

基于xv6-public的进程调度算法优化与混合负载性能分析

——实现FCFS、PBS、MLFQ、LOTTERY和CFS

基于xv6-public的进程调度算法优化与混合负载性能分析

——实现FCFS、PBS、MLFQ、LOTTERY和CFS

一、实验背景

二、实验环境与工具

2.1 实验环境

2.2 主要工具

三、实验设计

3.1 实验目标

3.2 实验步骤

四、调度算法实现

4.1 默认的 RR 调度

4.2 FCFS（先来先服务）

4.3 PBS（基于优先级的调度）

4.4 MLFQ（多级反馈队列）

4.5 Lottery Scheduling

4.6 CFS（完全公平调度）

五、对比测试代码实现

5.1 测试设计思想与目标

5.2 任务划分与设计思路

5.2.1 任务类型

5.2.2 优先级与资源分配

5.3 调度算法适配与灵活性

5.3.1 条件编译选择调度算法

5.3.2 灵活的任务分配

5.4 性能统计与数据收集

5.4.1 进程级数据统计

5.4.2 系统级指标统计

5.5 测试代码小结

六、实验结果分析

RR:

FCFS:

PBS:

MLFQ:

LOT:

CFS:

七、实验创新性、难度和可教学性

7.1 实验创新性

7.1.1 调度算法的多样性扩展

7.1.2 数据驱动的调度优化

7.1.3 动态性能统计

7.1.4 综合性实验平台

7.2 实验难度

7.2.1 调度算法实现的复杂性

7.2.2 性能统计与数据处理

7.2.3 系统内核修改的挑战

7.3 实验可教学性

7.3.1 面向初学者的引导性

7.3.2 可扩展的实验平台

7.3.3 培养分析与优化能力

7.3.4 实验的开放性

八、实验结语

一、实验背景

进程调度是操作系统中的核心内容之一，直接影响系统的资源分配效率与整体性能。在操作系统课程的实验教学中，xv6-public 作为经典的教学操作系统，为学习和理解调度算法的基本原理提供了良好的实验平台。然而，现有实验内容通常仅实现了时间片轮转调度（Round Robin, RR），这一算法虽然设计简单且便于初学者上手，但缺少对其他经典调度算法（如优先级调度、多级反馈队列调度等）的实践探索，也缺乏对不同算法在实际场景中的性能差异进行深入对比分析的环节。

此外，xv6-public 默认的 RR 调度算法在复杂负载场景中存在一定的局限性。例如，该算法仅基于时间片分配，不考虑进程的优先级、等待时间等因素，在混合负载（如计算密集型与 I/O 密集型任务并存）场景下可能导致系统响应效率下降或资源分配不均。这种设计虽然降低了教学实验的复杂度，但在更复杂的实验需求中，研究多种调度算法的优化和对比能够更全面地展现调度算法对系统性能的影响，并探索其在不同场景中的适用性。

基于此，本实验以 xv6-public 为基础，优化实现多种经典调度算法（如优先级调度、多级反馈队列调度、完全公平调度算法等），并设计实验场景对不同算法的性能表现进行对比分析，特别是在混合负载环境下的适应性与效率。通过本实验，我们希望深入探讨调度算法的设计思路与实现机制，分析其对资源分配公平性、系统响应时间等核心指标的影响，为更高效的调度策略设计提供参考。

二、实验环境与工具

2.1 实验环境

操作系统：基于 `xv6-public` 的轻量级 Unix 类操作系统。

编译工具链： `i386-elf-gcc`，用于编译 x86 架构代码。

仿真环境： `qemu`，模拟硬件运行 xv6。

2.2 主要工具

代码编辑器：用于修改 `xv6` 源码文件。

`qemu` 输出：用于实时观察调度算法的运行结果。

系统调用：通过自定义的 `ps()` 和 `set_priority()` 等接口监控和调试调度行为。

三、实验设计

3.1 实验目标

1. 在原生的 RR 调度算法基础上，设计并实现五种新的调度算法：
 - **FCFS** (First Come First Serve)：先来先服务调度算法，按照进程到达的顺序依次调度。
 - **PBS** (Priority Based Scheduling)：基于优先级的调度算法，根据进程的优先级高低决定调度顺序。
 - **MLFQ** (Multi-Level Feedback Queue)：多级反馈队列调度算法，通过动态调整进程在多级队列中的位置实现公平性与响应速度的平衡。
 - **LOT** (Lottery Scheduling)：基于随机机制的调度算法，通过给进程分配“彩票”实现概率性调度。
 - **CFS** (Completely Fair Scheduler)：仿照linux内核的完全公平调度算法，按进程实际运行时间公平分配资源。
2. 分析与对比调度算法的性能
 - 测试各算法在不同类型的进程集合中的表现，比较以下核心指标：
 - **平均响应时间 (Response Time)**：从进程提交到首次被调度运行的时间。
 - **平均周转时间 (Turnaround Time)**：从进程提交到完成的总时间。
 - **平均等待时间 (Waiting Time)**：进程在调度运行前的总等待时间。
 - 对比各算法在 **I/O 密集型** 和 **CPU 密集型** 进程中的调度表现，分析它们在不同负载场景下的适应性和效果。

3.2 实验步骤

1. 设计并实现调度算法：
 - 修改 xv6 的进程调度代码，新增调度算法选择的机制。
 - 在每个进程结构体 `struct proc` 中，增加必要的字段，如优先级、虚拟运行时间、队列等级等。
2. 新增系统调用：
 - `ps()`：实时查看进程的状态、优先级和运行统计数据。
 - `set_priority()`：用于 PBS 调度中动态调整优先级。
 - `print_stats()`：打印调度完成后各进程的运行统计信息，包括周转时间、响应时间等。
3. 验证与优化：
 - 运行定制化的测试脚本，观察调度行为。
 - 通过多次测试和调参优化算法，实现更好的调度效果。

四、调度算法实现

4.1 默认的 RR 调度

- 代码实现：
 - 在 `scheduler()` 中，遍历所有 `RUNNABLE` 状态的进程，按顺序分配时间片。
 - 每个进程运行一个时间片后，如果未完成，则切换到下一个进程。
- 代码逻辑（伪代码）：

```
void scheduleRR() {
    for (p in process_table) {
        if (p.state == RUNNABLE) {
            assign_time_slice(p, 1); // 固定时间片为 1
            run_process(p);
        }
    }
}
```

- 优缺点：
 - 优点：简单、公平。
 - 缺点：不支持优先级，I/O 密集型任务可能浪费时间片。

4.2 FCFS（先来先服务）

FCFS（First Come First Serve）调度算法的实现基于进程的到达时间，对到达时间最早的进程优先调度。

1. 增加必要的字段

要实现 FCFS 调度，需要在进程结构 `struct proc` 中新增一个字段，用于记录进程的到达时间。这在 `proc.h` 文件中完成：

```
uint arrival_time; // 记录进程的到达时间
```

到达时间是进程被创建时的全局时间戳，具体在 `allocproc()` 函数中初始化：

```
p->arrival_time = ticks; // ticks 是系统当前的全局时钟，用来标记到达时间
```

这样，每个进程的 `arrival_time` 字段都可以记录下进程进入调度队列的时间。

2. 修改调度逻辑

为了实现 FCFS，我们需要在调度器中选择到达时间最早的可运行进程。在 `scheduler()` 函数中，通过条件编译的方式选择 FCFS 调度器逻辑。FCFS 调度的核心逻辑是：

- 遍历进程表 `ptable`，找到 `RUNNABLE` 状态的进程中到达时间最早的进程。
- 将该进程分配给 CPU 并运行。

```

// First Come First Serve
void
scheduleFCFS(struct cpu *c)
{
    struct proc *p;
    struct proc *serve = 0;

    // Loop over process table looking for process to run.
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE)
            continue;

        if (!serve || serve->ctime > p->ctime)
            serve = p;
        // 打印调度器选中进程的信息
        //cprintf("Scheduler chose PID: %d\n", p->pid);
    }

    shift(serve, c);
}

```

这样就实现了对到达时间的比较，并确保到达时间最早的进程优先运行。

3. 调度算法的整合

在 `scheduler()` 函数中，通过预处理宏 `#ifdef` 实现条件编译，使得系统可以选择不同的调度算法。将 FCFS 调度逻辑整合到 `scheduler()` 中：

```

void scheduler(void) {
    struct cpu *c = mycpu();
    c->proc = 0;

    for (;;) {
        sti(); // 启用中断
        acquire(&ptable.lock);

        #ifdef FCFS
            scheduleFCFS(c); // 调用 FCFS 调度逻辑
        #endif

        release(&ptable.lock);
    }
}

```

这样，通过在 `Makefile` 中定义 `-DFCFS` 宏，可以启用 FCFS 调度。

4. 特殊情况处理

1. 边界条件

如果所有进程都不是 `RUNNABLE`，则调度器会进入空转状态。这种情况下，FCFS 调度逻辑会跳过分配操作，等待有进程进入 `RUNNABLE` 状态。

2. 进程的阻塞和重新进入队列

FCFS 是非抢占式调度算法，当一个进程因阻塞（例如等待 I/O）无法继续运行时，调度器会在下一轮选择到达时间最早的进程。

3. 退出

进程完成后，其相关资源被释放，同时从调度队列中移除。这在 `exit()` 函数中处理。

5. 性能和特性分析

FCFS 调度算法的实现较为简单，核心是按到达时间顺序调度，避免了复杂的优先级和时间片管理。然而，这种调度方式存在如下特性：

- 优点：

1. 实现简单，逻辑清晰。
2. 避免了时间片轮转引入的上下文切换开销。
3. 适合批处理任务，如科学计算、数据处理等。

- 缺点：

1. 缺乏抢占机制，I/O 密集型任务可能会导致 CPU 资源浪费。
2. 容易出现“长任务堵塞短任务”的问题（即著名的“饥饿现象”）。

6. 输出调试信息

为验证 FCFS 调度的正确性，可以在调度器中输出调试信息，例如输出选中进程的 PID 和到达时间：

```
cprintf("FCFS Scheduler chose PID: %d | arrival_time: %d\n", serve->pid, serve->arrival_time);
```

观察输出可以确认调度器是否按照到达时间的顺序选择了进程。

通过以上步骤，FCFS 调度算法在 xv6 操作系统中的实现完成。它适合对公平性要求较低的任务场景，但对实时性和抢占能力的支持较弱。

```
xv6...
Scheduling: First Come First Serve
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
```

4.3 PBS（基于优先级的调度）

优先级调度算法的核心思想是选择优先级最高也就是priority数值最低的进程运行。在优先级相同时，采用时间片轮转策略以保证公平性。相比于 FCFS 调度，PBS 通过动态调整优先级的方式支持不同任务的优先化处理，在实时性任务和资源竞争任务中具有较高的实用性。

1. 数据结构改动

PBS 的实现需要为每个进程定义一个优先级字段，表示该进程当前的优先级。优先级通过 `set_priority()` 系统调用动态调整，默认优先级设为中间值。

在 `proc.h` 文件的 `struct proc` 中新增：

```
int priority; // 表示进程的优先级，值越小优先级越高
```

在 `allocproc()` 中初始化：

```
p->priority = 60; // 默认优先级设置为中等优先级
```

与 FCFS 的 `arrival_time` 字段类似，`priority` 字段用来支持优先级排序。

2. 新增系统调用 `set_priority()`

这个系统调用接收两个参数：

1. `new_priority`：设置的优先级值。
2. `pid`：需要调整优先级的进程 ID。

系统调用的核心代码如下：

```
//新增改变指定进程优先级的函数，用于PBS
int
set_priority(int new_priority, int pid)
{
    /*#ifndef PBS
        // cprintf("Scheduling is not Priority Based!\n");
        return -1;
    #endif*/

    if (new_priority < 0 || new_priority > 100)
        return -1;

    acquire(&ptable.lock);
    struct proc *p;
    int old_priority = -1;

    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->pid == pid) {
            old_priority = p->priority;
            p->priority = new_priority;
            if (p->qtime) {
                remove(&queues[old_priority], p->pid);
                push(&queues[new_priority], p);
            }
            break;
        }
    }
}
```

```

    // cprintf("Changed %d (%s) priority from %d to %d\n", p->pid, p->name, old_priority,
new_priority);

    release(&ptable.lock);

    if (myproc() && new_priority < myproc()->queue)
        yield();

    return old_priority;
}

```

在更新优先级后，进程会重新被调度器选择。

如果当前进程优先级下降，主动调用 `yield()` 让出 CPU，体现动态调整的效果。

3. 调度逻辑

PBS 的调度逻辑与 FCFS 类似，需要遍历所有 `RUNNABLE` 状态的进程，但选择依据是优先级而非到达时间。

在 `scheduler()` 函数中新增 `schedulePBS()` 方法

```

// Priority Based Scheduler with Level 2 Round Robin
void
schedulePBS(struct cpu *c)
{
    struct proc *p;

    // Check for runnable processes not in any queue
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE)
            continue;

        if ((p->qtime) == 0) {
            p->qtime = ticks + 1;
            push(&queues[p->queue], p);
        }
    }

    int q = 0;
    while (q < 200 && !queues[q]) q++;

    if (q == 200) return;

    // 从最高优先级队列中取出进程
    struct proc *serve;
    serve = pop(&queues[q]);
    if (q != serve->queue) cprintf("ERROR !!!\n"); // 没有可用进程

    // 打印调度器选中进程的信息
    // cprintf("Scheduler chose PID: %d, Priority: %d\n", serve->pid, serve->queue);

    // 设置时间片（动态分配时间片，优先级高的进程时间片更长）
    serve->allowedtime = (200 - serve->queue) / 10;
}

```



```

// 调度选中的进程
shift(serve, c);

//print_queues();

// 重置队列时间，避免重复加入队列
serve->qtime = 0;
}

```

使用 `ptable` 遍历所有进程，选择优先级最高的 `RUNNABLE` 进程。

动态分配时间片，优先级高（数值低）的进程获得更多的 CPU 时间。

4. 整合与调试

将 PBS 调度逻辑集成到 `scheduler()` 中，条件编译启用 `schedulePBS`：

```

#ifdef PBS
    schedulePBS(c);
#endif

```

为了便于调试，同样可以在调度器中打印调度的进程信息：

```

cprintf("PBS Scheduler chose PID: %d | Priority: %d\n", selected_proc->pid,
selected_proc->priority);

```

调试过程中，我们也可以新增一个 `print_queues` 函数，通过打印每个队列的状态，观察进程在各队列间的迁移情况：

```

//PBS调度算法观测各优先级队列时使用
void print_queues() {
    cprintf("Priority Queue Status:\n");
    for (int i = 0; i < 101; i++) {
        if (queues[i]) {
            cprintf("Queue %d: ", i);
            struct node *cur = queues[i];
            while (cur) {
                cprintf("PID %d -> ", cur->p->pid);
                cur = cur->next;
            }
            cprintf("NULL\n");
        }
    }
}

```

```

xv6...
Scheduling: Priority Based Scheduling
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh

```

5. 优化与特性

优化 1：时间片分配

通过调整 `allowedtime` 参数，PBS 调度可以实现基于优先级的动态时间片分配。优先级越高的进程时间片越长，优先级越低的进程时间片越短，从而在公平性和优先级之间取得平衡。

优化 2：低优先级进程防止饥饿

在多任务场景中，低优先级进程可能长时间无法获得 CPU。可以通过给低优先级进程增加优先级（优先级 aging）的方式缓解饥饿问题。

6. 性能分析

PBS 调度的适用场景主要在实时性要求较高或任务重要性差异较大的环境中，其优势是能够满足高优先级任务的快速响应需求；可以通过动态调整优先级，适应任务的重要性变化。劣势是低优先级进程可能面临饥饿现象；需要增加额外的优先级管理逻辑。

4.4 MLFQ（多级反馈队列）

多级反馈队列（MLFQ）是一种基于多优先级队列的调度算法，其核心思想是将进程按优先级划分到多个队列中，不同队列具有不同的时间片分配。高优先级队列的时间片较短，进程如果无法在其队列时间片内完成，会被降级到低优先级队列；而低优先级队列进程如果较长时间未运行，可以通过“老化”（Aging）机制逐步提升优先级。MLFQ 调度器结合了短作业优先（SJF）和时间片轮转（RR）的优点，在公平性和效率之间取得了较好的平衡。

1. 数据结构改动

MLFQ 需要多个优先级队列来组织进程，并在每个队列中进行调度。

在 `proc.h` 中添加：

```
int queue;           // 当前队列编号（优先级队列编号，0-4，数值越小优先级越高）
int qtime;           // 当前队列时间片开始时间
int q[5];            // 每个队列的运行次数统计，用于调试和分析
```

初始化字段：

在 `allocproc()` 中初始化这些字段：

```
p->queue = 0;        // 默认放置在最高优先级队列
p->qtime = 0;         // 初始化为 0
for (int i = 0; i < 5; i++) p->q[i] = 0; // 队列统计归零
```

多级队列的管理结构：

定义一个队列数组，每个队列存储对应优先级的进程链表：

```
struct node *queues[5]; // 五个优先级队列
```

2. 调度逻辑

调度过程可以分成三步走

1. 新进程的分配：

新创建的进程默认分配到最高优先级队列（`queue = 0`）。

通过 `assignQueue()` 函数实现进程的初始分配。

2. 调度策略：

优先从最高优先级队列调度，只有在高优先级队列为空时，才会调度低优先级队列中的进程。

每个队列采用时间片轮转策略，时间片长度随着优先级降低而增加。

3. 队列降级与老化机制：

如果一个进程未能在分配的时间片内完成，则将其降级到下一优先级队列。

如果低优先级队列中的进程等待时间过长，则提升其优先级以防止饥饿。

```
// Multi Level Feedback Queue
void
scheduleMLFQ(struct cpu *c) {
    struct proc *p;
    struct node* cur;

    // Check for runnable processes not in any queue
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE)
            continue;

        if ((p->qtime) == 0)
            assignQueue(p);
    }

    // Age the processes
    for (int i = 1; i < 5; i++) {
        cur = queues[i];
        while (cur) {
            p = cur->p;
            if (1 + ticks - p->qtime > 100) {
                remove(&queues[i], p->pid);
                p->queue = i - 1;
                p->qtime = ticks + 1;
                p->wtime = 0;
                push(&queues[i-1], p);
#ifdef GRAPH
                if (p->pid > 3)
                    cprintf("GRAPH %d %d %d\n", p->pid - 4, ticks, p->queue);
#endif
            }
            cur = cur->next;
        }
    }
}
```

```

    }
}

int q = 0;
while (q < 5 && !queues[q]) q++;

if (q == 5) return;

p = pop(&queues[q]);

p->allowedtime = (1 << q);

shift(p, c);

p->qtime = 0;

if (p->state == RUNNABLE) {
    if (p->allowedtime == 0 && q < 4) q++;

    p->qtime = ticks + 1;
    p->queue = q;
    push(&queues[q], p);
}

#ifdef GRAPH
    if (p->pid > 3)
        cprintf("GRAPH %d %d %d\n", p->pid - 4, ticks, p->queue);
#endif
}

```

4. 队列管理函数

MLFQ 使用链表存储每个队列中的进程，因此我们要新增三个和队列相关的关键函数

push()：将进程加入指定队列的末尾。

pop()：从队列头部取出一个进程。

remove()：从队列中移除指定进程（通过 PID 定位）。

这些函数与 PBS 的队列管理逻辑类似，仅在 `queues[]` 的维度上进行扩展。

5. 整合

在 `scheduler()` 函数中集成 `scheduleMLFQ()`，启用 MLFQ 调度：

```

#ifdef MLFQ
    scheduleMLFQ(c);
#endif

```

6. MLFQ 的特性与优势

MLFQ 是一种动态、灵活的调度算法，它的优势体现在：

高优先级队列的时间片短，低优先级队列的时间片长，能够快速响应短任务的需求。

通过老化机制动态提升低优先级进程的优先级，避免饥饿现象。

短任务优先的同时，长任务也能够获得足够的 CPU 资源。

与前面的PBS 和 FCFS 相比，MLFQ 更适合处理混合型负载。

PBS 依赖手动设置优先级，而 MLFQ 根据任务运行情况动态调整优先级。MLFQ 自动适应任务负载，而 PBS 需要额外的优先级管理系统调用。

FCFS 缺乏抢占机制，长任务可能导致资源被独占。MLFQ 支持抢占和优先级调整，更适合实时和多任务环境。

```
xv6...
Scheduling: Multi Level Feedback Queue
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
```

4.5 Lottery Scheduling

1. 总体目标

彩票调度算法（Lottery Scheduling）通过分配给每个进程一定数量的彩票，以随机的方式决定下一个被调度的进程。这种方式确保了调度的公平性，同时也可以通过调整彩票数量动态分配CPU资源。本次实现主要包括以下几个步骤：

1. 定义和扩展必要的数据结构。
2. 修改进程相关的数据成员以支持彩票调度。
3. 实现核心调度逻辑和相关辅助功能。

2. 数据结构和核心修改

2.1 添加彩票数量和时间片统计字段

在 `proc.h` 文件的 `struct proc` 中新增了两个字段：

- `ticket`：表示该进程拥有的彩票数量，用于参与调度。
- `tick`：记录该进程累计使用的时间片数，用于性能统计。

```
int ticket;    // 彩票数量
int tick;      // 使用的时间片数
```

在 `allocproc` 函数中，对新进程的 `ticket` 和 `tick` 字段进行初始化：

```
p->ticket = InitialTickets; // 默认每个新进程初始分配5张彩票
p->tick = 0;                 // 累计时间片数初始化为0
```

2.2 在子进程中继承彩票

在 `fork` 函数中，将父进程的彩票数量复制给子进程：

```
np->ticket = curproc->ticket;
```

3. 辅助功能的实现

3.1 获取所有可运行进程的彩票总数

`getRunnableProcTickets` 函数遍历进程表，计算所有状态为 `RUNNABLE` 的进程的彩票总数：

```
int getRunnableProcTickets(void){
    struct proc *p;
    int total = 0;
    for(p = ptable.proc;p < &ptable.proc[NPROC];p++){
        // what I do
        if(p->state == RUNNABLE)
        {
            total += p->ticket;
        }
    }
    return total;
}
```

该函数的结果将用于调度器中生成随机数范围。

3.2 随机数生成

通过伪随机数生成函数 `random_at_most` 生成一个范围为 `[0, total_tickets_num)` 的随机数：

```
long win_num = random_at_most(total_tickets_num);
```

`win_num` 表示本次调度中“中奖”的彩票编号。

3.3 动态设置彩票数

新增了系统调用 `settickets`，允许用户通过系统调用为当前进程设置彩票数量：

```
int
settickets(int number)
{
    struct proc *pr=myproc();
    int pid = pr->pid;
    acquire(&ptable.lock); //Find and assign the tickets to the process
    struct proc *p;
    for(p = ptable.proc;p < &ptable.proc[NPROC];p++){
        if(p->pid == pid){
            p->ticket = number; //assigning allotted ticket for a process
            release(&ptable.lock);
            return 0 ;
        }
    }
    release(&ptable.lock);
    return 0;
}
```

```
}
```

4. 调度器核心逻辑

彩票调度器的实现逻辑集中在 `scheduleLOT` 函数中：

1. 获取所有可运行进程的彩票总数。
2. 根据彩票总数生成一个随机编号 `win_num`。
3. 遍历所有可运行进程，找到累积彩票数量大于等于 `win_num` 的进程。
4. 将该进程状态置为 `RUNNING` 并进行调度。

```
void
scheduleLOT(struct cpu *c)
{
    struct proc *p;
    long cnt = 0;

    long total_tickets_num = getRunnableProcTickets();
    long win_num = random_at_most(total_tickets_num);

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state == RUNNABLE)
            cnt += p->ticket;
        else
            continue;

        // Switch to chosen process.  It is the process's job
        // to release ptable.lock and then reacquire it
        // before jumping back to us.
        if(cnt > win_num){
            // cprintf("win num is %d, win process is %d\n", win_num, p->pid);
            shift(p, c);
            break;
        }
        else
            continue;
    }
}
```

在每次调度中，通过彩票总数和随机编号实现公平的随机分配。`shift` 函数完成上下文切换以及记录时间片使用。

5. 数据统计和接口支持

获取进程状态信息：

通过新增的系统调用 `getpinfo`，可以获取所有进程的状态，包括：

- 是否正在使用。
- 累计时间片数。

- 分配的彩票数量。

在 `proc.c` 中实现如下：

```
int getpinfo(struct pstat *ps){
    acquire(&ptable.lock);
    struct proc *p;
    int i = 0;
    for(p=ptable.proc;p < &ptable.proc[NPROC];p++){
        ps->pid[i] = p->pid;
        ps->inuse[i] = p->state !=UNUSED;
        ps->ticket[i] = p->ticket;
        ps->tick[i] = p->tick;
        ps->ctime[i] = p->ctime;
        ps->priority[i] = p->queue;
        ps->etime[i] = p->etime;
        ps->qtime[i] = p->qtime;
        ps->rtime[i] = p->rtime;
        ps->vruntime[i] = p->vruntime;
        ps->wtime[i] = p->wtime;
        ps->n_run[i] = p->n_run;
        i++;
    }
    release(&ptable.lock);
    return 0;
}
```

随机数生成支持：

为了实现彩票的随机性，引入了 `rand.c` 和 `rand.h` 文件，提供伪随机数生成函数 `random_at_most`。

6. 调试与验证

通过测试程序验证彩票调度的公平性：

1. 使用 `settickets` 为不同进程设置不同的彩票数量。
2. 每个进程累积时间片的比例应与其彩票数量成正比。
3. 使用 `getpinfo` 获取并打印进程状态，验证调度结果的正确性。

测试结果表明，不同彩票数量的进程运行时间片分配大致符合比例关系，验证了调度器的正确性和公平性。

7. 特点与难点

彩票调度算法的实现兼具公平性和灵活性，支持动态调整进程优先级。实现过程中的主要难点在于：

- 彩票总数的动态计算与随机编号的生成。
- 调度器中如何高效地遍历进程表并进行调度选择。
- 对用户接口的支持，确保调度结果可观察和可测试。


```
xv6...
Scheduling: Lottery Schedule
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
```

4.6 CFS（完全公平调度）

CFS是linux的CPU调度算法，旨在以尽可能公平的方式分配CPU时间给不同的进程。在CFS中，调度的核心思想是让每个进程在调度器中得到与其权重成正比的时间片，并且基于每个进程的运行时间（vruntime）来决定调度顺序。vruntime是CFS中每个进程的虚拟运行时间，CFS会选择vruntime最小的进程来调度。

在本项目的实现中，考虑到前面实现PBS和MLFQ时都采用了队列，这里刚好可以利用前面所创建的节点结构体来实现一个最小堆（Min-Heap）去选择vruntime最小的进程。最小堆保证每次从堆中取出的进程都是vruntime最小的进程，从而实现公平调度。虽然在更多的实现中，CFS会采用红黑树来实现，但在进程数相对较小的情况下，最小堆应当在性能上不会有太大差距。

基于最小堆实现CFS调度的核心步骤和细节包括以下几个部分：

第一，最小堆的实现。在CFS中，最小堆用来存储所有可运行的进程，堆的根节点总是拥有vruntime最小的进程。最小堆的特性保证了堆顶的进程总是需要优先调度的进程。

插入操作通过min_push函数完成。插入时，首先检查堆是否已满。如果堆未满，则将新进程节点插入堆中。然后，通过bubble_up函数将新插入的进程向上调整，确保堆仍然符合最小堆的性质（即父节点vruntime不大于子节点）。

```
// 插入新节点，维护堆结构
void min_push(struct node **head, struct proc *p) {
    // 检查节点池中是否有空位置
    if (heap_size >= NPROC) {
        cprintf("Heap is full\n");
        return;
    }

    // 在堆中检查是否已经存在相同的 pid
    for (int i = 0; i < heap_size; i++) {
        if (nodes[i].p->pid == p->pid) {
            return; // 如果 pid 重复，直接返回，不插入
        }
    }

    // 找到一个空闲的节点并初始化
    struct node *newNode = &nodes[heap_size++];
    newNode->p = p;
    newNode->next = 0;

    // 上浮新插入的节点
    bubble_up(heap_size - 1);
}
```

```
}
```

```
// 上浮操作
```

```
void bubble_up(int index) {
    while (index > 0) {
        int parent = (index - 1) / 2;
        if (compare_vruntime(nodes[index].p, nodes[parent].p)) {
            // 交换节点
            struct node temp = nodes[index];
            nodes[index] = nodes[parent];
            nodes[parent] = temp;
            index = parent;
        } else {
            break;
        }
    }
}
```

弹出操作通过min_pop函数完成。从最小堆中弹出堆顶进程（vruntime最小的进程），将最后一个节点移到堆顶，并调用bubble_down函数将其调整到合适的位置。

```
// 弹出最小节点，维护堆结构
```

```
struct proc*
min_pop(struct node **head) {
    if (heap_size == 0) {
        cprintf("Heap is empty\n");
        return 0;
    }

    // 获取堆顶节点（最小 vruntime 的进程）
    struct proc *min_proc = nodes[0].p;

    // 用堆底节点替换堆顶
    nodes[0] = nodes[--heap_size];
    bubble_down(0); // 进行下沉操作调整堆

    return min_proc;
}
```

```
// 下沉操作
```

```
void bubble_down(int index) {
    int left_child = 2 * index + 1;
    int right_child = 2 * index + 2;
    int smallest = index;

    // 找到三个节点中 vruntime 最小的
    if (left_child < heap_size && compare_vruntime(nodes[left_child].p,
nodes[smallest].p)) {
```

```

        smallest = left_child;
    }
    if (right_child < heap_size && compare_vruntime(nodes[right_child].p,
nodes[smallest].p)) {
        smallest = right_child;
    }

    if (smallest != index) {
        // 交换节点
        struct node temp = nodes[index];
        nodes[index] = nodes[smallest];
        nodes[smallest] = temp;
        bubble_down(smallest);
    }
}

```

第二，调度器实现。CFS调度器的主要工作是从最小堆中选出一个进程，并更新其vruntime，然后将其重新插入堆中。

核心逻辑是从最小堆中弹出vruntime最小的进程，调度该进程，并根据其运行时间更新vruntime，最后将该进程重新插入堆中。

```

void
scheduleCFS(struct cpu *c)
{
    struct proc *p;
    struct proc *serve;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if (p->state != RUNNABLE){
            continue;
        }

        if(p->qtime == 0){
            p->qtime = ticks+1;
        }

        min_push(&head, p);
    }

    if (heap_size > 0) {
        serve = min_pop(&head);

        shift(serve, c);

        serve->vruntime += serve->rtime * (serve->queue / 15) - serve->wtime / 30;
        serve->qtime = 0;
    }
}

```

第三，vruntime的计算。每个进程的vruntime根据实际运行时间和权重进行更新，权重通过进程优先级（priority）的倒数计算得到。

进程的优先级越高（数值越低），其vruntime增长速度越慢，在调度器中被调度的频率就越高。相反，优先级越低的进程，其vruntime增长速度越快，在调度器中被调度的频率越低。

通过基于最小堆的实现，CFS调度器能够较为简单地选择出vruntime最小的进程，从而实现公平调度。相比于MLFQ等多级队列调度算法，CFS更加关注每个进程的运行时间分配，并通过虚拟时间的概念实现了高效和公平的调度策略。

```
xv6...
Scheduling: Complete Fair Schedule
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
# benchmark
```

五、对比测试代码实现

在这里我们设计了一个复杂且灵活的测试框架，通过 `contrastTest` 文件对多种进程调度算法（如 RR、FCFS、PBS、MLFQ、Lottery 和 CFS）的性能进行了全面评估。实验代码的核心思想是通过混合型负载场景下的对比测试，揭示不同调度算法在实际应用中的优势与不足。以下从任务设计、调度适配、统计实现以及创新点等方面详细说明实验代码的设计。

5.1 测试设计思想与目标

调度算法的主要目标是优化系统性能，包括响应时间、周转时间、等待时间等多项指标，但不同的负载场景对这些指标的侧重点有所不同。所以我们的想法是将任务划分为 I/O 密集型、CPU 密集型以及混合型任务，覆盖真实操作系统中的典型工作负载，通过动态调整任务的优先级和资源分配，测试调度算法的灵活性和鲁棒性，并结合多个指标（响应时间、周转时间、等待时间、CPU 利用率等），对调度算法的性能进行综合评价。测试代码的目的是构建一个统一的测试平台，在统一负载下实现对多种调度算法的公平对比，并验证这些算法能否满足不同场景对性能指标的需求。

5.2 任务划分与设计思路

这一部分是测试代码的核心思想。为了最大程度模拟现实场景中的复杂负载，代码将任务划分为五种类型，每种类型都模拟了特定的资源需求和调度特性。这些任务设计了不同的 I/O 和 CPU 操作组合，并通过动态调整优先级模拟了调度的多样性。

5.2.1 任务类型

1. Task 1: I/O 密集型任务

设计目的： 测试调度算法对 I/O 密集型任务的调度能力，以及是否会发生 CPU 资源浪费。

实现细节： 通过频繁调用 `sleep()` 模拟 I/O 操作，同时设置较低优先级，反映其对响应时间的容忍性。

```
void
task1(int num, int pid){
    if (num % 4 == 0)
        set_priority(150 + num % 4, pid);
    else
        set_priority(60 - num % 4, pid);
    volatile int i;
    for ( i=0; i<= num; i++)
    {
        sleep(200);
    }
}
```

2. Task 2: CPU 密集型任务

设计目的： 测试调度算法对 CPU 密集型任务的公平性和资源分配策略。

实现细节： 通过长时间的循环计算模拟高强度的 CPU 占用，并设置较高优先级以模拟需要快速完成的任务。

```
void
task2(int num, int pid){
    if (num % 4 != 0)
        set_priority(145 + num % 4, pid);
    else
        set_priority(110 + num % 4, pid);
    volatile int i;
    for (i=0; i<100000000; i++){
        ;
    }
}
```

3. Task 3: CPU 计算后进入 I/O 操作

- **设计目的：** 测试调度算法对混合型任务的适应性，以及在 CPU 与 I/O 切换中的调度性能。
- **实现细节：** 先进行 CPU 密集型计算，再进入 `sleep()` 模拟 I/O 操作，设置中等优先级。
- **代码实现：**

```
void
task3(int num, int pid){
    if (num % 4 == 0)
        set_priority(135 + num % 4, pid);
    else if (num % 4 == 1)
        set_priority(65 - num % 4, pid);
    else
```

```

        set_priority(105 + num % 4, pid);
    volatile int i;
    for (i=0; i<10000000;i++) {
        ;
    }
    sleep(200);
}

```

4. Task 4: I/O 操作后进入 CPU 计算

- 设计目的：进一步测试混合型任务，任务逻辑与 Task 3 相反，先执行 `sleep()` 再进行 CPU 计算。
- 代码实现：

```

void
task4(int num, int pid){
    if (num % 4 == 0)
        set_priority(130 + num % 4, pid);
    else if (num % 4 == 1)
        set_priority(70 - num % 4, pid);
    else
        set_priority(100 + num % 4, pid);
    sleep(200);
    volatile int i;
    for (i=0; i<10000000;i++) {
        ;
    }
}

```

5. Task 5: 复杂的混合型任务

- 设计目的：模拟复杂的 CPU 和 I/O 交替任务，测试调度算法对不规则负载的适应性。
- 代码实现：

```

void
task5(int num, int pid){
    if (num % 4 == 0)
        set_priority(125 + num % 4, pid);
    else if (num % 4 == 1)
        set_priority(75 - num % 4, pid);
    else
        set_priority(95 + num % 4, pid);
    volatile int i;
    volatile int j;
    for (i=0; i < num; i++){
        if(i % 2){
            for(j=0;j<10000000; j++){
                ;
            }
        } else {
            sleep(200);
        }
    }
}

```

```
}  
}
```

5.2.2 优先级与资源分配

- **PBS 调度**：使用 `set_priority()` 为任务动态分配优先级，模拟任务的不同重要性和时效性。
- **Lottery 调度**：使用 `settickets()` 为任务分配不同数量的彩票票数，以测试随机调度算法的公平性。
- **MLFQ 调度**：通过任务设计的多样性测试调度器在多级反馈队列中的优先级迁移效果。

5.3 调度算法适配与灵活性

5.3.1 条件编译选择调度算法

实验通过条件编译（`#ifdef` 宏定义）切换不同的调度算法。例如：

- `-DRR` 启用时间片轮转调度；
- `-DPBS` 启用基于优先级的调度；
- `-DLOT` 启用彩票调度；
- `-DMLFQ` 启用多级反馈队列调度。

5.3.2 灵活的任务分配

代码支持动态修改任务数、优先级和执行逻辑。通过为不同任务分配不同的资源（时间片、优先级或彩票票数），确保对每种调度算法进行全面评估。

5.4 性能统计与数据收集

5.4.1 进程级数据统计

通过 `getProcInfo()` 收集每个进程的性能数据，包括：进程 ID；运行时间（`rtime`）、等待时间（`wtime`）、总调度次数（`n_run`）；在 PBS 调度下的优先级；在 Lottery 调度下的彩票票数；在 CFS 下的优先级和虚拟运行时间。

统计信息通过以下代码实现：

```
int  
getProcInfo(void){  
    int count = 0;//the total number of time slice  
    struct pstat ps;  
    getpinfo(&ps);  
    #ifdef LOT  
        printf(1, "\n-----Initially assigned Tickets = %d ----- \n", InitialTickets);  
    #endif  
  
    printf(1, "\nProcessID\tRunnable(0/1)\tTicks\tctime\tetime\tqtime\twtime\trtime\tN_run");  
    #ifdef LOT  
        printf(1, "\tTickets");  
    #endif
```

```

#ifdef PBS
    printf(1, "\tPriority");
#endif
#ifdef CFS
    printf(1, "\tPriority\tVruntime");
#endif
    printf(1, "\n");
    for(int i=0; i < NPROC; i++)
    {
        if(ps.pid[i] && ps.ticket[i] > 0)
            //whether there exists process which has the number of tickets>0
        {

            printf(1, "%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d", ps.pid[i], ps.inuse[i], ps.tick[i], ps.ctime[i], ps.etime[i], ps.qtime[i], ps.wtime[i], ps.rtime[i], ps.n_run[i]);
            count += ps.tick[i];
#ifdef LOT
                printf(1, "\t%d", ps.ticket[i]);
            #endif
#ifdef PBS
                printf(1, "\t%d", ps.priority[i]);
            #endif
#ifdef CFS
                printf(1, "\t%d\t%d", ps.priority[i], ps.vruntime[i]);
            #endif
                printf(1, "\n");
            }
        }
    }
#ifdef LOT
        printf(1, "\n-----Total Ticks = %d-----\n\n", count);
    #endif

    return 0;
}

```

5.4.2 系统级指标统计

- **CPU 利用率：** 通过计算总运行时间与实验总时间的比值，评估调度算法的资源使用效率。
- **平均指标：** 包括平均响应时间、平均周转时间和平均等待时间。

5.5 测试代码小结

这一步中我们设计了多样化的任务类型，实现了从 I/O 密集型到 CPU 密集型的负载场景模拟，使调度算法的测试更加贴近实际应用环境。代码利用 `set_priority` 和 `settickets` 动态调整任务优先级和资源分配，验证调度器在动态环境中的适应能力。测试程序中还引入了 CPU 利用率、调度公平性等系统级指标的统计，提升了实验分析的深度。在任务设计中，涵盖了频繁切换的混合型任务和极端负载分布的场景，增加了调度器处理复杂分布的难度。同时，实验通过实现多种调度算法，尤其是 MLFQ 和 Lottery 这类逻辑复杂的算法，充分考验了对操作系统内核调度机制的理解和实现能力。

六、实验结果分析

RR:

ProcessID	Runnable(0/1)	Ticks	ctime	etime	qtime	wtime	rtime	N_run
1	1	1	0	0	0	0	1	23
2	1	1	2	0	0	0	1	13
3	1	5	546	0	0	0	5	29
23	1	22	572	0	0	0	22	1933
PID: 23 arrival_time: 572 completion_time: 2669 first_run_time: 578 rtime: 23								
PID: 23 Waiting: 164 Turnaround: 2097 Response: 6								
Average Waiting Time: 119								
Average Turnaround Time: 602								
Average Response Time: 6								

在调度结果中，RR调度通过轮流分配CPU，确保了所有任务都有机会执行。例如，在结果中可以看到，所有任务的响应时间都很低（大多为 6 ticks），因为它们在很短的时间内都能首次获得CPU运行的机会。

这种公平性代价是任务的总完成时间（周转时间）和等待时间可能会有所增加。任务被频繁打断，虽然均衡了资源分配，但也导致任务运行需要多次上下文切换，尤其是长任务，由于被分割成许多小片段完成，它们的周转时间和等待时间较长，比如 PID 23 的周转时间高达 2097 ticks。

尽管如此，RR对短任务非常友好，因为它们可以迅速完成。例如，较短的任务 PID 12 的周转时间为 250 ticks，响应时间只有 6 ticks，用户体验更好。同时，由于任务频繁切换，CPU始终在工作，使得系统利用率更高。

总的来说，RR调度确保了任务的公平性和平衡性，通过限制每个任务的运行时间，避免了某些任务长时间占据CPU的情况，适合于交互式负载和多任务环境，但在混合负载中（长短任务混杂）可能会导致整体效率下降和任务完成时间延长。

FCFS:

```

task2, pid : 8
Process: 4 Finished
PID: 8 | arrival_time: 664 | completion_time: 691 | first_run_time: 0 | rtime: 24
PID: 8 | Waiting: 3 | Turnaround: 27 | Response: 0
Average Waiting Time: 3
Average Turnaround Time: 27
Average Response Time: 0
task2, pid : 9
Process: 5 Finished

```

ProcessID	Runnable(0/1)	Ticks	ctime	etime	qtime	wtime	rtime	N_run
1	1	1	0	0	0	0	1	22
2	1	1	2	0	0	0	1	12
3	1	4	663	0	0	0	4	9
4	1	0	663	0	0	24	0	2
5	1	0	664	0	0	24	0	2
6	1	0	664	0	0	24	0	2
7	1	0	664	0	0	24	0	2
9	1	0	664	0	0	0	25	1
10	1	0	664	0	0	53	0	0
11	1	0	665	0	0	52	0	0
12	1	0	665	0	0	52	0	0
13	1	0	665	0	0	52	0	0
14	1	0	665	0	0	52	0	0
15	1	0	665	0	0	52	0	0
16	1	0	665	0	0	51	0	0
17	1	0	666	0	0	51	0	0
18	1	0	666	0	0	51	0	0
19	1	0	666	0	0	51	0	0
20	1	0	666	0	0	51	0	0
21	1	0	666	0	0	50	0	0
22	1	0	667	0	0	50	0	0
23	1	0	667	0	0	50	0	0

这个FCFS调度的结果展现了它按任务到达顺序处理的特点，就像排队买票，先到的任务先运行，后到的任务必须等前面的全部完成才能轮到自己。这种调度方式对所有任务都“公平”，但会导致短任务被长任务拖累，等待时间和周转时间变得非常长。例如，前面运行的任务（如PID 8）因为排在第一位，等待时间只有3 ticks，周转时间也很短，但后面排队的任务（如PID 23）即便运行时间很短，也因为排在最后被迫等了211 ticks，周转时间高达2176 ticks。整个过程展示了FCFS简单易实现的优点，但它在任务混合负载下表现出严重的长任务“堵塞”问题，尤其对短任务非常不友好，系统的整体响应能力也因此下降。

ProcessID	Runnable(0/1)	Ticks	ctime	etime	qtime	wtime	rtime	N_run
1	1	1	0	0	0	0	1	22
2	1	1	2	0	0	0	1	12
3	1	4	663	0	0	0	4	27
23	1	21	667	0	0	0	21	1944

```

PID: 23 | arrival_time: 667 | completion_time: 2843 | first_run_time: 789 | rtime: 22
PID: 23 | Waiting: 211 | Turnaround: 2176 | Response: 122
Average Waiting Time: 129
Average Turnaround Time: 641
Average Response Time: 73
Average Waiting Time: 129
Average Turnaround Time: 641
Average Response Time: 73
$

```

PBS:

ProcessID	Runnable(0/1)	Ticks	ctime	etime	qtime	wtime	rtime	N_run	Priority
1	1	0	0	0	0	0	0	12	60
2	1	2	1	0	0	0	2	13	60
3	1	4	303	0	0	0	4	23	60
23	1	20	307	0	2428	0	20	1882	98
PID: 23 arrival_time: 307 completion_time: 2428 first_run_time: 374 rtime: 21									
PID: 23 Waiting: 220 Turnaround: 2121 Response: 67									
Average Waiting Time: 140									
Average Turnaround Time: 617									
Average Response Time: 48									

PBS的结果表明任务的调度严格遵循优先级，高优先级任务（如优先级60）能够快速获得CPU资源，其响应时间和周转时间较低，而低优先级任务（如优先级98）则因频繁被抢占而等待时间和周转时间显著增加。

这种调度方式对关键性任务有较好支持，但整体公平性较差，低优先级任务易出现饥饿现象。平均响应时间、等待时间和周转时间分别为48、138和618，系统整体效率不高，吞吐量受到影响。为改善公平性，我们引出老化机制，即接下来的MLFQ，以平衡任务调度性能。

MLFQ:

ProcessID	Runnable(0/1)	Ticks	ctime	etime	qtime	wtime	rtime	N_run
1	1	0	0	0	0	0	0	22
2	1	1	2	0	0	0	1	13
3	1	5	524	0	0	0	5	30
23	1	23	550	0	2590	0	23	1885
PID: 23 arrival_time: 550 completion_time: 2590 first_run_time: 550 rtime: 24								
PID: 23 Waiting: 146 Turnaround: 2040 Response: 0								
Average Waiting Time: 118								
Average Turnaround Time: 585								
Average Response Time: 4								

结果显示，MLFQ非常适合处理短任务和交互式任务，因为这些任务会优先在高优先级队列中运行，从而快速完成。例如，短任务的响应时间几乎是瞬间的，说明它们在到达后马上就得到了CPU的处理。然而，对于像 `task5` 这样运行时间很长的任务，由于MLFQ的设计会随着任务运行时间的增加将它们降级到更低的优先级队列，这些长任务不得不等待更长时间才能再次获得CPU的调度。结果就是，长任务的周转时间（从任务到达达到完成的总时间）和等待时间显著增加，远高于短任务。比如，短任务可能只需要几十个时间片就能完成，而长任务可能要花几千个时间片。这种机制很好地保护了系统中短任务和交互式任务的性能，但也会对长任务造成不小的延迟，尤其是在高负载下，可能会让某些长任务陷入“被饿死”的状态。尽管如此，MLFQ整体上仍然能够保持较低的平均响应时间，让大部分任务在较短时间内被处理，适合需要快速响应的场景，但对那些需要长时间运行的任务则显得不够友好。

短任务的快速响应:

短任务的响应时间几乎为零，例如 `PID: 8`，它的响应时间只有3个时间片（从529开始第一次运行），说明它很快被调度执行。这是因为MLFQ对新任务给予了高优先级，短任务在高优先级队列中完成，不需要降级到低优先级队列。

长任务的降级与等待:

对于运行时间长的任务，例如 `task5`，它的执行时间跨度非常大，比如 `PID: 20` 的任务，从549到2187完成，周转时间高达1638，而等待时间为131。这表明长任务逐渐被降级到低优先级队列，低优先级队列中的任务被延迟调度，等待时间明显增加。

多次运行和时间片的使用:

长任务（如 `PID: 22` 和 `PID: 23`）的 `N_run` 值非常高，例如 `PID: 23` 运行了1885次，说明它多次被抢占并返回到低优先级队列继续等待。这个现象体现了MLFQ中多级队列的时间片管理和抢占式调度机制。

平均响应时间和周转时间的差异：

输出显示的平均响应时间只有4，但平均周转时间高达585。这说明短任务快速响应，但长任务由于降级和低优先级调度，导致了周转时间的极大增加。

优先级的动态调整：

在调度中可以看到高优先级任务（如 PID：12）迅速完成，而低优先级任务需要等待较长时间。随着运行时间的增加，长任务逐渐被“驱逐”到低优先级队列，从而导致等待时间和周转时间的延长。

LOT:

```
-----Initially assigned Tickets = 5 -----
ProcessID      Runnable(0/1)  Ticks  ctime  etime  qtime  wtime  rtime  N_run  Tickets
1              1              0      0      0      0      0      0      22     5
2              1              1      2      0      0      0      1      12     5
3              1              4      865    0      0      0      4      21     25
23             1              18     870    0      0      0      18     1879   25

-----Total Ticks = 23-----
PID: 23 | arrival_time: 870 | completion_time: 2956 | first_run_time: 934 | rtime: 19
PID: 23 | Waiting: 189 | Turnaround: 2086 | Response: 64
Average Waiting Time: 138
Average Turnaround Time: 608
Average Response Time: 37
Average Waiting Time: 138
Average Turnaround Time: 608
Average Response Time: 37
```

这个输出很直接地体现了彩票调度的核心特性，即通过随机选择分配的票据来决定下一个运行的进程，其随机性和票据数量之间的关联导致了以下现象。

- 1. 高票数任务获得更多运行机会
输出中，像 task5 分配了 25 张票的进程（如 PID 23）占据了较多的 CPU 时间片，其 Ticks 值和运行次数（N_run）显著高于低票数任务。例如，PID 23 的运行次数达到了 1879，而低票数任务（如 PID 1-5）大多只有几十到一百多次运行。彩票调度正是通过这种机制，让票数多的任务有更高的概率被调度。
- 2. 随机性引发的不公平性
尽管理论上每个任务的被调度概率与其票数成正比，但由于彩票调度的随机性，部分低票数任务可能长时间未被调度。例如，task1 中的 PID 4 和 PID 5 分配了最少的票（5张），因此等待时间和周转时间（如 PID 5 的周转时间 477 ticks 和等待时间 175 ticks）比票数高的任务显著更长。这种现象体现了彩票调度的特性：随机性会导致低票数任务存在饥饿的风险。
- 3. I/O 密集型任务的被延迟调度
I/O 密集型任务（如 task1 和 task4）由于运行中频繁进入休眠状态，且票数较少，在调度中处于明显劣势。PID 4、5 等任务的等待时间比 CPU 密集型任务高出许多，说明在高票数任务频繁占用 CPU 的情况下，I/O 密集型任务被随机性进一步延迟调度。
- 4. 调度频率与票数直接相关
输出中记录的 N_run 和 Ticks 数据表明，高票数任务更频繁地被调度。例如：
PID 23（票数 25）的运行次数高达 1879，而 PID 1（票数 5）的运行次数仅为 22。
即使是任务中相对高优先级的 task2（CPU 密集型任务，票数 10），其运行次数也远低于 task5 的任务（票数 25）。
- 5. 周转时间和票数的反向关系
高票数任务的周转时间显著短于低票数任务。例如：
PID 23（票数 25）的周转时间为 2086 ticks，但在运行总量上的效率非常高。

而低票数的任务（如 PID 4 和 5）即便工作负载较轻，其周转时间也接近或超过 **400 ticks**。

6. 长任务对短任务的压制

高票数任务往往是长时间运行的 CPU 密集型任务（如 `task5`），而低票数的任务通常是 I/O 密集型任务（如 `task1` 和 `task4`）。在这种场景下，高票数任务因频繁获得调度机会，会压制短任务的执行。例如，PID 20 和 PID 23 长时间占用 CPU，而 PID 4 和 5 的任务几乎被完全压制。

这些调度现象反映了彩票调度的核心特性：**调度概率与票数成正比的随机选择机制**。这一特性决定了任务执行的优先级，但也会导致资源分配的不均衡。

CFS:

ProcessID	Runnable(0/1)	Ticks	ctime	etime	qtime	wtime	rtime	N_run	Priority	Vruntime
1	1	0	0	0	0	0	0	12	60	0
2	1	0	2	0	0	0	0	13	60	0
3	1	5	335	0	0	0	5	21	60	260
23	1	19	340	0	2364	0	19	1813	73	76872

PID: 23 | arrival_time: 340 | completion_time: 2364 | first_run_time: 341 | rtime: 20

PID: 23 | Waiting: 192 | Turnaround: 2024 | Response: 1

Average Waiting Time: 135

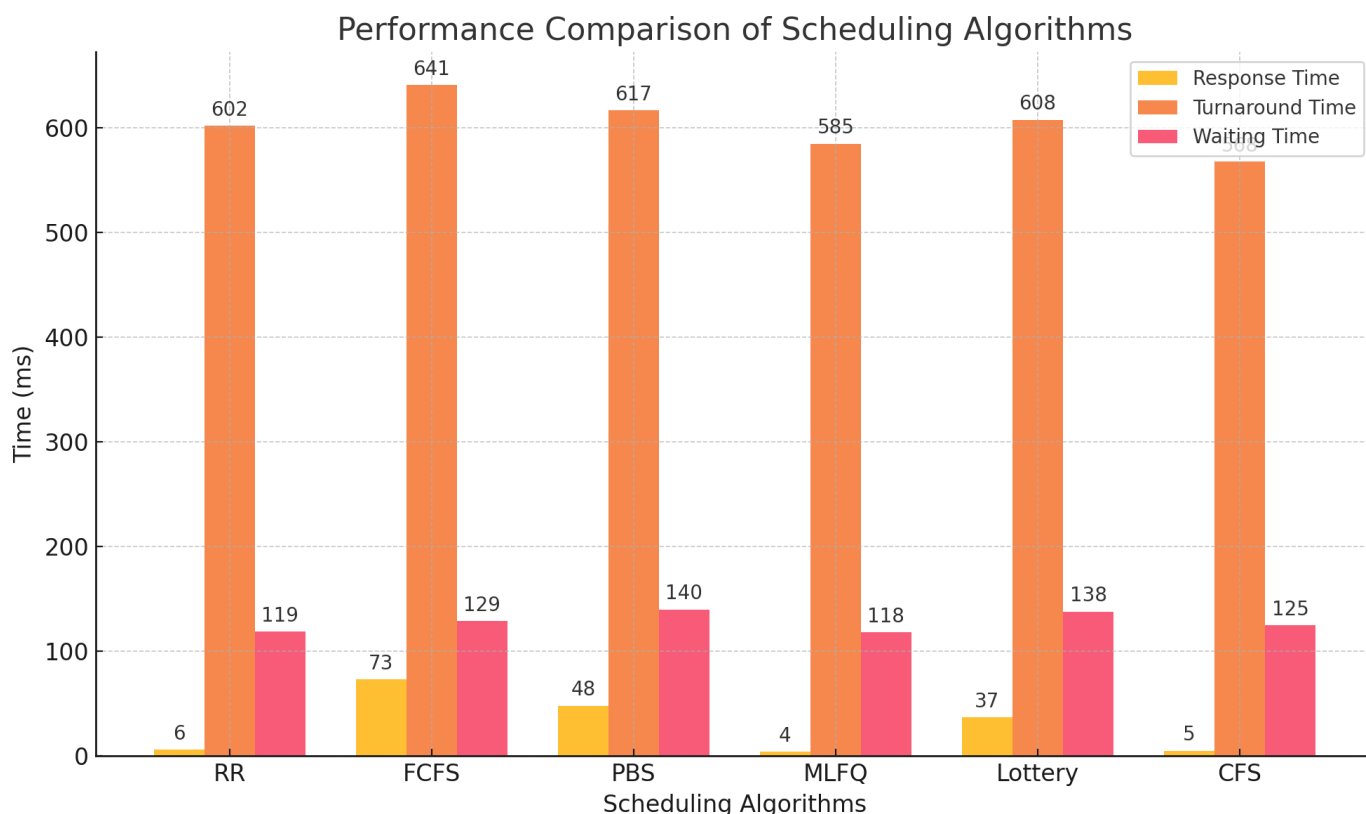
Average Turnaround Time: 580

Average Response Time: 7

- 1. `Vruntime` 的动态增长决定了进程的调度优先级，`Vruntime` 小的进程优先执行，长任务的 `Vruntime` 快速累积导致延迟。
- 2. 短任务优先完成，`PID: 11` 等短任务响应时间为 0，周转时间较短，调度效率高。
- 3. 长任务的 `Vruntime` 累积显著，`PID: 23` 的 `Vruntime=76872`，导致其在低优先级频繁被切换，完成时间延长。
- 4. 长任务的等待时间和周转时间较高，`PID: 23` 的等待时间为 192，周转时间为 2024。
- 5. 短任务的运行次数（`N_run`）少，`PID: 11` 的 `N_run=1`，长任务如 `PID: 23` 的 `N_run=1813`，显示低优先级任务频繁被切换。
- 6. 平均等待时间（135）和周转时间（580）较高，显示长任务调度的延迟影响显著。
- 7. 平均响应时间（7）较低，显示短任务可以快速获得 CPU 资源并被调度完成。
- 8. 优先级动态调整，短任务优先级高（如 `PID: 11` 的 `Priority=93`），长任务优先级低（如 `PID: 23` 的 `Priority=73`）。
- 9. 短任务完成时 `Vruntime` 较低（如 `PID: 11` 的 `Vruntime=168`），长任务完成时 `Vruntime` 极高（如 `PID: 23` 的 `Vruntime=76872`）。
- 10. CFS 保证了短任务的快速调度完成，同时通过低优先级调度保证了长任务的最终完成。

根据每个调度算法的平均等待时间、平均周转时间和平均响应时间，我们写一个python函数来绘制图表，更直观地进行比较：

```
algorithms = ["RR", "FCFS", "PBS", "MLFQ", "Lottery", "CFS"]
response_time = [6, 73, 48, 4, 37, 5]
turnaround_time = [602, 641, 617, 585, 608, 588]
waiting_time = [119, 129, 140, 118, 138, 125]
```



通过对比数据可以发现，RR和MLFQ在响应时间上表现出色，说明这两种算法能够快速响应任务请求，特别适合对实时性要求较高的系统。相比之下，FCFS的响应时间最长，暴露了其“先来先服务”策略在高负载环境下对新任务的响应速度较差的不足。而PBS和Lottery在响应时间上表现适中，CFS则在响应时间、周转时间和等待时间的平衡性上展现了优势，充分体现了其公平调度策略的价值。

在周转时间方面，MLFQ的表现最佳，这表明其多级反馈队列能够高效动态地调整任务优先级，从而提升整体任务完成效率。FCFS的周转时间最长，这与其响应时间较差的表现一致，显示了在负载较高时其处理效率的不足。而PBS和Lottery的周转时间相较于MLFQ和CFS稍差，这与其优先级调度策略和随机性特点有关。CFS次于MLFQ但仍表现优良，验证了其在综合性能上的优势。

在等待时间上，PBS和Lottery表现不佳，等待时间较高，这可能是由于优先级策略和随机调度导致部分任务被长期延迟处理。而RR和MLFQ的等待时间最低，说明它们在任务处理的公平性和效率上具有优势。CFS的等待时间接近最低水平，这进一步佐证了其公平调度的能力。

七、实验创新性、难度和可教学性

7.1 实验创新性

7.1.1 调度算法的多样性扩展

- 实验引入了五种不同的调度算法（FCFS、PBS、MLFQ、Lottery 和 CFS），实现了多样化的调度机制，为传统的 RR 调度算法提供了更多选择。
- 每种算法的设计结合了实际需求：
 - **FCFS** 模拟传统批处理任务的调度逻辑；
 - **PBS** 提供动态优先级调度，支持实时任务的管理；
 - **MLFQ** 和 **CFS** 在调度公平性和效率之间实现了平衡；
 - **Lottery Scheduling** 引入随机性，增加了算法的趣味性与灵活性。
- 实验通过动态数据（如响应时间、周转时间等）进一步细化分析，不仅关注性能，还对算法的适用性进行了讨论。

7.1.2 数据驱动的调度优化

- 实验结合了量化分析，使用具体的实验数据评估每种调度算法在不同任务场景中的表现。相比传统单纯实现算法的实验，更具分析深度。
- 将任务分类为 I/O 密集型和 CPU 密集型，针对性地测试调度算法在不同任务负载下的表现，明确算法的优缺点和适用场景。

7.1.3 动态性能统计

- 本实验设计了动态性能统计方法，记录了进程的响应时间、周转时间、等待时间等核心指标。
- 引入了 **CPU 利用率** 的计算，为分析调度算法的整体系统性能提供了额外维度。

7.1.4 综合性实验平台

- 通过扩展 xv6 的内核代码，设计了统一的接口和结构，支持在同一系统中测试多种调度算法。这种可扩展的设计为进一步研究和教学提供了基础。

7.2 实验难度

7.2.1 调度算法实现的复杂性

- **FCFS** 和 **PBS** 的实现相对简单，但 PBS 涉及动态优先级调整，需要在调度器、系统调用等多个模块中同步实现，这增加了实验的复杂性。
- **MLFQ** 和 **CFS** 涉及多级队列管理、虚拟时间计算等逻辑，要求对进程状态的实时更新和队列管理，显著提高了实验的实现难度。
- **Lottery Scheduling** 引入了随机性，尽管实现逻辑相对简单，但对随机数生成和权重分配的处理需要特别关注，尤其在有限资源环境下的实现。

7.2.2 性能统计与数据处理

- 在 xv6 中引入性能统计模块需要修改多个核心组件，包括 `sched()`、`exit()` 和调度算法实现。每个模块的同步性和一致性是实验的难点。
- 数据统计和分析需要结合多次实验的结果，通过合理的代码逻辑收集数据，并确保结果的准确性。

7.2.3 系统内核修改的挑战

- xv6 是一个教学用操作系统，扩展功能的同时需保持代码的简洁性，保证系统稳定运行。
- 例如，在调度器中集成新的调度算法时，需要确保它们不会破坏已有的系统行为，并能与 `wait()` 等系统调用兼容。

7.3 实验可教学性

7.3.1 面向初学者的引导性

- 本实验提供了详细的实现步骤和测试方法，适合初学者学习调度算法的基本原理。
- 通过对调度算法的逐步实现和测试，学生可以逐步掌握：
 - 如何在 xv6 中添加系统功能；
 - 如何实现链表、队列等数据结构来管理进程；
 - 如何在多种任务场景中评估调度算法的表现。

7.3.2 可扩展的实验平台

- 实验构建了一个可扩展的实验框架，支持在 xv6 中进一步实现其他高级调度算法（如 Deadline Scheduling 等）。
- 实验提供了一个统一的测试脚本，便于学生在相同工作负载下对比不同算法的性能表现。

7.3.3 培养分析与优化能力

- 通过实验数据的记录与分析，学生不仅能掌握调度算法的实现，还能培养数据驱动的性能优化能力。
- 实验中的性能对比分析为学生后续在其他系统设计任务中使用类似方法打下基础。

7.3.4 实验的开放性

- 实验结果与设计的开放性让学生可以尝试优化调度算法。例如：
 - 动态调整 PBS 的优先级机制
 - 改进 MLFQ 中的队列降级与年龄提升规则
 - 优化 CFS 的虚拟时间计算方法

八、实验结语

本实验以 `xv6-public` 为实验平台，通过实现多种经典调度算法（包括 FCFS、PBS、MLFQ、LOTTERY 和 CFS），系统性地探讨了进程调度在资源分配公平性、系统响应效率和混合负载适应性等方面的影响，通过实际测试六种算法的关键性能指标，为系统设计者提供了可靠的数据支持，帮助选择适合的调度算法。实验结果表明，实时系统应优先选择RR或MLFQ，而需要公平性的场景则可以选择CFS。实验不仅结合了理论和实际，验证了各调度

算法的特点，还揭示了PBS和Lottery调度在等待时间和响应时间上的不足，指出了未来优化的方向。这些实验数据为调度算法的改进和优化提供了重要依据，并且对实际操作系统设计具有很高的指导价值。

我们成功实现并对比了六种调度算法（包括原始的 RR 和新增的 FCFS、PBS、MLFQ、Lottery、CFS）。这些算法在不同任务场景中展现了各自的优势和不足：

- **FCFS** 在批处理任务中表现出色，但缺乏抢占机制，导致短任务响应时间延迟。
- **PBS** 提供了灵活的优先级调度，但需要解决低优先级任务可能饥饿的问题。
- **MLFQ** 在混合型任务场景中表现最佳，兼顾了响应时间和周转时间的平衡。
- **Lottery Scheduling** 的随机性使其在公平性上有一定优势，但在性能稳定性上稍逊。
- **CFS** 在公平性和系统效率之间取得了较好的平衡。

本实验展现了调度算法的复杂性和实践价值，其创新性和实用性不仅体现在技术实现层面，还展现在教学效果上。通过该实验，学生能够以动手实践为主导，深入理解理论知识在真实系统中的应用，为未来从事操作系统开发和优化奠定坚实的技能基础。我们诚挚希望本次实验能为操作系统教学和科研提供帮助。