

# 进程调度实验报告

本实验基于复旦大学2024os课程的scheduler lab，在原有的轮转调度基础上，实现了静态优先级调度、动态优先级调度、彩票调度、步长调度等调度方式，并且从轮转时间、响应时间、公平性（包括饥饿）、适应性等角度进行分析。

## 一、实验原理和背景知识

### 1. 轮转调度 (Round-Robin Scheduling)

原理：

好处：

缺点：

### 2. 静态优先级调度 (Static Priority Scheduling)

原理：

好处：

缺点：

### 3. 动态优先级调度 (Dynamic Priority Scheduling)

原理：

好处：

缺点：

### 4. 彩票调度 (Lottery Scheduling)

原理：

好处：

缺点：

### 5. 步长调度 (Stride Scheduling)

原理：

好处：

缺点：

调度算法的比较

## 二、实验内容

- 获取进程运行状态
- 设置进程优先级
- 随机数函数
- 优先级调度（静态、动态）
- 彩票调度
- 步长调度

## 三、测试与分析

- 测试代码实现
- 测试结果分析
  - 轮转调度 (Round Robin)
  - 静态优先级调度 (Static Priority Scheduling)
  - 动态优先级调度 (Dynamic Priority Scheduling)
  - 彩票调度 (Lottery Scheduling)
  - 步长调度 (Stride Scheduling)

综合结论：

# 一、实验原理和背景知识

---

- 我们本学期课程实验中实现了一个简单的轮转调度，考虑到还学习了很多其他调度算法的理论，如：静态优先级调度、动态优先级调度、彩票调度、步长调度等，在这里对其一一实现。MLFQ算法由于和我实现的动态优先级调度基本一致，在此没有明确实现（事实上将我实现的动态优先级调度改几个超参数基本就等效MLFQ算法）。

以下是所提到的调度算法的主要原理及其原理优缺点分析：

---

## 1. 轮转调度 (Round-Robin Scheduling)

**原理：**

- 每个进程被分配一个固定的时间片，按顺序依次执行，时间片结束时切换到下一个进程。
- 如果一个进程没有在时间片内完成，则重新放回队列尾部，等待下一轮。

**好处：**

- **公平性：**每个进程都能获得均等的CPU时间，不会有进程饥饿问题。
- **响应性：**适合交互式任务，能够快速响应用户的输入。

**缺点：**

- **上下文切换开销大：**频繁的上下文切换会导致系统开销增加。
  - **不适合长任务：**时间片过短时，长任务可能多次被中断，效率降低。
- 

## 2. 静态优先级调度 (Static Priority Scheduling)

**原理：**

- 每个进程分配一个固定的优先级，调度时总是选择优先级最高的进程。
- 优先级在进程生命周期内不变。

**好处：**

- **简单高效：**适用于实时系统，能够保证高优先级任务优先完成。
- **高优先级任务延迟最小：**重要任务的执行得到保障。

**缺点：**

- **饥饿问题：**低优先级任务可能长期得不到调度。
  - **适应性差：**优先级一旦设定就无法动态调整，难以适应实时环境中的变化。
-

### 3. 动态优先级调度 (Dynamic Priority Scheduling)

#### 原理：

- 每个进程的优先级会根据一定规则动态调整，比如随着等待时间增加提高优先级、长时间运行的进程降低优先级。
- 常用于解决静态优先级中的饥饿问题。

#### 好处：

- **避免饥饿**：等待时间长的任务可以逐渐提升优先级，最终得到调度。
- **更灵活**：能够根据任务状态动态调整优先级，更加适应复杂场景。

#### 缺点：

- **实现复杂**：动态调整优先级的规则可能需要额外的计算开销。
  - **不确定性**：可能导致优先级波动，影响系统的预测性。
- 

### 4. 彩票调度 (Lottery Scheduling)

#### 原理：

- 给每个进程分配一定数量的彩票，调度时随机抽取一张彩票，拥有该彩票的进程获得执行权。
- 彩票数量通常与任务的权重相关，例如更重要的任务分配更多彩票。

#### 好处：

- **公平性和灵活性结合**：理论上每个进程都有被选中的机会，同时高优先级任务的执行概率更大。
- **适应性强**：可以通过动态调整彩票数量来改变优先级。

#### 缺点：

- **随机性带来的不确定性**：调度可能不够平稳，有时会导致高优先级任务延迟执行。
  - **复杂性**：彩票的管理和概率计算可能带来额外开销。
- 

### 5. 步长调度 (Stride Scheduling)

#### 原理：

- 每个进程分配一个步长值（与优先级相关联），并使用行程值（pass）来记录每个任务的执行顺序。
- 行程值越小，任务获得执行的频率越高。
- 调度时选择最小行程值的任务执行，并增加其步长值。

#### 好处：

- **公平性强**：能够精确控制任务的执行频率，避免随机性。
- **适应性好**：可以动态调整步长值，灵活实现优先级调度。

缺点：

- **计算复杂性**：需要实时维护虚拟时间和步长值，带来额外的系统开销。
- **初始配置难**：步长值的设置可能影响调度效果。

调度算法的比较

调度算法	效率（平均轮转时间）	公平性（响应时间极差，轮转时间极差）	饥饿问题	实现复杂度	适应性
轮转调度	低	高	无	简单	较低
静态优先级调度	高	中	存在	简单	较低
动态优先级调度	高	高	无	较复杂	高
彩票调度	中（概率）	高（概率）	无	中等	高
步长调度	中	高	无	较复杂	高

二、实验内容

首先展示共用部分主要代码，然后每个进程调度算法依次展示。部分小修改不展示。

为了展示调度算法的特点，我们设置程序只使用单核 CPU

共用部分

# 1: 获取进程运行状态

## 关键代码:

```
/*
 *      kernel/proc.c
 */

// UNUSED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE
int on_state_change(int cur_state, int nxt_state, struct proc *p) {

    uint64 current_time = ticks; // 假设 ticks 是系统全局时间

    // 设置进程初次调度的时刻
    if(p->first_run_time==0 && p->normal==1 && p->start != 0 &&
    cur_state==RUNNABLE && nxt_state==RUNNING){
        p->first_run_time=current_time;
        //printf("%d\n", (int)current_time);
    }
    // 计算在当前状态下的持续时间
    uint64 duration = current_time - p->start;
    // 更新当前状态的结束时间
    if (p->start != 0) { // 如果进程之前已经进入某个状态
        // 根据当前状态更新对应的时间
        switch (cur_state) {
            case RUNNING:
                p->running_time += duration;
                break;
            case RUNNABLE:
                p->runnable_time += duration;
                break;
            case SLEEPING:
                p->sleep_time += duration;
                break;
            // 可以根据需要处理其他状态
            default:
                break;
        }
    }

    // 更新状态的开始时间和结束状态
    p->start = current_time;
    return 0; // 无需返回值
}
```

## 设计思路:

- 上面是进程转换函数
- 对于每个状态转换的进程，如果是初次由 `RUNNABLE` 变为 `RUNNING` 会记录该进程的首次运行时间，用于之后获得响应时间。
- 此外对于每个进程，每从一个状态转移到另一状态，累加其对应的状态时间

## 2: 设置进程优先级

### 关键代码:

```
/*
 *      kernel/proc.c
 */

// set priority [0-3] to a given process [pid]
// -1 means error, 0 means success

int set_priority(int priority, int pid) {

    initlock(&random_lock, "random_lock"); // 初始化锁
    // 检查优先级范围
    if (priority < 0 || priority > 3) {
        return -1; // 返回错误
    }
    // 遍历进程表，找到指定的进程
    struct proc *p;
    for (p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);

        if (p->pid == pid) { // 找到指定的进程
            p->priority = priority; // 设置优先级
            p->tix=(4-priority); // 设置彩票数目
            p->normal=1; // 设置为正常进程
            release(&p->lock);
            return 0; // 返回成功
        }
        release(&p->lock);
    }
    return -1; // 找不到进程，返回错误
}
```

### 设计思路:

- 上面是优先级设置函数
- 根据参数设置 `pid` 对应进程的优先级并且将 `(4-priority)` 设置为进程彩票数。

## 3: 随机数函数

### 关键代码:

```
/*
 *      kernel/proc.c
 */

static uint32 lcg_seed = 12345; // 随机数种子
static struct spinlock random_lock; // 用于保护 lcg_seed 的锁

// 随机数生成函数
int random(int max) {
    acquire(&random_lock); // 获取锁，确保线程安全

    // 使用线性同余法生成伪随机数
    lcg_seed = (lcg_seed * 1103515245 + 12345) % (1 << 31);

    int result = lcg_seed % (max + 1); // 生成 0 到 max 之间的随机数

    release(&random_lock); // 释放锁

    return result;
}
```

### 设计思路:

- 由于 xv6 没有随机数库，我查阅资料实现了一个简单的线性同余法生成伪随机数函数。
- 该函数主要用于彩票调度使用，同时用在需要优先级的调度中：如果两个进程优先级相同，则随机选择一个调度，以此避免顺序的影响并增加随机性。

## 各个进程调度算法实现

下面介绍每个进程调度算法实现，这些代码主要部分都在 `kernel/proc.c/scheduler` 这个 `nonreturn` 函数中，调度算法的选择通过宏定义条件编译：通过修改 `kernel/proc.h` 的宏定义来设置 `scheduler` 函数的具体算法。

# 1: 优先级调度 (静态、动态)

## 关键代码:

```
/*
 *      kernel/proc.c
 */

// 动态调度专用: 设置进程运行时间和等待时间
void update_ticks(struct proc *p) {
    if (p->state == RUNNABLE) {
        p->wait_ticks++; // 增加等待时间
    }
    if (p->state == RUNNING) {
        p->cpu_ticks++; // 增加 CPU 时间
    }
}

// 根据运行等待状况, 调整优先级
void adjust_priority(struct proc *p) {
    // 如果进程的 CPU 时间过长, 降低优先级
    if (p->cpu_ticks >= 10) { // 假设每 10 个时钟周期调整一次
        if (p->priority < 3) {
            p->priority++; // 降低优先级
        }
        p->cpu_ticks = 0; // 重置 CPU 时间
    }

    // 如果进程在等待队列中等待时间过长, 提升优先级
    if (p->wait_ticks >= 20) { // 假设等待时间超过 20 个时钟周期
        if (p->priority > 0) {
            p->priority--; // 提升优先级
        }
        p->wait_ticks = 0; // 重置等待时间
    }
}

// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns. It loops, doing:
//  - choose a process to run.
//  - swtch to start running that process.
//  - eventually that process transfers control
//    via swtch back to the scheduler.
void scheduler(void) {

    struct cpu *c = mycpu();

    c->proc = 0;
    int cur=0;
    int flag=1;
```



```

for(;;){
if(flag||(ticks-cur)>=1) {
    flag=0;
    cur=ticks;
    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();

    int found = 0;

    #if defined RR
    // 略
    #elif defined PR
    // Priority scheduling, iterating over max_p to find process with highest
priority
    // First find the process with the highest priority and is RUNNABLE
    // If found such max_p, copy to p, and run it.

    struct proc *p;

    // 1. 更新每个进程的ticks
    for (p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);
        update_ticks(p); // 更新进程的ticks
        release(&p->lock);
    }

    // 2. 调整每个进程的优先级
    for (p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);
        adjust_priority(p); // 调整优先级

        release(&p->lock);
    }

    struct proc *max_p = 0; // 用于存储优先级最高的进程

    // 在遍历过程中，先锁定所有进程
    for (p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock); // 获取当前进程的锁

        if (p->state == RUNNABLE) {
            // 如果当前进程的优先级更高（数值更小）或者还没有找到RUNNABLE进程
            if (max_p == 0 || p->priority < max_p->priority) {
                // 如果当前进程有更高的优先级，更新 max_p
                //printf("%d",p->priority );
                if (max_p != 0) {
                    release(&max_p->lock); // 释放之前选中的进程锁
                }
                max_p = p; // 更新 max_p 为当前进程
            } else if (p->priority == max_p->priority) {
                // 如果当前进程和 max_p 的优先级相同
                if (random(1)) {
                    // 选择当前进程
                    if (max_p != 0) {

```

```

        release(&max_p->lock); // 释放之前选中的进程锁
    }
    max_p = p;
} else {
    release(&p->lock);
}
} else {
    release(&p->lock);
}
} else {
    release(&p->lock);
}
}

// If found such max_p, copy to p, and run it.
if (max_p != 0) {

    p = max_p;
    on_state_change(RUNNABLE, RUNNING, p);

    p->state = RUNNING;
    c->proc = p;
    swtch(&c->context, &p->context);
    c->proc = 0;
    release(&p->lock);
    found = 1;
}

#elif defined LS
//略
#elif defined SS
//略
#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");
}
}
}
}

```

## 设计思路：

- `update_ticks` 函数设置更新进程运行时间和等待时间，
- 在上述函数基础上，根据运行等待状况，`adjust_priority` 函数调整优先级
- **注意：调整的幅度是超参数。如果设置为运行完一定时间片下降一级，而等待一段时间直接上升为最高优先级，那么就等价于 MLFQ 调度。如果设置运行时间和等待时间都大于一个极大的数才调整优先级，那么就退化为 静态优先级调度**
- `#elif defined PR` 后的是主要代码，先更新每个进程的运行时间和等待时间ticks，再调整每个进程的优先级。对于处于等待状态的进程，遍历所有进程找到最高优先级的进程，如果有多个相同最高优先级的进程就随机选择一个。对于选中的进程，并调用上下文切换函数 `swtch` 将 CPU 控制权切换到选中进程。调度完成后，清空 CPU 的进程信息并释放锁。

## 实验测试结果

- 静态调度

```
$ 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: 81 ticks| Running Time: 68 ticks| Sleep Time: 12 ticks|
Response Time: 1 ticks
PID: 5 | Runnable Time: 163 ticks| Running Time: 65 ticks| Sleep Time: 12 ticks|
Response Time: 163 ticks
PID: 6 | Runnable Time: 241 ticks| Running Time: 65 ticks| Sleep Time: 12 ticks|
Response Time: 241 ticks
PID: 8 | Runnable Time: 82 ticks| Running Time: 68 ticks| Sleep Time: 12 ticks|
Response Time: 2 ticks
PID: 7 | Runnable Time: 321 ticks| Running Time: 65 ticks| Sleep Time: 12 ticks|
Response Time: 321 ticks
Average Turnaround Time: 255 ticks
min_turnaround_time Time: 161 ticks
max_turnaround_time Time: 398 ticks
Average response Time: 145 ticks
min_response_time Time: 1 ticks
max_response_time Time: 321 ticks
```

- 动态调度

```
$ 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: 308 ticks| Running Time: 65 ticks| Sleep Time: 12 ticks|
Response Time: 1 ticks
PID: 5 | Runnable Time: 231 ticks| Running Time: 63 ticks| Sleep Time: 12 ticks|
Response Time: 20 ticks
PID: 6 | Runnable Time: 174 ticks| Running Time: 64 ticks| Sleep Time: 12 ticks|
Response Time: 41 ticks
PID: 7 | Runnable Time: 60 ticks| Running Time: 64 ticks| Sleep Time: 12 ticks|
Response Time: 60 ticks
PID: 8 | Runnable Time: 251 ticks| Running Time: 64 ticks| Sleep Time: 12 ticks|
Response Time: 2 ticks
Average Turnaround Time: 280 ticks
min_turnaround_time Time: 136 ticks
max_turnaround_time Time: 385 ticks
Average response Time: 24 ticks
min_response_time Time: 1 ticks
max_response_time Time: 60 ticks
```

## 2: 彩票调度

### 关键代码:

```
/*
 *      kernel/proc.c
 */

// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run.
//  - swtch to start running that process.
//  - eventually that process transfers control
//    via swtch back to the scheduler.
void scheduler(void) {

    struct cpu *c = mycpu();
```

```

c->proc = 0;
int cur=0;
int flag=1;
for(;;){
if(flag||(ticks-cur)>=1) {
    flag=0;
    cur=ticks;
    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();

    int found = 0;

    #if defined RR
    // 略
    #elif defined PR
    //略
    #elif defined LS
        struct proc *p=0;
        struct proc *sp=0;

        int counter = 0;

        // 获取彩票数目
        int summ=0;
        for (p = proc; p < &proc[NPROC]; p++) {
            acquire(&p->lock); // 获取当前进程的锁

            if (p->state == RUNNABLE) {

                summ+=p->tix;

            }
            release(&p->lock);
        }

        // 确认赢家位置
        int winner = random(summ);

        // 确认赢家进程
        for (p = proc; p < &proc[NPROC]; p++) {
            acquire(&p->lock); // 获取当前进程的锁
            if (p->state == RUNNABLE) {

                counter+=p->tix;
                if (counter>=winner) {
                    sp=p;
                    break;
                }
            }
            release(&p->lock);
        }

        if (sp != 0) {
            p=sp;

```

```

        on_state_change(RUNNABLE, RUNNING, p);

        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);
        c->proc = 0;
        release(&p->lock);
        found = 1;
    }

    #elif defined SS
    //略
    #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");
    }
}
}
}
}

```

## 设计思路：

- `#elif defined LS` 后的是主要代码。
- 首先遍历所有 `RUNNABLE` 进程求和彩票数 `summ`。
- 然后，随机函数 `random(summ)` 生成一个范围在 `[0, summ)` 内的随机数，用于确定抽到的“彩票”编号。
- 再然后，遍历所有 `RUNNABLE` 进程，累积其彩票数到 `counter`。当 `counter >= winner` 时，表示随机数对应的“彩票”已被找到，当前进程 `p` 即为赢家，指针 `sp` 被赋值为该进程。一旦找到，立即跳出循环。
- 对于选中的进程，并调用上下文切换函数 `swtch` 将 CPU 控制权切换到赢家进程。调度完成后，清空 CPU 的进程信息并释放锁。

## 实验测试结果

- 彩票调度

```
$ 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: 134 ticks| Running Time: 64 ticks| Sleep Time: 12 ticks|
Response Time: 5 ticks
PID: 8 | Runnable Time: 183 ticks| Running Time: 64 ticks| Sleep Time: 12 ticks|
Response Time: 2 ticks
PID: 5 | Runnable Time: 221 ticks| Running Time: 63 ticks| Sleep Time: 12 ticks|
Response Time: 9 ticks
PID: 6 | Runnable Time: 276 ticks| Running Time: 63 ticks| Sleep Time: 12 ticks|
Response Time: 16 ticks
PID: 7 | Runnable Time: 315 ticks| Running Time: 64 ticks| Sleep Time: 12 ticks|
Response Time: 1 ticks
Average Turnaround Time: 301 ticks
min_turnaround_time Time: 210 ticks
max_turnaround_time Time: 391 ticks
Average response Time: 6 ticks
min_response_time Time: 1 ticks
max_response_time Time: 16 ticks
```

### 3: 步长调度

#### 关键代码:

```
/*
 *      kernel/proc.c
 */

// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns. It loops, doing:
//  - choose a process to run.
//  - swtch to start running that process.
//  - eventually that process transfers control
//    via swtch back to the scheduler.
void scheduler(void) {

    struct cpu *c = mycpu();

    c->proc = 0;
    int cur=0;
    int flag=1;
    for(;;){
```

```

if(flag||(ticks-cur)>=1) {
    flag=0;
    cur=ticks;
    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();

    int found = 0;

    #if defined RR
    // 略
    #elif defined PR
    //略
    #elif defined LS
    //略

    #elif defined SS

    struct proc *p;

    struct proc *min_p = 0;

    // 在遍历过程中，先锁定所有进程
    for (p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock); // 获取当前进程的锁

        if (p->state == RUNNABLE) {
            // 如果当前进程的路径小，更新
            if (min_p == 0 || p->pass< min_p ->pass) {

                if (min_p != 0) {
                    release(&min_p ->lock); // 释放之前选中的进程锁
                }
                min_p = p; // 更新
            } else if (p->pass== min_p ->pass) {

                if (random(1)) {

                    if (min_p != 0) {
                        release(&min_p ->lock);
                    }
                    min_p = p;
                }else{
                    release(&p->lock);
                }
            }else{
                release(&p->lock);
            }
        }
    }
}

```



```

// If found such max_p, copy to p, and run it.
if (min_p != 0) {

    p = min_p ;
    on_state_change(RUNNABLE, RUNNING, p);

    p->state = RUNNING;
    p->pass+=(int)(100/p->tix);
    c->proc = p;
    swtch(&c->context, &p->context);
    c->proc = 0;
    release(&p->lock);
    found = 1;
}

#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");
}
}
}
}

```

## 设计思路：

- `#elif defined ss` 后的是主要代码。
- 对于处于等待状态的进程，遍历所有进程找到路径最小的进程，如果有多个路径最小的进程就随机选择一个。
- 对于选中的进程，用 `p->pass+=(int)(100/p->tix)` 将其路径增加其步长，这里实际上让优先级高（彩票多）的进程“走得慢”，从而获得更多的调用机会。
- 调用上下文切换函数 `swtch` 将 CPU 控制权切换到选中进程。调度完成后，清空 CPU 的进程信息并释放锁。

## 实验测试结果

- 步长调度

```
$ 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: 196 ticks| Running Time: 65 ticks| Sleep Time: 12 ticks|
Response Time: 5 ticks
PID: 8 | Runnable Time: 201 ticks| Running Time: 67 ticks| Sleep Time: 12 ticks|
Response Time: 1 ticks
PID: 5 | Runnable Time: 240 ticks| Running Time: 65 ticks| Sleep Time: 12 ticks|
Response Time: 3 ticks
PID: 6 | Runnable Time: 282 ticks| Running Time: 65 ticks| Sleep Time: 12 ticks|
Response Time: 2 ticks
PID: 7 | Runnable Time: 323 ticks| Running Time: 64 ticks| Sleep Time: 12 ticks|
Response Time: 4 ticks
Average Turnaround Time: 325 ticks
min_turnaround_time Time: 273 ticks
max_turnaround_time Time: 399 ticks
Average response Time: 3 ticks
min_response_time Time: 1 ticks
max response time Time: 5 ticks
```

## 三、测试与分析

### 1. 测试代码实现

关键代码：

```
/*
 *      user/priostat.c
 */

#include "kernel/param.h"
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

uint64 MAGIC_NUM = 114514;
uint64 pids[1024];    // 保存每个子进程的pid
uint64 results[1024]; // 保存每个子进程的计算结果

// Process Statistics Test
// Description:
// A sample calculation Task is involved here
// The test will check if the process statistics are correctly updated
```

```

// according to different types of Scheduler [RR / PR]
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;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: stat <n>\n");
        exit(1);
    }

    int n = atoi(argv[1]);
    if (n <= 0) {
        printf("Error: Invalid number of processes\n");
        exit(1);
    }

    // 用于统计所有进程的总运行时间、等待时间、休眠时间
    int total_run_time = 0;
    int total_wait_time = 0;
    int total_sleep_time = 0;

    // 用于统计所有进程的minroundtime
    int min_turnaround_time= 1000000;

    // 用于统计所有进程的maxroundtime
    int max_turnaround_time= 0;

    int total_response_time=0;
    // 用于统计所有进程的minroundtime
    int min_response_time= 1000000;

    // 用于统计所有进程的maxroundtime
    int max_response_time= 0;

    for (int i = 0; i < n; i++) {
        int pid = fork();
        if (pid < 0) {
            printf("Error: Fork failed\n");

```

```

        exit(1);
    }

    if (pid == 0) {
        // 子进程执行大规模计算
        uint64 ret = big_calculation();
        pids[i] = getpid();
        results[i] = ret;
        // just for debugging
        sleep(1);
        exit(0); // 子进程退出
    } else {
        // 根据i % 4依次设置优先级
        int priority = i % 4; // 0到3的优先级
        printf("Set priority %d to PID %d\n", priority, pid);
        set_priority(priority, pid);
    }
}

// 父进程等待所有子进程并获取调度统计
for (int i = 0; i < n; i++) {
    int runnable_time, running_time, sleep_time, created_time, first_run_time;

    // wait_sched(int *runnable_time, int *running_time, int *sleep_time);
    int pid = wait_sched(&runnable_time, &running_time,
&sleep_time, &created_time, &first_run_time); // 调用wait_sched获取统计信息

    if (pid >= 0) {
        int response_time = first_run_time - created_time;

        printf("PID: %d | Runnable Time: %d ticks | Running Time: %d ticks |
Sleep Time: %d ticks | Response Time: %d ticks\n",
            pid, runnable_time, running_time, sleep_time, response_time);

        int per_sum = runnable_time + running_time + sleep_time;
        min_turnaround_time = (min_turnaround_time > per_sum) ?
per_sum : min_turnaround_time;
        max_turnaround_time = (max_turnaround_time < per_sum) ?
per_sum : max_turnaround_time;
        total_run_time += running_time;
        total_wait_time += runnable_time;
        total_sleep_time += sleep_time;

        min_response_time = (min_response_time > response_time) ?
response_time : min_response_time;
        max_response_time = (max_response_time < response_time) ?
response_time : max_response_time;
        total_response_time += response_time;
    } else {
        printf("Error: wait_sched failed\n");
        exit(1);
    }
}

```

```

    }
}

// 计算并输出平均轮转时间 (Turnaround Time)
int avg_turnaround_time = (total_run_time + total_wait_time +
total_sleep_time) / n;
printf("Average Turnaround Time: %d ticks\n", avg_turnaround_time);
printf("min_turnaround_time Time: %d ticks\n", min_turnaround_time);
printf("max_turnaround_time Time: %d ticks\n", max_turnaround_time);

// 计算并输出平均响应时间
int avg_response_time = total_response_time / n;
printf("Average response Time: %d ticks\n", avg_response_time);
printf("min_response_time Time: %d ticks\n", min_response_time);
printf("max_response_time Time: %d ticks\n", max_response_time);

exit(0);
}

```

## 设计思路:

这个测试代码构建了5个子进程，优先级分别设置为0,0,1,2,3。程序运行结束后，最后统计每个进程的轮转、响应时间。并且进一步得到当前调度算法的平均、最小、最大轮转时间和平均、最小、最大响应时间。方便之后分析

## 测试方式:

- 首先修改 `kernel/proc.h` 中最下面几行的 `#define` ~ 选择调度方式，如下选择了轮转调度:

```

// RR轮转，PR为优先级调度（静态和动态），LS为彩票调度，SS为步长调度
#define RR
//#define PR
//#define LS
//#define SS

```

- 如果选中了优先级调度，可以修改 `kernel/proc.c` 中的 `adjust_priority(struct proc *p)` 函数内的超参数来选择调度具体方式（默认为动态优先级调度）：如果设置为运行完一定时间片下降一级，而等待一段时间直接上升为最高优先级，那么就等价于 `MLFQ` 调度。如果设置运行时间和等待时间都大于一个极大的数才调整优先级，为 `静态优先级调度`。
- 最后 `make qemu` 后在 `qemu` 命令行输入 `priostat 5` 进行测试

## 2. 测试结果分析

调度算法	avg turnaround	min turnaround	max turnaround	avg response	min response	max response	图片链接
轮转调度	344	342	345	0	0	0	<a href="#">查看图片</a>
静态优先级调度	255	161	398	145	1	321	<a href="#">查看图片</a>
动态优先级调度	280	136	385	24	1	60	<a href="#">查看图片</a>
彩票调度	301	210	390	6	1	16	<a href="#">查看图片</a>
步长调度	325	273	399	3	1	5	<a href="#">查看图片</a>

注：时间单位为 `ticks`。

再次展示原理对比表格：

调度算法	效率（平均轮转时间）	公平性（响应时间极差，轮转时间极差）	饥饿问题	实现复杂度	适应性
轮转调度	低	高	无	简单	较低
静态优先级调度	高	中	存在	简单	较低
动态优先级调度	高	高	无	较复杂	高
彩票调度	中（概率）	高（概率）	无	中等	高

调度算法	效率 (平均轮转时间)	公平性 (响应时间极差, 轮转时间极差)	饥饿问题	实现复杂度	适应性
步长调度	中	高	无	较复杂	高

分析结果和原理的匹配：

### 1. 轮转调度 (Round Robin)

- 原理：
  - 效率低：平均轮转时间较长。
  - 公平性高：每个任务获得的CPU时间片相等。
  - 饥饿问题无：任务轮流被调度。
  - 实现复杂度简单：只需循环队列。
- 测试结果：
  - avg turnaround 是 344，高于其他调度算法，表明效率确实较低。
  - min turnaround 和 max turnaround 差异小 (342~345)，说明公平性较高。
  - avg response 为 0，符合轮转调度中任务能迅速开始执行的特点。
  - 与原理匹配良好。

### 2. 静态优先级调度 (Static Priority Scheduling)

- 原理：
  - 效率高：高优先级任务更早完成。没有冗余的过多切换开销。
  - 公平性中等：优先级低的任务可能等待时间较长。
  - 饥饿问题存在：低优先级任务可能永远得不到执行。
  - 实现复杂度简单：按优先级排序。
- 测试结果：
  - avg turnaround 为 255，明显优于轮转调度，符合效率高的预期。
  - min turnaround 低至 161，说明高优先级任务被迅速完成。
  - max turnaround 达到 398，反映低优先级任务延迟较久，公平性较差。
  - avg response 为 145，显示高优先级任务较快开始，但低优先级任务响应时间明显拖长 (max response 321)。
  - 总体符合原理。

### 3. 动态优先级调度 (Dynamic Priority Scheduling)

- 原理：
  - 效率高：动态调整优先级确保高效率。
  - 公平性高：避免饥饿问题。
  - 饥饿问题无：低优先级任务的优先级会随时间增加。
  - 实现复杂度较复杂：需动态计算优先级。

- 测试结果：
  - `avg turnaround` 为 280，比静态优先级略高，符合高效率预期。
  - `min turnaround` 136，显示高优先级任务迅速完成。
  - `max turnaround` 385，比静态优先级低，显示公平性更高。
  - `avg response` 为 24，`response` 极差小，表明响应时间普遍较短，符合动态调整优先级的优势，相比静态优先级调度明显优化。
  - 符合原理。

## 4. 彩票调度 (Lottery Scheduling)

- 原理：
  - 效率中：基于概率分配时间片。
  - 公平性高：理论上每个任务都有机会获得时间片。
  - 饥饿问题无：通过随机分配避免。
  - 实现复杂度中等：需管理彩票分配。
- 测试结果：
  - `avg turnaround` 为 301，中规中矩，符合效率中等的特性。
  - `min turnaround` 210，`max turnaround` 390，显示任务执行时间分布较广，符合随机性特点。
  - `avg response` 为 6，表明任务响应速度都较快，`response` 极差小，符合公平性的特点。
  - 符合原理。

## 5. 步长调度 (Stride Scheduling)

- 原理：
  - 效率中等：根据步长大小分配时间片。
  - 公平性高：任务按步长精确控制调度。
  - 饥饿问题无：步长确保所有任务都能被调度。
  - 实现复杂度较复杂：需计算和管理步长。
- 测试结果：
  - `avg turnaround` 为 325，略高于彩票调度，效率中等。
  - `min turnaround` 273，`max turnaround` 399，范围较窄，显示公平性较高。
  - `avg response` 为 3，`response` 极差小，反映任务响应时间极短并且差距小，响应快速且非常的公平。
  - 符合原理。

---

## 综合结论：

1. 与原理匹配性：
  - 测试结果整体与各调度算法的理论特性一致。
2. 适应性与特点：
  - 静态优先级调度适合高优先级任务为主的场景，但可能造成饥饿。



- 动态优先级和步长调度实现较复杂，但综合性能优异，适合对公平性和效率要求较高的场景。