR CODE */
        /* IMPORTANT FUNCTIONS:
         * run_timer_list
         *----------------------
         * you should update your lab5 code (just add ONE or TWO lines of code):
         *    Every tick, you should update the system time, iterate the timers,
and trigger the timers which are end to call scheduler.
         *    You can use one funcitons to finish all these things.
         */
//这部分在lab5中是这样的：
    ticks ++;
    if (ticks % TICK_NUM == 0) {
        assert(current != NULL);
        current->need_resched = 1;
    }
```

根据提示，只需调用 run_timer_list 这一函数即可完成所有工作：

```
    ticks++;
    assert(current != NULL);
    run_timer_list();
    break;
```

练习1：使用Round Robin调度算法

- 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

  sched_class结构体定义如下：

  其中除了char *name是调度器的名字外，还定义了5个函数指针，其指向的函数通过传入参数和函数名不难看出其完成的功能。

  - init：初始化就绪队列；
  - enqueue：将某个进程加入就绪队列；
  - dequeue：将某个进程从就绪队列中删除（runnable->running）；
  - pick_next：选择下一个可执行进程（也就是dequeue的目标进程）；
  - proc_tick：计时器，用于判断时间片是否用完。

  ```
  struct sched_class {
      // the name of sched_class
      const char *name;
      // Init the run queue
  ```

```
    void (*init)(struct run_queue *rq);
    // put the proc into runqueue, and this function must be called with
rq_lock
    void (*enqueue)(struct run_queue *rq, struct proc_struct *proc);
    // get the proc out runqueue, and this function must be called with
rq_lock
    void (*dequeue)(struct run_queue *rq, struct proc_struct *proc);
  // choose the next runnable task
    struct proc_struct *(*pick_next)(struct run_queue *rq);
  // dealer of the time-tick
    void (*proc_tick)(struct run_queue *rq, struct proc_struct *proc);
};
```

接下来分别查看各函数的定义（涉及到的函数均在default_sched.c中）：

1. RR_init()：

```
static void RR_init(struct run_queue *rq) { // 初始化就绪队列
    list_init(&(rq->run_list)); // 初始化队头
    rq->proc_num = 0; // 初始化进程数为0
}

//其中run_queue结构体定义如下
struct run_queue {
    list_entry_t run_list; // 队头
    unsigned int proc_num; // 就绪队列中进程数
    int max_time_slice; // 每个进程所能使用的最大时间片
  // For LAB6 ONLY
    skew_heap_entry_t *lab6_run_pool; // 这是在练习2中使用的优先队列中的节点
};
```

2. RR_enqueue()：进队有两种情况，一是把新进程加入就绪队列，二是正在运行的进程时间片到了重新加入队尾。

```
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))); // 前提是进程控制块指针非空（否则
也不用进队了）
    list_add_before(&(rq->run_list), &(proc->run_link)); // 把PCB中的
run_link插入rq->run_list
    if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice)
{ // 如果该进程的时间片为0或者该时间片大于所能分配给进程的最大时间片，则需要修改时间片
        proc->time_slice = rq->max_time_slice;
    }
    proc->rq = rq; // 加入进程池
    rq->proc_num ++; // 就绪进程数加一
}
//我的理解是proc->rq=rq指示当前进程在rq这个就绪队列中，但其在队列中具体的位置由
proc->run_link在rq->run_list中的位置决定
```

3. RR_dequeue()：出队列也就是进程状态由runnable转移到running。

```
 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); // 同时满足该进程控制块指针非空并且进程在给定的就绪队列中才继续往下
    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
}
```

4. RR_pick_next()：上一个函数RR_dequeue()将一个进程从就绪队列中移除，而至于移除哪一个进程则是本函数给出的结果。

```
//对于RR调度算法来说，采用FIFO策略即可，即选择队列中第一个元素
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)); // 选取就绪进程队列中的第一个元素
    if (le != &(rq->run_list)) { // le==&(rq->run_list)意味队列空，return NULL
        return le2proc(le, run_link); // 否则由le，run_link找到对应的PCB并返回
    }
    return NULL;
}
```

5. RR_proc_tick()：该函数在发生时钟中断的时候调用。

```
static void RR_proc_tick(struct run_queue *rq, struct proc_struct *proc)
{
    if (proc->time_slice > 0) {
        proc->time_slice --; // 每次时钟中断time_slice减1
    }
    if (proc->time_slice == 0) { // 如果本次中断时time_slice==0
      proc->need_resched = 1; // 则设置此进程的PCB成员变量need_resched为1，表示该进程时间片用完了。在下个时钟中断执行trap函数时会检测到proc->need_resched = 1，从而调用schedule函数，把当前执行进程放回就绪队列末尾，并从就绪队列中按pick_next的结果取出下一个进程执行。
    }
}
```

最后，用sched_class定义了一个类：

```
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,
};
```

RR_init是对就绪队列的初始化。在ucore运行的过程中，如果有进程要加入就绪队列，就需要调用RR_enqueue，前面说过入队列包括了两种情况，然而对于该函数来讲其实并不关心是哪种情况，总之先把proc->run_link加入队尾，然后检查时间片，若需要重新赋值就再赋一次值，就绪进程数+1。若调用了schedule函数，则涉及到三个操作，一是需要将当前运行的进程加入就绪队列，二是要从该队列中选取一个进程，三是要把这个进程出队列，那么这个过程就会涉及到RR_pick_next以及RR_dequeue，在RR调度策略里，选择很简单，只需选取队首元素即可，出队也是一样，直接把队列中该进程的run_link删除即可，然后就绪进程数-1。最后就是RR_proc_tick，这个函数在发生时钟中断时调用，用来把当前执行进程的time_slice-1，如果某次发现time_slice减到了0，说明该进程时间片到了，need_resched置1，这样在下一次时钟中断时就会调用schedule完成进程的切换。

- 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

  我的理解是，多级反馈队列应该是指多个就绪队列，主要是针对从running状态回到runnable状态的进程。由于使用一个就绪队列完全是可以的，那么使用多个队列的不同就是可以设置队列优先级。故可以考虑给出多个队列，不同队列之间具有不同的优先级，在 pick_next 的时候优先考虑选择优先级高的就绪队列，而单个队列内部依旧采用RR调度。至于优先级的划分，如果是新入队的进程可以考虑放到优先级最高的队列，而从CPU上退下来的进程可以根据时间片使用情况来进行区分（如时间片用完，则放到低优先级，其余按剩余时间的多少按优先级从高到低排列，这里和上文中RR调度就有一点区别了，RR调度是当时间片用完才发生进程切换，而此时每一次时钟中断都会导致 time_slice-1 以及进程切换）。

练习2：实现 Stride Scheduling 调度算法

基本思想：（参照实验指导书的内容）

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（只与进程的优先权有关系）。
4. 在一段固定的时间之后，回到 2 步骤，重新调度当前stride最小的进程。

最后还可以看到一段以优先队列实现 Stride Scheduling 算法的伪码：

- init(rq):

  – Initialize rq->run_list

  – Set rq->lab6_run_pool to NULL

  – Set rq->proc_num to 0

- enqueue(rq, proc)

  – Initialize proc->time_slice

  – Insert proc->lab6_run_pool into rq->lab6_run_pool

  – rq->proc_num ++

- dequeue(rq, proc)

  – Remove proc->lab6_run_pool from rq->lab6_run_pool

  – rq->proc_num --

- pick_next(rq)

  – If rq->lab6_run_pool == NULL, return NULL

– Find the proc corresponding to the pointer rq->lab6_run_pool

– proc->lab6_stride += BIG_STRIDE / proc->lab6_priority

– Return proc

- proc_tick(rq, proc):

  – If proc->time_slice > 0, proc->time_slice --

  – If proc->time_slice == 0, set the flag proc->need_resched

下面就按照要求先用default_sched_stride_c的内容覆盖default_sched.c。打开现在的default_sched.c发现其中内容仍然包含

- init
- enqueue
- dequeue
- pick_next
- proc_tick

这五个函数，这不过现在是stride_init，stride_enqueue之类……而这5个函数，就是本次实验中我们要补充的内容。

同样的，最后还有一个类的声明：

```
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,
};
```

并且在开头还有一个函数 proc_stride_comp_f 用来比较两个进程的stride值（直接相减）。

```
static int
proc_stride_comp_f(void *a, void *b)
{
     struct proc_struct *p = le2proc(a, lab6_run_pool);
     struct proc_struct *q = le2proc(b, lab6_run_pool);
     int32_t c = p->lab6_stride - q->lab6_stride;
     if (c > 0) return 1;
     else if (c == 0) return 0;
     else return -1;
}
```

通过阅读代码可以发现，本实验是提供了两种数据结构的实现方法，一种是之前RR调度里使用的队列（list），另一种就是优先队列（实际就是实现了一个斜堆 skew_heap —— 树形结构），考虑到 Stride Scheduling 算法的核心是要从就绪队列中选取一个 stride 值最小的进程，那么使用 skew_heap 显然会有更高的效率，因为只需要取根节点即可而不用遍历整个队列，于是我也采用 skew_heap 的方式来实现。其中各函数的注释还是较为详细，对比之前给出的伪码外加入队函数 skew_heap_insert 和出队函数 skew_heap_remove 的使用方法在实验指导书中也有给出，因此代码并不难写。

下面还是一个一个来看，我主要挑各函数里面与RR调度算法的区别来讲：

- stride_init()：与之前的区别只是加入了对 skew_heap 的初始化

```
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(&(run_list)); // 这条语句实际没什么用（因为没有用list）
    rq->lab6_run_pool = NULL;
    rq->proc_num = 0;
}
```

- stride_enqueue()：区别就在之前是把proc->run_link加入rq->run_list，但是现在不使用list而是用skew_heap，所以是把 proc->lab6_run_pool 插入到以 rq->lab6_run_pool 为根的斜堆中去，插入同时根据 proc_stride_comp_f 比较的结果调整根节点。其余都没区别。

```
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
      */
    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) { // 如果
该进程的时间片为0或者该时间片大于所能分配给进程的最大时间片，则需要修改时间片
        proc->time_slice = rq->max_time_slice;
    }
    proc->rq = rq; // 加入进程池
    rq->proc_num ++; // 就绪进程数加一
}
```

- stride_dequeue()：list_del_init(&(proc->run_link)); 变为 skew_heap_remove(rq->lab6_run_pool, &(proc->lab6_run_pool), proc_stride_comp_f);

```
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
      */
    rq->lab6_run_pool = skew_heap_remove(rq->lab6_run_pool, &(proc-
>lab6_run_pool), proc_stride_comp_f);
    rq->proc_num --; // 就绪进程数减一
}
```

- stride_pick_next()：skew_heap 的查找优越性就体现在这里。

```
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); // 直接取根即可
    if (p->lab6_priority == 0)
        p->lab6_stride += BIG_STRIDE;
    else
        p->lab6_stride += BIG_STRIDE / p->lab6_priority;
    return p;
}
```

- stride_proc_tick()：无变化。

```
static void
stride_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
     /* LAB6: YOUR CODE */
    if (proc->time_slice > 0) {
        proc->time_slice --; // 每次时钟中断time_slice减1
    }
    if (proc->time_slice == 0) { // 如果本次中断时time_slice==0
        proc->need_resched = 1; // 则设置此进程的PCB成员变量need_resched为1，表示该进程时间片用完了。在下个时钟中断执行trap函数时会检测到proc->need_resched = 1，从而调用schedule函数，把当前执行进程放回就绪队列末尾，而从就绪队列中按pick_next的结果取出下一个进程执行。
    }
}
```

提问：

1. 如何证明 STRIDE_MAX – STRIDE_MIN <= PASS_MAX ?

   考虑将STRIDE_MIN移到式子右边，得到 STRIDE_MAX <= PASS_MAX + STRIDE_MIN 即 STRIDE_MAX <= STRIDE_MIN_NEXT ，其中 STRIDE_MIN_NEXT 是当前有最小步进值的进程被调度后STRIDE加上PASS之后新的STRIDE值。如果 STRIDE_MAX > STRIDE_MIN_NEXT ，那意味着有一个进程（假设为p）永远不会被调度，并且其余进程的STRIDE不断虽然不断增加但却永远也到达不了 STRIDE_MAX 即当前 p 的 STRIDE 值，这可能不太符合算法的思想，所以有 STRIDE_MAX <= STRIDE_MIN_NEXT ，即 STRIDE_MAX – STRIDE_MIN <= PASS_MAX 。

2. 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?

   由于`max_stride - min_stride <= BIG_STRIDE`，并且 ucore 中 BigStride 用的是无符号 32 位整数，最大值只能是2^32-1，而又因为是无符号的，因此，最小只能为 0，而且我们需要把 32 位无符号整数进行比较，需要保证任意两个进程 stride 的差值在 32 位有符号数能够表示的范围之内，故 BIG_STRIDE 的最大值为 (2^32-1)/2 。

这个在 default_sched.c 中开头也要求了我们来自己定义BigStride，这里就定义的是最大值 (2^32-1)/2 即 0x7FFFFFFF。

```
/* You should define the BigStride constant here*/
/* LAB6: YOUR CODE */
#define BIG_STRIDE    0x7FFFFFFF /* ??? */
```

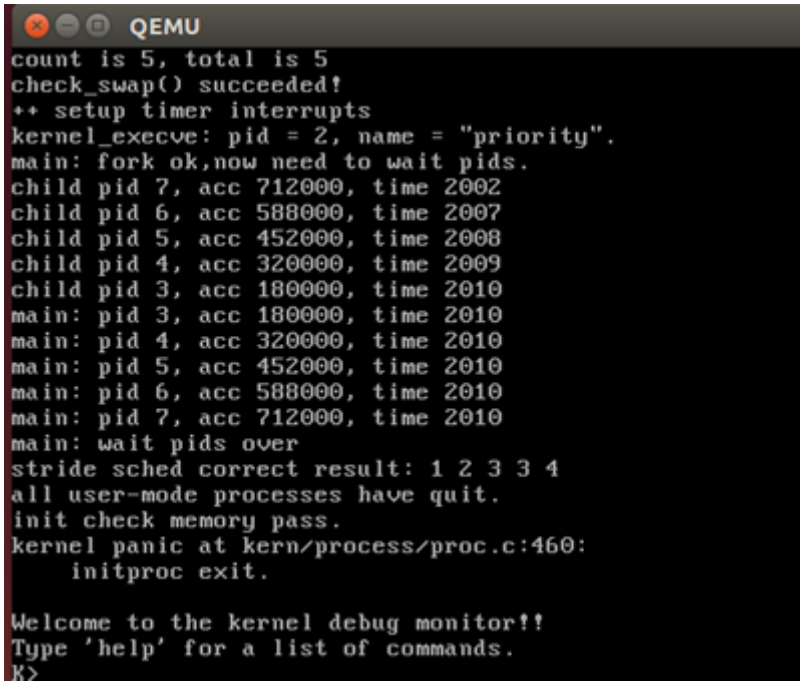
三、实验结果：

执行make grade：

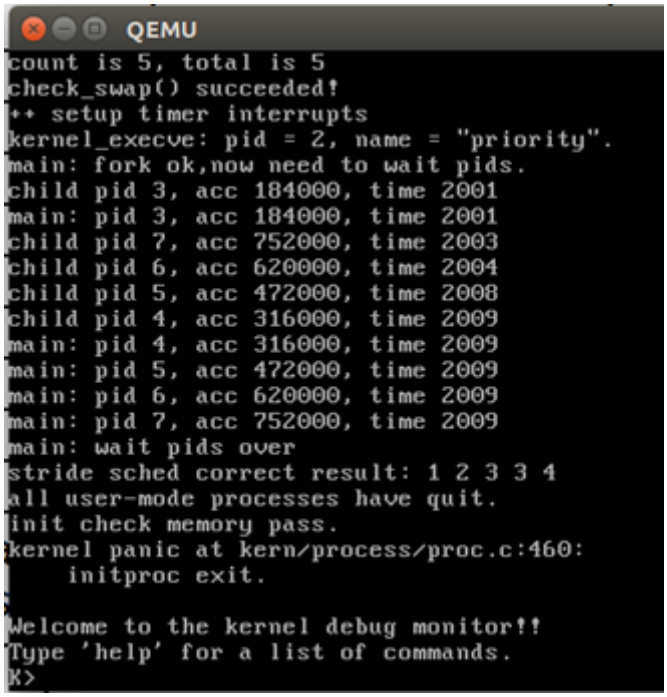
```
   !! error: missing 'check_slab() succeeded!'

forktree:                 (1.9s)
  -check result:                                WRONG
   !! error: missing 'init check memory pass.'

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

matrix:                   (11.5s)
  -check result:                                WRONG
   !! error: missing 'init check memory pass.'

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

priority:                 (21.9s)
  -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
```

这个地方不太清楚为什么Total Score只有91..但是我用答案 make 出来也是这样...

执行make qemu：



执行make run-priority：



从这个输出来看，似乎也没什么问题。

四、实验总结：

本次实验还是以看代码为主，理解了已有的 RR 调度算法的流程，在此基础之上完成 Stride Scheduling 调度算法并不是很难。本质上来说，它们完成的功能是一样的，只是采用了不同的数据结构，因而在不同的数据结构上会有不同的算法。但是除了具体实现的函数存在差别以外，思路和流程以及函数完成的逻辑功能其实是一致的（除了pick_next）。那么通过本次实验，主要就是认识了进程调度过程中需要完成的工作，但总体来说认识还很片面，包括实验报告中的语言大多是比较概括性的，不是太具体。还有就是对于不同的调度算法（不局限于这里的 RR 和 Stride Scheduling ）其 pick_next 函数肯定是不同的，但是其他诸如入队出队的操作除了取决于数据结构有不同的实现外应该好像都没有什么太大区别。