

# 模块实验创新：基于 xv6 的进程状态统计与动态优先级调度优化

## 1. 实验背景

在多任务操作系统中，调度器的设计直接影响系统性能和用户体验。传统的时间片轮转（RR）调度和静态优先级调度（SPS）虽各有优点，但往往难以同时兼顾任务的公平性和效率：

- 时间片轮转：公平性高，但无法优先处理高优先级任务。
- 静态优先级调度：高优先级任务得以优先处理，但容易导致低优先级任务饥饿（Starvation）。

**动态优先级调度（DPS）**通过结合 Aging机制，动态提升等待时间较长的低优先级任务优先级，平衡了公平性和效率。本实验的重点是实现和验证动态优先级调度在解决多任务调度问题中的优越性。

本次实验创新基于复旦大学操作系统课程Lab（基于xv6）进行迭代更新，主要修改代码已在下方实验实现中标出。

## 2. 实验目标与实现

### 2.1 任务一：获取进程运行状态

**实验目标：**实现系统调用 `wait_sched`，统计进程在以下状态的时间：

- `RUNNING`：进程实际占用 CPU 的时间；
- `RUNNABLE`：进程处于可运行状态但未获得 CPU 的时间；
- `SLEEPING`：进程因 I/O 或其他原因阻塞的时间。

#### 设计思路

- 在 `proc.h` 中为 `struct proc` 添加时间统计相关的成员变量：

```
uint64 created_time;    // 创建时间
uint64 finish_time;    // 完成时间
uint64 running_time;    // 运行时间
uint64 runnable_time;  // 可运行状态时间
uint64 sleep_time;     // 睡眠时间
uint64 start;          // 当前状态开始时间
uint64 end;            // 当前状态结束时间
```

- 使用 `on_state_change` 函数，在状态切换时记录每种状态的累计时间：

```
// UNUSED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE
int on_state_change(int cur_state, int nxt_state, struct proc *p) {
    uint64 current_time = ticks; // 当前时间
    p->end = current_time;       // 记录状态结束时间
    // 根据当前状态累加对应的时间
    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;
    default:
        break;
}
// 如果状态切换到 RUNNING, 重置等待时间
if (nxt_state == RUNNING) {
    p->wait_time = 0; // 清空等待时间
}
// 更新状态切换时间
p->start = current_time;
return 0;
}

```

3. 修改涉及状态切换的关键函数（如 `allocproc`、`scheduler`、`exit` 等），插入 `on_state_change` 调用。
4. 在 `proc.c` 中实现 `wait_sched` 系统调用，返回子进程的状态时间数据：

```

// get the running time, sleeping time, runnable time when the child process
returns
int wait_sched(int *runnable_time, int *running_time, int *sleep_time) {
    struct proc *np;
    int havekids, pid;
    struct proc *p = myproc();
    acquire(&p->lock);
    for (;;) {
        havekids = 0;
        for (np = proc; np < &proc[NPROC]; np++) {
            if (np->parent == p) {
                acquire(&np->lock);
                havekids = 1;
                if (np->state == ZOMBIE) {
                    // 子进程已经退出, 记录其 PID
                    pid = np->pid;
                    // 传递运行状态时间到用户空间
                    if (copyout(p->pagetable, (uint64)runnable_time, (char *)&np-
>runnable_time, sizeof(np->runnable_time)) < 0 ||
                        copyout(p->pagetable, (uint64)running_time, (char *)&np-
>running_time, sizeof(np->running_time)) < 0 ||
                        copyout(p->pagetable, (uint64)sleep_time, (char *)&np-
>sleep_time, sizeof(np->sleep_time)) < 0) {
                        release(&np->lock);
                        release(&p->lock);
                        return -1; // 出错, 返回 -1
                    }
                }
                // 释放子进程资源
                freeproc(np);
                release(&np->lock);
                release(&p->lock);
                return pid; // 成功返回子进程的 PID
            }
        }
    }
    release(&np->lock);
}

```

```

    }
}
// 如果没有子进程或父进程被杀死
if (!havekids || p->killed) {
    release(&p->lock);
    return -1;
}
// 等待子进程状态变化
sleep(p, &p->lock);
}
}

```

## 2.2 动态优先级调度的设计

### 核心思想

动态优先级调度通过 Aging（老化）机制解决低优先级任务饥饿问题。具体实现如下：

1. 每个进程的优先级范围为 0~3（0 为最高优先级，3 为最低优先级）。
2. 进程在每次调度中累积 `wait_time`（等待时间）。当等待时间超过阈值（`AGING_THRESHOLD`），进程优先级提升（数值减小）。
3. 调度器总是选择 `RUNNABLE` 状态中优先级最高的进程运行。

### 实现细节

- 在 `proc.h` 中为每个进程添加 `priority` 和 `wait_time` 字段。
- 修改调度器
  - 遍历所有 `RUNNABLE` 状态的进程。
  - 如果进程的 `wait_time` 超过阈值，提升优先级并重置等待时间。
  - 选择优先级最高的进程切换到运行状态。
- Aging 阈值设置为 `1 tick`，使优先级调整效果显著。

### 调度器核心代码实现

```

void scheduler(void) {
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0; // 当前 CPU 上没有正在运行的进程
    for (;;) {
        // 开启中断，防止死锁
        intr_on();
        int found = 0;
        #if defined RR
        // Round-Robin 调度
        for (p = proc; p < &proc[NPROC]; p++) {
            acquire(&p->lock);
            if (p->state == RUNNABLE) {
                // TODO: 状态切换
                on_state_change(p->state, RUNNING, p);
                // 运行选中的进程
                p->state = RUNNING;
                c->proc = p;
            }
        }
        if (!found) {
            // 如果没有可运行的进程，就休眠
            sleep(c, &c->lock);
        }
    }
}

```

```

    swtch(&c->context, &p->context); // 切换到进程上下文
    // 进程执行完毕，释放当前 CPU
    c->proc = 0;
    found = 1;
}
release(&p->lock);
}
#elif defined PR
// 优先级调度（支持动态 Aging）
struct proc *max_p = 0; // 当前优先级最高的进程
int highest_priority = 4; // 优先级范围是 0-3，初始化为比最低优先级更大的值
for (p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if (p->state == RUNNABLE) {
        // Aging 机制：如果进程等待时间超过阈值且优先级不是最高，则提升优先级
        p->wait_time++;
        if (p->wait_time >= 20 && p->priority > 0) {
            printf("Process %d: priority boosted from %d to %d (aging)\n",
                p->pid, p->priority, p->priority - 1);
            p->priority--; // 提升优先级（数字越小优先级越高）
            p->wait_time = 0; // 重置等待时间
        }
        // 找到当前优先级最高的进程
        if (p->priority < highest_priority) {
            highest_priority = p->priority;
            max_p = p;
        }
    }
}
release(&p->lock);
}
// 如果找到最高优先级的进程，运行它
if (max_p != 0) {
    acquire(&max_p->lock);
    if (max_p->state == RUNNABLE) {
        // 状态切换到 RUNNING
        on_state_change(max_p->state, RUNNING, max_p);
        max_p->state = RUNNING;
        max_p->wait_time = 0; // 重置等待时间
        c->proc = max_p;
        // 切换到选中的进程
        swtch(&c->context, &max_p->context);
        // 切换回来后，释放 CPU
        c->proc = 0;
        found = 1;
    }
    release(&max_p->lock);
}
#endif
// 如果没有 RUNNABLE 的进程，等待中断
if (found == 0) {
    intr_on();
    asm volatile("wfi"); // 等待中断
}
}
}

```

## 2.3 实验测试

实验程序模拟 5 个子进程，分配不同的初始优先级和计算任务（工作负载）。测试分为两种调度模式：

- 1. 动态优先级调度 (Dynamic Priority Scheduling) 。
- 2. 静态优先级调度 (Static Priority Scheduling) 。

### 实验代码

## 3. 实验结果与分析

### 3.1 实验数据

#### 任务一：进程状态统计

运行 `stat` 程序，输入 `stat 5`，得到以下结果：

```
$ stat 5
PID: 4 | Runnable Time: 36 ticks | Running Time: 49 ticks | Sleep Time: 12 ticks
PID: 7 | Runnable Time: 39 ticks | Running Time: 61 ticks | Sleep Time: 12 ticks
PID: 5 | Runnable Time: 37 ticks | Running Time: 63 ticks | Sleep Time: 12 ticks
PID: 6 | Runnable Time: 44 ticks | Running Time: 60 ticks | Sleep Time: 12 ticks
PID: 8 | Runnable Time: 44 ticks | Running Time: 60 ticks | Sleep Time: 12 ticks

Average Turnaround Time: 110 ticks
```

#### 任务二：静态与动态优先级调度

运行 `priostat` 程序，输入 `priostat 5`，分别测试静态和动态优先级调度：

##### 静态优先级调度

```
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 | Priority | Runnable Time | Running Time | Sleep Time
----|-----|-----|-----|-----
4   | 0       | 9           | 48          | 12
8   | 0       | 7           | 52          | 12
5   | 1       | 25          | 57          | 12
6   | 2       | 60          | 59          | 12
7   | 3       | 95          | 51          | 12

Average Turnaround Time: 104 ticks
```

##### 动态优先级调度

```
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

Process 6: priority boosted from 2 to 1 (aging)

Process 7: priority boosted from 3 to 2 (aging)

PID	Priority	Runnable Time	Running Time	Sleep Time
4	0	7	43	0
5	0	15	47	0
6	0	25	52	0
7	0	61	45	0
8	0	57	54	0

Average Turnaround Time: 84 ticks

### 3.2 分析与对比

- 任务一：实现了 `wait_sched` 系统调用，成功统计进程在各状态的运行时长，为调度算法性能分析提供了数据支持。
- 任务二：
  - 静态优先级调度中，低优先级任务（如 PID 7）长期处于 `RUNNABLE` 状态，造成饥饿问题。
  - 动态优先级调度通过 Aging 机制解决了饥饿问题，显著降低了平均轮转时间（104 ticks → 84 ticks）。

### 4. 结论

- 动态优先级调度通过 Aging 机制有效缓解了饥饿问题，改善了多任务系统的公平性。
- 实验结果证明了动态优先级调度在复杂任务场景下的优势。

### 改进方向

- 多级队列调度**：结合动态优先级调度，支持任务分类（如 I/O 密集型与 CPU 密集型）。
- 实时任务支持**：在调度算法中加入实时任务调度策略（如 EDF）。

通过此次实验，我们验证了动态优先级调度的优势，并为进一步优化调度算法奠定了基础。