Virtualizing the CPU

Multi-Level Feedback Queue & Proportional Share

Liu yufeng

Fx_yfliu@163.com

Hunan University

Multi-level Feedback Queue (MLFQ)

- The Multi-level Feedback Queue (MLFQ) scheduler was first described by **Corbato** et al. in 1962 in a system known as the Compatible Time-Sharing System (**CTSS**).

- This work, along with later work on Multics, led the ACM to award Corbato its highest honor, the **Turing Award**.

- The fundamental problem MLFQ tries to address is two-fold.

- First, it would like to optimize **turnaround time**, such as SJF (or STCF); **unfortunately**, the OS doesn’t generally know how long a job will run for.

- Second, MLFQ would like to minimize **response time**; **unfortunately**, algorithms like Round Robin reduce response time but are terrible for turnaround time.

关键问题：没有完备的知识如何调度？

没有工作长度的先验（priori）知识，如何设计一个能同时减少响应时间和周转时间的调度程序？

提示：从历史中学习

多级反馈队列是用历史经验预测未来的一个典型的例子，操作系统中有很多地方采用了这种技术（同样存在于计算机科学领域的很多其他地方，比如硬件的分支预测及缓存算法）。如果工作有明显的阶段性行为，因此可以预测，那么这种方式会很有效。当然，必须十分小心地使用这种技术，因为它可能出错，让系统做出比一无所知的时候更糟的决定。

MLFQ: Basic Rules

- The MLFQ has a number of distinct **queues**, each assigned a different **priority level**.

- At any given time, a job that is ready to run is **on a single queue**.

- MLFQ uses priorities to decide which job should run at a given time: a job with higher priority (i.e., a job on a higher queue) is chosen to run.

- Of course, more than one job may be on a given queue, and thus have the *same* priority.

- In this case, we will just use round-robin scheduling among those jobs.

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

The key to MLFQ scheduling therefore lies in how the scheduler sets priorities.

- Rather than giving a fixed priority to each job, MLFQ **varies** the priority of a job based on its **observed behavior**.

- If a job repeatedly relinquishes（放弃） the CPU while waiting for input from the keyboard, MLFQ will keep its priority high.

- If a job uses the CPU intensively（集中）, MLFQ will reduce its priority.

- In this way, MLFQ will try to **learn** about processes as they run, and thus use the history of the job to predict its **future** behavior.

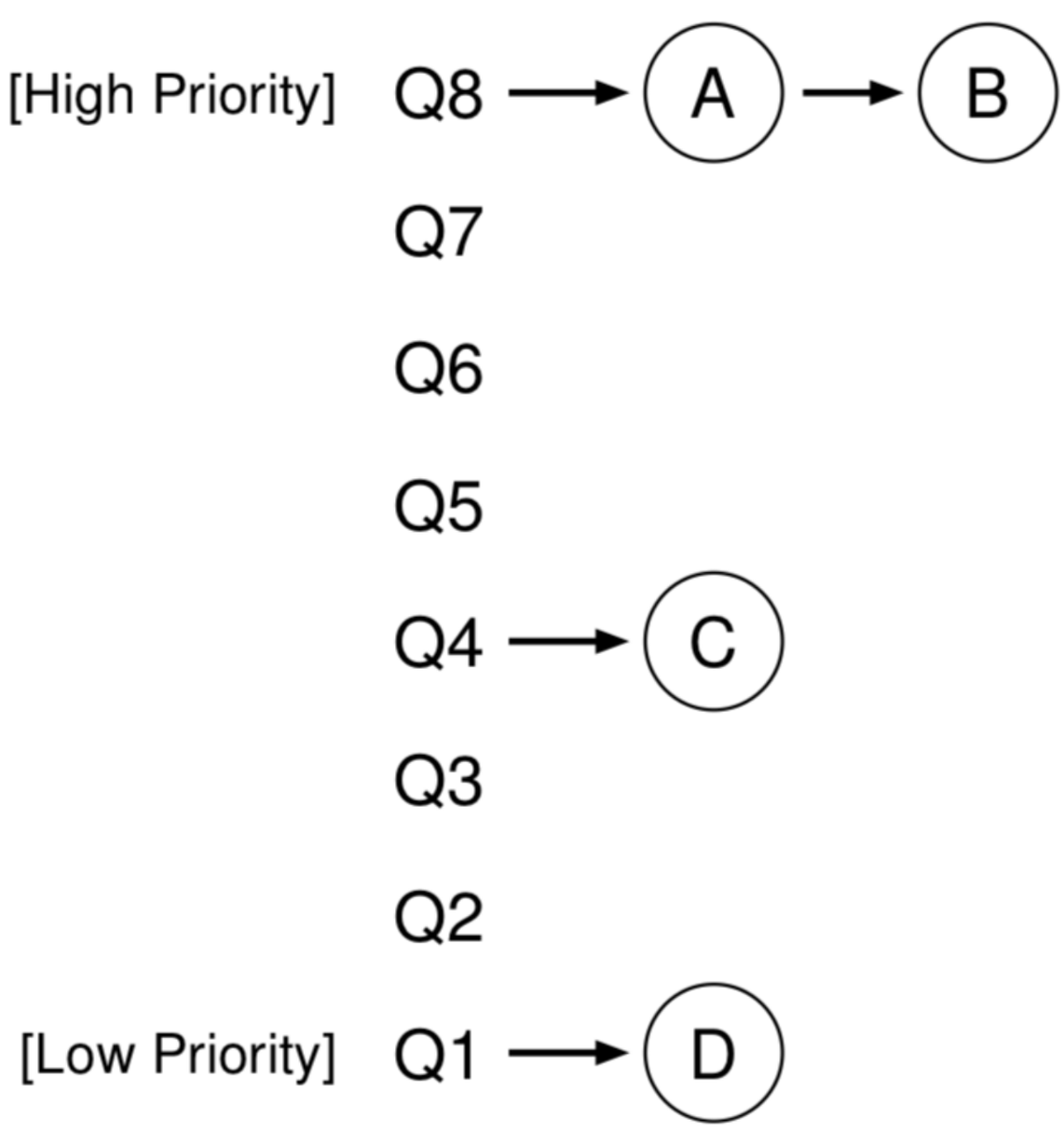


Figure 8.1: **MLFQ Example**

存在什么问题?

饥饿

解决方法：动态改变优先级

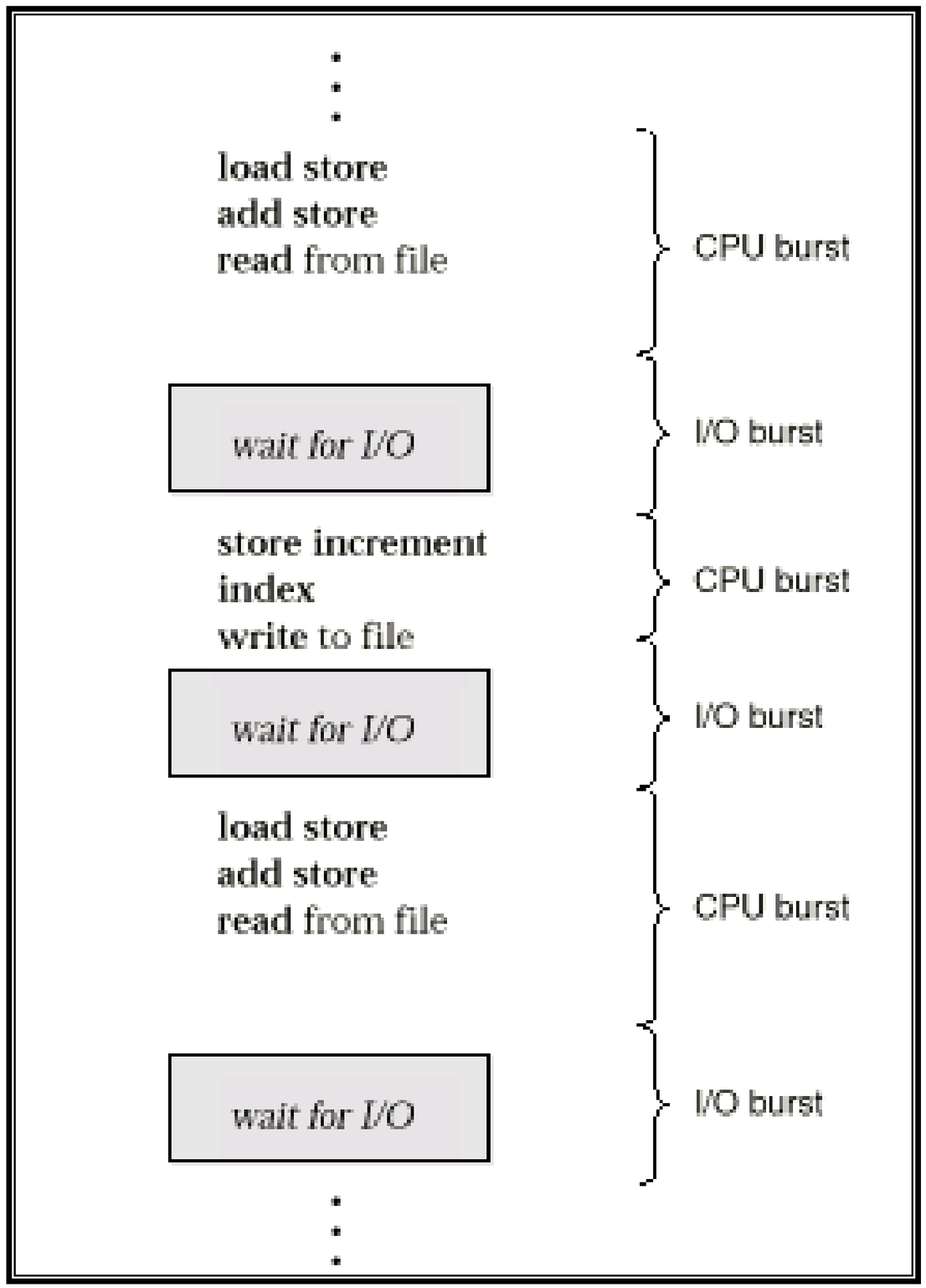
1973 年，工作人员关闭MIT 的IBM 7094 时，他们发现一个在1967 年提交的低优先权进程还没有运行

Attempt #1: How To Change Priority

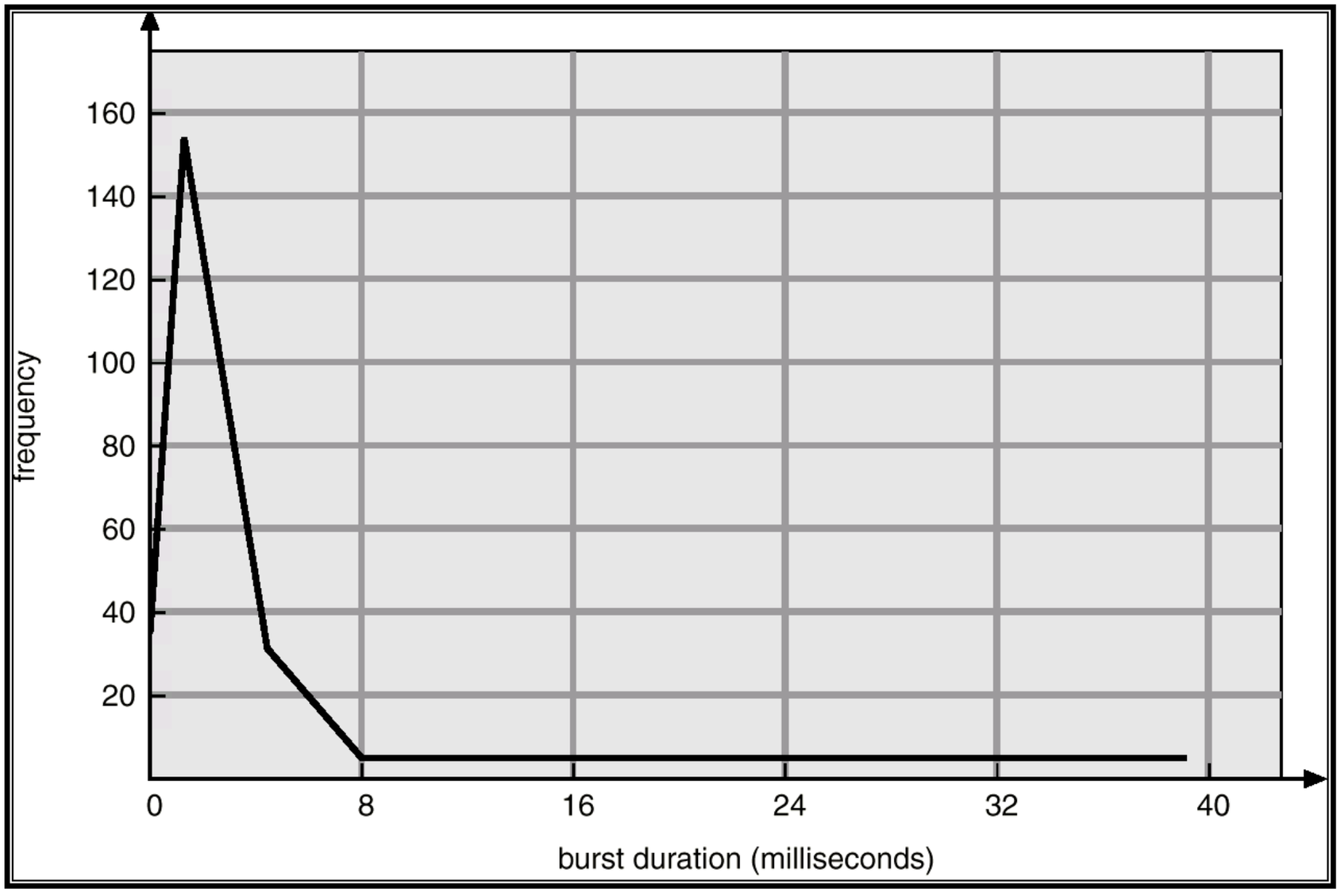
- Keep in mind our workload: a mix of short-running interactive jobs, and some longer-running “CPU-bound” jobs (response time isn’t important).

- **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue).
- **Rule 4a:** If a job uses up an entire time slice while running, its priority is *reduced* (i.e., it moves down one queue).
- **Rule 4b:** If a job gives up the CPU before the time slice is up, it stays at the *same* priority level.

时间片一般多大？


load store
add store
read from file
wait for I/O
store increment
index
write to file
wait for I/O
load store
add store
read from file
wait for I/O
CPU burst
I/O burst
CPU burst
I/O burst
CPU burst
I/O burst

Histogram of CPU-burst Times



时间片的大小通常为 10~100ms

Figure 8.2: **Long-running Job Over Time**

- 该工作首先进入最高优先级（Q2）。执行一个 10ms 的时间片后，调度程序将工作的优先 级减 1，因此进入 Q1。在 Q1 执行一个时间片后，最终降低优先级进入系统的最低优先级（Q0），一直留在那里.

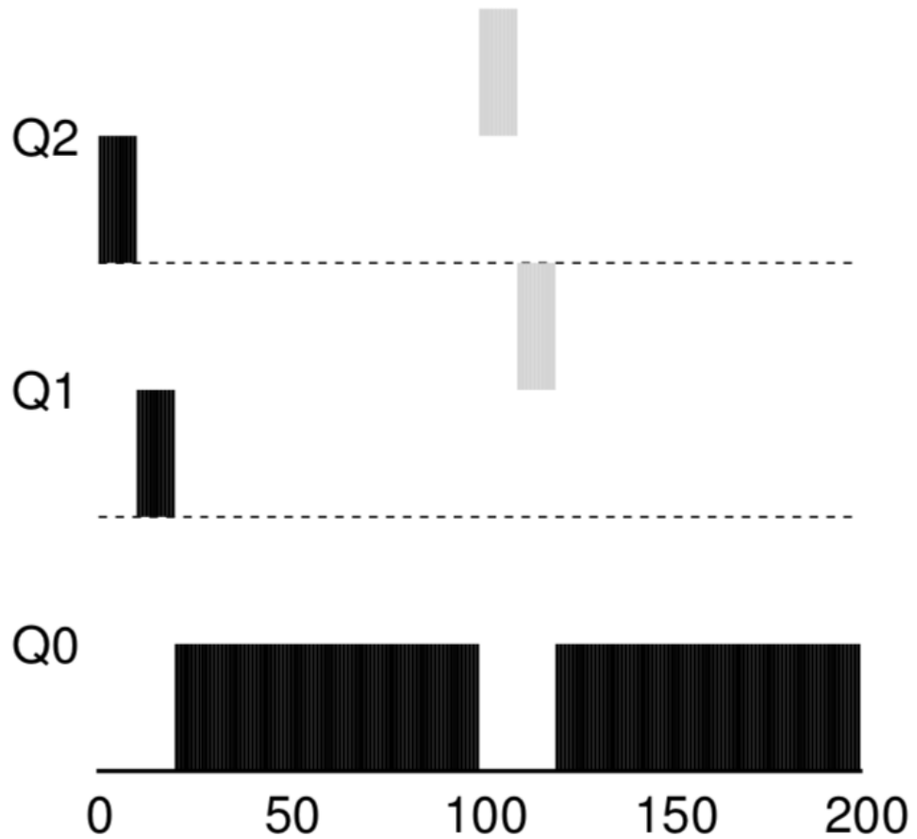


Figure 8.3: **Along Came An Interactive Job**

- A 是一个长时间运行的 CPU 密集型工作，B 是一个运行时间很短的交互型工作。假设 A 执行 一段时间后 B 到达
- A（用黑色表示）在最低优先级队列执行（长时间运行 的 CPU 密集型工作都这样）。B（用灰色表示）在时间 T=100 时到达，并被加入最高优先级 队列
- 这个算法的一个主要目标：如果不知道工作是短工作还是长工作，那么就在开始的时候假设其是短工作，并赋予最高优先级。如果确实是短 工作，则很快会执行完毕，否则将被慢慢移入低优先级队列（例如A），而这时该工作也被认为是长 工作了。通过这种方式，MLFQ 近似于 SJF

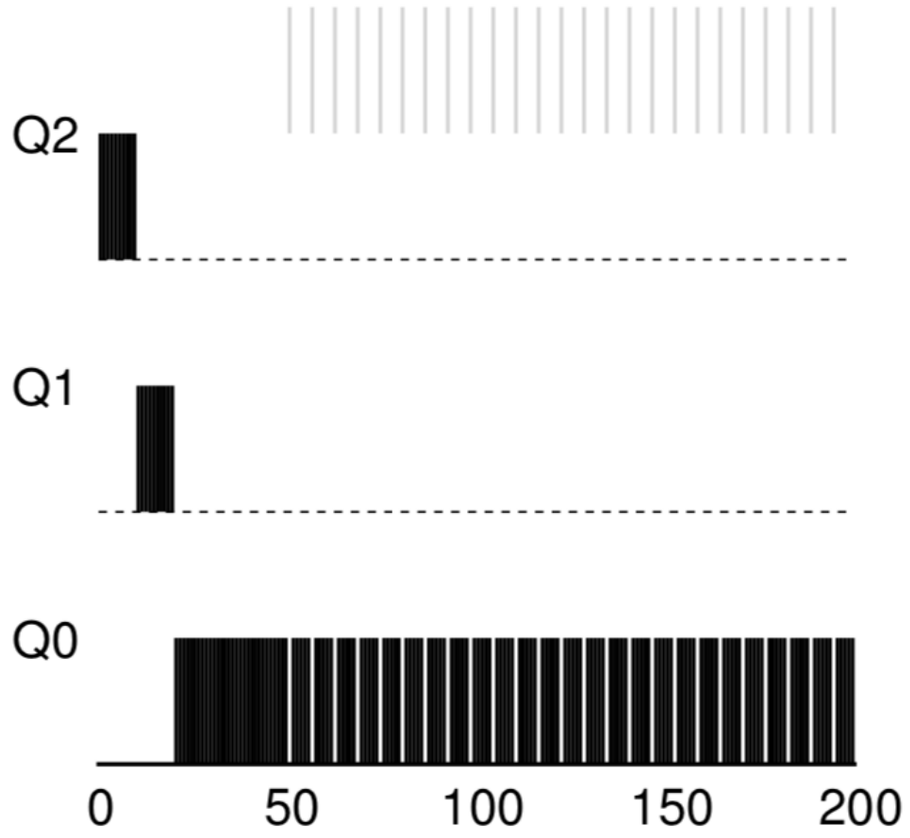


Figure 8.4: **A Mixed I/O-intensive and CPU-intensive Workload**

- 规则 4b，如果进程在时间片用完之前主动放弃 CPU， 则保持它的优先级不变。
- 假设交互型工作中有大量的 I/O 操作（比 如等待用户的键盘或鼠标输入），它会在时间片用完之前放弃 CPU。在这种情况下，我们不 想处罚它，只是保持它的优先级不变
- ，交互型工作 B（用灰色表示）每执行 1ms 便需要进行 I/O 操作，它与长时间运行的工作 A（用黑色表示）竞争 CPU。MLFQ 算法保持 B 在最高优先 级，因为 B 总是让出 CPU。如果 B 是交互型工作，MLFQ 就进一步实现了它的目标，让交 互型工作快速运行

Problems With Our Current MLFQ

- 如果系统有“太多”交互型工作，就会不断占用 CPU，导致长工作永远无法得到 CPU（它们饿死了）。即使在这种情况下，我们希望这些长 工作也能有所进展。

- Second, a smart user could rewrite their program to **game the scheduler**. （进程在时间片用完之前，调用一个 I/O 操作（比如访问一个无关的文件），从而 主动释放 CPU）

- 一个程序可能在不同时间表现不同。一个计算密集的进程可能在某段时间表现为 一个交互型的进程。用我们目前的方法，它不会享受系统中其他交互型工作的待遇。

Attempt #2: The Priority Boost

- What could we do in order to guarantee that CPU-bound jobs will make some progress?

- The simple idea here is to periodically **boost** the priority of all the jobs in system.

- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.

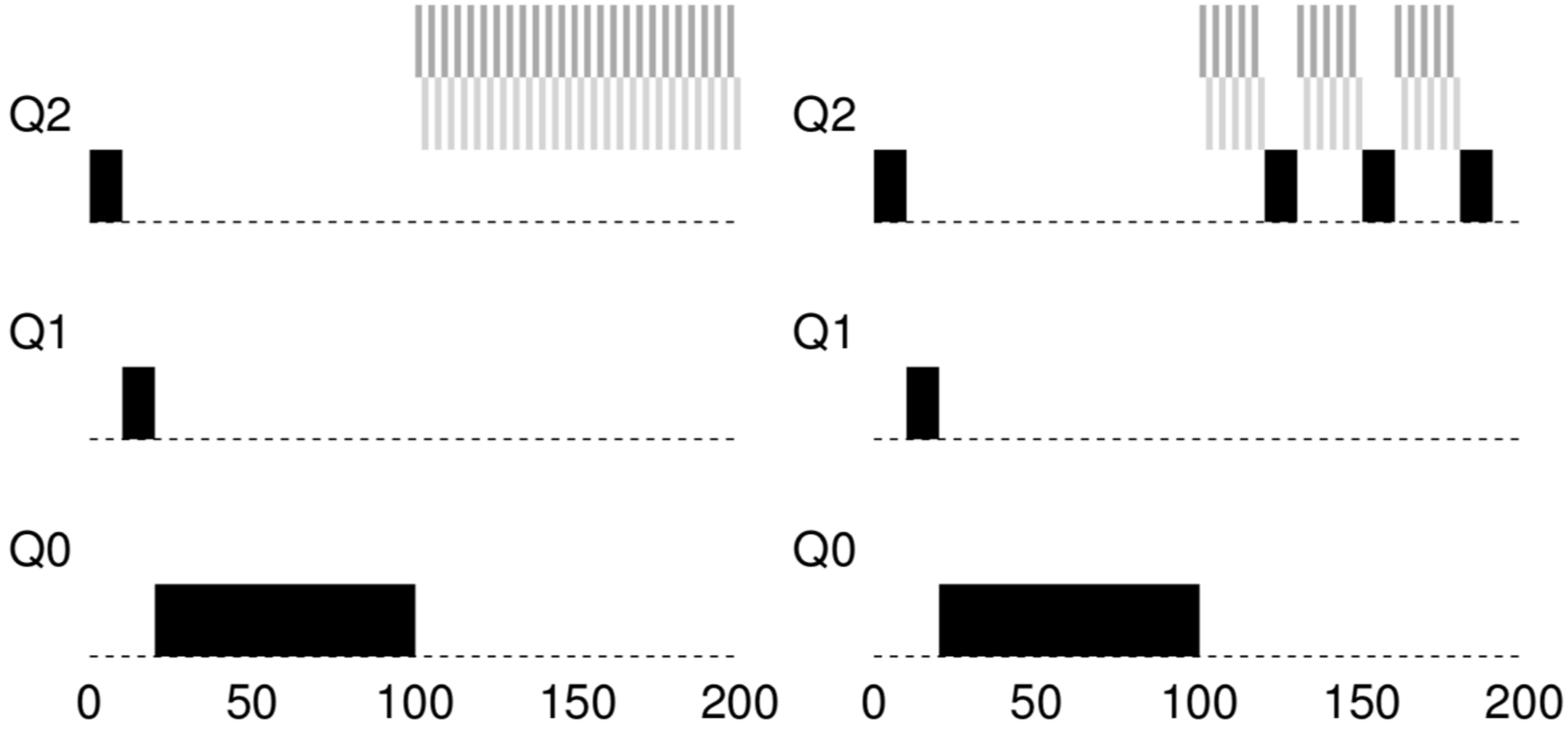


Figure 8.5: **Without (Left) and With (Right) Priority Boost**

- Our new rule solves two problems.
- First, processes are **guaranteed not to starve**: by sitting in the top queue, a job will share the CPU with other high-priority jobs in a round-robin fashion.
- Second, if a CPU-bound job has become interactive, the scheduler treats it properly once it has received the priority boost.

Attempt #3: Better Accounting

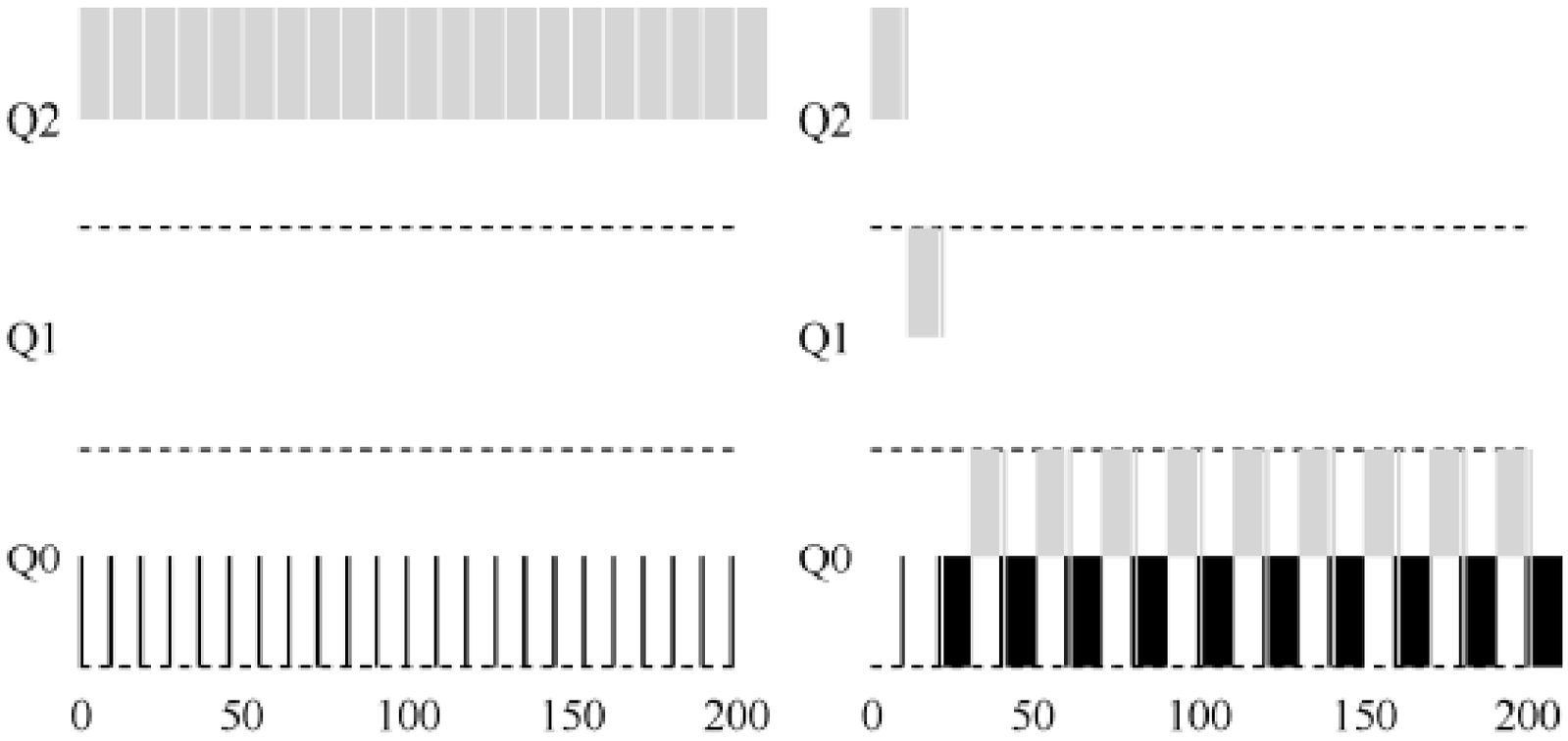


图 8.6　不采用愚弄反制（左）和采用（右）

- How to prevent gaming of our scheduler?
- The solution is to perform better **accounting** of CPU time at each level of the MLFQ.

规则4：调度程序应该记录一个进程在某一层中消耗的总时间，而不是在调度时重新计时。只要进程用完了自己的配额，就将它降到低一优先级的队列中去。不论它是一次用完的，还是拆成很多次用完

Tuning MLFQ And Other Issues

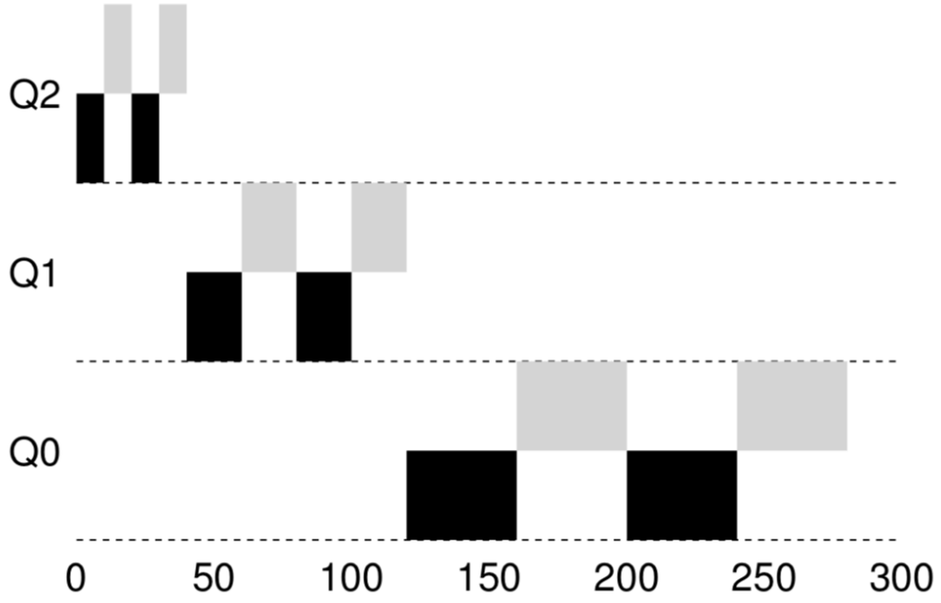


Figure 8.7: **Lower Priority, Longer Quanta**

- How to **parameterize** such a scheduler.
- For example, how many queues should there be? How big should the time slice be per queue? How often should priority be boosted in order to avoid starvation and account for changes in behavior?

这些问题都没有显而易见的答案，因此只有利用对工作负载的经验，以及后续对调度程序的调优，才会导致令人满意的平衡。

大多数的 MLFQ 变体都支持不同队列可变的时间片长度。高优先级队列通常只有较短的时间片(比如 10ms 或者更少)，因而这一层的交互工作可以更快地切换。相反，低优先级队列中更多的是 CPU 密集型工作，配置更长的时间片会取得更好的效果。

Summary

- MLFQ 规则

  规则 1：如果 A 的优先级 > B 的优先级，运行 A（不运行 B）。

  规则 2：如果 A 的优先级 = B 的优先级，轮转运行 A 和 B。

  规则 3：工作进入系统时，放在最高优先级（最上层队列）。

  规则 4：一旦工作用完了其在某一层中的时间配额（无论中间主动放弃了多少次 CPU），就降低其优先级（移入低一级队列）。

  规则 5：经过一段时间 S，就将系统中所有工作重新加入最高优先级队列。

- For this reason, **many systems**, including BSD UNIX derivatives, Solaris, and Windows NT and subsequent Windows operating systems **use a form of MLFQ as their base scheduler**.

Proportional Share

- **Proportional-share** (**fair-share**) scheduler is based on a simple concept:
  - instead of optimizing for turnaround or response time, a scheduler might try to guarantee that **each job obtain a certain percentage of CPU time**.
- An excellent early example of proportional-share scheduling is known as **lottery scheduling**.

Basic Concept: Tickets Represent Your Share

- **Tickets** are used to represent the share of a resource that a process should receive.

- The percent of tickets that a process has represents its share of the system resource.

- Imagine two processes, A and B, and further that A has 75 tickets while B has only 25. Thus, A receives 75% of the CPU and B the remaining 25%.

- Lottery scheduling achieves this **probabilistically**.
- The scheduler must know how many total tickets there are (in our example, there are 100).
- The scheduler then picks a winning ticket, which is a number from 0 to 99. Assuming A holds tickets 0 through 74 and B 75 through 99, the winning ticket simply determines whether A or B runs.

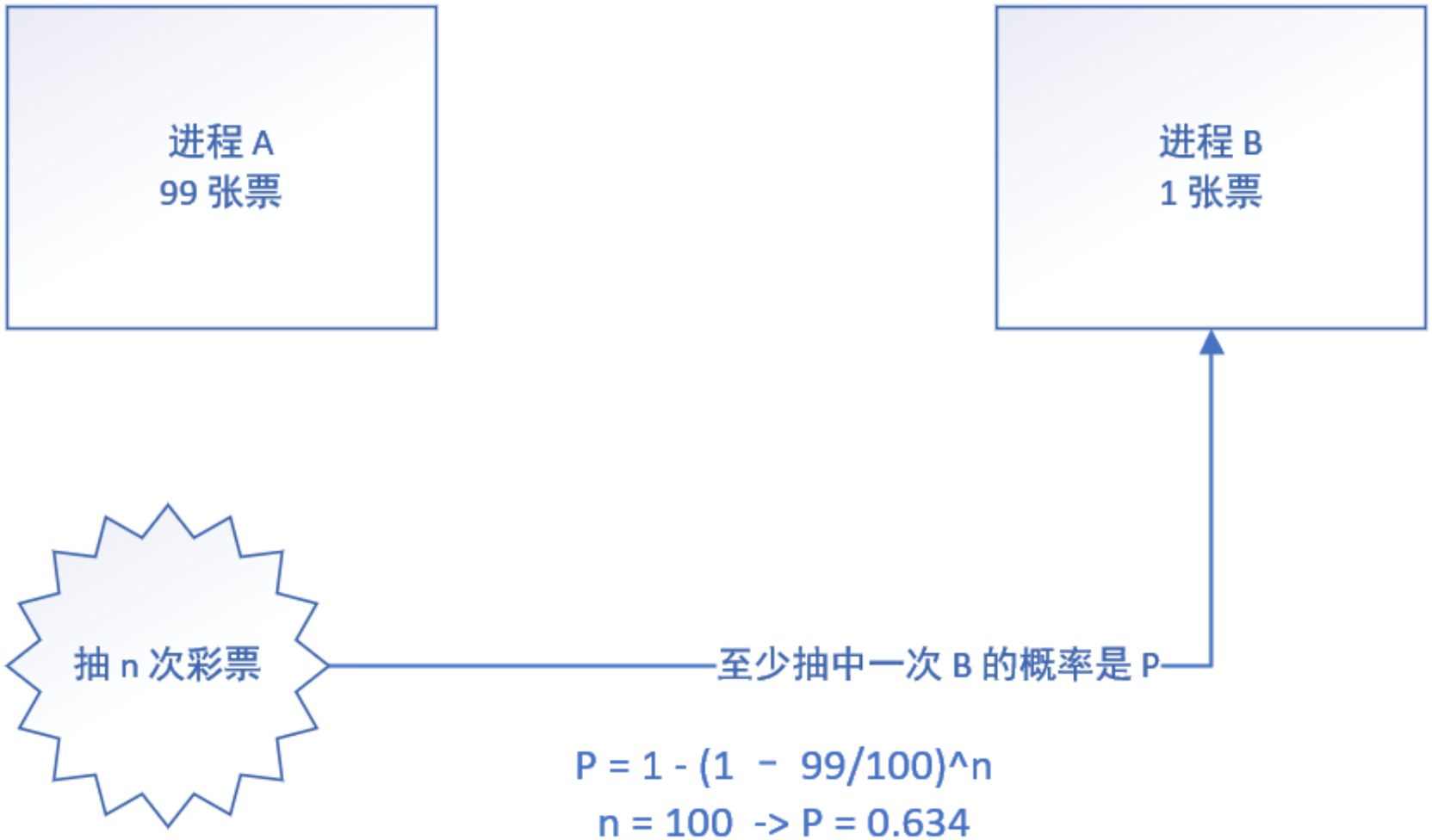
Here is an example output of a lottery scheduler's winning tickets:

```
63 85 70 39 76 17 29 41 36 39 10 99 68 83 63 62 43  0 49
```

Here is the resulting schedule:

```
A     A  A     A  A  A  A  A  A     A     A  A  A  A  A
   B        B                    B     B
```

- In our example above, B only gets to run 4 out of 20 time slices (20%), instead of the desired 25% allocation.
- However, the longer these two jobs compete, the more likely they are to achieve the desired percentages.



$$P = 1 - (1 - 99/100)^n$$

n = 100 -> P = 0.634

n = 200 -> P = 0.866

...

随着 n 增大，B 被至少选中一次的概率也越来越大，杜绝了饥饿

彩票机制

- 彩票货币（ticket currency）：加入一个抽象层，可以让任务组自由修改组内的份额

- 假设用户 A 和用户 B 每人拥有 100 张彩票。用户 A 有两个工作 A1 和 A2，他以自己的货币，给每个工作 500 张彩票（共 1000 张）。用户 B 只运行一个工作，给它 10 张彩 票（总共 10 张）。操作系统将进行兑换，将 A1 和 A2 拥有的 A 的货币 500 张，兑换成全局 货币 50 张。类似地，兑换给 B1 的 10 张彩票兑换成 100 张。然后会对全局彩票货币（共 200 张）举行抽奖，决定哪个工作运行。

```
User A -> 500 (A's currency) to A1 -> 50 (global currency)
       -> 500 (A's currency) to A2 -> 50 (global currency)
User B -> 10 (B's currency) to B1 -> 100 (global currency)
```

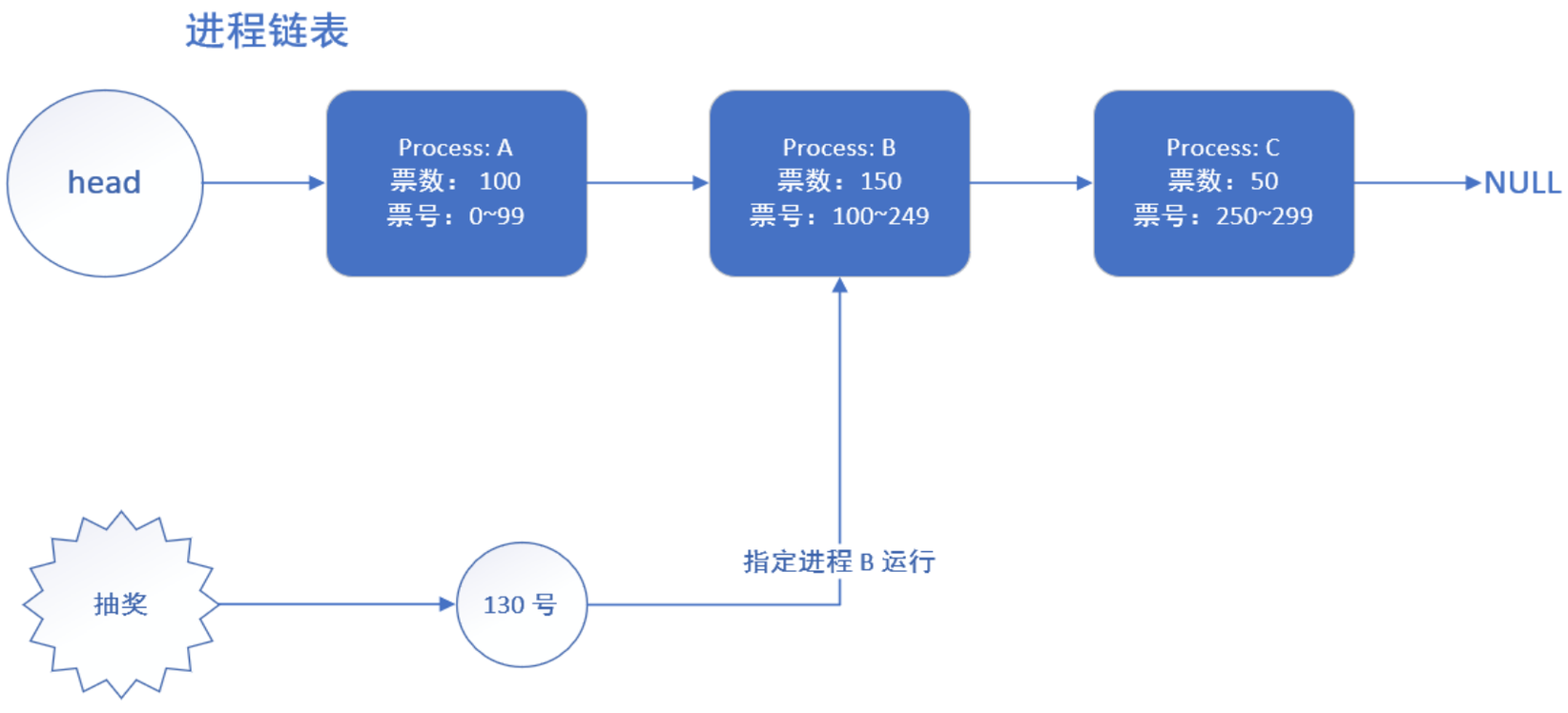
彩票机制

- 彩票转让（ticket transfer）：一个进程可以临时将自己 的彩票交给另一个进程

- 彩票通胀（ticket inflation）：利用通胀，一个进程可以临时提升或 降低自己拥有的彩票数量。

Implementation

- Assume we keep the processes in a list. Here is an example comprised of three processes, A, B, and C, each with some number of tickets.
- To make a scheduling decision, we first have to pick a **random number** (the winner) from the total number of tickets (**400**). Let’s say we pick the number **300**.
- Then, we simply traverse the list, with a simple counter used to help us find the winner.



```
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 while (current) {
11      counter = counter + current->tickets;
12      if (counter > winner)
13          break; // found the winner
14      current = current->next;
15 }
16 // 'current' is the winner: schedule it...
```

Figure 9.1: **Lottery Scheduling Decision Code**

Lottery Fairness

- 为了更好地理解彩票调度的运行过程，现在简单研究一下两个互相竞争任务的完成时间，每个任务都有相同数目的 100 张彩票，以及相同的运行时间 R。
- 希望两个任务在大约同时完成，但由于彩票调度算法的随机性，有时一个任务会先于另一个完成。
- 为了量化这种区别，我们定义了一个简单的指标 U (**unfairness metric**)，将两个任务完成时刻相除得到 U 的值。比如，运行时间 R 为 10，第一个任务在时刻 10 完成，另一个在 20，U=10/20=0.5。如果两个任务几乎同时完成，U 的值将接近于 1。
- 我们的目标是: 完美的公平调度程序可以做到 U=1。

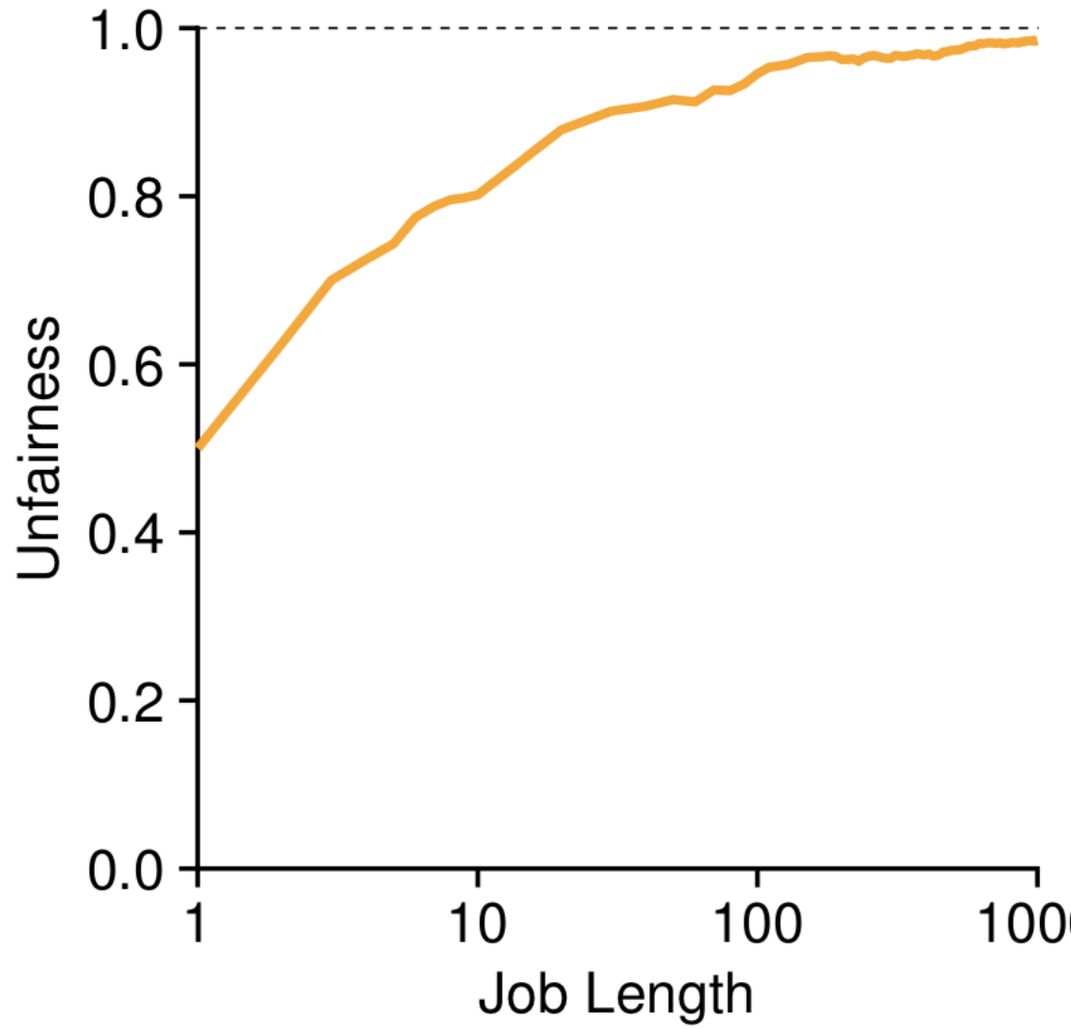


Figure 9.2: **Lottery Fairness Study**

- 图 9.2 展示了当两个任务的运行时间从 1 到1000 变化时，30 次试验的平均 U 值。

- 可以看出，当任务执行时间很短时，平均不公平度非常糟糕。只有当任务执行非常多的时间片时，彩票调度算法才能得到期望的结果。

Why Not Deterministic?

- Why use randomness at all?

- Randomness gets us a simple scheduler, it occasionally will not deliver the exact right proportions, especially over short time scales.

- For this reason, Waldspurger invented **stride scheduling**（步长调度）, a **deterministic** fair-share scheduler.

- 系统中的每个工作都有自己的步长，这个值与票数成反比
- A、B、C三个进程的票数分别为100、50、250，我们用一个大数除以每个工作的票数，就可以得到每个工作的步长。比如用10000作为大数，则A、B、C的步长分别为100，200，40

```
curr = remove_min(queue);    // pick client with min pass
schedule(curr);              // run for quantum
curr->pass += curr->stride; // update pass using stride
insert(queue, curr);         // return curr to queue
```

- The scheduler uses the **stride（步长）** and **pass（总体进展）** to determine which process should run next.
- 步长调度的基本思路是：每次需要进行调度时，选择目前拥有最小行程值的进程，并且在运行之后将该进程的行程值增加一个步长。

Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	Who Runs?
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	...

Figure 9.3: **Stride Scheduling: A Trace**

- 初始行程值都为 0。因此，最初，所有进程都可能被选择执行。假设选择 A（任意的，所有具有同样低 的行程值的进程，都可能被选中）。A 执行一个时间片后，更新它的行程值为 100。然后运 行 B，并更新其行程值为 200。最后执行 C，C 的行程值变为 40。这时，算法选择最小的行 程值，是 C，执行并增加为 80（C 的步长是 40）。然后 C 再次运行（依然行程值最小），行程值增加到 120。现在运行 A，更新它的行程值为 200（现在与 B 相同）。然后 C 再次连续 运行两次，行程值也变为 200。

- C运行了5次，A运行了2次，B运行了1次，正好是票数的比例——250、100、50

Why use lottery scheduling at all?

- 彩票调度有一个步长调度没有的优势——不需要全局状态.
- 假如一个新的进程在执行过程中加入系统，应该怎么设置它的行程值呢？按步长算法，初始值设置成0？这样的话，在追赶上其他进程的行程值之前，它会一直独占 CPU。
- 彩票调度算法不需要对每个进程记录**全局状态**，只需要用新进程的票数更新全局的总票数就可以了。因此彩票调度算法能够更合理地处理新加入的进程。

彩票调度

- 缺点：票数分配问题没有确定的解决方式，因此实际中很少应用
- 特定领域例如容易确定份额比例的领域里，比例份额调度程序就可能更有用——比如在虚拟数据中心，你可能希望 1/4 的 CPU 时间给 Windows 虚拟机，剩下的时间给 Linux 虚拟机
- 视频服务器，在该视频服务器上若干进程正在向其客户提供视频流，每个视频流的帧速率都不相同。假设这些进程需要的帧速率分别是10、20和25帧/秒。如果给这些进程分别分配10、20和25张彩票，那么它们会自动地按照大致正确的比例（即10：20：25）划分CPU的使用

[WW94] “Lottery Scheduling: Flexible Proportional-Share Resource Management” Carl A. Waldspurger and William E. Weihl

OSDI ’94, November 1994

关于彩票调度的里程碑式的论文，让调度、公平分享和简单随机算法的力量在操作系统社区重新焕发了活力。

[W95] “Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management” Carl A. Waldspurger Ph.D. Thesis, MIT, 1995

Waldspurger 的获奖论文，概述了彩票和步长调度。如果你想写一篇博士论文，总应该有一个很好的例子，让你有个努力的方向：这是一个很好的例子。

End