

# 一、实验内容

## 1.1. 获取进程运行状态

### 1.1.1 实验步骤

#### a. 首先实现on\_state\_change函数

- 由于每次进入on\_state\_change函数前，均已经获取了进程的锁，故在函数中获取已运行的时间长度时，可以无需获取锁，以避免阻塞
- 而后将p->end赋值为当前时间，并根据不同的当前状态，在不同状态的累计运行时间上添加p->end-p->start
- 而后将p->start赋值为当前时间，以便于下一次调用时进行计算

```
int on_state_change(int cur_state, int nxt_state, struct proc *p) {
    p->end = ticks;
    switch(cur_state){
        case RUNNING:{
            p->running_time += p->end - p->start;
            break;
        }
        case RUNNABLE:{
            p->runnable_time += p->end - p->start;
            break;
        }
        case SLEEPING:{
            p->sleep_time += p->end - p->start;
            break;
        }
    }
    p->start = ticks;
    return 0;
}
```

#### b. 而后添加wait\_sched系统调用

- 添加系统调用的一系列流程在上次lab中已经说明，不再赘述，仅对实现进行说明
- 首先需要在进行状态变化的地方加入on\_state\_change函数以进行运行时长收集
  - userinit

```
on_state_change(p->state, RUNNABLE, p);
```

- fork

```
on_state_change(np->state, RUNNABLE, np);
```

- exit

```
on_state_change(p->state, ZOMBIE, p);
```

- scheduler

```
on_state_change(p->state, RUNNING, p);
```

- yield

```
on_state_change(p->state, RUNNABLE, p);
```

- sleep

```
on_state_change(p->state, SLEEPING, p);
```

- wakeup

```
on_state_change(p->state, RUNNABLE, p);
```

- kill

```
on_state_change(p->state, RUNNABLE, p);
```

- 而后由于wait\_sched与wait函数的功能大致相同，因此只需要将wait函数中返回子进程结束状态的语句替换为如下即可

```

if(copyout(p->pagetable, (uint64)runable_time, (char*)&np->runable_time, sizeof(np-
>runable_time)) < 0){
    release(&np->lock);
    release(&p->lock);
    return -1;
}
if(copyout(p->pagetable, (uint64)running_time, (char*)&np->running_time, sizeof(np-
>running_time)) < 0){
    release(&np->lock);
    release(&p->lock);
    return -1;
}
if(copyout(p->pagetable, (uint64)sleep_time, (char*)&np->sleep_time, sizeof(np-
>sleep_time)) < 0){
    release(&np->lock);
    release(&p->lock);
    return -1;
}

```

c. 最后在sysproc.c中添加如下函数，以便syscall调用即可

```

uint64 sys_wait_sched(void) {
    uint64 runable_time, running_time, sleep_time;

    if(argaddr(0, &runable_time) < 0){
        return -1;
    }
    if(argaddr(1, &running_time) < 0){
        return -1;
    }
    if(argaddr(2, &sleep_time) < 0){
        return -1;
    }

    return wait_sched((int*)runable_time, (int*)running_time, (int*)sleep_time);
}

```

## 1.1.2 实验测试结果（默认RR调度策略）

PID	Runnable Time	Running Time	Sleep Time
4	56 ticks	91 ticks	12 ticks
5	53 ticks	103 ticks	12 ticks
8	55 ticks	101 ticks	12 ticks
6	58 ticks	99 ticks	12 ticks

PID	Runnable Time	Running Time	Sleep Time
7	62 ticks	96 ticks	12 ticks

**Average Turnaround Time:** 166 ticks

```
hart 2 starting
hart 1 starting
init: starting sh
$ stat 5
PID: 4 | Runnable Time: 56 ticks | Running Time: 91 ticks | Sleep Time: 12 ticks
PID: 5 | Runnable Time: 53 ticks | Running Time: 103 ticks | Sleep Time: 12 ticks
PID: 8 | Runnable Time: 55 ticks | Running Time: 101 ticks | Sleep Time: 12 ticks
PID: 6 | Runnable Time: 58 ticks | Running Time: 99 ticks | Sleep Time: 12 ticks
PID: 7 | Runnable Time: 62 ticks | Running Time: 96 ticks | Sleep Time: 12 ticks
Average Turnaround Time: 166 ticks
$
```

## 1.2 优先级调度

### 1.2.1 实验步骤

a. 首先实现set\_priority系统调用

- 首先实现set\_priority
  - 遍历proc，寻找符合pid的进程
  - 获取锁后设置优先级，然后释放锁

```
int set_priority(int priority, int pid) {
    struct proc* p;
    for (p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);
        if (p->pid == pid) {
            p->priority = priority;
            release(&p->lock);
            return 0;
        }
        release(&p->lock);
    }
    return -1;
}
```

- 而后实现sys\_set\_priority，便于syscall调用，其中只需要利用argint函数获取相应的pid和priority然后调用set\_priority即可

```
uint64 sys_set_priority(void) {
    int pid, priority;

    if(argint(0, &priority) < 0){
        return -1;
    }
    if(priority < 0 || priority > 3){
        return -1;
    }
    if(argint(1, &pid) < 0){
        return -1;
    }
    return set_priority(priority, pid);
}
```

## b. 而后实现scheduler的PR部分

- 首先设置一个外层循环，当没有任何可运行进程时跳出
- 在循环中，需要遍历proc找到优先级最高的进程
  - 这里初始化highest\_priority为4，以便处理找到第一个RUNNABLE进程的情况
  - 在每次找到优先级最高的RUNNABLE进程时，**需要获取其锁，并在找到下一个RUNNABLE并且优先级更高的进程时才能释放**，以避免在遍历过程中该进程的状态发生变化(由于有多CPU)
  - 若找到了一个可以调度的进程，则执行与RR一致的操作即可，并需要将found置为0
  - 否则退出循环

```
for(;;){
    // Priority scheduling, iterating over max_p to find process with highest
    priority
    struct proc* max_p = 0;
    int highest_priority = 4;
    // First find the process with the highest priority and is RUNNABLE
    for (p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);
        if (p->state == RUNNABLE && p->priority < highest_priority) {
            if(max_p != 0){
                release(&max_p->lock);
            }
            highest_priority = p->priority;
            max_p = p;
            found = 1;
            continue;
        }
        release(&p->lock);
    }
}
```

```

        // If found such max_p, copy to p, and run it.
    if(found == 1){
        p = max_p;
        // TODO: on state change
        on_state_change(p->state, RUNNING, p);

        // Switch to chosen process. It is the process's job
        // to release its lock and then reacquire it
        // before jumping back to us.
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);

        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0;
        release(&p->lock);
        found = 0;
    }else{
        break;
    }
}

```

## 1.2.2 实验测试结果

```

init: starting sh
$ priostat 5
Set priority 0 to PID 4
Set priority 1 to PID 5
Set priority 2 to PID 6
Set priority 3 to PID 7
Set priority 0 to PID 8
PID: 4 | Runnable Time: 0 ticks | Running Time: 89 ticks | Sleep Time: 12 ticks
PID: 8 | Runnable Time: 0 ticks | Running Time: 91 ticks | Sleep Time: 12 ticks
PID: 5 | Runnable Time: 1 ticks | Running Time: 98 ticks | Sleep Time: 12 ticks
PID: 6 | Runnable Time: 72 ticks | Running Time: 79 ticks | Sleep Time: 12 ticks
PID: 7 | Runnable Time: 97 ticks | Running Time: 65 ticks | Sleep Time: 12 ticks
Average Turnaround Time: 130 ticks

```

PID	Runnable Time	Running Time	Sleep Time
4	0 ticks	89 ticks	12 ticks
8	0 ticks	91 ticks	12 ticks
5	1 tick	98 ticks	12 ticks
6	72 ticks	79 ticks	12 ticks
7	97 ticks	65 ticks	12 ticks

## 二、问题解答

---

### 2.1

---

- PR
  - 在PR策略下，PID为4、5、8的3个优先级较高的进程处于RUNNABLE状态的时间较短，能更优先获得CPU资源
  - 但是PID为6、7的2个进程相对前3个进程处于RUNNABLE状态的时间明显更长，这在实际系统中可能会导致某些进程长期得不到执行
  - 由于该策略下可以根据优先级调度进程，故在令运行时间需要较短的进程具有较高优先级时，可以让平均周转时间降低
- RR
  - 在RR策略中，所有进程处于RUNNABLE状态的时间较为平均，且响应时间较短
  - 但是由于较为频繁的上下文切换，所以可能在此方面有较大的开销，且平均周转时间也会较长

### 2.2

---

- PR
  - 在该策略下，每个进程被赋予一个优先级，调度器选择优先级高的进程执行
  - 低优先级的进程可能长时间得不到执行
  - 适用场景：该策略适用于需要优先处理高优先级任务的场景，例如实时系统等任务重要性具有显著差异的系统
- RR
  - 在该策略下，每个进程被分配一个固定的时间片，如果一个进程消耗完了其时间片，便会被挂起并切换到下一个进程
  - 所有进程都以相同的时间片进行调度，避免了某个进程长时间占用CPU的情况
  - 适用场景：该策略适用于时间共享系统、多用户环境等要求资源分配更具公平性的场景

### 2.3

---

- 任务一和任务二的代码实现已在上述部分进行说明