

xv6 实验文档：进程调度算法实现

扩展自复旦大学操作系统课程实验

一、实验目标

本实验的目标是实现和修改 xv6 操作系统中的进程调度算法。原xv6本只支持轮询调度算法(Round Robin, RR)，我们将实现四种不同的调度算法，并根据不同的配置进行切换。具体来说，我们最终实现了以下几种调度算法：

- **轮询调度 (Round Robin, RR)**
- **优先级调度 (Priority Scheduling, PR)**
- **步长调度 (Stride Scheduling, Stride)**
- **多级反馈队列调度 (Multilevel Feedback Queue, MLFQ)**

通过实验，学习和理解调度算法的实现细节，深入了解操作系统调度器的工作原理，并评估不同调度算法在多任务环境下的性能表现。

二、实验环境

- **操作系统**：xv6
- **开发工具**：GCC 编译器，QEMU 虚拟机
- **硬件要求**：虚拟化支持的计算机（通过 QEMU 模拟硬件环境）

三、实验总体过程

1. **进程状态管理**：实现 `RUNNABLE`、`RUNNING`、`SLEEPING` 等进程状态的管理。
2. **调度算法的实现**：
 - **轮询调度**：每个进程按照顺序轮流执行，每次执行一个固定时间片。
 - **优先级调度**：每个进程根据预先设置的优先级进行调度，优先级高的进程先执行。
 - **步长调度**：每个进程拥有一个步长，调度时选择“pass”值最小的进程执行，步长调度可以实现更公平的资源分配。
 - **多级反馈队列调度**：将进程分为多个队列，每个队列有不同的优先级，进程随着时间片的耗尽在队列之间移动。
3. **调度器实现**：修改 `scheduler()` 函数，支持以上四种调度算法。
4. **调度算法的扩展**：可以通过宏定义切换不同的调度算法。

四、调度器实现

在 xv6 操作系统中，调度器的实现位于 `proc.c` 文件中的 `scheduler()` 函数。该函数是操作系统调度进程的核心，负责选择一个进程并将其切换到 CPU 上执行。

1. **轮询调度 (RR)**：

- 通过遍历进程表，选择处于 `RUNNABLE` 状态的进程。
- 进程按顺序轮流执行，每个进程占用固定的时间片。

2. **优先级调度 (PR)** :

- 每个进程有一个优先级值，优先级低的进程先执行。
- 进程会根据优先级进行选择，优先级高的进程优先调度。

3. **步长调度 (Stride)** :

- 每个进程有一个步长 (stride) , 它表示进程在调度中占有的权重。
- 进程的 `pass` 值根据步长递增，选择 `pass` 值最小的进程进行调度。

4. **多级反馈队列 (MLFQ)** :

- 将进程分为多个优先级队列，较高优先级队列的进程先执行。
- 如果进程的时间片用完，会从高优先级队列降级到低优先级队列。

五、核心代码分析

5.1 `proc`结构体的扩展

我们在 `struct proc` 结构体中添加了多个新的字段，用于支持调度算法的扩展和进程的状态统计。以下是这些字段的详细解释：

```
struct proc {
    struct spinlock lock;

    // p->lock must be held when using these:
    enum procstate state;      // 进程的当前状态 (RUNNING, RUNNABLE, SLEEPING, ZOMBIE等)
    struct proc *parent;      // 父进程指针
    void *chan;                // 如果非零，表示进程在 chan 上睡眠
    int killed;                // 如果非零，表示进程已被终止
    int xstate;                // 退出状态，传递给父进程的 wait
    int pid;                   // 进程ID

    // these are private to the process, so p->lock need not be held.
    uint64 kstack;             // 进程内核栈的虚拟地址
    uint64 sz;                  // 进程的内存大小 (字节)
    pagetable_t pagetable;     // 用户页表
    struct trapframe *trapframe; // trampoline.S 的数据页
    struct context context;     // swtch() 调用的上下文
    struct file *ofile[NOFILE]; // 打开的文件
    struct inode *cwd;          // 当前工作目录
    char name[16];              // 进程名称 (调试用)

    // Newly added for Lab3
    uint64 created_time;        // 进程创建时间 (用于测量进程生命周期)
    uint64 finish_time;         // 进程完成时间
    uint64 running_time;        // 进程运行时间 (进程在 RUNNING 状态下的时间)
    uint64 runnable_time;       // 进程可运行时间 (进程在 RUNNABLE 状态下的时间)
    uint64 sleep_time;          // 进程睡眠时间 (进程在 SLEEPING 状态下的时间)
```

```
uint64 start;           // 进程状态开始时间（用于跟踪进程各状态的持续时间）
uint64 end;             // 进程状态结束时间（同上）

int priority;           // 进程优先级，范围为 [0,1,2,3]，其中 0 为最高优先级，3 为最低优先级
int weight;             // 进程的权重（用于步长调度）
int stride;             // 步长调度中的步长（stride）
int pass;               // 步长调度中的虚拟时间（pass value）

int queue_level;        // 当前进程所在的队列（多级反馈队列调度）
int time_quantum;        // 当前队列的时间片（每个队列有不同的时间片大小）
int ticks;              // 当前队列时间片已使用的时间
};
```

1. `created_time`、`finish_time`:

- `created_time`: 记录进程的创建时间，通常在进程启动时设置为系统的当前时间。
- `finish_time`: 记录进程的完成时间，通常在进程退出时设置为当前时间。

2. `running_time`、`runnable_time`、`sleep_time`:

- `running_time`: 记录进程处于 `RUNNING` 状态下的时间。该字段反映了进程实际运行的时间，通常在进程切换到 `RUNNING` 状态时开始计时，切换到其他状态时停止计时。
- `runnable_time`: 记录进程处于 `RUNNABLE` 状态下的时间。这可以用来衡量进程等待执行的时间。
- `sleep_time`: 记录进程处于 `SLEEPING` 状态下的时间。当进程处于睡眠状态时，这个字段会增加。

3. `start` 和 `end`:

- `start`: 记录进程每次状态变化时的开始时间。可以用来跟踪进程的状态切换时间。
- `end`: 记录进程每次状态变化时的结束时间。与 `start` 配合使用，可以精确计算每个状态的持续时间。

4. `priority`、`weight`、`stride`、`pass`:

- `priority`: 表示进程的优先级，数值越小优先级越高。优先级用于支持 **优先级调度** (Priority Scheduling)，在调度时选择优先级最高的进程。
- `weight`: 表示进程的权重，通常用于 **步长调度** (Stride Scheduling) 中，权重越大的进程可以分配到更多的 CPU 时间。
- `stride`: 步长调度中的步长值。每个进程的步长决定了其执行的频率，步长越小，进程的优先级越高，能够获得更多的 CPU 时间。
- `pass`: 步长调度中的虚拟时间 (pass value)，它随着进程的步长变化而递增。进程的虚拟时间决定了进程被调度的优先级。`pass` 值最小的进程会先执行。

5. `queue_level`、`time_quantum`、`ticks`:

- `queue_level`: 表示进程当前所在的队列。在 **多级反馈队列调度** (MLFQ) 中，进程被划分为多个优先级队列，每个队列对应一个不同的优先级。
- `time_quantum`: 表示当前队列的时间片大小。在多级反馈队列调度中，每个队列的时间片大小可以不同，通常高优先级队列的时间片较小。

- **ticks**: 表示当前队列中进程已使用的时间片。每当进程在该队列中运行时, **ticks** 会递增, 直到该进程的时间片用尽并降级到下一个队列。

5.2 Scheduler()函数的修改

我们根据不同的调度算法对 **xv6** 调度器 (**scheduler()** 函数) 进行修改。修改后的调度器能够支持 **RR**、**PR**、**Stride** 和 **MLFQ** 四种调度算法, 并在系统运行时根据定义的调度策略选择合适的进程进行调度。

以下是调度器实现的详细解析, 涵盖了每种调度算法的核心逻辑。

```
void scheduler(void) {
    struct proc *p;
    struct cpu *c = mycpu();

    c->proc = 0;
    for (;;) {
        // Avoid deadlock by ensuring that devices can interrupt.
        intr_on();

        int found = 0;

        // Note here that we have two different scheduling algorithms
        #if defined RR
        for (p = proc; p < &proc[NPROC]; p++) {
            acquire(&p->lock);
            if (p->state == RUNNABLE) {
                // 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);
                // on_state_change(RUNNING, RUNNABLE, p);
                // Process is done running for now.
                // It should have changed its p->state before coming back.
                c->proc = 0;

                found = 1;
            }
            release(&p->lock);
        }
        #elif defined PR
        // Priority scheduling
        struct proc *max_p;
        max_p = 0;
        int highest_priority = 4; // Initialize to lowest priority
```

```
// Find the highest priority RUNNABLE process
for (p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if (p->state == RUNNABLE && p->priority < highest_priority) {
        if (max_p) release(&max_p->lock);
        max_p = p;
        highest_priority = p->priority;
    } else {
        release(&p->lock);
    }
}

// If a RUNNABLE process was found, schedule it
if (max_p) {
    found = 1;
    on_state_change(max_p->state, RUNNING, max_p);
    max_p->state = RUNNING;
    c->proc = max_p;
    swtch(&c->context, &max_p->context);
    c->proc = 0;
    release(&max_p->lock);
}
#elif defined Stride

// Stride Scheduling
struct proc *min_p = 0;
int min_pass = INT_MAX;

// Find the RUNNABLE process with the smallest pass value
for (p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if (p->state == RUNNABLE) {
        if (p->pass < min_pass) {
            if (min_p) release(&min_p->lock);
            min_p = p;
            min_pass = p->pass;
        } else {
            release(&p->lock);
        }
    } else {
        release(&p->lock);
    }
}

// If a RUNNABLE process was found, schedule it
if (min_p) {
    found = 1;

    // Execute the process
    on_state_change(min_p->state, RUNNING, min_p);
    min_p->state = RUNNING;
    c->proc = min_p;
    swtch(&c->context, &min_p->context);
    c->proc = 0;
}
```

```
// Update the pass value of the chosen process
min_p->pass += min_p->stride; // Update virtual time

release(&min_p->lock);
}

#elif defined MLFQ

// 假设我们有 NQUEUE 个队列，按优先级从高到低
for (int i = 0; i < NQUEUE; i++) {
    // 遍历队列 i 中的所有 RUNNABLE 进程
    for (p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);
        if (p->state == RUNNABLE && p->queue_level == i) {
            // 检查进程是否已耗尽当前队列的时间片
            if (p->ticks >= p->time_quantum) {
                // 时间片用完，降级到下一个队列
                p->queue_level = (p->queue_level < NQUEUE - 1) ? p->queue_level + 1 : p-
>queue_level;
                p->ticks = 0; // 重置时间片
            }

            // 选择一个进程并开始调度
            found = 1;
            on_state_change(p->state, RUNNING, p);
            p->state = RUNNING;
            c->proc = p;

            // 切换到该进程并运行
            swtch(&c->context, &p->context);
            c->proc = 0;

            // 更新当前进程的时间片
            p->ticks++; // 增加当前队列的已使用时间

            // 如果进程还没运行完，保持其在当前队列；如果运行完，进行降级
            if (p->ticks >= p->time_quantum) {
                p->queue_level = (p->queue_level < NQUEUE - 1) ? p->queue_level + 1 : p-
>queue_level;
                p->ticks = 0; // 重置时间片
            }

            release(&p->lock);
            break; // 只选择一个进程进行调度
        }
        release(&p->lock);
    }
    if (found) break; // 如果在某个队列中找到了可调度的进程，跳出循环
}
#endif

// The same as Round-Robin, if no RUNNABLE process is found, we will wait for interrupt
if (found == 0) {
```

```
    intr_on();
    asm volatile("wfi");
}
}
```

5.2.1 轮转调度算法 (Round-Robin, RR)

- **轮转调度算法**是一种简单的调度算法，每个进程按时间轮流获取 CPU 时间片。
- 在实现中，遍历所有进程（`proc` 数组），对于每个 `RUNNABLE` 状态的进程，分配 CPU 并执行。调度时进程的 `state` 被设置为 `RUNNING`，然后使用 `swtch()` 进行上下文切换。

5.2.2 优先级调度算法 (Priority Scheduling, PR)

- **优先级调度算法**通过 `priority` 字段选择优先级最低的进程进行调度（优先级数值越小，优先级越高）。
- 在该算法中，遍历所有进程，找到优先级最高的 `RUNNABLE` 进程，并将其切换到 `RUNNING` 状态执行。该进程的 `state` 被设置为 `RUNNING`，并执行上下文切换。

5.2.3 步长调度算法 (Stride Scheduling)

- **步长调度算法**使用虚拟时间（`pass`）和步长（`stride`）来决定进程的调度顺序。每个进程都有一个 `pass` 值，`pass` 值最小的进程将优先被调度。
- 每当进程运行后，`pass` 值会根据其步长（`stride`）进行更新。这样，步长较小的进程会被调度得更多。

5.2.4 多级反馈队列调度算法 (MLFQ)

- **多级反馈队列调度算法**结合了轮转调度和优先级调度，进程根据其行为（如时间片用尽）动态地在不同优先级队列间移动。
- 每个队列对应不同的时间片（`time_quantum`），进程在每个队列中运行的时间用 `ticks` 进行跟踪。如果进程耗尽了当前队列的时间片，它会被降级到一个较低的优先级队列。

5.3 调度算法的测试与验证

1. **`wait_sched`**：该系统调用允许父进程等待子进程的结束，并返回子进程的相关状态信息，包括可运行时间、运行时间和休眠时间。主要仿照`wait`系统调用扩展而来
2. **`on_state_change`**：该函数用于跟踪进程状态的变化，并在进程状态发生变化时记录状态持续时间的统计信息。

三、实验实现

在本实验中，主要实现了两个功能：

1. **`wait_sched` 系统调用**：
 - 该调用使父进程能够等待其子进程的结束，并收集关于子进程在各个状态下的时间信息（可运行时间、运行时间和休眠时间）。
 - 当子进程处于 `ZOMBIE` 状态时，父进程将得到该子进程的相关时间数据，并返回子进程的 PID（进程 ID）。

- 如果没有子进程，或者当前进程被杀死，系统调用将返回 `-1`。
- 通过使用 `copyout` 函数，将这些数据从内核空间拷贝到用户空间。

```
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) {
                    pid = np->pid;
                    int runnable = np->runnable_time;
                    int running = np->running_time;
                    int sleeping = np->sleep_time;

                    if ((runnable_time && copyout(p->pagetable, (uint64)runnable_time, (char
*)&runnable, sizeof(runnable)) < 0) ||
                        (running_time && copyout(p->pagetable, (uint64)running_time, (char
*)&running, sizeof(running)) < 0) ||
                        (sleeping && copyout(p->pagetable, (uint64)sleep_time, (char *)&sleeping,
sizeof(sleeping)) < 0)){

                        release(&np->lock);
                        release(&p->lock);
                        return -1;
                    }

                    // 设置结束时间
                    np->finish_time = ticks;
                    freeproc(np);
                    release(&np->lock);
                    release(&p->lock);
                    return pid;
                }
                release(&np->lock);
            }
        }
    }
    if (!havekids || p->killed) {
        release(&p->lock);
        return -1;
    }
    sleep(p, &p->lock);
}
}
```


- **进程查找与判断**: 通过遍历所有进程 (`proc` 数组), 寻找当前进程 `p` 的子进程。如果找到子进程且该子进程处于 `ZOMBIE` 状态 (即进程已经结束并等待父进程回收), 则父进程会获取该子进程的运行状态信息 (如: 运行时间、可运行时间和休眠时间)。
- **数据复制**: 使用 `copyout` 函数将子进程的运行状态信息从内核空间拷贝到用户空间。如果复制失败, 函数返回 `-1`。
- **清理与回收**: 子进程的结束时间被设置为当前系统时间 (`ticks`), 并通过 `freeproc(np)` 释放该进程。然后返回子进程的 `PID`, 表示该进程已经结束。

该函数确保了父进程能够准确地获取其子进程的状态信息, 并避免了死锁问题。当没有子进程或者父进程被杀死时, 函数会返回 `-1`。

2. `on_state_change` 函数:

- 用于处理进程状态的变更, 更新进程的状态持续时间, 并根据当前状态对相关时间进行累积统计 (运行时间、可运行时间、休眠时间)。
- 状态变更时, 更新 `start` 时间戳, 以便下一次状态变更时能够计算持续时间。

```
int on_state_change(int cur_state, int nxt_state, struct proc *p) {
    uint64 current_ticks = ticks;

    uint64 time_in_state = current_ticks - p->start;

    switch (cur_state) {
        case RUNNING:
            p->running_time += time_in_state;
            break;
        case RUNNABLE:
            p->runnable_time += time_in_state;
            break;
        case SLEEPING:
            p->sleep_time += time_in_state;
            break;
        case UNUSED:
        case ZOMBIE:
            break;
    }

    p->start = current_ticks;

    p->state = nxt_state;

    return 0;
}
```

- **状态持续时间统计**: 该函数计算进程在当前状态下的持续时间, 并根据状态类型更新相应的统计数据:
 - 如果当前状态是 `RUNNING`, 则增加 `running_time`。
 - 如果当前状态是 `RUNNABLE`, 则增加 `runnable_time`。
 - 如果当前状态是 `SLEEPING`, 则增加 `sleep_time`。
 - 对于 `UNUSED` 和 `ZOMBIE` 状态, 不进行时间统计。
- **时间戳更新**: 函数通过更新 `start` 时间戳来准备下一次状态变更的持续时间计算。

- **状态变更**：将进程的当前状态更新为 `nxt_state`。

该函数在每次进程状态发生变更时被调用，用于精确统计每种状态下的时间，使得系统能够准确地跟踪进程的生命周期及其资源消耗。

六、实验结果的测试

我们编写了一个名为 `priostat` 的用户程序，执行以下操作：

1. **初始化参数和验证输入**：用户输入参数 `n`，指定要创建的子进程数量。
2. **创建子进程并执行计算**：
 - 父进程使用 `fork` 创建 `n` 个子进程。
 - 每个子进程执行一个大规模的计算任务 (`big_calculation`)，并返回计算结果。
 - 在子进程中，我们为每个进程设置不同的优先级、权重、队列等级和时间片等调度参数。
3. **收集调度统计数据**：
 - 每个子进程完成计算后，父进程通过调用 `wait_sched` 系统调用来收集该子进程的运行时间、等待时间和休眠时间。
4. **计算和输出调度统计数据**：
 - 父进程累加所有子进程的运行时间、等待时间和休眠时间。
 - 计算所有进程的平均周转时间 (Turnaround Time)。

1. `big_calculation` 函数

```
uint64 big_calculation() {
    uint64 i, j, sum = 0;
    for (i = 0; i < MAGIC_NUM; i++) {
        for (j = 0; j < i; j++) {
            if (j & 1) {
                sum -= j;
            } else {
                sum += j;
            }
        }
        sum *= i;
        sum /= (i - j + 1);
        if (i % 11451 == 0) {
            sleep(1);
        }
    }
    return sum;
}
```

- 该函数执行一个大规模的计算任务，涉及两个循环，进行一些加减运算，并通过 `sleep(1)` 模拟进程的休眠操作。
- 该函数返回最终的计算结果。

2. 创建子进程并设置调度参数

```
int priority = i % 4; // 设置优先级
set_priority(priority, pid);

int weight = (i % 4) + 1;
set_weight(weight, pid);

int queue_level = i % 3; // 设置队列等级
int time_quantum = (i % 3) + 5; // 设置时间片
set_mlfq_initial(pid, queue_level, time_quantum, priority);
```

- 每个子进程被赋予一个优先级（0 到 3）、权重（1 到 4）和队列等级（0 到 2）。
- 使用 `set_priority`、`set_weight` 和 `set_mlfq_initial` 设置进程的调度参数。
- 这些设置将影响进程的调度策略（如优先级调度、MLFQ 调度等）。

3. 等待子进程并收集状态统计

```
int pid = wait_sched(&runable_time, &running_time, &sleep_time);
```

- 在父进程中，使用 `wait_sched` 系统调用等待子进程结束，并获取每个子进程的状态信息（如可运行时间、运行时间和休眠时间）。

4. 计算和输出平均周转时间

```
int avg_turnaround_time = (total_run_time + total_wait_time + total_sleep_time) / n;
printf("Average Turnaround Time: %d ticks\n", avg_turnaround_time);
```

- 根据所有子进程的运行时间、等待时间和休眠时间，计算并输出平均周转时间。

实际测试运行说明及截图验证

通过运行 `priostat` 程序，验证了进程调度统计功能是否正确。输出包括每个进程的 PID 以及其对应的运行时间、可运行时间和休眠时间。此外，程序还计算并输出了所有进程的平均周转时间。

1. 切换调度算法 在`proc.h`代码文件的末尾对使用的调度算法进行定义，从而选择需要使用的调度算法。

```
126 // TODO: using RR for Round-Robin, using PR for Priority-Scheduling, using Stride for Stride-Scheduling,
127 //use MLFQ for Multilevel Feedback Queue
128
129 // #define RR
130 // #define PR
131 // #define Stride
132 #define MLFQ
133
```

2. 实际测试 `make qemu`启动后，运行定义的用户调用`priostat` [测试的目的进程数]

3. 对比不同调度算法

```

init: starting on
$ priostat 7
Set priority 0 to PID 4
Set Weight 1 to PID 4
Set MLFQ - Queue Level: 0, Time Quantum: 0, Priority: 0 to PID 4
Set priority 1 to PID 5
Set Weight 2 to PID 5
Set MLFQ - Queue Level: 1, Time Quantum: 4, Priority: 1 to PID 5
Set priority 2 to PID 6
Set Weight 3 to PID 6
Set MLFQ - Queue Level: 2, Time Quantum: 8, Priority: 2 to PID 6
Set priority 3 to PID 7
Set Weight 4 to PID 7
Set MLFQ - Queue Level: 0, Time Quantum: 0, Priority: 3 to PID 7
Set priority 0 to PID 8
Set Weight 1 to PID 8
Set MLFQ - Queue Level: 1, Time Quantum: 4, Priority: 0 to PID 8
Set priority 1 to PID 9
Set Weight 2 to PID 9
Set MLFQ - Queue Level: 2, Time Quantum: 8, Priority: 1 to PID 9
Set priority 2 to PID 10
Set Weight 3 to PID 10
Set MLFQ - Queue Level: 0, Time Quantum: 0, Priority: 2 to PID 10
PID: 5 | Runnable Time: 111 ticks | Running Time: 89 ticks | Sleep Time: 12 ticks
PID: 4 | Runnable Time: 117 ticks | Running Time: 84 ticks | Sleep Time: 12 ticks
PID: 6 | Runnable Time: 112 ticks | Running Time: 91 ticks | Sleep Time: 12 ticks
PID: 8 | Runnable Time: 116 ticks | Running Time: 89 ticks | Sleep Time: 12 ticks
PID: 9 | Runnable Time: 115 ticks | Running Time: 90 ticks | Sleep Time: 12 ticks
PID: 7 | Runnable Time: 124 ticks | Running Time: 84 ticks | Sleep Time: 12 ticks
PID: 10 | Runnable Time: 121 ticks | Running Time: 85 ticks | Sleep Time: 12 ticks
Average Turnaround Time: 216 ticks

```

上图为RR轮转调度：时间片设为5，然后对所有进程进行轮转调度

```
init: starting sh
$ priostat 7
Set priority 0 to PID 4
Set Weight 1 to PID 4
Set MLFQ - Queue Level: 0, Time Quantum: 0, Priority: 0 to PID 4
Set priority 1 to PID 5
Set Weight 2 to PID 5
Set MLFQ - Queue Level: 1, Time Quantum: 4, Priority: 1 to PID 5
Set priority 2 to PID 6
Set Weight 3 to PID 6
Set MLFQ - Queue Level: 2, Time Quantum: 8, Priority: 2 to PID 6
Set priority 3 to PID 7
Set Weight 4 to PID 7
Set MLFQ - Queue Level: 0, Time Quantum: 0, Priority: 3 to PID 7
Set priority 0 to PID 8
Set Weight 1 to PID 8
Set MLFQ - Queue Level: 1, Time Quantum: 4, Priority: 0 to PID 8
Set priority 1 to PID 9
Set Weight 2 to PID 9
Set MLFQ - Queue Level: 2, Time Quantum: 8, Priority: 1 to PID 9
Set priority 2 to PID 10
Set Weight 3 to PID 10
Set MLFQ - Queue Level: 0, Time Quantum: 0, Priority: 2 to PID 10
PID: 4 | Runnable Time: 11 ticks | Running Time: 75 ticks | Sleep Time: 12 ticks
PID: 8 | Runnable Time: 11 ticks | Running Time: 75 ticks | Sleep Time: 12 ticks
PID: 5 | Runnable Time: 9 ticks | Running Time: 83 ticks | Sleep Time: 12 ticks
PID: 9 | Runnable Time: 80 ticks | Running Time: 84 ticks | Sleep Time: 12 ticks
PID: 6 | Runnable Time: 102 ticks | Running Time: 80 ticks | Sleep Time: 12 ticks
PID: 10 | Runnable Time: 108 ticks | Running Time: 80 ticks | Sleep Time: 12 ticks
PID: 7 | Runnable Time: 165 ticks | Running Time: 74 ticks | Sleep Time: 12 ticks
Average Turnaround Time: 160 ticks
```

上图为PR优先级调度:按照进程号模4得到优先级, 直接对进程进行按照优先级的调度

```
hart 1 starting
hart 2 starting
init: starting sh
$ priostat 7
Set priority 0 to PID 4
Set Weight 1 to PID 4
Set MLFQ - Queue Level: 0, Time Quantum: 0, Priority: 0 to PID 4
Set priority 1 to PID 5
Set Weight 2 to PID 5
Set MLFQ - Queue Level: 1, Time Quantum: 4, Priority: 1 to PID 5
Set priority 2 to PID 6
Set Weight 3 to PID 6
Set MLFQ - Queue Level: 2, Time Quantum: 8, Priority: 2 to PID 6
Set priority 3 to PID 7
Set Weight 4 to PID 7
Set MLFQ - Queue Level: 0, Time Quantum: 0, Priority: 3 to PID 7
Set priority 0 to PID 8
Set Weight 1 to PID 8
Set MLFQ - Queue Level: 1, Time Quantum: 4, Priority: 0 to PID 8
Set priority 1 to PID 9
Set Weight 2 to PID 9
Set MLFQ - Queue Level: 2, Time Quantum: 8, Priority: 1 to PID 9
Set priority 2 to PID 10
Set Weight 3 to PID 10
Set MLFQ - Queue Level: 0, Time Quantum: 0, Priority: 2 to PID 10
PID: 7 | Runnable Time: 28 ticks | Running Time: 81 ticks | Sleep Time: 12 ticks
PID: 6 | Runnable Time: 64 ticks | Running Time: 80 ticks | Sleep Time: 12 ticks
PID: 10 | Runnable Time: 52 ticks | Running Time: 89 ticks | Sleep Time: 12 ticks
PID: 9 | Runnable Time: 93 ticks | Running Time: 82 ticks | Sleep Time: 12 ticks
PID: 5 | Runnable Time: 91 ticks | Running Time: 88 ticks | Sleep Time: 12 ticks
PID: 8 | Runnable Time: 138 ticks | Running Time: 85 ticks | Sleep Time: 12 ticks
PID: 4 | Runnable Time: 135 ticks | Running Time: 89 ticks | Sleep Time: 12 ticks
Average Turnaround Time: 182 ticks
$
```

上图为Stride步长调度：按照进程号模4得到每一个进程的权重，并根据权重计算步长

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ priostat 7
Set priority 0 to PID 4
Set Weight 1 to PID 4
Set MLFQ - Queue Level: 0, Time Quantum: 0, Priority: 0 to PID 4
Set priority 1 to PID 5
Set Weight 2 to PID 5
Set MLFQ - Queue Level: 1, Time Quantum: 4, Priority: 1 to PID 5
Set priority 2 to PID 6
Set Weight 3 to PID 6
Set MLFQ - Queue Level: 2, Time Quantum: 8, Priority: 2 to PID 6
Set priority 3 to PID 7
Set Weight 4 to PID 7
Set MLFQ - Queue Level: 0, Time Quantum: 0, Priority: 3 to PID 7
Set priority 0 to PID 8
Set Weight 1 to PID 8
Set MLFQ - Queue Level: 1, Time Quantum: 4, Priority: 0 to PID 8
Set priority 1 to PID 9
Set Weight 2 to PID 9
Set MLFQ - Queue Level: 2, Time Quantum: 8, Priority: 1 to PID 9
Set priority 2 to PID 10
Set Weight 3 to PID 10
Set MLFQ - Queue Level: 0, Time Quantum: 0, Priority: 2 to PID 10
PID: 4 | Runnable Time: 28 ticks | Running Time: 61 ticks | Sleep Time: 12 ticks
PID: 5 | Runnable Time: 56 ticks | Running Time: 64 ticks | Sleep Time: 12 ticks
PID: 6 | Runnable Time: 48 ticks | Running Time: 81 ticks | Sleep Time: 12 ticks
PID: 7 | Runnable Time: 64 ticks | Running Time: 78 ticks | Sleep Time: 12 ticks
PID: 10 | Runnable Time: 116 ticks | Running Time: 86 ticks | Sleep Time: 12 ticks
PID: 8 | Runnable Time: 144 ticks | Running Time: 67 ticks | Sleep Time: 12 ticks
PID: 9 | Runnable Time: 156 ticks | Running Time: 66 ticks | Sleep Time: 12 ticks
Average Turnaround Time: 171 ticks
$ █
```

上图为MLFQ调度：按照进程号依次加入到三个队列中，每个队列设置依次增大的时间片和依次减小的优先级

六、实验总结

通过以上修改，我们实现了四种不同的调度算法，并且通过实验验证了它们的有效性：

- 1. **轮转调度 (RR)**：保证了系统的公平性，每个进程均等地分配 CPU 时间片。
- 2. **优先级调度 (PR)**：高优先级进程被优先调度，有效地确保了紧急任务能够及时处理。
- 3. **步长调度 (Stride)**：提供了一种资源分配公平的方法，按虚拟时间调度，防止某些进程长期占用 CPU。
- 4. **多级反馈队列调度 (MLFQ)**：综合了轮转调度和优先级调度的优点，能够动态调整进程的优先级，提高了系统的响应性。

七、参考文献

- xv6 官方代码及文档
- 复旦大学操作系统课程系列实验