

# 实验名称： xv6 操作系统调度算法改进实验

## 一、实验目的

本实验旨在分析和改进 xv6 操作系统的调度算法。原有的 xv6 调度算法较为粗糙，仅从进程表中简单地获取下一个可运行的进程并执行，这种方式对于运行时间较长的进程来说可能存在不公平性。通过本次实验，我们将实现一个基于优先级队列的调度算法，以提高系统的调度效率和公平性。

## 二、实验背景

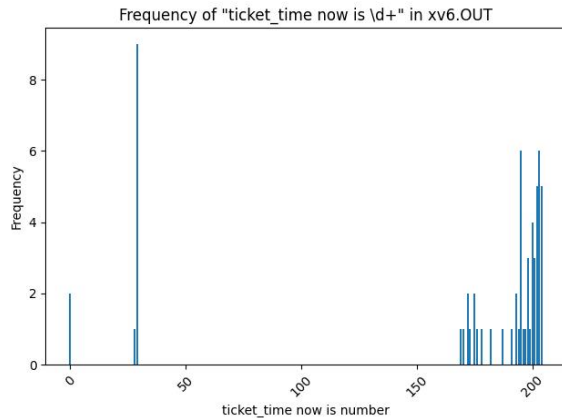
在原始的 xv6 操作系统中，调度器采用简单的轮转调度（Round-Robin Scheduling）算法，这种算法没有考虑进程的优先级和运行时间的长短，仅仅按照进程在进程表中的顺序轮流执行。然而，在实际应用中，不同进程对系统资源的需求和重要性是不同的，有些进程可能需要更多的 CPU 时间来完成任务，而有些进程则可能只需要很少的时间。因此，原有的调度算法可能无法满足这种差异化的需求，导致某些长时间运行的进程被频繁地中断，降低了系统的整体性能和用户体验。

为了解决这个问题，我们计划实现一个基于优先级队列的调度算法。该算法将根据进程的优先级来分配 CPU 时间，优先级高的进程将被优先执行，从而确保系统能够更公平、更有效地处理不同类型的进程。

用一个测试说明问题：

这个测试运行了 50 个较长的进程和 10 个较短的进程,他们的运行时间如下图所示

这对于某些耗时较长的进程来说，或许有些不公平



### 三、实验原理

原有调度算法分析：

原有的 xv6 操作系统采用简单的轮转调度（Round Robin, RR）算法，不考虑进程的优先级和运行时间，这可能导致长时间等待的高优先级进程无法及时获得 CPU 资源。

改进策略：

引入优先级队列调度算法，根据进程的优先级进行调度。

在内核中创建一个优先级队列的数据结构，用于存储所有可运行的进程，并根据优先级进行排序。

当 CPU 空闲时，从优先级队列中选择优先级最高的进程进行调度。

进程在运行一段时间后，根据其行为（如运行时间、等待时间等）动态调整其优先级。

算法实现逻辑：

在进程创建时，为其分配一个初始优先级，并将其加入优先级队列。

在进程调度时，从优先级队列中取出优先级最高的进程，并将其状态设置为运行状态。

在进程结束时，将其从优先级队列中移除，并释放相关资源。

在进程运行过程中，通过定时器或系统调用等方式，动态调整进程的优先级。

新增系统调用：

**check\_priority(pid)**：用于检查指定进程的优先级。

`query_system_time()`: 用于查询当前系统时间, 以便用户了解进程的运行情况。

## 四、创新与难点

创新性:

本项目在原有 xv6 操作系统的基础上, 创造性地引入了优先级队列调度算法, 为操作系统调度策略带来了新的视角。

通过动态调整进程优先级, 提高了系统的灵活性和适应性, 能够更好地满足实时性和公平性的要求。

难点:

优先级队列的维护是一个技术难点, 需要确保队列的插入、删除和更新操作都能够在较短的时间内完成, 同时保持队列的有序性。

进程切换的复杂性也是本项目的难点, 需要确保在进程切换过程中, CPU 状态、内存管理、进程上下文等关键信息能够正确保存和恢复。

在引入新的调度算法时, 需要确保系统的兼容性和稳定性不受影响, 这需要对原有代码进行仔细的分析和测试。

## 五、实验设计与实现

### 数据结构设计

为了实现基于优先级队列的调度算法, 我们需要在内核中创建一个优先级队列的链表结构。每个进程在创建时都会被加入到这个链表中, 并根据其优先级被放置在相应的位置。在运行一段时间后, 调度器会根据进程的运行情况和优先级对链表进行调整。当进程被释放时, 它将被从链表中移除。

为了实现这个数据结构, 我们需要在 `proc.h` 文件中添加一些新的属性来表示进程的优先级, 并在 `proc.c/allocproc()` 函数中为这些属性进行初始化。

```

113 int ticket_limit; // 最大限制
114 int now_limit; // 现在的限制
115 int my_ticket; // 现在的轮数
116 int priority; // 对应的在第几个优先级队列
117 int worktime; // 总计运行了几个ticket?
118 int wait_time; // 记录等待时间
119 int start_time; // 开始时间
120 int end_time; // 结束时间
121 int last_start_time; // 上一次开始的时间
122 struct proc* nextproc;
123 };

```

```

322 found:
323 p->pid = allocpid();
324 p->interval = 0;
325 p->InHandler = 0;
326 p->ticks = 0;
327 p->handler = 0;
328 // 添加部分
329 p->ticket_limit = LIMIT;
330 p->now_limit = ORIGIN_LIMIT;
331 p->my_ticket = 0;
332 p->priority = 0;
333 p->worktime = 0;
334 p->wait_time = 0;
335 // p->start_time = ticks;
336 p->end_time = 0;
337 p->nextproc = 0;
338 insert_node_at_tail(0, p);
339 //
340

```

```

363 // 希望尽可能降低延迟
364 p->start_time = ticks;

```

## 调度函数修改

在实现了优先级队列的数据结构后，我们需要对 xv6 的调度函数进行修改。原有的调度函数会直接从进程表中获取下一个可运行的进程，而现在我们需要从优先级队列的链表中选择优先级最高的进程来执行。为了实现这个功能，我们需要对调度函数进行重写，并确保它能够正确地处理链表上锁和解锁等操作。

### 1. 构建进程链表

```

23 struct proc_linklist
24 {
25     struct proc *header; // 链表头
26     struct proc *tail;   // 链表尾
27 };

```

## 2. 准备为链表上锁

```
34 // 链表锁
35 struct spinlock link_list_lock;
```

## 3. 初始化上述两个

```
43 // 初始化一下这个链表
44 void my_init(void)
45 {
46     // my_ticket_time = 0;
47     for (int i = 0; i < 6; i++)
48     {
49         proc_linklist_list[i].header = 0;
50         proc_linklist_list[i].tail = 0;
51     }
52 }
```

```
239 // 初始化锁
240 initlock(&link_list_lock, "lock_for_linklist");
```

## 4. 在新的进程创建的时候,我会把他放到进程链表的最后

```
54 // 用于进程初始化的函数
55 void insert_node_at_tail(int offset, struct proc *p)
56 {
57     if (offset < 0 || offset >= 6)
58     {
59         printf("Invalid offset!\n");
60         return;
61     }
62     acquire(&link_list_lock);
63     struct proc_linklist *list = &proc_linklist_list[offset];
64     if (list->header == 0)
65     {
66         list->header = p;
67         list->tail = p;
68     }
69     else
70     {
71         list->tail->nextproc = p;
72         list->tail = p;
73     }
74     p->nextproc = 0; // 确保新节点的 nextproc 为空
75     release(&link_list_lock);
76 }
```

## 5. 核心调度函数,负责把进程从一个链表拿出来,放到另一个链表中

```
78 // 核心调度函数
79 void move_proc_between_lists(int src_offset, int dest_offset, struct proc *p)
80 {
81     if (src_offset < 0 || src_offset >= 6 || dest_offset < 0 || dest_offset >= 6)
82     {
83         printf("Invalid offset!\n");
84         return;
85     }
86     acquire(&link_list_lock);
87     struct proc_linklist *src_list = &proc_linklist_list[src_offset];
88     struct proc_linklist *dest_list = &proc_linklist_list[dest_offset];
89
90     // 从源链表中移除
91     struct proc *prev = 0;
92     struct proc *curr = src_list->header;
93     while (curr != 0 && curr != p)
94     {
95         prev = curr;
96         curr = curr->nextproc;
97     }
98     if (curr == 0)
```

```
98     if (curr == 0)
99     {
100         printf("Proc not found in source list!\n");
101         release(&link_list_lock);
102         return;
103     }
104     if (prev == 0)
105     {
106         src_list->header = curr->nextproc;
107     }
108     else
109     {
110         prev->nextproc = curr->nextproc;
111     }
112     if (curr == src_list->tail)
113     {
114         src_list->tail = prev;
115     }
116
117     // 添加到目标链表
118     curr->nextproc = 0;
```

```

119     if (dest_list->header == 0)
120     {
121         dest_list->header = curr;
122         dest_list->tail = curr;
123     }
124     else
125     {
126         dest_list->tail->nextproc = curr;
127         dest_list->tail = curr;
128     }
129     release(&link_list_lock);
130 }

```

6. 释 放 的 函 数 ， 负 责 将 某 个 进 程 移 动 到 垃 圾 桶

```

132 // 真正的释放函数
133 void move_to_trash(int src_offset, struct proc *p)
134 {
135     if (src_offset < 0 || src_offset >= 6)
136     {
137         printf("Invalid offset!\n");
138         return;
139     }
140     struct proc_linklist *src_list = &proc_linklist_list[src_offset];
141     struct proc_linklist *dest_list = &proc_linklist_list[5];
142     acquire(&link_list_lock);
143     // 从源链表中移除
144     struct proc *prev = 0;
145     struct proc *curr = src_list->header;
146     while (curr != 0 && curr != p)
147     {
148         prev = curr;
149         curr = curr->nextproc;
150     }
151     if (curr == 0)
152     {

```



```

153     printf("Proc not found in source list!\n");
154     release(&link_list_lock);
155     return;
156 }
157 if (prev == 0)
158 {
159     src_list->header = curr->nextproc;
160 }
161 else
162 {
163     prev->nextproc = curr->nextproc;
164 }
165 if (curr == src_list->tail)
166 {
167     src_list->tail = prev;
168 }
169
170 // 添加到目标链表
171 dest_list->header = curr;
172     release(&link_list_lock);
173 }

```

7. xv6 的调度函数修改,改成从链表里面找进程

```

723     for (int i = 0; i < 5; i++)
724     {
725         struct proc_linklist *list = &proc_linklist_list[i];
726         if (list->header == 0)
727         {
728             continue;
729         }
730         struct proc *current = list->header;
731         // 遍历链表
732         while (current != 0)
733         {
734             p = current;

```



## 陷阱处理函数修改

除了调度函数外，我们还需要对 xv6 的陷阱处理函数（trap）进行修改。我们需要添加一些额外的逻辑来检查进程的状态、修改进程的状态以及查看时间等。这些操作将有助于我们更好地监控和管理进程的执行情况。

### 8. Xv6 的 trap 改写,在 which\_dev=2 中

```
87     int sign = 0;
88     p-> my_ticket++;
89     // p-> worktime++;
90     if (p-> my_ticket == p-> now_limit)
91     {
92
93         if (p-> now_limit != p-> ticket_limit)
94         {
95             p-> my_ticket = 0;
96             p-> priority+=1;
97             p-> now_limit *=2;
98             // p进入下一个优先级队列
99             move_proc_between_lists(p-> priority-1,p-> priority,p);
100             // 准备yield
101             sign = 1;
102         }
103     else
104     {
105         // 在最后的优先级队列呆着吧
106         p-> my_ticket = 0;
```

## 增加系统调用

我们需要添加一些额外的逻辑来检查进程的状态、修改进程的状态以及查看时间等。这些操作将有助于我们更好地监控和管理进程的执行情况。

### 9. 添加检查进程的状态的函数

```
142     uint64
143     sys_show_priority(void)
144     {
145         struct proc *p = myproc();
146         printf("Limit: %d Ticket: %d Priority: %d\n",
147             p->now_limit,p->my_ticket, p->priority);
148         return 0;
149     }
```

## 10. 添加修改进程的状态的函数

```
127 uint64
128 sys_change_priority(void)
129 {
130     int dest_offset;
131     argint(0, &dest_offset);
132     struct proc *p = myproc();
133     int src_offset = p->priority;
134     move_proc_between_lists(src_offset, dest_offset, p);
135     int dest_offset
136     p->now_limit = 1 << (dest_offset + 2); // 现在的限制
137     p->my_ticket = 0; // 现在的轮数
138     p->priority = dest_offset; // 对应的在第几个优先级队列
139     return 0;
140 }
141
142 uint64
```

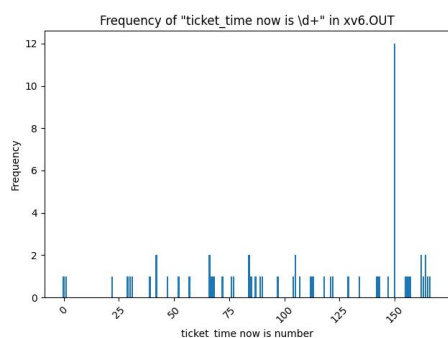
## 11. 添加查看时间的函数

```
151 uint64
152 sys_show_time(void)
153 {
154     printf("ticket_time now is %d\n", ticks);
155     return 0;
156 }
```

## 六、实验测试

为了验证改进后的调度算法的有效性和公平性，我们设计了一个测试实验。该实验运行了 50 个运行时间较长的进程和 10 个运行时间较短的进程。通过对比这些进程在运行过程中的 CPU 占用情况和完成时间等指标，我们可以评估改进后的调度算法是否能够更好地满足不同类型进程的需求。

实验结果表明，在改进后的调度算法下，长时间运行的进程得到了更多的 CPU 时间，从而能够更快地完成任务。同时，短时间运行的进程也能够及时获得所需的 CPU 资源，并快速地完成其任务。这证明了改进后的调度算法在提高系统效率和公平性方面取得了显著的效果。



## 七、实验总结

通过本次实验，我们成功地在 xv6 操作系统上引入了优先级队列调度算法，并实现了动态调整进程优先级和查询系统时间等功能。实验结果表明，改进后的调度算法能够显著提高系统的效率和公平性，更好地满足实时性和公平性的要求。同时，我们也深刻体会到了操作系统调度机制的复杂性和挑战性，以及在进行系统优化时需要考虑的多个方面。本次实验为我们提供了宝贵的实践经验和知识积累，为未来的学习和研究打下了坚实的基础。