华东师范大学计算机科学与技术学院上机实践报告

课程名称：操作系统	**年级**：2022 级	**上机实践日期**：2024 年 6 月 10 日
指导教师：李东	**姓名**：岳锦鹏	**学号**：10213903403
实验名称：实验六十七　调度器、同步互斥		

一、 完成相关实验内容后，回答以下问题：

Lab6:

1、 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。

先看代码：

```
// The introduction of scheduling classes is borrrowed from Linux, and makes the
// core scheduler quite extensible. These classes (the scheduler modules) encapsulate
// the scheduling policies.
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);
    /* for SMP support in the future
     *  load_balance
     *     void (*load_balance)(struct rq* rq);
     *  get some proc from this rq, used in load_balance,
     *  return value is the num of gotten proc
     *  int (*get_proc)(struct rq* rq, struct proc* procs_moved[]);
     */
};
```

`name` 即为调度类的名称，`init` 用来初始化运行队列（数据结构不一定是链表，可以是任何数据结构），`enqueue` 是（创建新的进程的时候）进程入队的函数；`dequeue` 是（进程结束的时候）进程出队的函数；`pick_next` 用来在就绪（RUNNABLE）状态的进程中选出下一个将要调度的进程；`proc_tick` 是在时钟中断时需要执行的函数。

`labcodes_answer/lab6_result` 中实现的是 stride 调度算法，而 Round Robin 调度算法可以在 `labcodes/lab6` 中找到：

```
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,
};
```

对于 Round Robin(时间片轮转), μCore 的调度执行过程如下：调用 `RR_init` 初始化一个队列。之后每次触发时钟中断时，进入 `trap_dispatch` ，根据 `labcodes_answer/lab7_result/kern/trap/trap.c` ：

```
/* 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.
    */
```

应该更新系统计时器，并且调用调度器的中断处理函数 `RR_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;
    }
}
```

在中断处理函数中更新进程还剩余的时间片（`proc->time_slice` ，如果时间片用完了就把进程标记为需要重新调度。执行完 `trap_dispatch` 之后，在 `trap` 函数中有这样几行：

```
if (current->need_resched) {
    schedule();
}
```

这就是在当前进程需要重新调度时再次执行 `schedule` ，再看 `schedule` 的代码：

```
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);
        }
        if ((next = sched_class_pick_next()) != NULL) {
```

```
            sched_class_dequeue(next);
        }
        if (next == NULL) {
            next = idleproc;
        }
        next->runs ++;
        if (next != current) {
            proc_run(next);
        }
    }
    local_intr_restore(intr_flag);
}
```

这里当前进程是 PROC_RUNNABLE 状态，所以会把当前进程放入队列（末尾），之后把下一个进程（从队首）取出，如果下一个进程不是 idle 进程，就执行下一个进程。下一个进程继续执行直到时间片用完（或提前结束或进入等待状态），这样就实现了时间片轮转。

2、 如何在 uCore OS 中设计实现“多级反馈队列调度算法”？请给出概要设计，鼓励给出详细设计。

- 初始化函数：初始化所有的多级队列，每个队列有不同的时间片大小；
- 入队函数：新的进程应该进入最高优先级的队列；
- 出队函数：进程在哪个队列就从哪个队列里出队；
- 选出下一个调度的进程：按照优先级从高到低遍历每个队列，找到一个不为空的队列后，选出队首的进程；
- 时钟中断时执行：如果当前进程的时间片用完，这时候就需要把这个进程从当前队列中取出，放到更低的优先级队列中；如果已经是最低优先级队列，就按照时间片轮转的方式，从队首取出放到队尾；
- 时钟中断时，如果比当前进程更高的优先级队列中存在进程，那么将当前进程的状态改为 PROC_RUNNABLE 并调用 schedule 进行重新调度，这是为了实现高优先级对低优先级的抢占式调度；
- IO 中断时，将当前进程标记为等待 IO；等到调用 wakeup_proc 唤醒进程时，如果此进程被标记了等待 IO，那么此时将进程放到最高优先级队列，此时该进程会被优先调度，这是为了确保 IO 密集型的进程优先得到调度。

Lab7:

1、 uCore OS 与理论课的信号量机制的实现方案有何不同？请给出理论课中的信号量机制的实现方案的概要设计。

理论课中的信号量机制的实现方案直接简单地关闭中断，修改信号量的值，再开中断。但是在 μCore 中在进入临界区之后要进行进程的等待和唤醒。也就是在 __down 函数中，关闭中断后，访问 sem->value ，如果此信号量的值大于 0 那么正常修改后开中断。否则，说明进程需要等待，这时重新开启中断后会将进程放入等待队列，并且触发一次进程调度。

```
    bool intr_flag;
    local_intr_save(intr_flag);
```

```
    if (sem->value > 0) {
        sem->value --;
        local_intr_restore(intr_flag);
        return 0;
    }
    wait_t __wait, *wait = &__wait;
    wait_current_set(&(sem->wait_queue), wait, wait_state);
    local_intr_restore(intr_flag);

    schedule();
```

当该进程被唤醒，并且被再次调度到时，就将其移出等待队列：

```
    local_intr_save(intr_flag);
    wait_current_del(&(sem->wait_queue), wait);
    local_intr_restore(intr_flag);

    if (wait->wakeup_flags != wait_state) {
        return wait->wakeup_flags;
    }
```

在 __up 中，关闭中断后，如果当前的等待队列中有进程，那么需要唤醒队首的一个进程，并且将信号量的值加一，之后再开中断。

```
        if ((wait = wait_queue_first(&(sem->wait_queue))) == NULL) {
            sem->value ++;
        }
```

理论课中的信号量机制的实现方案只需要在 up 里关中断，修改信号量的值，开中断，在 down 里关中断，修改信号量的值，开中断。

2、 用户级信号量与内核级信号量有何不同？请给出在现有的内核级信号量基础上实现用户级信号量的概要设计。

用户级信号量的创建、访问、修改都是在用户态完成的，而内核级信号量的创建、访问、修改都是在内核态完成的。

在现有的内核级信号量基础上实现用户级信号量，最简单的方式是使用内核级信号量对一个用户空间中的变量当作临界资源访问，比如可以在用户空间中这样实现 PV 操作（以下 P 用 acquire 表示，V 用 release 表示）

```
#define SLEEP_TIME 10 // 检测间隔

typedef struct {
    int value; // 用户级信号量的值
    semaphore_t sem; // 内核级信号量
} user_semaphore;

void acquire(*user_semaphore) {
    while (1) { // 不断检测用户级信号量的值
```

```
        down(user_semaphore->sem); // 进入临界区
        if (user_semaphore->value > 0) {
            user_semaphore->value--; // 修改用户级信号量
            up(user_semaphore->); // 退出临界区
            return;
        }
        up(user_semaphore->sem); // 退出临界区
        do_sleep(SLEEP_TIME); // 用户进程将自己挂起
    }
}

void release(*user_semaphore) {
    down(user_semaphore->sem); // 进入临界区
    user_semaphore->value++; // 修改用户级信号量
    up(user_semaphore->sem); // 退出临界区
}
```

这里不需要 init 函数，因为创建变量的时候可以直接给 value 赋初值。

可以作个类比，由于不能一直处于关中断的状态（万一因为故障没有开中断），所以用关中断实现内核级信号量用来实现内核线程的同步互斥；由于不能一直处于内核态（权限过高不安全），所以用内核级信号量实现用户级信号量用来实现用户进程或线程的同步互斥。

当然也可以使用系统调用来实现用户级信号量，这里就不详述了。

二、 程序设计与实现的基本思路

虽然是任选一题完成，但是写完一个后顺手就把另一个写了。

Lab6:

1、 彩票进程调度算法,还是比较容易的,每个进程设置了一个新的属性:`uint32_t lab6_ticket_start;` 用来表示进程的第一个彩票的位置，并且使用原先就有的 `uint32_t lab6_priority;` 来表示进程的彩票个数(优先级越高彩票越多)。也就是 `lab6_ticket_start` 到 `lab6_ticket_start + lab6_priority` 这一部分都是进程拥有的彩票。

2、 但是后来发现这样的话在进程退出时很麻烦，彩票中间空了一段的话就很难随机抽取彩票了，所以不用 `uint32_t lab6_ticket_start;` 了，而是在选择下一个进程时把队列中的进程彩票数量（优先级）加起来和随机到的彩票比较；

3、 在运行队列中加了一个 `uint32_t lab6_total_num;` 属性，用来记录目前所有的进程总共有多少彩票；

4、 init 函数没什么改的，enqueue 函数里需要每次把 `lab6_total_num` 加上当前进程的优先级，dequeue 函数也是每次把 `lab6_total_num` 减去当前进程的优先级。这里要注意优先级可能为 0，但是如果一个进程有 0 张彩票的话那这个进程永远不会被调度到了，所以这里优先级加了 1。但在 `lab6_set_priority` 这个函数中已经有类似的操作了：

```
//FOR LAB6, set the process's priority (bigger value will get more CPU time)
void
lab6_set_priority(uint32_t priority)
{
    if (priority == 0)
        current->lab6_priority = 1;
    else current->lab6_priority = priority;
}
```

不过如果创建进程时没有调用 lab6_set_priority 那么默认的优先级好像就是 0，所以自己计算的时候加一还是有必要的。

5、 proc_tick 函数也不需要改，彩票调度也是每过一个时间片发一次彩票；

6、 最重要的就是 ticket_pick_next 函数了，每次调用时先随机抽取一张彩票 target_index：

```
uint32_t target_index = rand() % rq->lab6_total_num;
```

然后遍历队列，每次把 temp_ticket_num 加上当前进程的优先级（加一），如果加上之后达到了 target_index，那么也就说明这个彩票在这个进程拥有的彩票范围之间，那么就选择这个进程：

```
struct proc_struct *p;
int32_t temp_ticket_sum = 0;
while (le != &rq->run_list)
{
    p = le2proc(le, run_link);
    temp_ticket_sum += p->lab6_priority + 1;
    if (temp_ticket_sum >= target_index)
        break;
    le = list_next(le);
}
```

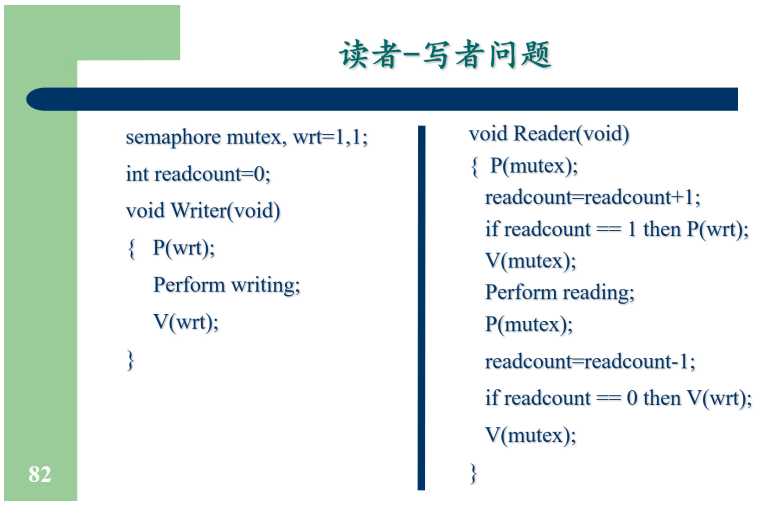
7、 之后就是要检验这个调度是否有问题了，只需要在 proc.c 中创建多个优先级不同的进程即可，会自动完成调度。(这里优先级为了看起来方便就使用了 (rand() % 100) * 100，也可以设置其他的优先级)

```
static int
my_test_user_main(void *arg) {
    int num = (int)arg;
    int priority = (rand() % 100) * 100;
    cprintf("process %d, priority %d\n", num, priority);
    lab6_set_priority(priority);
    cprintf("process %d end\n", num);
}
```

```
int i = 0;
for (; i < 10; i++) {
    int pid = kernel_thread(my_test_user_main, (void *)i, 0);
    struct proc_strucht *proc = find_proc(pid);
}
```

Lab7:

1、 读者写者问题的同步关系，更简单了，课件中已经给了代码，只需要稍微改改就能用：



2、 原先的代码在 lab7_result/kern/sync/check_sync.c 实现了哲学家就餐问题，这里就新建了一个 read_write_sync.c 实现读者写者问题，并在 proc.c 中将原先的 check_sync 改为 read_write_sync 。

3、 这里的读者写者问题只有一个缓冲区，信号量定义了两个，semaphore_t mutex 用来互斥，与课件中的 mutex 一样，semaphore_t synchronization 用来同步，相当于课件中的 wrt ，read_pos_write_neg 相当于课件中的 readcount 。

4、 但是对于信号量的初始化的操作要在主进程里做而不能在定义时赋初值，这里的主进程就是 read_write_sync ，在主进程中随机创建读者进程和写者进程（直到达到目标数量），并随机延迟，这样就能体现出读者和写者在不同情况下的互斥与同步是否正确；

5、 读者和写者的读操作和写操作也用 do_sleep 来模拟延迟，并且在每个读者或写者完成所有操作后打印读取的结果或写入的结果便于观察。

三、 **代码**

1、 https://gitea.shuishan.net.cn/10213903403/os_kernel_lab

2、 也可以看上传的附件。