调度-多级反馈队列

邰颖
南京大学
智能科学与技术学院

多级反馈队列（MLFQ）

首次在兼容的分时系统（Compatible Time-Shareing System, CTSS）中由 Corbato 描述

目标：

1.通过优先运行较短的进程，**优化周转时间（SJF）**

2.在没有预知进程运行时间的情况下，**最小化响应时间**

3.一种从过去学习以预测未来的调度策略

多级反馈队列（MLFQ）：基本规则

- MLFQ 有多个不同的队列
  - 每个队列被分配不同的**优先级**。
  - 一个准备运行的进程被分配到一个单独的队列。
- **规则：**
  1. 调度程序选择一个**高优先级队列**中的进程在 CPU 上运行
  2. 在同一队列中的进程之间使用**轮转调度**（因为它们具有相同的优先级）

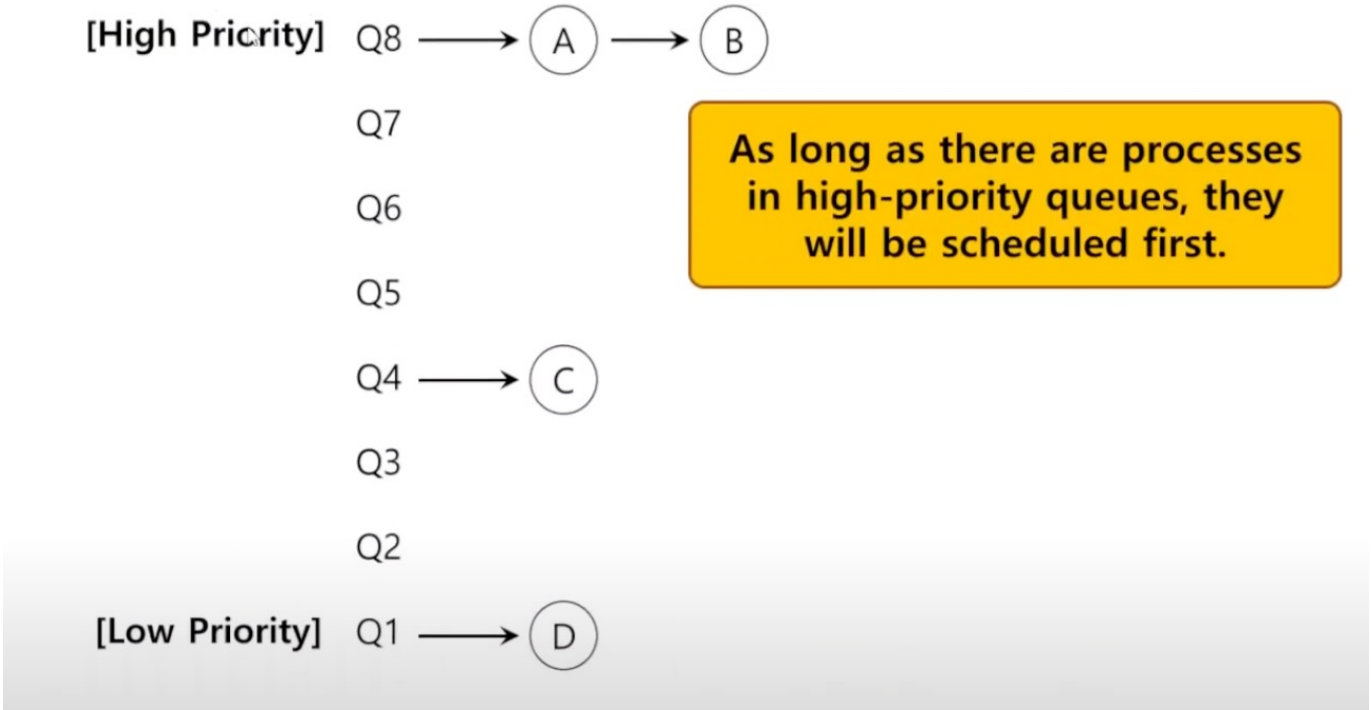
Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).
Rule 2: If Priority(A) == Priority(B), A & B run in RR.

多级反馈队列（MLFQ）：基本规则

- 关键是调度器如何设置队列的优先级。
- MLFQ 根据进程的**观察行为**来调整优先级，而不是根据固定的优先级值。

- **示例：**
  - 一个进程在等待 I/O 时反复释放 CPU（一个交互式进程）→ 保持其高优先级。
  - 一个长时间密集使用 CPU 的进程（一个 CPU 密集型进程）→ 降低其优先级。

多级反馈队列调度（MLFQ）：例子



方法 #1：如何更改优先级

规则 3：当进程进入系统时，它被放置在**最高优先级队列**中。

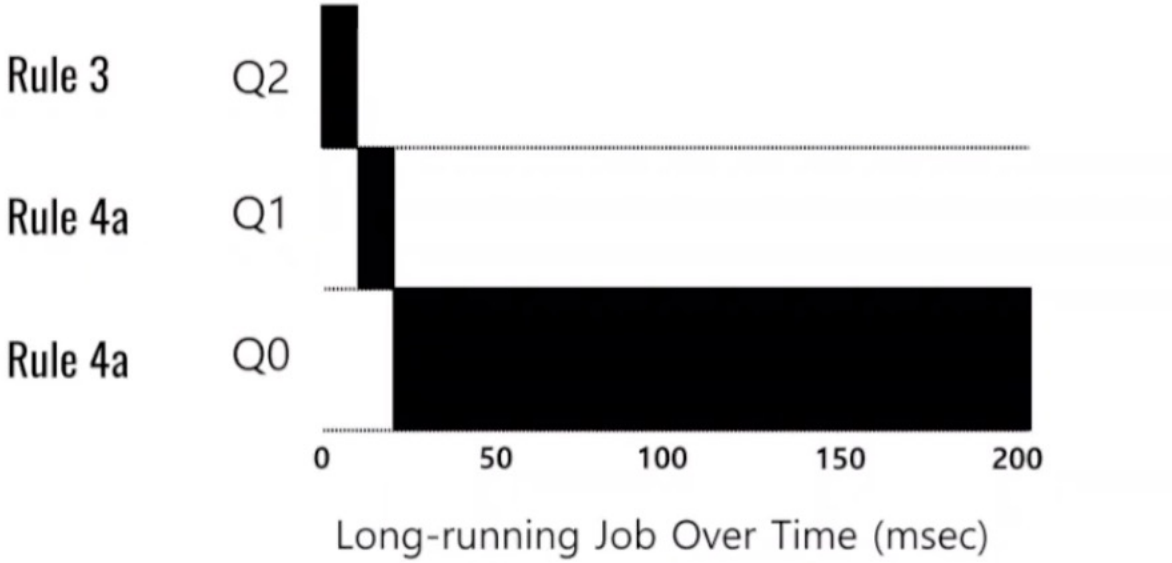
规则 4a：如果一个进程在运行时使用了整个时间片，它的优先级将降低（即，它会下移到较低的队列）。

规则 4b：如果一个进程在时间片结束之前放弃了 CPU，它将保持在当前优先级水平。

总结：通过这种方式，MLFQ 可以近似模拟最短作业优先（SJF）调度策略。

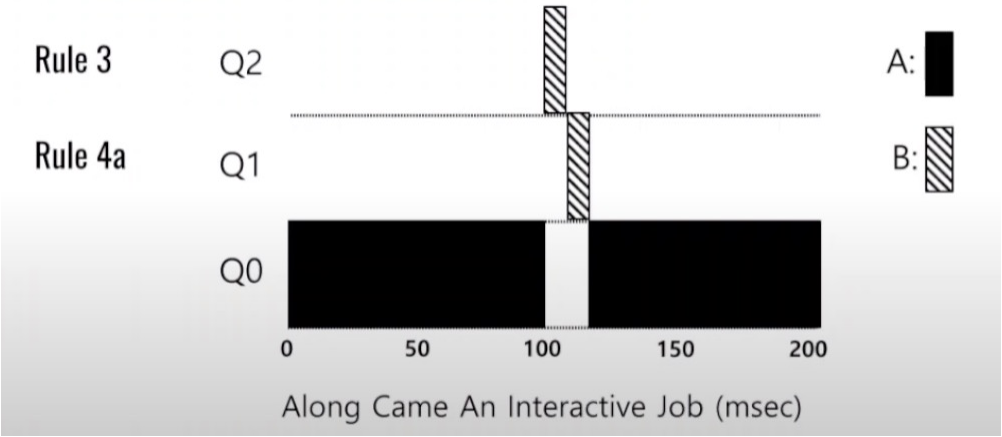
示例 1：一个长时间运行的进程

- 一个三队列调度器，时间片为 10 毫秒
  - **规则 3**：进程进入时被放入**Q2**队列（最高优先级队列）。
  - **规则 4a**：如果进程使用了整个时间片，它的优先级降低，移到**Q1**队列。
  - **规则 4a**：如果进程继续使用整个时间片，它的优先级继续降低，移到**Q0**队列。



示例 2：一个短进程的到来

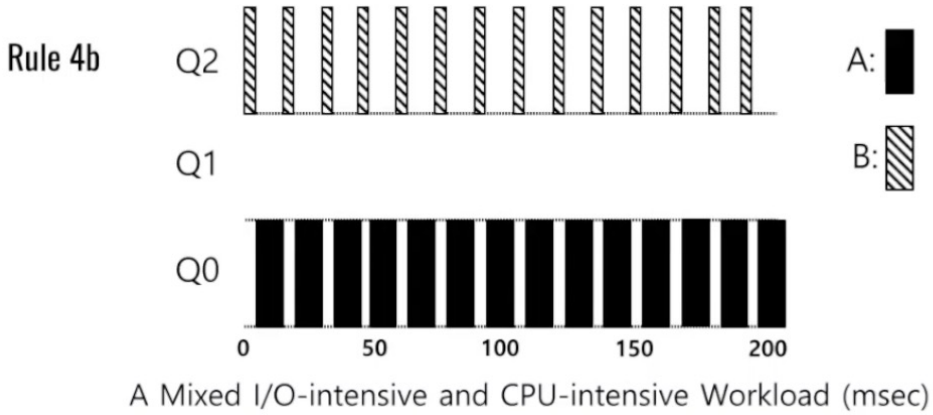
- 假设：
  - 进程 A：一个长时间运行的 CPU 密集型进程
  - 进程 B：一个短时间运行的交互式进程（运行时间为 20 毫秒）
  - 进程 A 已经运行了一段时间，然后进程 B 在时间 T=100 时到达。



- 调度程序并不完全知道新进程的运行时间，但它假设新进程是短时间运行
- 如果假设正确，进程将较早完成，这种方式接近模拟最短作业优先（SJF）调度策略

示例 3：I/O 操作怎么办？

- 假设：
  - **进程 A**：一个长时间运行的 CPU 密集型进程。
  - **进程 B**：一个交互式进程，它只需要 1 毫秒的 CPU 时间，然后执行 I/O 操作。



A Mixed I/O-intensive and CPU-intensive Workload (msec)

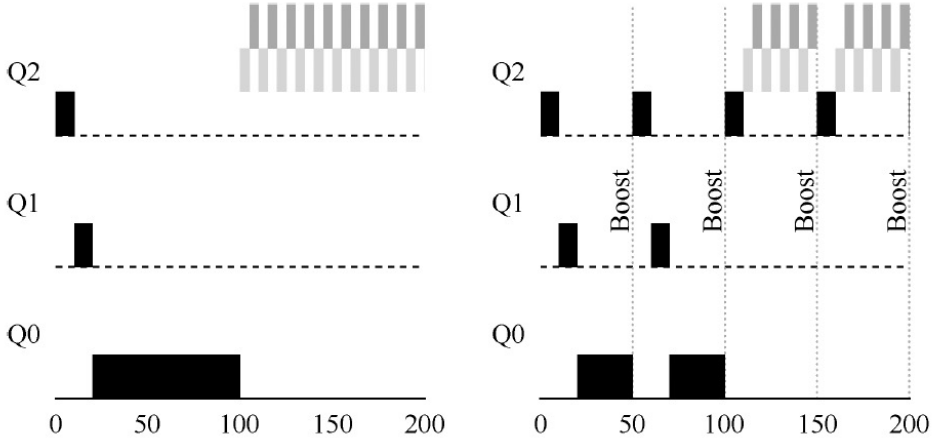
- 不要因 CPU 被提前放弃（在时间片完成之前）而惩罚进程（即降低优先级）
- MLFQ 方法保持交互式进程在最高优先级队列中。

基本 MLFQ 和方法 #1 的问题

- **饿死问题**
  - 如果系统中有“太多”交互式进程，长时间运行的进程将永远无法获得任何 CPU 时间。
- **进程可能会随着时间改变其行为**
  - 一个 CPU 密集型进程可能会变成 I/O 密集型进程。
- **调度器作弊**
  - 进程在时间片的 99% 时间内运行后，发出 I/O 操作——这样就没有消耗整个时间片，进程仍然保留在当前队列中。
  - 进程因此获得了更高比例的 CPU 时间

方法 #2：优先级提升

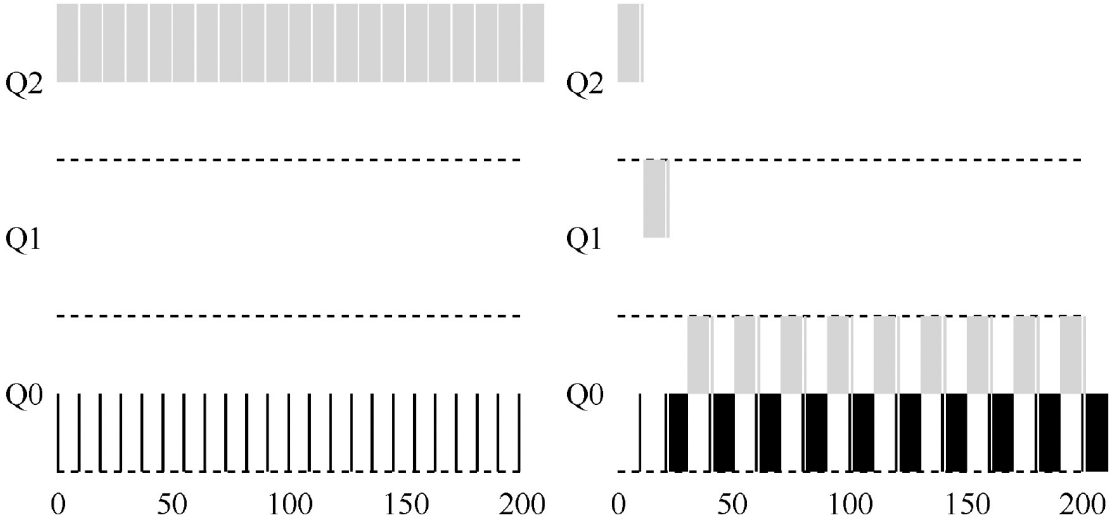
- 解决了**饿死问题**和**CPU 密集型进程变为交互式进程**的问题。
- **规则 5**：在经过一段时间 S 后，将系统中所有进程移动到最高优先级队列。
- **S 的合理值是多少？一个有趣的问题**
  - **示例**：一个长时间运行的进程 A 和两个短时间运行的交互式进程 B 和 C，每 50 毫秒进行一次优先级提升。



- 左边没有优先级提升，长工作在两个短工作到达后被饿死。
- 右边每50ms 就有一次优先级提升（这里只是举例，这个值可能过小），每过50ms 就被提升到最高优先级，从而定期获得执行。

方法 #3：更好的记录每个队列中进程消耗的 CPU 时间

- 防止进程作弊（gaming the scheduler）
- **解决方案：**
  - **规则 4**（重写规则 4a 和 4b）：一旦一个进程在某个队列中**消耗了时间片**（无论它放弃了多少次 CPU），它的优先级将降低（即，它会移到较低的队列）。

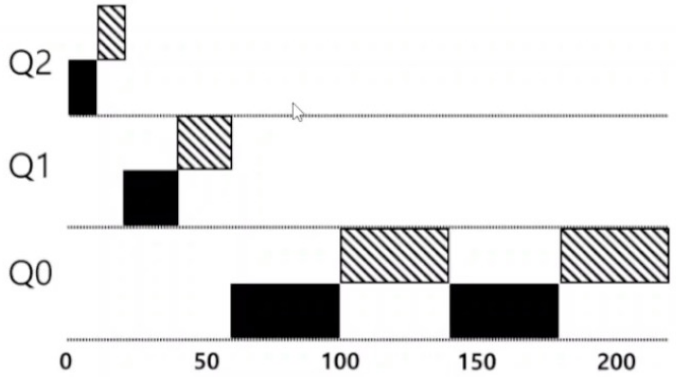


- **左图（没有作弊容忍）：**进程按照时间片进行调度。
- **右图（有作弊容忍）：**进程在每个队列中消耗时间片后，其优先级会下降，确保 CPU 时间得到更公平的分配。

调整 MLFQ 和其他问题

- 我们如何对 MLFQ 调度器进行参数化？
  - **队列的数量，每个队列的时间片，S 的最佳值**，等等

- **时间片设计方面，通常：**
  - 高优先级队列分配**短时间片**。例如：10 毫秒或更少。
  - 低优先级队列分配**较长时间片**。例如：40 毫秒。



Example) 10ms for the highest queue, 20ms for the middle, 40ms for the lowest

- 对于最高优先级队列分配 10 毫秒，次优先级队列分配 20 毫秒，最低优先级队列分配 40 毫秒。
- 低优先级队列会有更长的时间片

调整 MLFQ 和其他问题（续）

- 对于 **Solaris - 分时调度类（TS）**
  - 它提供了一组表来决定进程在其生命周期中如何调整优先级，每层的时间片多大，以及多久提升一个工作的优先级
  - 60 个队列
  - 时间片长度逐渐增加
    - 最高优先级：20 毫秒
    - 最低优先级：几百毫秒
  - 优先级大约每 1 秒提升一次

- 对于 **FreeBSD**，使用数学公式来确定优先级，使用量会随时间衰减。

- **对于其他操作系统**，操作系统/内核服务会被赋予更高的优先级。
  - 允许用户提供建议来帮助设置优先级级别，例如：使用 **nice** 命令。

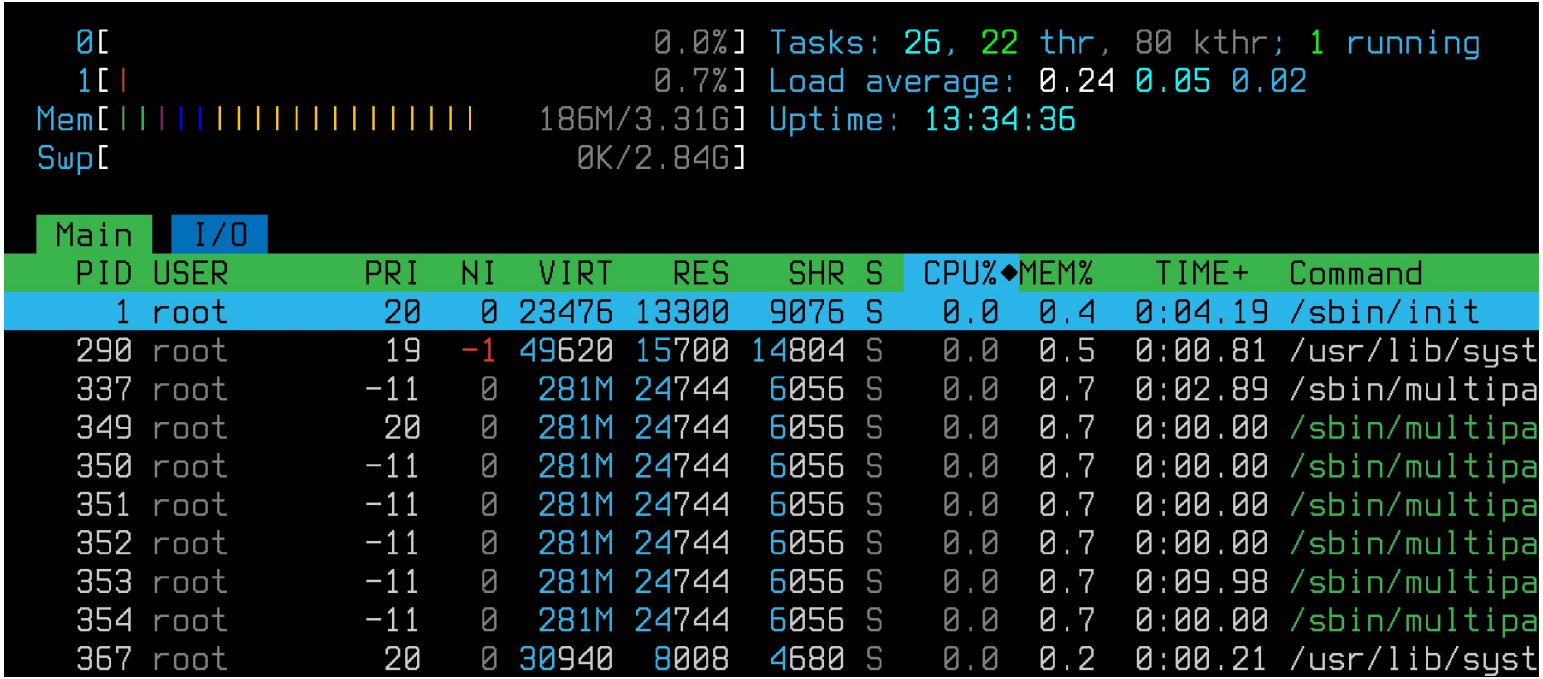
nice指令

man nice

```
NICE(1)                          User Commands                          NICE(1)

NAME
       nice - run a program with modified scheduling priority

SYNOPSIS
       nice [OPTION] [COMMAND [ARG]...]

DESCRIPTION
       Run COMMAND with an adjusted niceness, which affects process scheduling.
       With no COMMAND, print the current niceness.  Niceness values range from
       -20  (most  favorable  to  the  process)  to  19 (least favorable to the
       process).

       Mandatory arguments to long options are mandatory for short options too.

       -n, --adjustment=N
              add integer N to the niceness (default 10)

       --help display this help and exit

       --version
              output version information and exit
 Manual page nice(1) line 1 (press h for help or q to quit)
```

htop查看NI

htop -> NI(nice值，表示进程调度优先级；负值表示较高的优先级)



Nice指令调整优先级

./cpu hello
htop –p%pid%

nice -11 ./cpu hello
htop –p%pid%

```
  0[                          0.0%] Tasks: 26, 22 thr, 80 kthr; 1 running
  1[|                         0.7%] Load average: 0.24 0.05 0.02
Mem[|||||||||||||||||     186M/3.31G] Uptime: 13:34:36
Swp[                        0K/2.84G]

  Main   I/O
    PID USER      PRI  NI  VIRT   RES   SHR S CPU%◆MEM%   TIME+  Command
      1 root       20   0 23476 13300  9076 S  0.0  0.4  0:04.19 /sbin/init
    290 root       19  -1 49620 15700 14804 S  0.0  0.5  0:00.81 /usr/lib/syst
    337 root      -11   0  281M 24744  6056 S  0.0  0.7  0:02.89 /sbin/multipa
    349 root       20   0  281M 24744  6056 S  0.0  0.7  0:00.00 /sbin/multipa
    350 root      -11   0  281M 24744  6056 S  0.0  0.7  0:00.00 /sbin/multipa
    351 root      -11   0  281M 24744  6056 S  0.0  0.7  0:00.00 /sbin/multipa
    352 root      -11   0  281M 24744  6056 S  0.0  0.7  0:00.00 /sbin/multipa
    353 root      -11   0  281M 24744  6056 S  0.0  0.7  0:09.98 /sbin/multipa
    354 root      -11   0  281M 24744  6056 S  0.0  0.7  0:00.00 /sbin/multipa
    367 root       20   0 30940  8008  4680 S  0.0  0.2  0:00.21 /usr/lib/syst
```

多级反馈队列调度（MLFQ）小结

- 优化后的 MLFQ 规则：
  - **规则 1**：如果 Priority(A) > Priority(B)，则 A 运行（B 不运行）。
  - **规则 2**：如果 Priority(A) = Priority(B)，则 A 和 B 使用轮转调度（RR）。
  - **规则 3**：当一个进程进入系统时，它被放置在最高优先级队列。
  - **规则 4**：一旦进程在某个队列中消耗完其时间片（无论它已放弃多少次 CPU），其优先级将降低（即，它会移到较低的队列）。
  - **规则 5**：经过一段时间 S 后，将系统中所有的进程移动到最高优先级队列。

调度-比例份额

邰颖
南京大学
智能科学与技术学院

调度：比例份额

比例共享调度器（公平共享调度器）

- 保证每个进程获得**一定百分比**的 CPU 时间

- 不一定优化周转时间或响应时间

基本概念：彩票数表示份额

- 彩票
  - 代表进程（或用户）应获得的资源份额
  - 彩票的百分比代表其在系统资源中所占的份额
- 示例
  - 有两个进程，A 和 B：
    - 进程 A 拥有 75 张彩票 → 获得 75% 的 CPU 时间
    - 进程 B 拥有 25 张彩票 → 获得 25% 的 CPU 时间

方法 #1: 彩票调度

- 每个时间片都可以进行抽奖
- 调度器随机选择一张**中奖彩票**（概率性/随机化）
  - 运行持有中奖彩票的进程
- **示例：**总共有 100 张彩票
  - 进程 A 拥有 75 张彩票：0 ~ 74
  - 进程 B 拥有 25 张彩票：75 ~ 99

Scheduler's winning tickets:	63	85	70	39	76	17	29	41	36	39	10	99	68	83	63
Resulting scheduler:	A	B	A	A	B	A	A	A	A	A	A	B	A	B	A

- 这两个进程运行的时间越长，它们得到的CPU 时间比例就会越接近期望。

为什么使用随机？

- **随机** 可以避免奇怪的边界情况行为

- **随机** 是轻量级的
  - 减少了每个进程的状态记录

- **随机** 是快速的
  - 从生成器生成一个随机数通常很快（硬件支持）

彩票调度中的若干机制

#1：彩票货币

- 用户（在多用户系统中）可以将彩票分配给他们的进程，可以使用他们想要的任何货币（本地货币）。
- 系统将这种货币转换为正确的全局彩票。

- **例子：**总共有200张彩票（全局彩票）
  - 进程A有100张彩票
  - 进程B有100张彩票

User A → *500* (A's local currency) to A1 → *50* (global currency)
→ *500* (A's local currency) to A2 → *50* (global currency)

User B → *10* (B's local currency) to B1 → *100* (global currency)

用户A：
→ 500（A的本地货币）分配给A1 → 50（全局彩票）
→ 500（A的本地货币）分配给A2 → 50（全局彩票）

用户B：
→ 10（B的本地货币）分配给B1 → 100（全局彩票）

彩票机制

#2：彩票转让

•一个进程可以暂时将其**彩票**交给另一个进程。

•在客户端-服务器应用中很有用，服务器代表客户端执行任务。

#3：彩票通胀

•一个进程可以暂时增加或减少其拥有的**彩票**数量。

•如果任何一个进程需要更多的CPU时间，它可以增加其**彩票**数量。

•假设一组进程之间相互信任，以防止滥用。

彩票调度的实现

示例：有三个进程，A、B 和 C。总共有400张票。随机抽签，假设winner=300

•将这些进程放在一个链表中：


```
1       // counter: used to track if we've found the winner yet
2       int counter = 0;
3
4       // winner: use some call to a random number generator to
5       // get a value, between 0 and the total # of tickets
6       int winner = getrandom(0, totaltickets);
7
8       // current: use this to walk through the list of jobs
9       node_t *current = head;
10
11      // loop until the sum of ticket values is > the winner
12      while (current) {
13              counter = counter + current->tickets;
14              if (counter > winner)
15                      break; // found the winner
16              current = current->next;
17      }
18      // 'current' is the winner: schedule it...
```

What optimizations can you think of?

彩票调度的实现

- 如何评估？
- 定义公平性指标
  - **第一个进程完成的时间** 除以 **第二个进程完成的时间**。
- 示例：
  - 假设有两个进程，每个进程的运行时间为 **R = 10**。
    - 第一个进程在 **10** 时间点完成。
    - 第二个进程在 **20** 时间点完成。
  - **公平性指标 F** = 第一个进程完成的时间 ÷ 第二个进程完成的时间

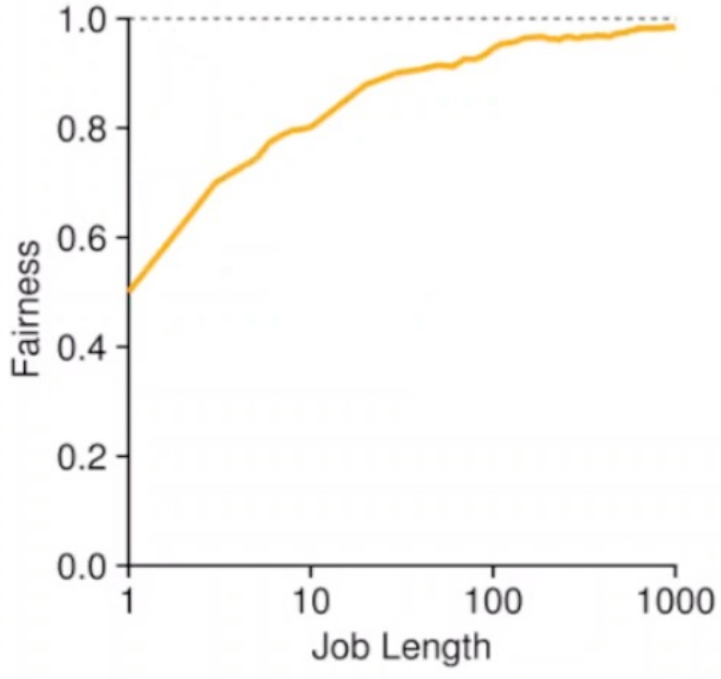
  即：F = 10 ÷ 20 = 0.5。

- 解释：
  - **F** 值接近 1 时，表示两个进程几乎同时完成，系统比较公平。
  - **F** 值较小则说明第一个进程完成得较早，第二个进程可能处于等待状态，系统的公平性较差。

彩票调度的公平性研究

模拟场景：

- 有两个进程，运行时间 **R**（作业长度）范围从 1 到 1000。
- 每个进程有相同数量的票（100张）。



当运行时间不太长时，平均公平性可能会比较严重。

如何分配票数？

- 一种方法是假设用户最了解情况，因此让用户来分配票数。
- 但这仍然是一个开放的问题。

方法#2：步长调度

- 随机性偶尔会带来不好的结果，尤其是对于运行时间较短的进程。
- 这是Waldspurger发明的一个**确定性公平共享调度器**（deterministic fair-share scheduler）
- 每个进程的步长（Stride）计算方式是：
  - 步长 =（某个大的数字）/（该进程的票数）
  - 例如：假设大数字为10000
    - 如果进程A有100张票，那么A的步长就是10000/100 = 100
    - 如果进程B有50张票，那么B的步长就是10000/50 = 200
- 当一个进程运行时，按它的步长增加一个计数器（也叫做行程（pass）值）
  - 选择运行Pass值最小的进程

```
current = remove_min(queue);          // pick client with minimum pass
schedule(current);                    // use resource for quantum
current->pass += current->stride;     // compute next pass using stride
insert(queue, current);               // put back into the queue
```

Pseudocode for Stride Scheduling

步长调度示例

表 9.1　　步长调度：记录

行程值（A）（步长=100）	行程值（B）（步长=200）	行程值（C）（步长=40）	谁运行
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	……

任意的，所有具有同样低的行程值的进程，都可能被选中

- 彩票调度算法只能一段时间后，在概率上实现比例，而步长调度算法可以在每个调度周期后做到完全正确。
- 如果一个新的进程进入，并且它的Pass值为0，它将垄断CPU资源！而彩票调度可以更合理地处理新加入的进程，因为它**不需要为每个进程维护全局状态**。

方法 #3：Linux完全公平调度器（CFS）

- 目标：公平地将CPU资源均匀分配给所有竞争的进程
- 使用基于计数的技术，称为虚拟运行时间（virtual runtime，vruntime）
- 当进程运行时，其虚拟运行时间会增加（可能与物理时间的增加速率相同）
- 当进行调度决策时，选择虚拟运行时间最小的进程
- 使用周期性的时钟中断——只能在固定的时间间隔做出决策
  - 如果时间片不是时钟中断间隔的整数倍，不会受到影响

方法 #3：Linux完全公平调度器（CFS）

- **控制参数：**
  - **sched_latency** — 确定一个进程应该运行多长时间后再考虑进行调度切换（典型值为48ms）
    - 进程的时间片 = (sched_latency / n 进程数)
    - 例如：假设n=4，时间片 = 48/4 = 12ms
  - **min_granularity** — 解决进程过多时（当n非常大时，时间片变得很小）
    - 即使n很大时，也要为进程分配最小的时间片，通常设置为6ms

方法 #3：Linux完全公平调度器（CFS）

- 加权（**nice值**） — 允许用户/管理员控制进程优先级（nice和renice指令）
  - nice值的范围：[-20, +19]，其中 (+) 表示较低的优先级， (-) 表示较高的优先级
  - nice值映射到权重表：

```
static const int prio_to_weight[40] = {
    /* -20 */ 88761, 71755, 56483, 46273, 36291,
    /* -15 */ 29154, 23254, 18705, 14949, 11916,
    /* -10 */  9548,  7620,  6100,  4904,  3906,
    /*  -5 */  3121,  2501,  1991,  1586,  1277,
    /*   0 */  1024,   820,   655,   526,   423,
    /*   5 */   335,   272,   215,   172,   137,
    /*  10 */   110,    87,    70,    56,    45,
    /*  15 */    36,    29,    23,    18,    15,
};
```

  - **有效时间片** 计算公式：

$$\text{time\_slice}_k = \frac{\text{weight}_k}{\sum_{i=0}^{n-1} \text{weight}_i} \cdot \text{sched\_latency}$$

方法 #3：Linux完全公平调度器（CFS）

- **虚拟运行时间调整**（实际运行时间随着时间的推移不断增加）

$$vruntime_i = vruntime_i + \frac{weight_0}{weight_i} \cdot runtime_i$$

- 示例：
  - 场景：进程A的nice值为-5，进程B的nice值为0。
  - **时间片**
    - 从权重表中得知，进程A的权重为3121，time_slice(A): (3121/4145)*48 = 36ms
    - 从权重表中得知，进程B的权重为1024，time_slice(B): (1024/4145)*48 = 12ms
  - **虚拟运行时间**
    - vruntime(A) = vruntime(A) + (1024/3121)*run_time(A)
    - vruntime(B) = vruntime(B) + (1024/1024)*run_time(B)
    - 因此，进程A的虚拟运行时间将以进程B的三分之一的速度增加，A将有较小的虚拟运行时间，因此优先级更高。

方法 #3：Linux完全公平调度器（CFS）

- 如何高效地（尽可能快速地）找到下一个要运行的进程？

- 红黑树（**Red-Black Trees**）
  - 自平衡的二叉树，相比于链表的线性搜索时间，具有对数级**O(log n)**的搜索时间

  - CFS只将**正在运行/可运行的进程**放入红黑树中

方法 #3：Linux完全公平调度器（CFS）

- 红黑树（RB树）示例
  - 给定：有10个进程，虚拟运行时间分别为1, 5, 9, 10, 14, 18, 17, 21, 22, 24
  - 如果使用排序列表，下一步要运行的进程就是排序列表中的第一个元素
    - 如果插入一个新进程，在最坏的情况下可能需要遍历列表中所有元素
  - 使用红黑树（虚拟运行时间作为Key）提高了效率，大多数操作的时间复杂度是对数级

方法 #3：Linux完全公平调度器（CFS）

- **如何处理I/O和睡眠进程（长时间睡眠）？**
  - •唤醒的睡眠进程可能会垄断CPU，因为它的虚拟运行时间（vruntime）一段时间未更新，因此会较小
  - •解决方案：通过将新唤醒进程的虚拟运行时间设置为红黑树中**最小的虚拟运行时间**，来调整新进程的虚拟运行时间
  - •短时间休眠的进程不会获得公平的CPU份额

- **CFS的其他有趣功能**
  - •提高性能的启发式算法
  - •处理多个CPU
  - •为进程组进行调度

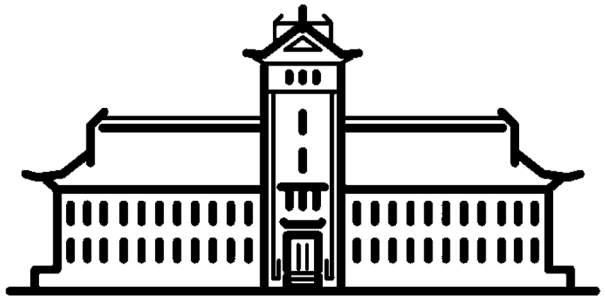
相关的Linux指令

- man sched
- sysctl -A | grep "sched" | grep -v "domain"
- sudo nice --10 ./cpu.elf
- ps -l -p `pidof cpu.elf`
- cat /proc/sched_debug
- cat /proc/schedstat
- cat /proc/`pidof cpu.elf`/sched
- sudo renice -n -20 -p `pidof cpu.elf`
- sudo chrt -r -p 40 `pidof cpu.elf`
- sudo chrt -o -p 0 `pidof cpu.elf`

比例份额调度小结

- 本章介绍了比例份额调度的概念，并简单讨论了三种方法
  - 彩票调度
  - 步长调度
  - Linux完全公平调度

谢谢



南京大學
NANJING UNIVERSITY