调度

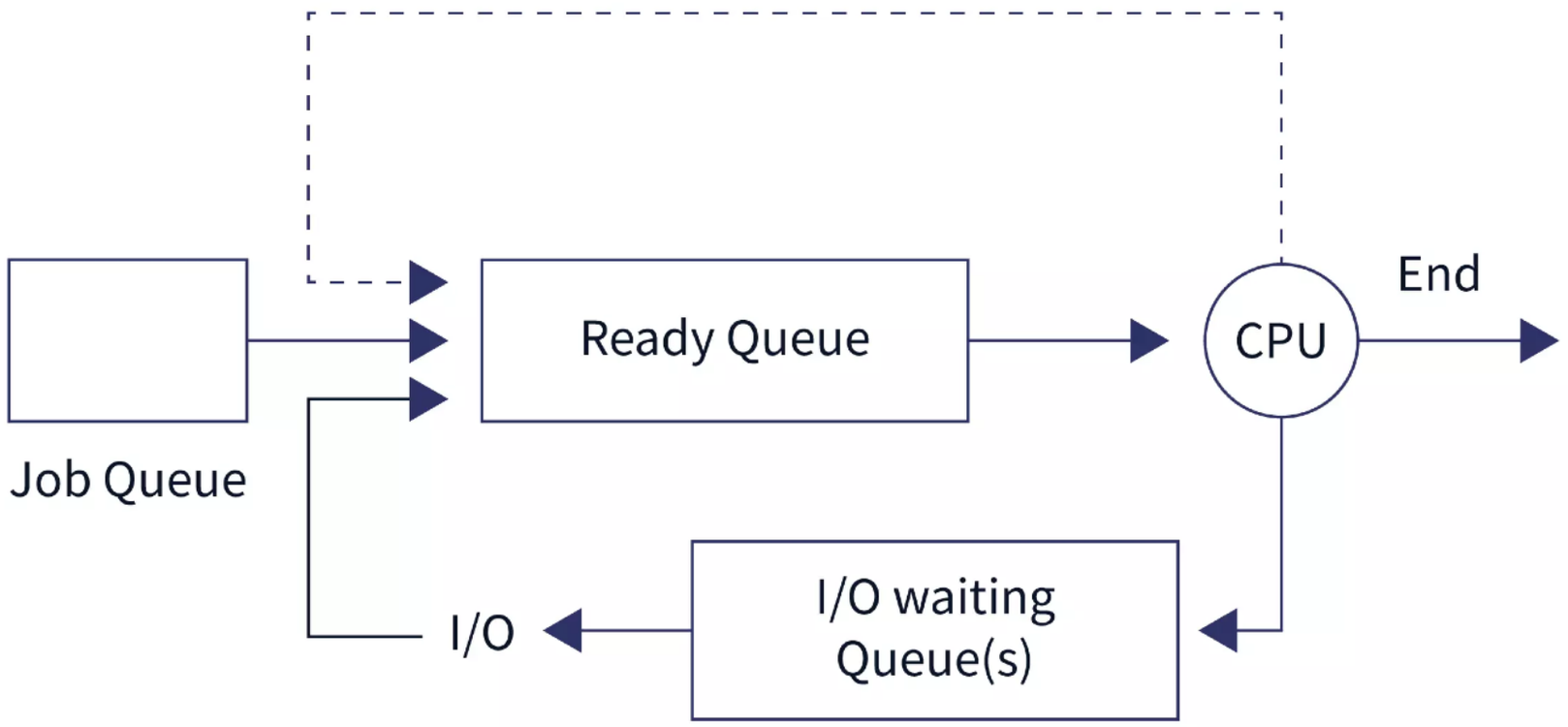
Section 2: Part II

2026

调度

通过对计算任务进行资源的合理分配以满足系统的既定目标

- 根据系统实现，调度对象可以是进程或线程 (这里统一用进程代表)
  - Job Scheduling: 决定将哪个待执行进程载入内存
  - CPU Scheduling: 决定将哪个进程放到 CPU 上执行



调度

通过对计算任务进行资源的合理分配以满足系统的既定目标

- 根据调度的时机，调度策略可分为可抢占和不可抢占
  - Non-preemptive: 处于运行状态的进程总是持续运行，直到自愿让出 CPU 为止 (调用 exit, yield 或在某事件上阻塞)
  - Preemptive: 当前正在运行的进程可被打断并变为 Ready 状态，使得其它等待队列里的进程能调度运行 (在系统调用或中断时)

调度目标

如何评价一个调度策略的优劣?

- 吞吐量 (throughput): 单位时间完成执行的进程数
- 周转时间 (turnaround time): 从进程创建到完成的时间间隔 (*burst time + waiting time*)

Batch System

- 响应时间 (response time): 从进程创建到第一次获得执行的时间间隔
- 等待时间 (waiting time): 进程花在等待 CPU 上的时间
- 可预测性 (predictability): 多次重复执行的表现差异

Interactive System

- 截止时间 (deadlines): 是否能在截止时间前执行完成

Real-Time System

调度策略

针对批处理系统 (batch system) 的调度策略

- First-Come First-Served (FCFS)
- Shortest Job First (SJF)
- Shortest Time-to-Completion First (STCF)

First-Come First-Served (FCFS)

先来先服务: 按照进程到达系统的先后顺序进行不可抢占式调度

Jobs	Burst Time
P1	3
P2	3
P3	24

Average turnaround time = (3+6+30)/3 = 13

Average response time = (0+3+6)/3 = 3

First-Come First-Served (FCFS)

先来先服务: 按照进程到达系统的先后顺序进行不可抢占式调度

Jobs	Burst Time
P1	3
P2	3
P3	24

Average turnaround time = (30+27+24)/3 = 27

Average response time = (27+24+0)/3 = 17

First-Come First-Served (FCFS)

先来先服务: 按照进程到达系统的先后顺序进行不可抢占式调度

- FCFS 对进程到达系统的顺序敏感
  - 护航效应 (convoy effect): 长作业会推迟后续短作业的执行
  - 如果长作业先到达，系统的平均周转时间和响应时间会明显增加
- FCFS 是否会导致饿死 (starvation)?
  - 总是存在一直得不到调度执行的进程?

Shortest Job First (SJF)

最短任务优先: 优先调度运行时间短的进程

Jobs	Burst Time
P1	3
P2	6
P3	24
P4	1

Average turnaround time = (4+10+34+1)/4 = 12.25

Average response time = (1+4+10+0)/4 = 3.75

Shortest Job First (SJF)

最短任务优先: 优先调度运行时间短的进程

- 如果所有进程同时达到系统，SJF 能获得最优的平均周转时间
  - 假设存在一个不是 SJF，但在平均周转时间上最优的替代策略
  - 因为该策略不是 SJF，所以在某个时间点它会选择一个并非当前运行时间最短的进程来调度执行
- 此时，如果我们选择优先调度当前运行时间最短的进程，则可以减少整个系统的平均周转时间
- 因此，不是 SJF 的替代策略不会是最优策略

Shortest Job First (SJF)

最短任务优先: 优先调度运行时间短的进程

- SJF 是否会受护航效应 (convoy effect) 的影响?
  - 任何不可抢占式 (non-preemptive) 调度策略都会
- SJF 是否会导致饿死 (starvation)?
  - 任何总是偏好 (favour) 某种特定属性的调度策略都会

Shortest Time-to-Completion First

最短剩余时间优先: 始终选择当前剩余执行时间最短的进程

- SJF 的可抢占版本

Jobs	Burst Time	Arrive Time
P1	3	10
P2	6	1
P3	24	0
P4	16	18

优化周转时间

SJF & STCF 的目的都是优化系统的周转时间 (turnaround time)

⚠️ 但会导致饿死：优先调度短作业 ➔ 尽可能推迟长作业执行

⚠️ 更困难的是，需要提前知道每个进程所需的“执行时间”

- 让用户在创建进程时告诉操作系统?
- 基于某种历史数据估算?

调度策略

针对交互式系统 (interactive system) 的调度策略

- Round Robin (RR)
- Multi-Level Feedback Queue (MLFQ)
- Proportional Fair Sharing (Lottery and Stride Scheduling)
- Completely Fair Scheduler (CFS)

Round Robin (RR)

时间片轮转: 每个进程分配一个固定时间片 (time slice / quantum)
运行，时间片结束后切换到下一个进程 (基于时钟中断的抢占式调度)

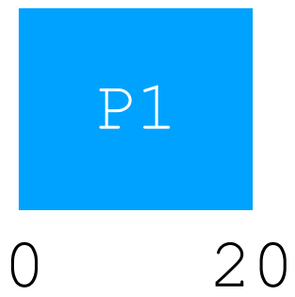
```
Time Slice = 20

Jobs        Burst Time

 P1              53

 P2               8

 P3              68

 P4              24
```

Round Robin (RR)

时间片轮转: 每个进程分配一个固定时间片 (time slice / quantum)
运行，时间片结束后切换到下一个进程 (基于时钟中断的抢占式调度)

```
Time Slice = 20

Jobs        Burst Time

 P1              53 -> 33
 P2              8
 P3              68
 P4              24
```

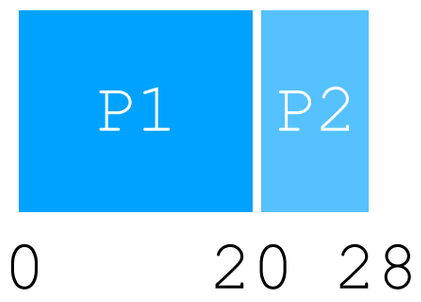
The job is preempted if it has not completed



Round Robin (RR)

时间片轮转: 每个进程分配一个固定时间片 (time slice / quantum)
运行，时间片结束后切换到下一个进程 (基于时钟中断的抢占式调度)

```
Time Slice = 20

Jobs        Burst Time
 P1              33
 P2              8 -> 0
 P3              68
 P4              24
```

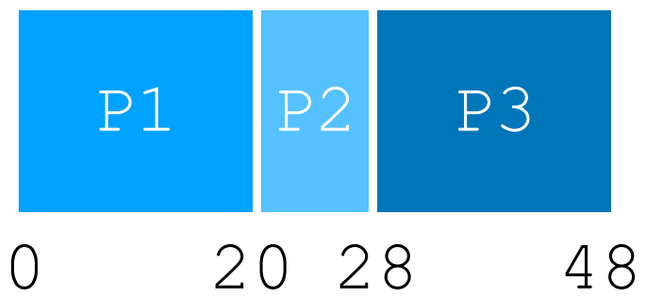


Directly switch if the job finishes early

Round Robin (RR)

时间片轮转: 每个进程分配一个固定时间片 (time slice / quantum)
运行，时间片结束后切换到下一个进程 (基于时钟中断的抢占式调度)

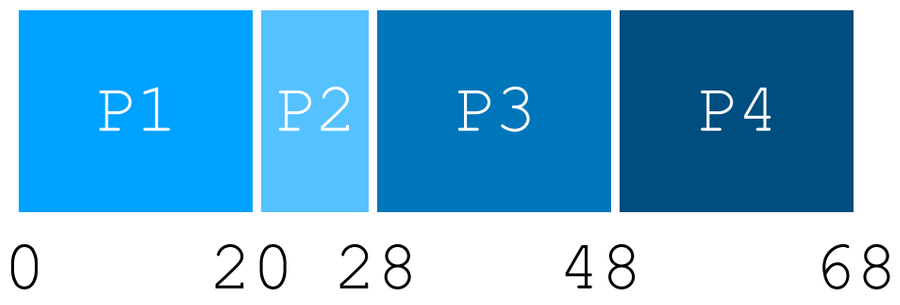
```
Time Slice = 20

Jobs        Burst Time

 P1              33
 P2              0
 P3              68 -> 48
 P4              24
```



Round Robin (RR)

时间片轮转: 每个进程分配一个固定时间片 (time slice / quantum)
运行，时间片结束后切换到下一个进程 (基于时钟中断的抢占式调度)

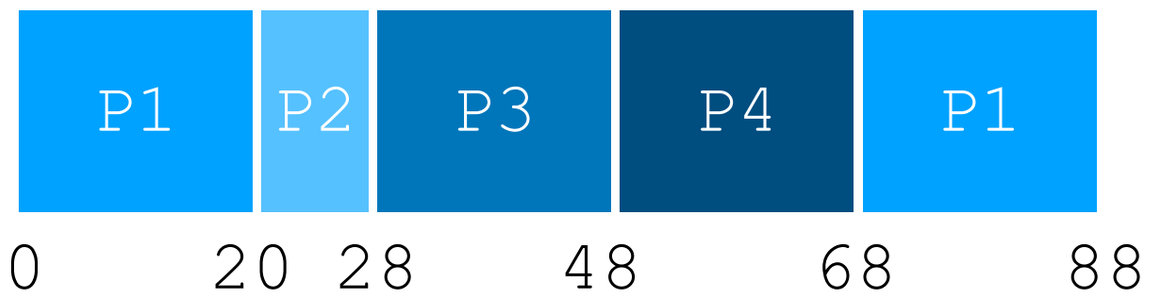
```
Time Slice = 20

Jobs        Burst Time

 P1              33
 P2              0
 P3              48
 P4              24 -> 4
```



Round Robin (RR)

时间片轮转: 每个进程分配一个固定时间片 (time slice / quantum)
运行，时间片结束后切换到下一个进程 (基于时钟中断的抢占式调度)

```
Time Slice = 20

Jobs        Burst Time

 P1              33 -> 13
 P2              0
 P3              48
 P4              4
```



Round Robin (RR)

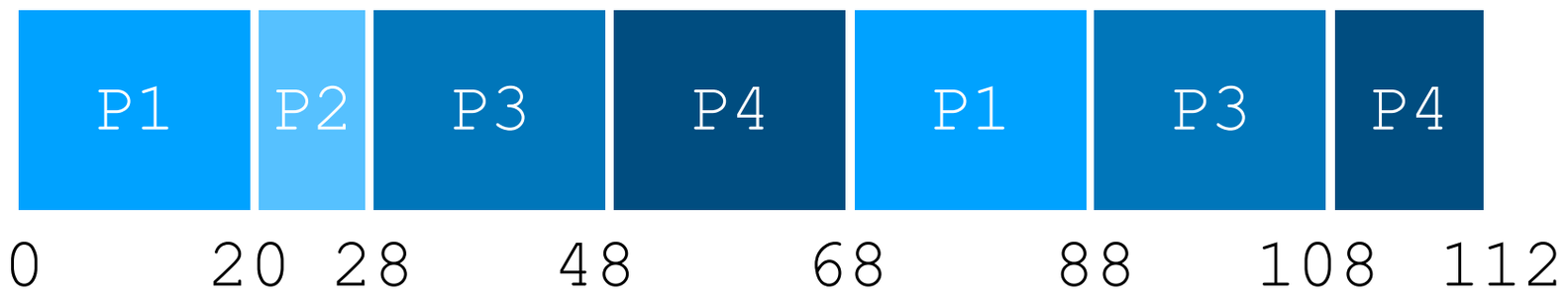
时间片轮转: 每个进程分配一个固定时间片 (time slice / quantum)
运行，时间片结束后切换到下一个进程 (基于时钟中断的抢占式调度)

```
Time Slice = 20

Jobs        Burst Time

 P1              13
 P2              0
 P3              48 -> 28
 P4              4
```

Round Robin (RR)

时间片轮转: 每个进程分配一个固定时间片 (time slice / quantum)
运行，时间片结束后切换到下一个进程 (基于时钟中断的抢占式调度)

```
Time Slice = 20

Jobs        Burst Time

 P1              13
 P2              0
 P3              28
 P4              4 -> 0
```



Round Robin (RR)

时间片轮转: 每个进程分配一个固定时间片 (time slice / quantum)
运行，时间片结束后切换到下一个进程 (基于时钟中断的抢占式调度)

Average turnaround time = (125+28+153+112)/4 = 104.25

Average waiting time = (72+20+85+88)/4 = 66.25

- P1 = 0+(68-20)+(112-88) = 72
- P2 = (20-0) = 20
- P3 = (28-0)+(88-48)+(125-108)+0 = 85
- P4 = (48-0)+(108-68)= 88

Round Robin (RR)

时间片轮转: 每个进程分配一个固定时间片 (time slice / quantum)
运行，时间片结束后切换到下一个进程 (基于时钟中断的抢占式调度)

- RR 是否会受护航效应 (convoy effect) 的影响?
  - 每个进程每次最多运行一个时间片长度
- RR 是否会导致饿死 (starvation)?
  - 对于 N 个进程，每个进程最多等待 $N - 1$ 个时间片

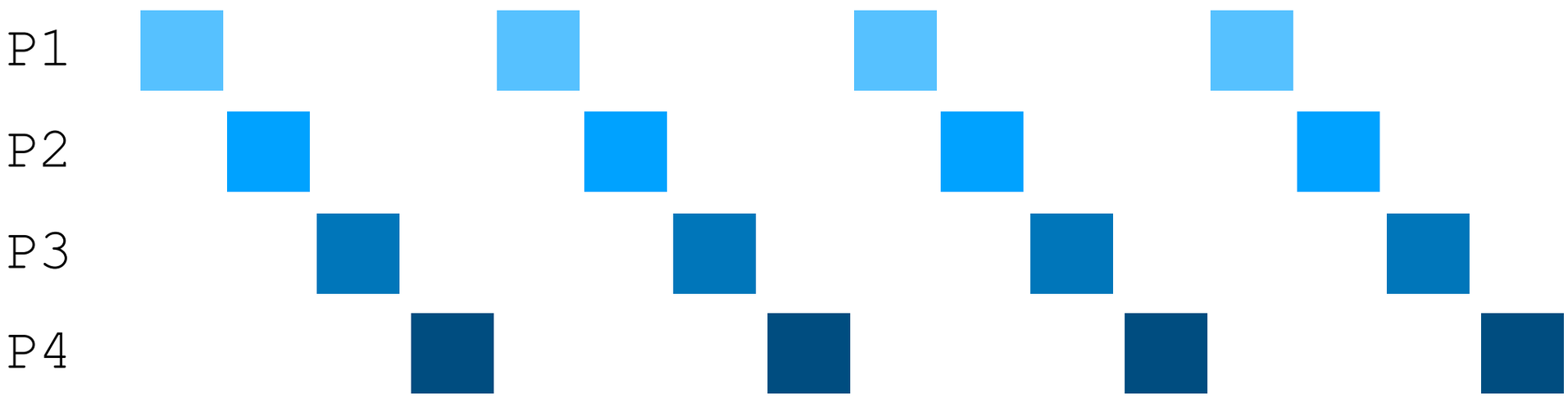
Round Robin (RR)

⚠️ 但 RR 中存在不可忽视的进程间 context switch 开销

- RR 设计的一个核心在于时间片的长度 (length of time slice)
  - 如果太短，大量 CPU 时间将花费在进程 context switch，而不是进程计算任务的执行
  - 如果太长，RR 就变成了 FCFS
- 需要让时间片长度足够 "补偿" 进程间切换的成本
  - 如果 time slice = 10 ms，switch cost = 1 ms，则有 10% 的 CPU 时间用于 switch 而浪费了
  - 如果 time slice = 100 ms，则少于 1% 的时间用于 switch

Round Robin (RR)

⚠️ 当多个执行时间相同的进程同时到达时，RR 会导致较长的平均周转时间 (并且还有较高的 switch 开销)



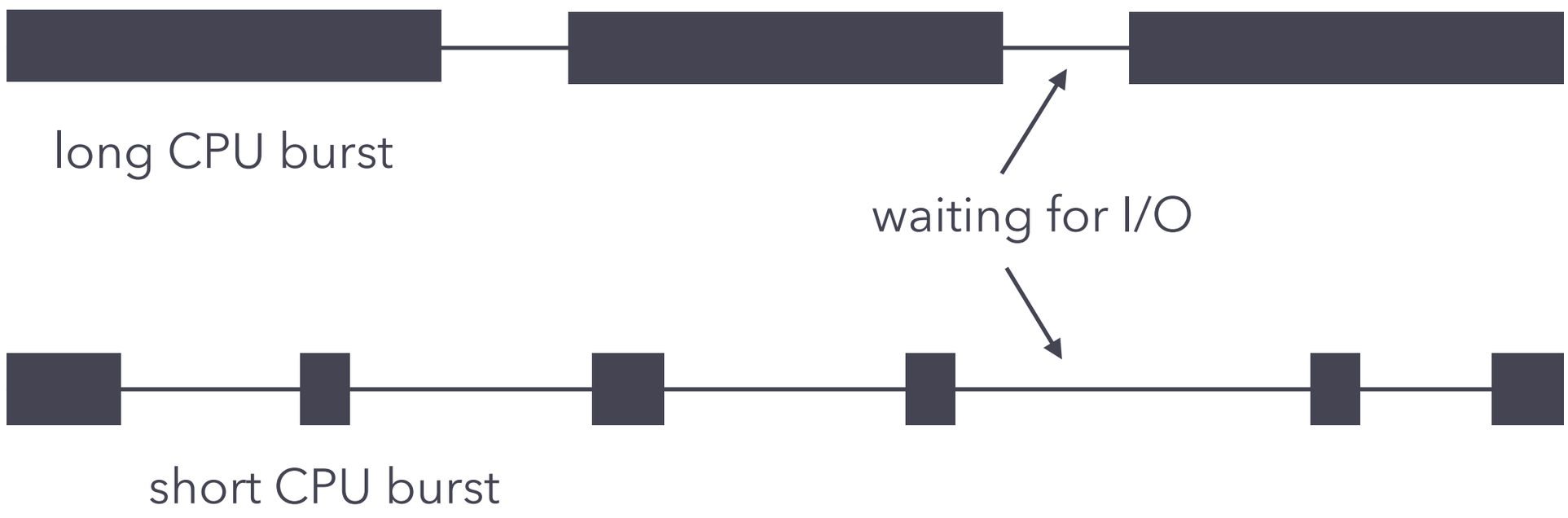
Round Robin

FIFO / SJF

Round Robin (RR)

⚠️ 当系统中同时存在 CPU-bound 和 I/O-bound 进程时，RR 也可能会导致较长的响应时间

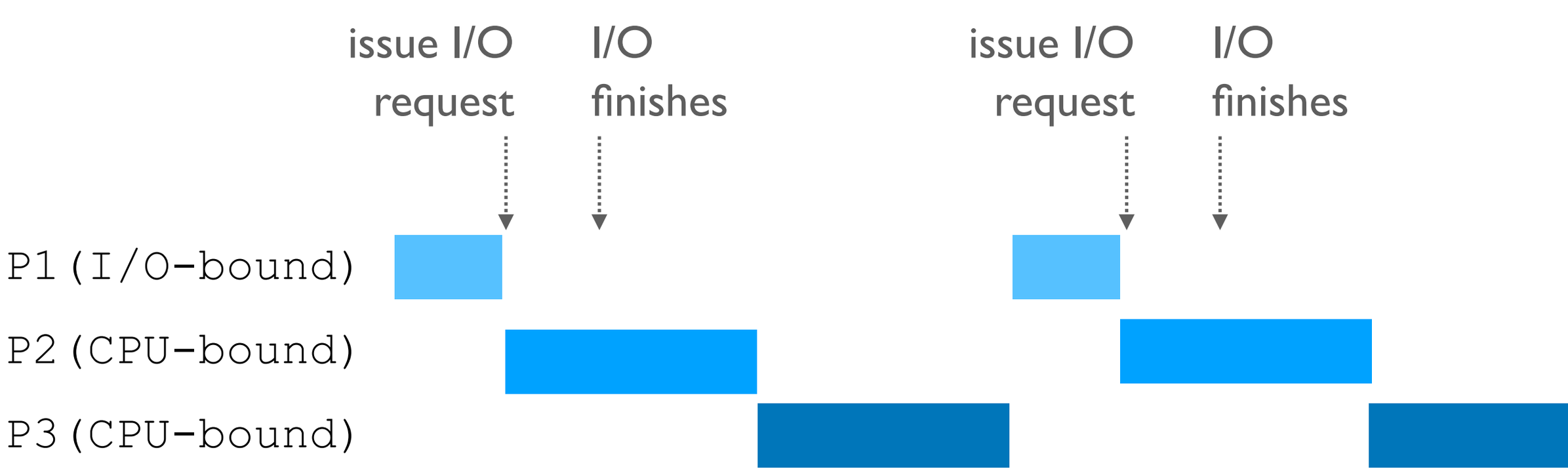
- CPU-bound: 执行主要依赖 CPU，不需要经常等待外部资源
- I/O-bound: 大部分时间都在等待 I/O (也就是需要和外界频繁交互)



Round Robin (RR)

⚠️ 当系统中同时存在 CPU-bound 和 I/O-bound 进程时，RR 也可能会导致较长的响应时间

- 即使 I/O 很快完成 (例如在编辑器中打字)，I/O-bound 进程仍需要等待其它 CPU-bound 进程用完他们的时间片后才会被调度执行



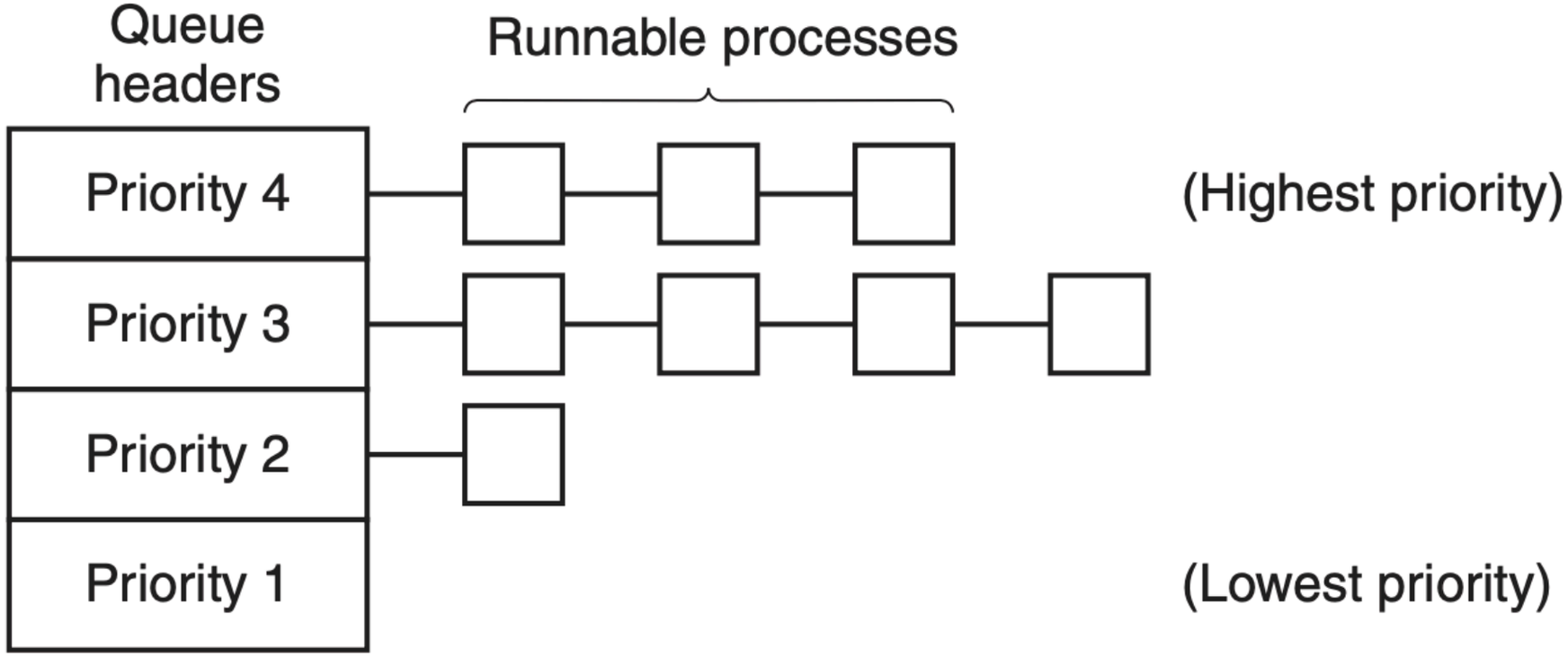
优化周转时间 vs. 响应时间

- 优化进程的周转时间 (for batch jobs)?
  - 谁执行时间短谁优先 (SJF/STCF)
  - 但 OS 很难精确知道不同进程的执行时间
- 优化进程的响应时间 (responsive to interactive users)?
  - 尽可能在不同进程间轮转 (RR)
  - 但可能导致较长的周转时间
- OS 在调度进程运行前并不知道其是 CPU-bound 还是 I/O-bound, 并且进程可能会在运行过程中改变其行为

优先级调度

为不同进程分配不同的优先级 (放入不同的 priority queue)，然后总是优先调度优先级更高的进程 (同一队列中的进程可用 RR 调度)

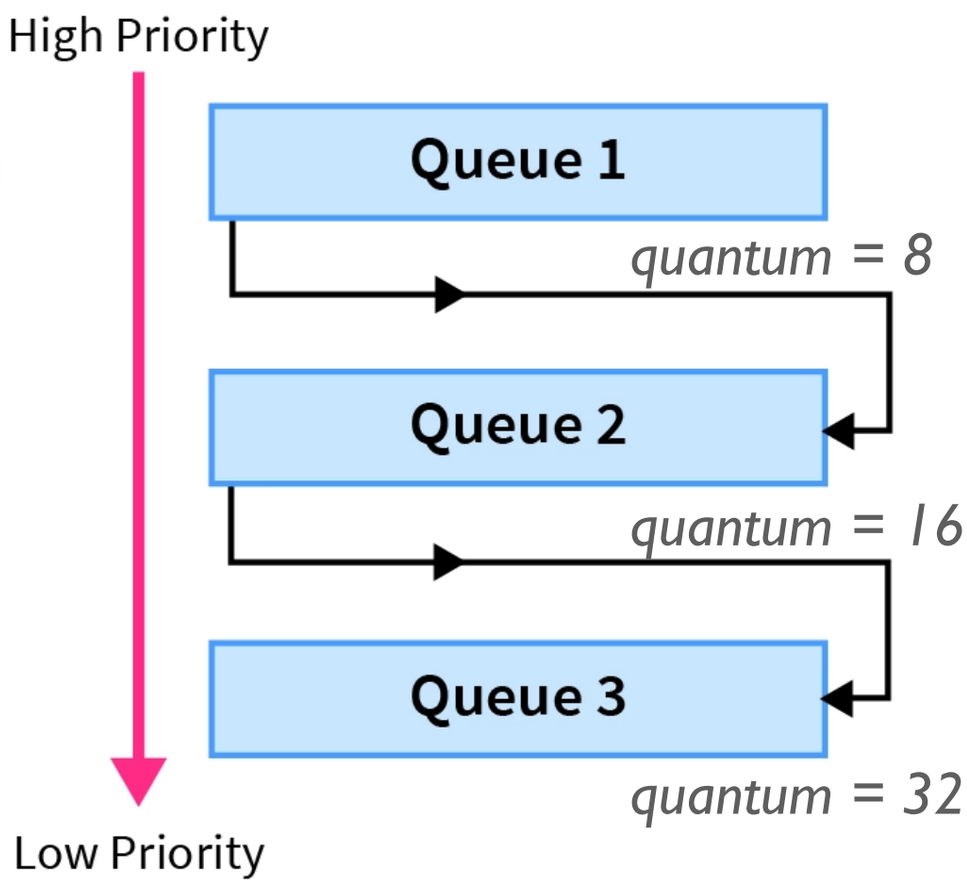
- RR 其实假设系统中的所有进程具有相同的优先级
- 但是，如何确定优先级?



Multi-Level Feedback Queue (MLFQ)

多级反馈队列

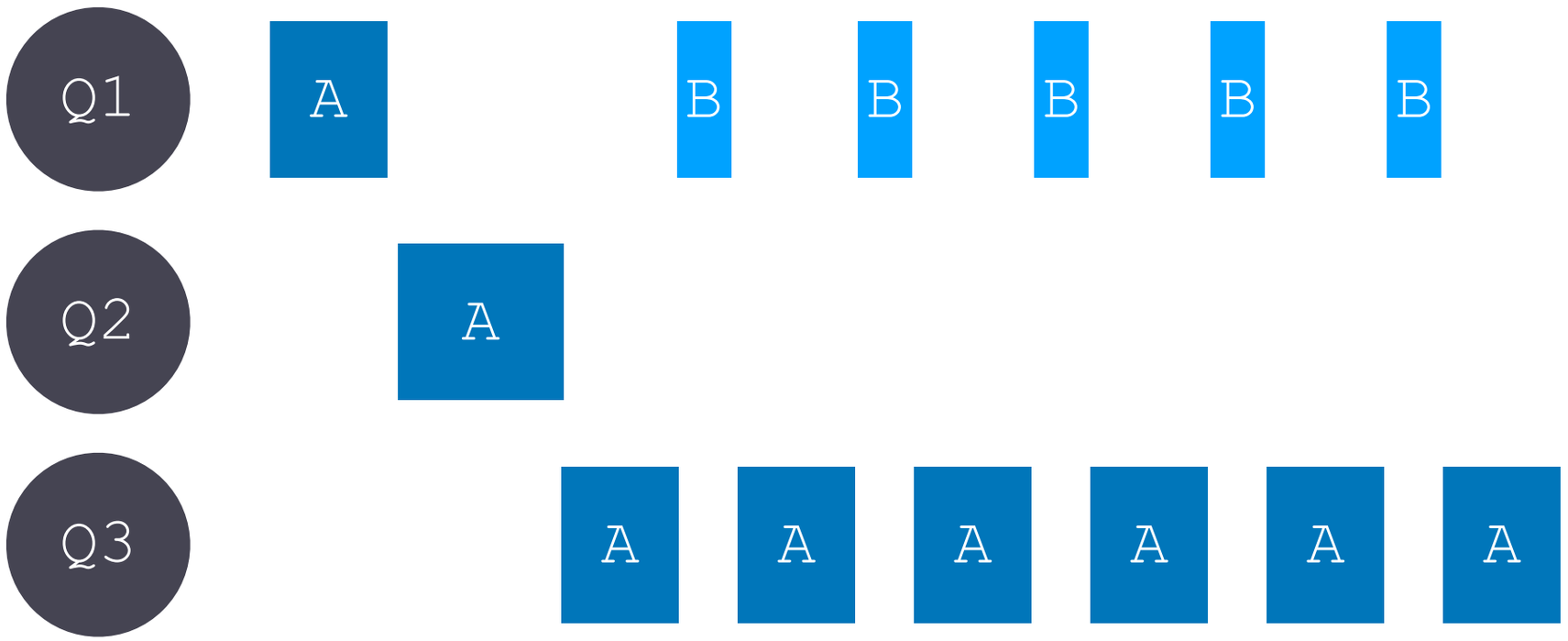
- 具有多个优先级队列 (Multi-Level Queue)
  - 可给予低优先级队列更长的时间片
- 根据进程表现的行为来在不同的队列间移动进程 (Feedback)
- 进程进入系统时，首先放置在最高优先级队列
- 如果进程用完了该队列对应的时间片份额，降低其优先级
- 如果进程在时间片结束前主动放弃CPU，仍停留在当前优先级



Multi-Level Feedback Queue (MLFQ)

根据进程的历史行为来猜测其是 CPU-bound 还是 I/O-bound

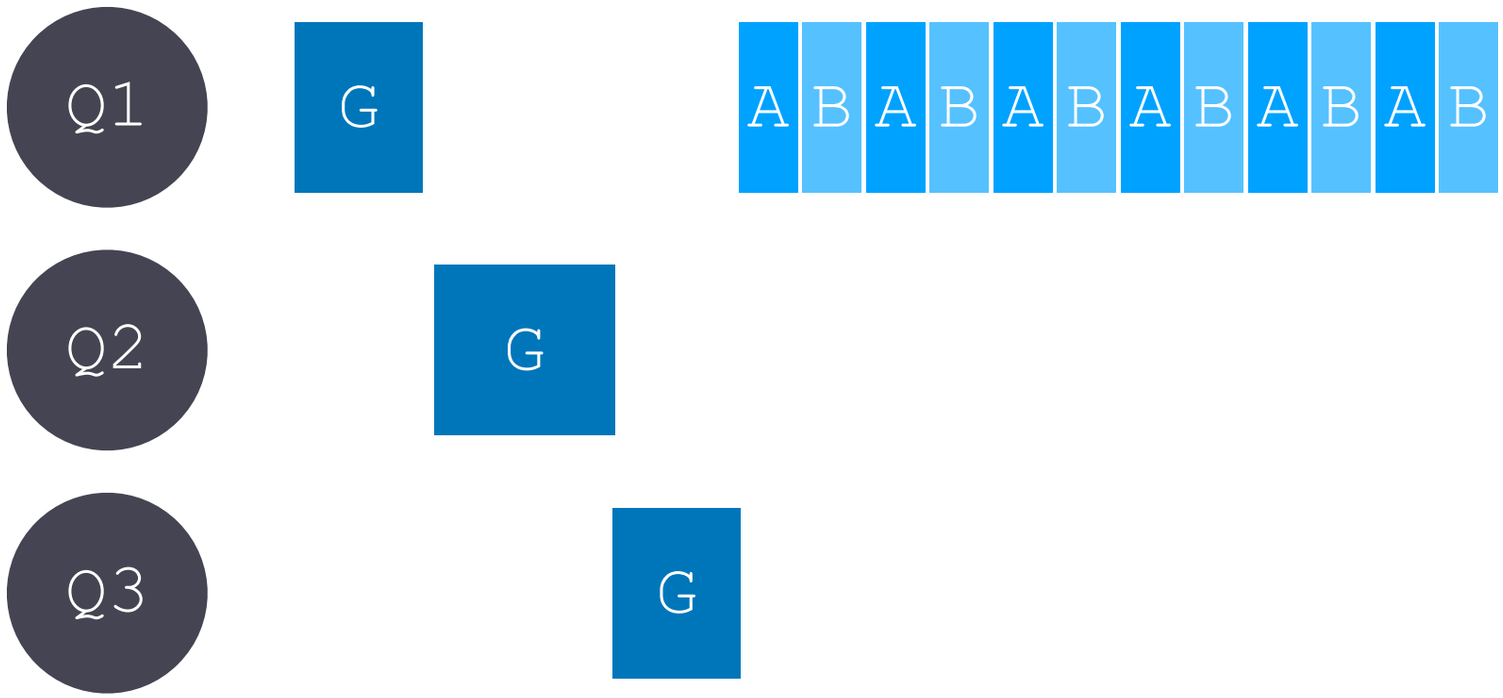
- 对于 CPU-bound，如果执行时间短则使其快速完成 (某种 SJF 的近似)，否则优先级会逐步降低 (a long running job)
- 对于 I/O-bound，确保其停留在较高优先级以确保快速响应



Multi-Level Feedback Queue (MLFQ)

⚠️ MLFQ 存在饿死 (starvation) 的问题

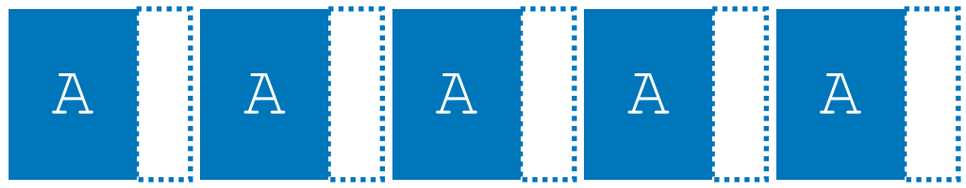
- 如果系统中总是存在高优先级进程 (大量交互式任务)，则低优先级进程 (纯计算任务) 总是不会被调度执行
  - 如果一个进程首先是 CPU-bound，然后转换为 I/O-bound?
- 可定期提高/重置进程的优先级 (priority boosting) 来避免



Multi-Level Feedback Queue (MLFQ)

⚠️ MLFQ 有可能会被恶意进程愚弄 (gamed)

- 如果用户知道 OS 的详细调度策略，则可以精心编写代码来使进程获得本不该属于其的更多计算资源
- 可通过持续追踪进程的累计执行时间 (better accounting) 来避免
- 在用完累计份额后，无论是否在时间片结束前 yield 都降低优先级



Before the time slice is over, give up the CPU (e.g., sleep for a few milliseconds)

Multi-Level Feedback Queue (MLFQ)

MLFQ 对于 I/O-bound 交互式进程能提供很好的快速响应，同时也能平衡 CPU-bound 进程的运行

- MLFQ 可以有很多不同的实现方式
  - 优先级队列的个数
  - 进程首次进入系统时的初始队列
  - 每个队列的时间片长度、以及采用的调度算法
    (例如，高优先队列采用 RR，低优先级队列采用 FCFS)
  - 降低优先级、以及提高优先级的策略
    (例如，确保某类进程最低只会到某一优先级)
- 目前很多 OS 都采用 MLFQ 的某种变种实现来进行调度

Fair Share Scheduling

除了设计一种调度策略去优化周转时间/响应时间，另一种思路是确保系统中的所有进程能以某种方式公平分享 CPU 资源

- 公平分享 (fair share)
  - 完全平均 (like RR): *N* 个进程，每个分配 *1/N* 的 CPU 时间
  - 按比例分配 (proportional): 给某些进程更多的份额
- 在调度时，如果一个进程已使用的 CPU 时间不足 (或超过) 其份额，则在后续给予其更多 (或更少) 的 CPU 时间
- 这同时也确保了所有进程都能往前推进 (no starvation)

Lottery Scheduling

彩票调度: 用进程持有的彩票 (ticket) 数量来代表其 CPU 使用时间份额

- 给每个进程分配一定数量的彩票
- 在每个时间片，随机选择一个中奖彩票
  - 拥有更多彩票的进程：更大的中奖机会 ➔ 更多的 CPU 时间
- 为了避免饥饿，确保每个进程至少持有一张彩票

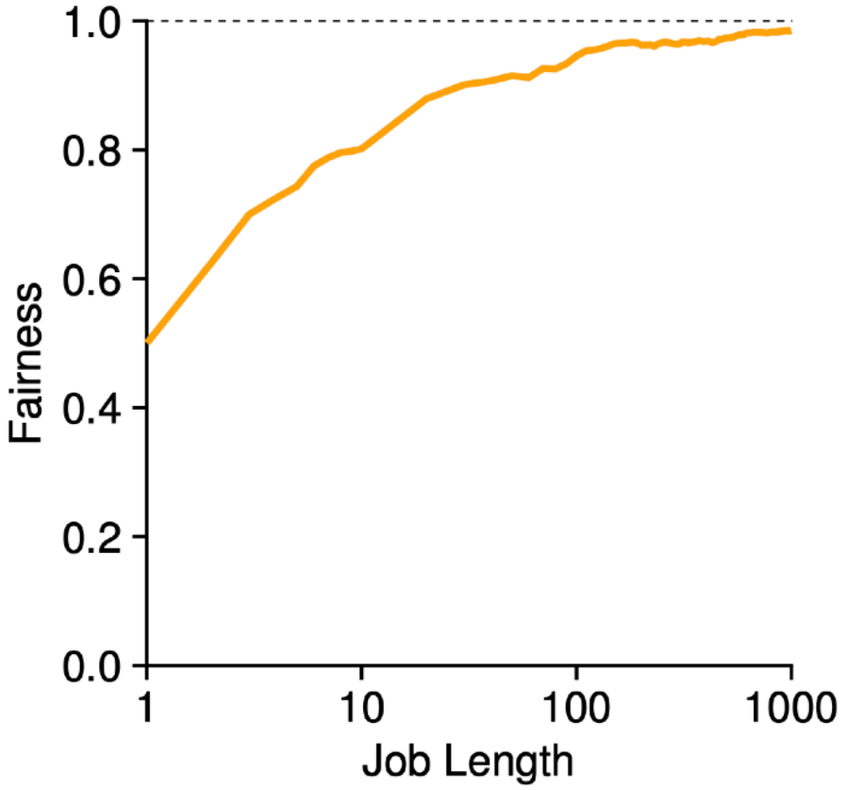
```
63 85 70 39 76 17 29 41 36 39 10 99 68 83 63 62 43  0 49 12

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

Suppose that A has 75 tickets while B has only 25

Lottery Scheduling

彩票调度: 用进程持有的彩票 (ticket) 数量来代表其 CPU 使用时间份额

- 实现仅依赖每个进程持有的彩票数目，不需要追踪额外信息
- 进程间可以通过交换彩票来促进协作 (e.g., 把自己的彩票给另一个进程，以使其能尽快执行某些任务)
- 应用灵活，可通过分配不同彩票模拟不同的调度策略 (e.g., 给短作业进程更多彩票来模拟 SJF)
- 但是，选择的随机性会导致一些问题
  - 缺少对下次调度执行进程的控制
  - 对短时间运行的进程不一定公平



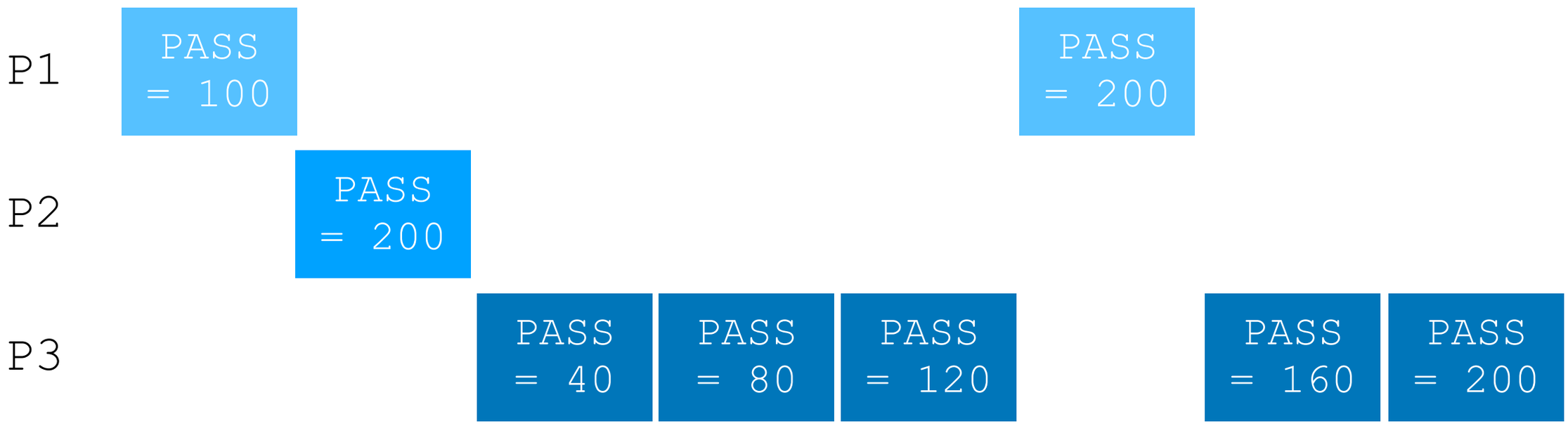
Stride Scheduling

步幅调度: 一种确定性的 fair-share 调度策略

- 每个进程具有一个 `pass` 值和一个 `stride` 值
  - 进程每调度执行一个时间片: `proc->pass += proc->stride`
  - 给予进程的 `stride` 值越小，进程占有的 CPU 时间份额越多
- 调度器选择选择 `pass` 值最小的进程来执行
- 需要追踪进程的全局状态信息

Stride Scheduling

步幅调度: 一种确定性的 fair-share 调度策略



假设 P1 stride = 100, P2 stride = 200, P3 stride = 40

(P1, P2 和 P3 以 2:1:5 的方式分享 CPU 资源)

真实世界的调度

Linux Scheduler

- 早期的 Linux 使用 RR
- 从 Linux 2.2 开始，将任务 (task) 分为三类:
  - `SCHED_FIFO` (real-time)
  - `SCHED_RR` (real-time)
  - `SCHED_NORMAL` (non real-time)
- 只要系统中存在处于就绪状态的 real-time 任务，则 non real-time 任务就不会被调度执行
  - 这里的 real-time 并不是真的实时系统中的任务
  - 仅是赋予任务更高优先级的一种方式

真实世界的调度

Linux Scheduler

- 对于普通任务 (non real-time tasks) 的调度策略
  - **O(n) scheduler**: Linux 2.4 to Linux 2.6
  - **O(1) scheduler**: Linux 2.6 to 2.6.22
  - **CFS scheduler**: Linux 2.6.23 onwards

O(n) Scheduler

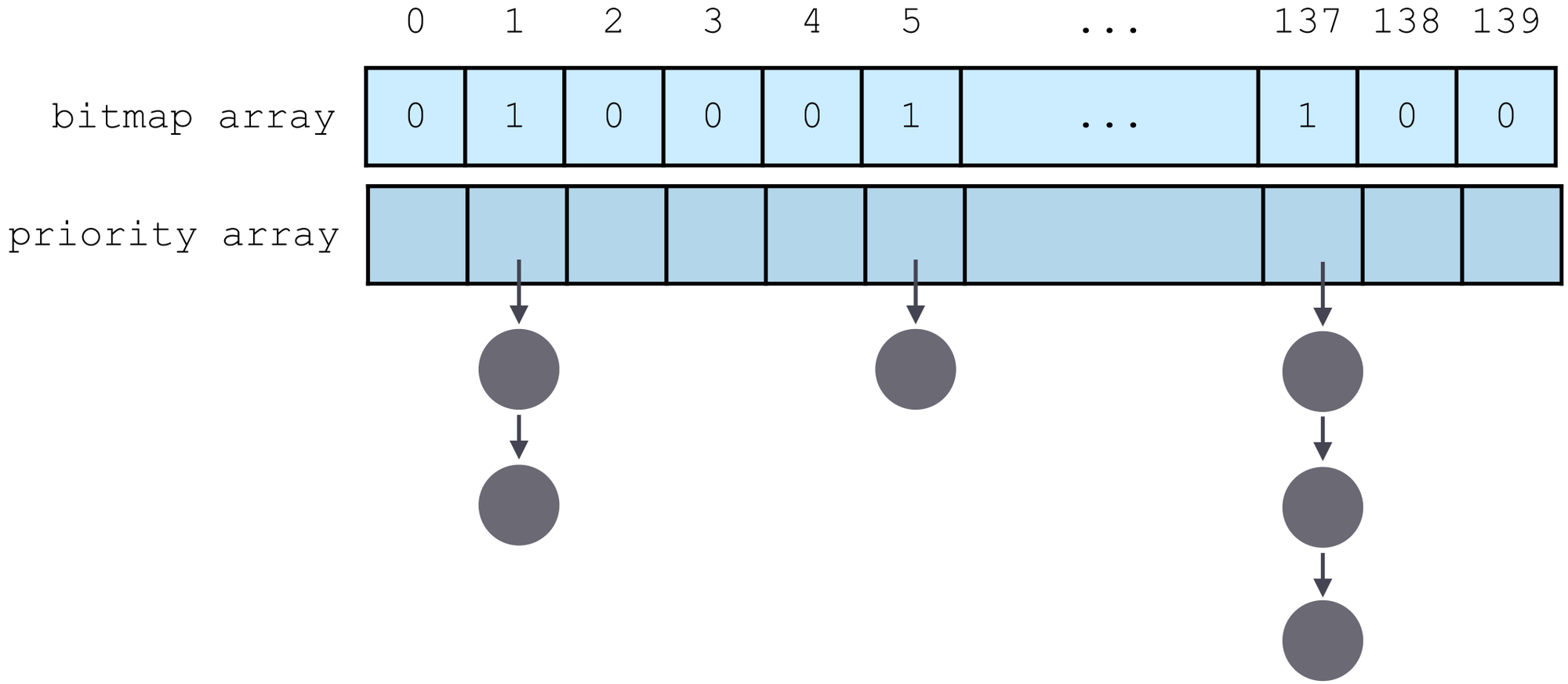
在内核中维护一个存储所有就绪任务的队列 (queue)，每次都从该队列中根据某种指标 (goodness value) 选择下一个执行任务

- 如果一个任务没有用完其 time slice 就结束，则下一次允许该任务执行更长时间
- 每次调度都需要遍历队列来进行选择
- 任务数量少时的合理选择
- 但存在可扩展性问题、并且多核时仍只有一个全局队列

O(1) Scheduler

总是以 O(1) 时间复杂度选择下一个调度执行的任务

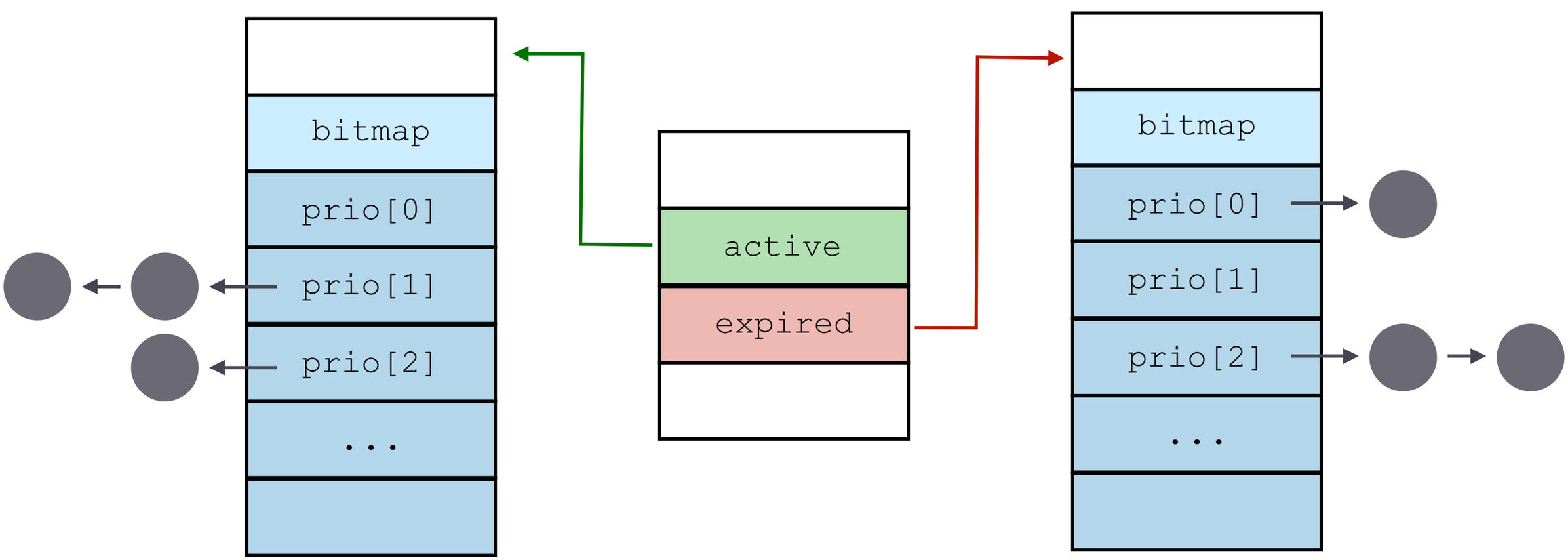
- 维护一个 bitmap array 和一个 priority array (每个核维护自己的队列)
- 寻找优先级最高的任务 → 寻找 bitmap array 中的第一个 1
  - 硬件提供相应的指令 (例如 x86 中的 bit scan forward 指令 `bsfl`)



O(1) Scheduler

总是以 O(1) 时间复杂度选择下一个调度执行的任务

- 维护两个 priority arrays (active & expired)
  - 当一个任务用完 time slice，重新计算优先级，并移到 expired array
  - 当 active array 为空时，交换指向 active 和 expired array 的指针



O(1) Scheduler

通过 `nice` 值 [-20, 19] 来表示优先级 (默认值为 0)

- 正数代表较低的优先级 (you would nice up your job)
- 负数代表较高的优先级 (not nice)
- 只有系统管理员可以将 `nice` 设为负数

```
top - 22:12:36 up 13:11,  0 users,  load average: 4.00, 3.81, 2.57
Tasks:   6 total,   5 running,   1 sleeping,   0 stopped,   0 zombie
%Cpu(s): 20.6 us,  0.2 sy,  4.4 ni, 74.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  2046748 total,   106004 free,   257492 used,  1683252 buff/cache
KiB Swap:  1048572 total,  1048572 free,        0 used.  1626020 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
   30 root      20   0    4624    808    744 R  41.0  0.0   6:44.49 sh
   32 root      20   0    4624    868    800 R  41.0  0.0   5:17.14 sh
   28 root      25   5    4624    884    816 R  13.3  0.0   7:15.03 sh
   34 root      30  10    4624    804    740 R   4.3  0.0   0:22.34 sh
    1 root      20   0   18504   3420   2976 S   0.0  0.2   0:00.06 bash
   37 root      20   0   36612   3184   2740 R   0.0  0.2   0:00.01 top
```

O(1) Scheduler

依赖复杂的启发式规则动态调整任务的优先级 (猜测任务的特征)

- 一种 MLFQ 策略的实现
- 偏好于交互式 (interactive) 任务

bonus = min(10, average sleep time / 100 ms)

Bonus increases as the process sleeps more

priority = max(100, min(static priority - bonus + 5, 139))

minimum value is still 100

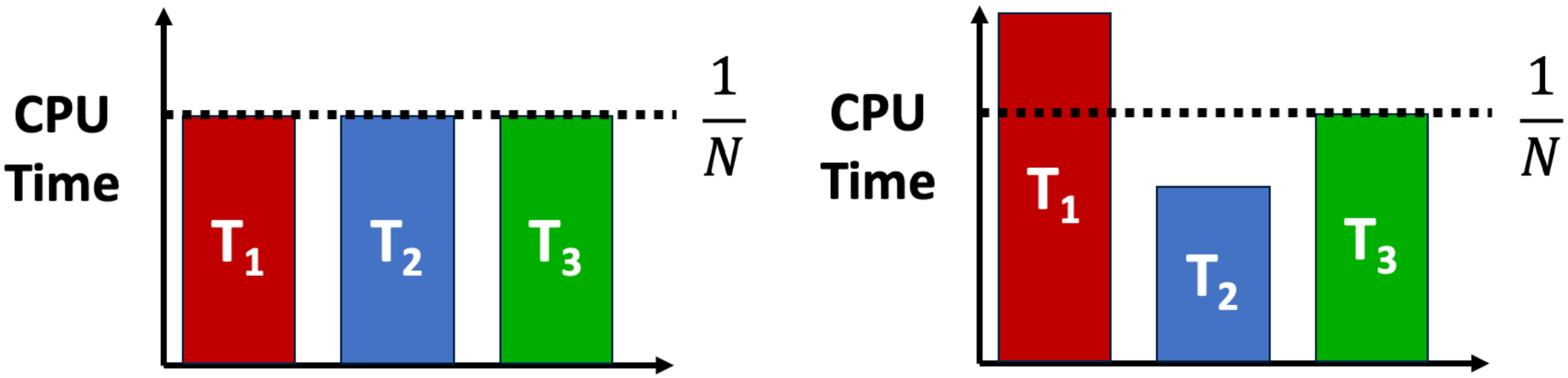
bonus is subtracted to increase priority

maximum value is still 139

Completely Fair Scheduler (CFS)

Linux CFS 调度器尝试在系统所有任务之间公平地分配 CPU

- 实现一种 counting-based fair-share 调度
  - 每个任务运行时，追踪其 CPU 使用时间
  - 当发生调度时，CFS 选择当前 CPU 使用时间最小的任务来运行



N processes "simultaneously" execute on *1/N* of CPU

Completely Fair Scheduler (CFS)

设定一个调度周期 sched_latency，在每个调度周期内所有任务都会被调度一次 (避免静态时间片下调度时延随任务个数上升而上升)

- 在 **equal share** 下，time slice = sched_latency / # tasks
- 假设 sched_latency = 20 ms
  - 如果有 4 个进程，则 time slice = 5 ms
  - 如果有 200 个进程，则 time slice = 0.1 ms？
- 设定一个最小时间片粒度 min_granularity

Completely Fair Scheduler (CFS)

在 equal share 的基础上，CFS 允许用户通过给予某些任务更高优先级的方式来让该任务获得 CPU 的 higher share (proportional share)

- 基于权重 (优先级差异) 来计算每个任务的 time slice

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

Completely Fair Scheduler (CFS)

在 equal share 的基础上，CFS 允许用户通过给予某些任务更高优先级的方式来让该任务获得 CPU 的 higher share (proportional share)

- 将进程的 `nice` 值映射到相应的权重
- 从 "分配绝对时间片" 转变为 "分配相对 CPU 份额" (`nice` 值每相差 1，CPU 运行时间份额相差大约 10%)

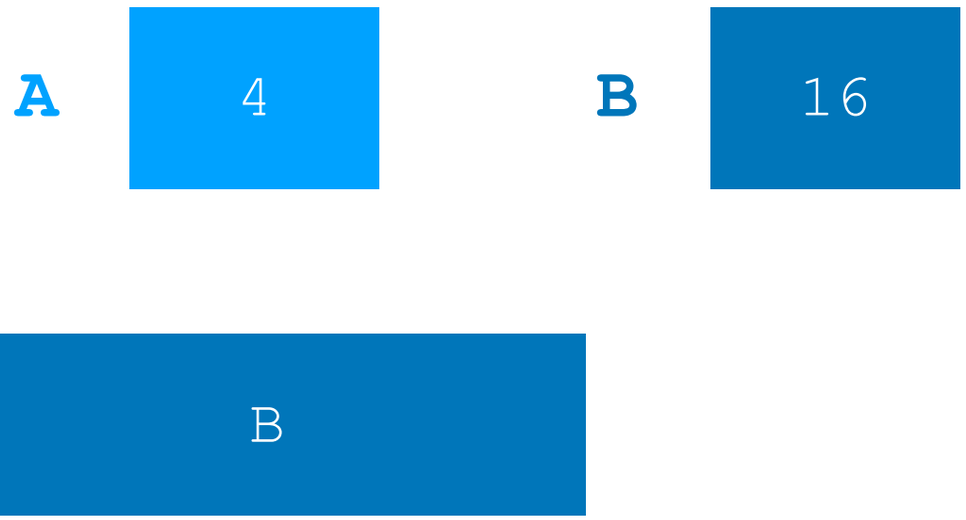
```
// weight = 1024 / 1.25^nice
static const int prio_to_weight[40] = {
    /* -20 */ 88761, 71755, 56483, 46273, 36291,
    /* -15 */ 29154, 23254, 18705, 14949, 11916,
    /* -10 */  9548,  7620,  6100,  4904,  3906,
    /*  -5 */  3121,  2501,  1991,  1586,  1277,
    /*   0 */  1024,   820,   655,   526,   423,
    /*   5 */   335,   272,   215,   172,   137,
    /*  10 */   110,    87,    70,    56,    45,
    /*  15 */    36,    29,    23,    18,    15,
};
```

Completely Fair Scheduler (CFS)

```
Target latency = 20 ms
Minimum granularity = 1 ms
A weight = 1, slice = 4 ms
B weight = 4, slice = 16 ms
```

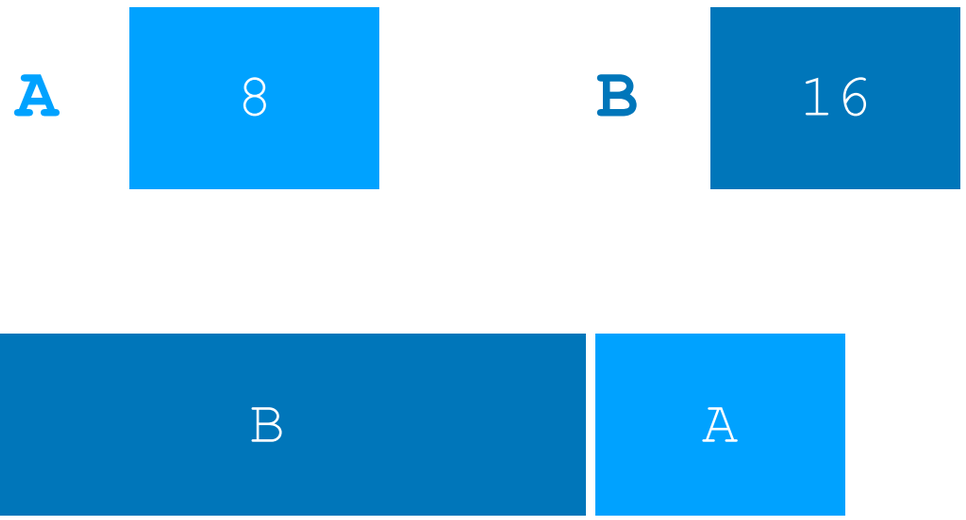
Completely Fair Scheduler (CFS)

```
Target latency = 20 ms
Minimum granularity = 1 ms
A weight = 1, slice = 4 ms
B weight = 4, slice = 16 ms
```



Completely Fair Scheduler (CFS)

```
Target latency = 20 ms
Minimum granularity = 1 ms
A weight = 1, slice = 4 ms
B weight = 4, slice = 16 ms
```

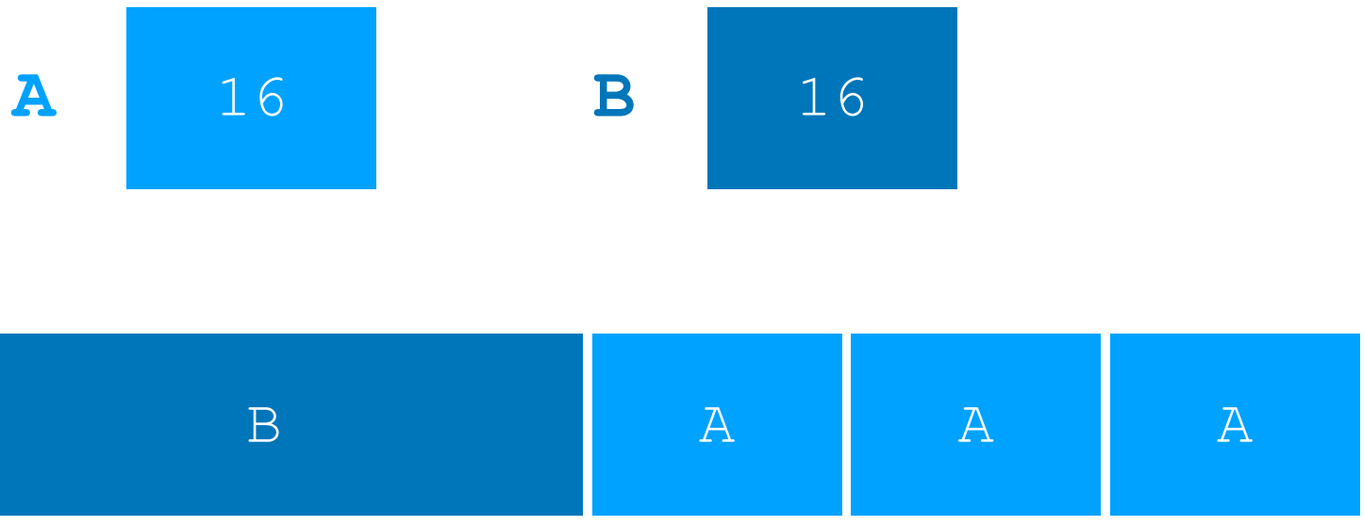


Completely Fair Scheduler (CFS)

```
Target latency = 20 ms
Minimum granularity = 1 ms
A weight = 1, slice = 4 ms
B weight = 4, slice = 16 ms
```

Completely Fair Scheduler (CFS)

```
Target latency = 20 ms
Minimum granularity = 1 ms
A weight = 1, slice = 4 ms
B weight = 4, slice = 16 ms
```



A and B got 50% of the CPU !

Completely Fair Scheduler (CFS)

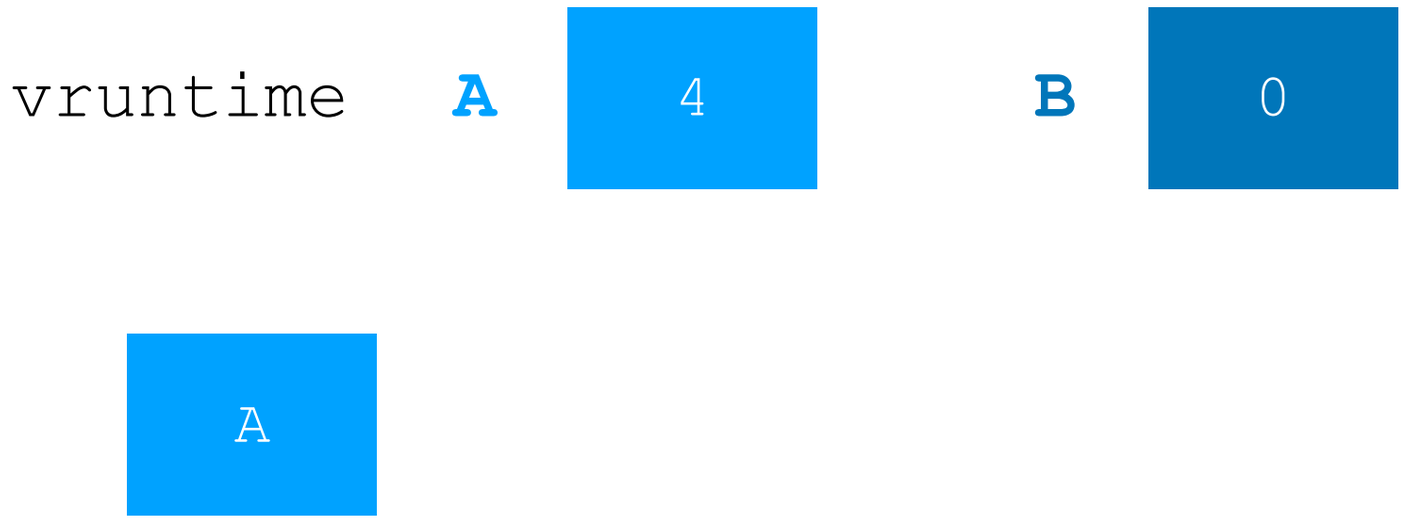
在 equal share 的基础上，CFS 允许用户通过给予某些任务更高优先级的方式来让该任务获得 CPU 的 higher share (proportional share)

- 追踪任务的虚拟 CPU 使用时间 vruntime 而不是真实 CPU 使用时间
  - 使得不同的任务具有不同的执行速率
  - 任务的权重越高，其 vruntime 的累积速度越慢

$$\texttt{vruntime}_i = \texttt{vruntime}_i + \frac{\texttt{weight}_0}{\texttt{weight}_i} \cdot \texttt{runtime}_i$$

Completely Fair Scheduler (CFS)

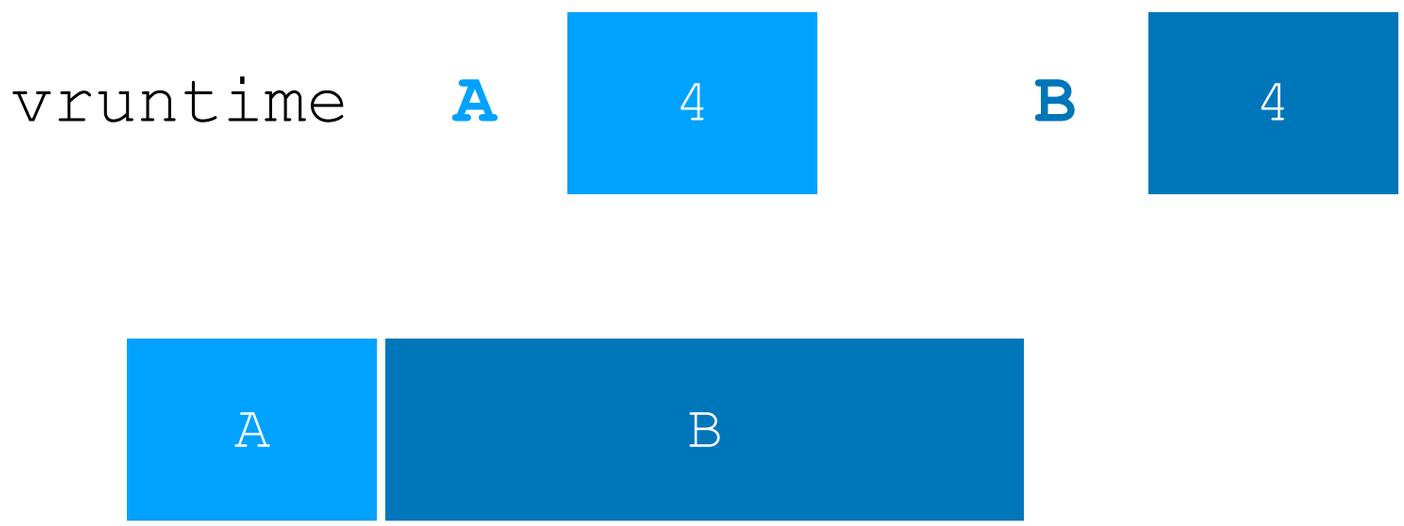
```
Target latency = 20 ms
Minimum granularity = 1 ms
A weight = 1, slice = 4 ms
B weight = 4, slice = 16 ms
```



virtual runtime = physical runtime * 1 / weight = 4

Completely Fair Scheduler (CFS)

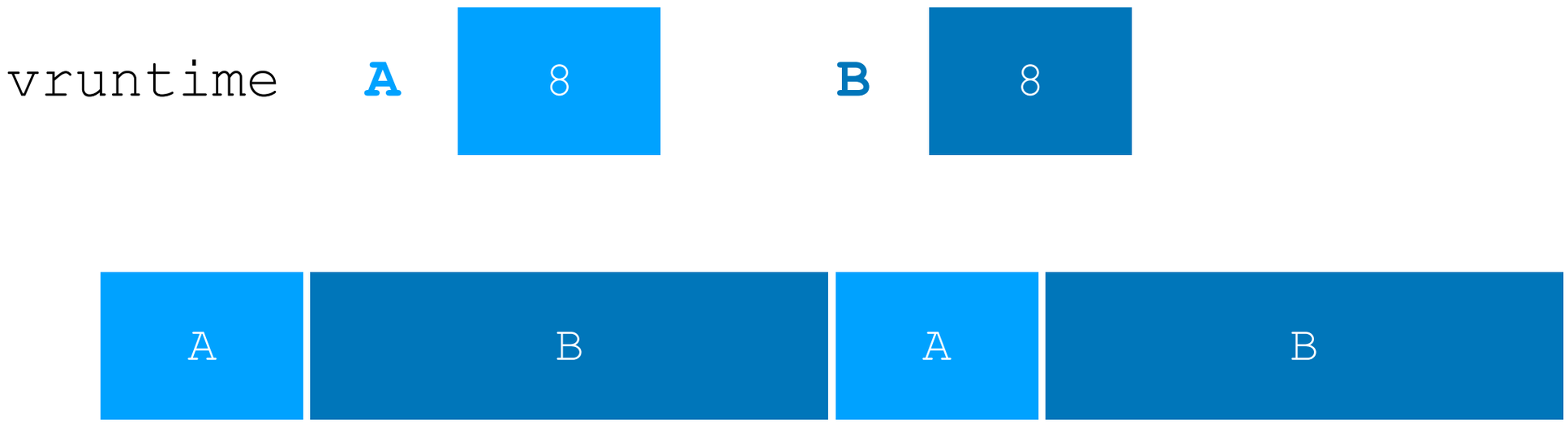
```
Target latency = 20 ms
Minimum granularity = 1 ms
A weight = 1, slice = 4 ms
B weight = 4, slice = 16 ms
```



virtual runtime = physical runtime * 1 / weight = 16 * 1/4 = 4

Completely Fair Scheduler (CFS)

```
Target latency = 20 ms
Minimum granularity = 1 ms
A weight = 1, slice = 4 ms
B weight = 4, slice = 16 ms
```

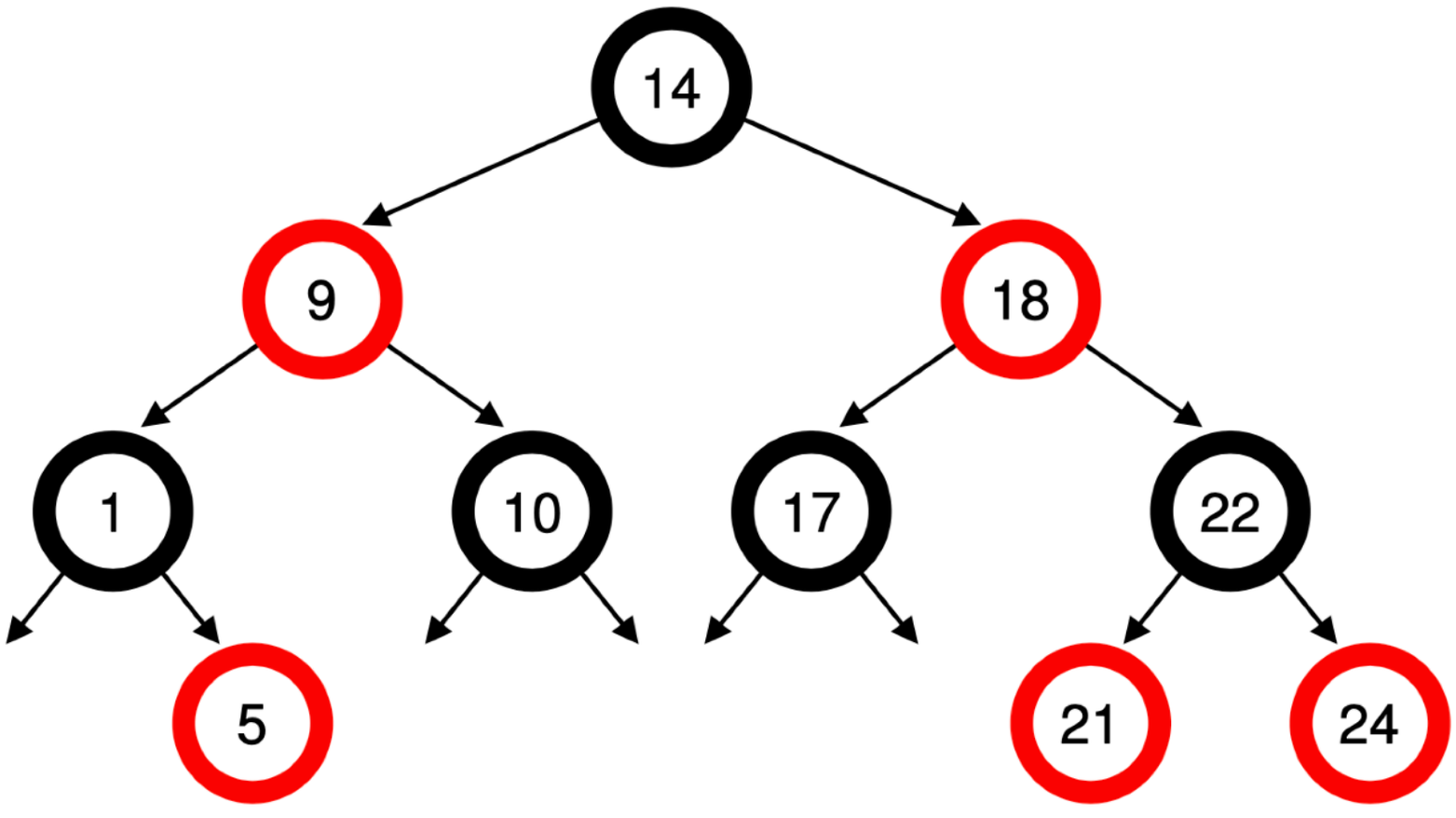


CFS shares virtual runtime equally

Completely Fair Scheduler (CFS)

每次调度时需要查找当前 vruntime 值最少的任务

- 使用列表等简单数据结构难于扩展到大量任务存在的场景
- CFS 使用红黑树来存储当前可运行的任务



Completely Fair Scheduler (CFS)

频繁执行 I/O 或睡眠的任务可能无法获得公平的 CPU 份额

- 假设进程 A 一直运行计算任务，进程 B 睡眠 10 秒
  - 当 B 睡眠结束时，其 vruntime 将比 A 晚 10 秒
  - 此时 B 将独占 (monopolise) CPU，并潜在导致 A 的饿死
- 可以在进程唤醒时修改其 vruntime 来避免这一问题 (例如更改为当前所有进程的最小值)

多核 CPU 的调度 *

- 由一个 CPU (主服务器) 负责调度，其它 CPU 仅执行代码
  - 实现简单，避免了数据共享和同步的需求
  - 但主服务器会成为潜在的性能瓶颈
- 对于对称多处理器 (Symmetric Multiprocessing / SMP)，每个处理器自己进行调度
  - 多个处理器共享一个就绪队列?
  - 不同处理器各自维护一个就绪队列?

多核 CPU 的调度 *

- SMP 下需要保持所有 CPU 负载均衡 (load balancing) 以提高效率
  - 推迁移 (push migration): 周期性检查每个处理器的负载，如果发现过载，则将任务从过载的 CPU 推送到其他 CPU
  - 拉迁移 (pull migration): 空闲 CPU 从繁忙 CPU 中拉取任务
- 负载均衡会破坏 CPU 的亲和性 (affinity)
  - 当一个任务在一个 CPU 上运行时，缓存中存储了该任务的数据
  - 为了保持系统的良好性能，需要有一个保持亲和性的调度策略
    - 不能简单强制绑定任务到某个 CPU
    - 也不能哪个 CPU 空闲了就直接把其它就绪任务放到上面运行

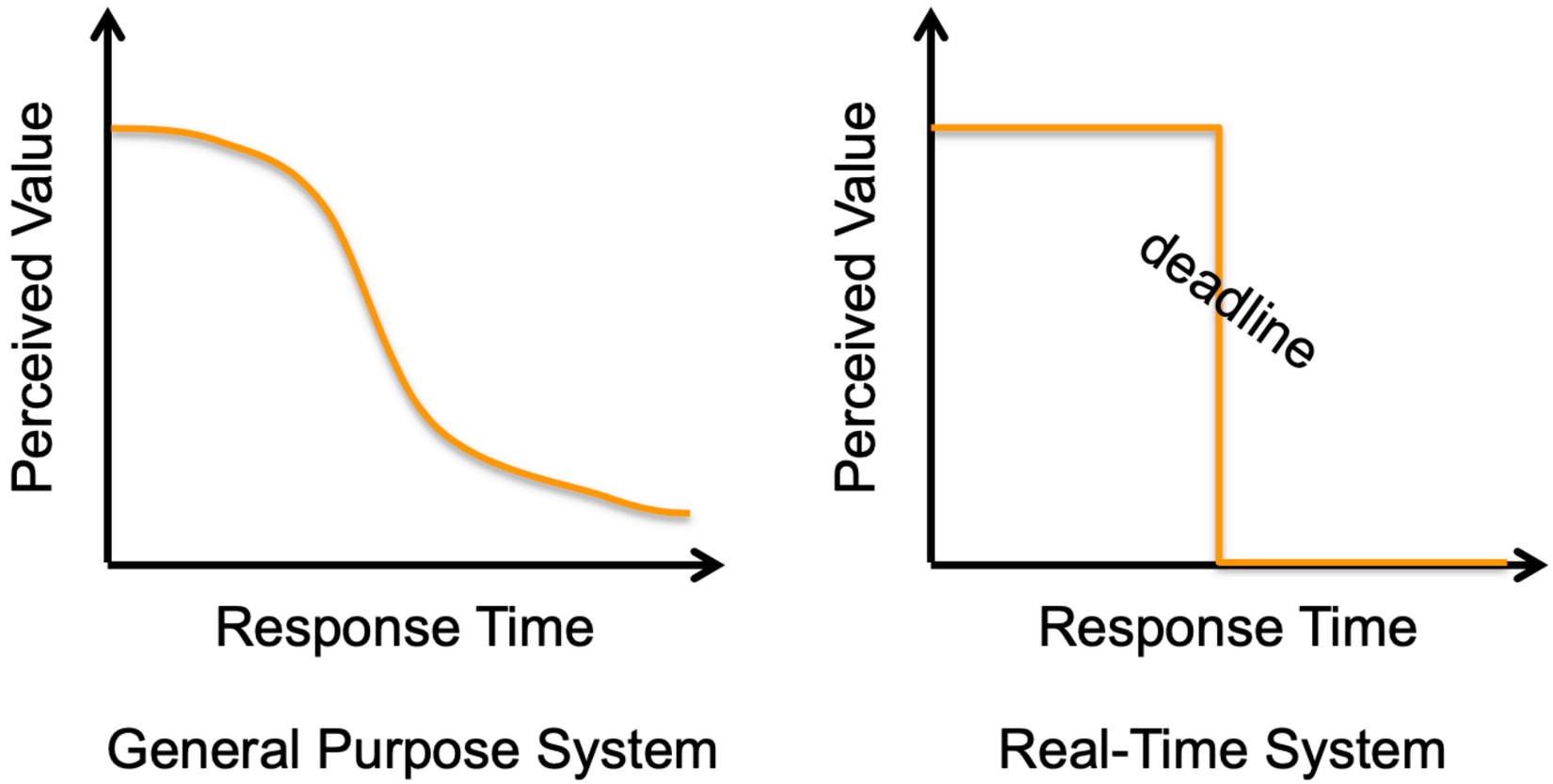
调度策略 *

针对实时系统 (real-time system) 的调度策略

- Rate Monotonic (RM)
- Earliest Deadline First (EDF)

实时系统

在实时系统 (real-time system) 中，任务是否能在截止时间 (deadline) 前完成是调度决策的关键因素

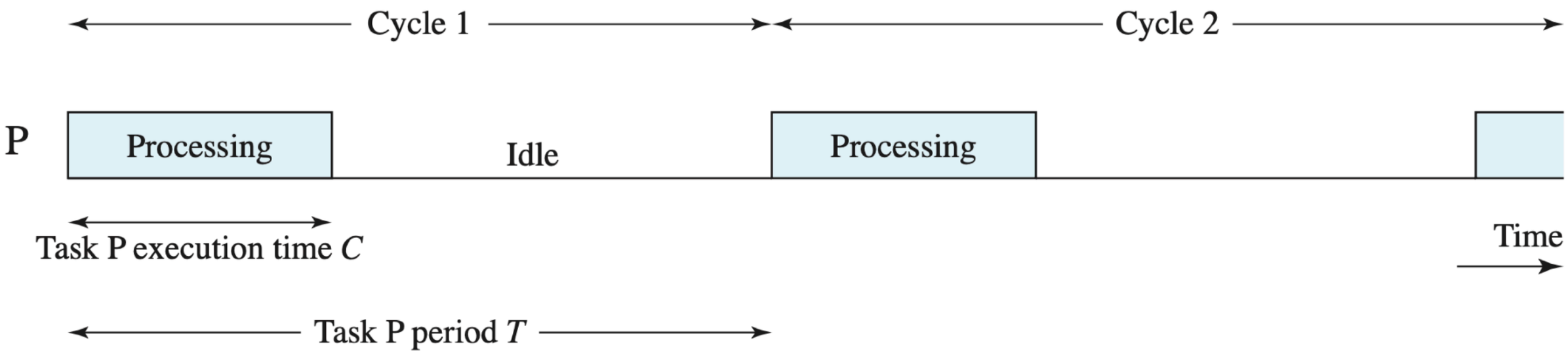


Having the right answer but having it too late
is often just as bad as not having it at all

实时系统调度

考虑一种针对周期性 (periodic) 任务的实时系统调度

- 每个任务有一个处理时间 (processing time) C 和周期 (period) T
  - 任务每次达到所需的 CPU 处理时间都相同
  - 任务每次周期的结束作为该次任务的截止时间



实时系统调度

考虑一种针对周期性 (periodic) 任务的实时系统调度

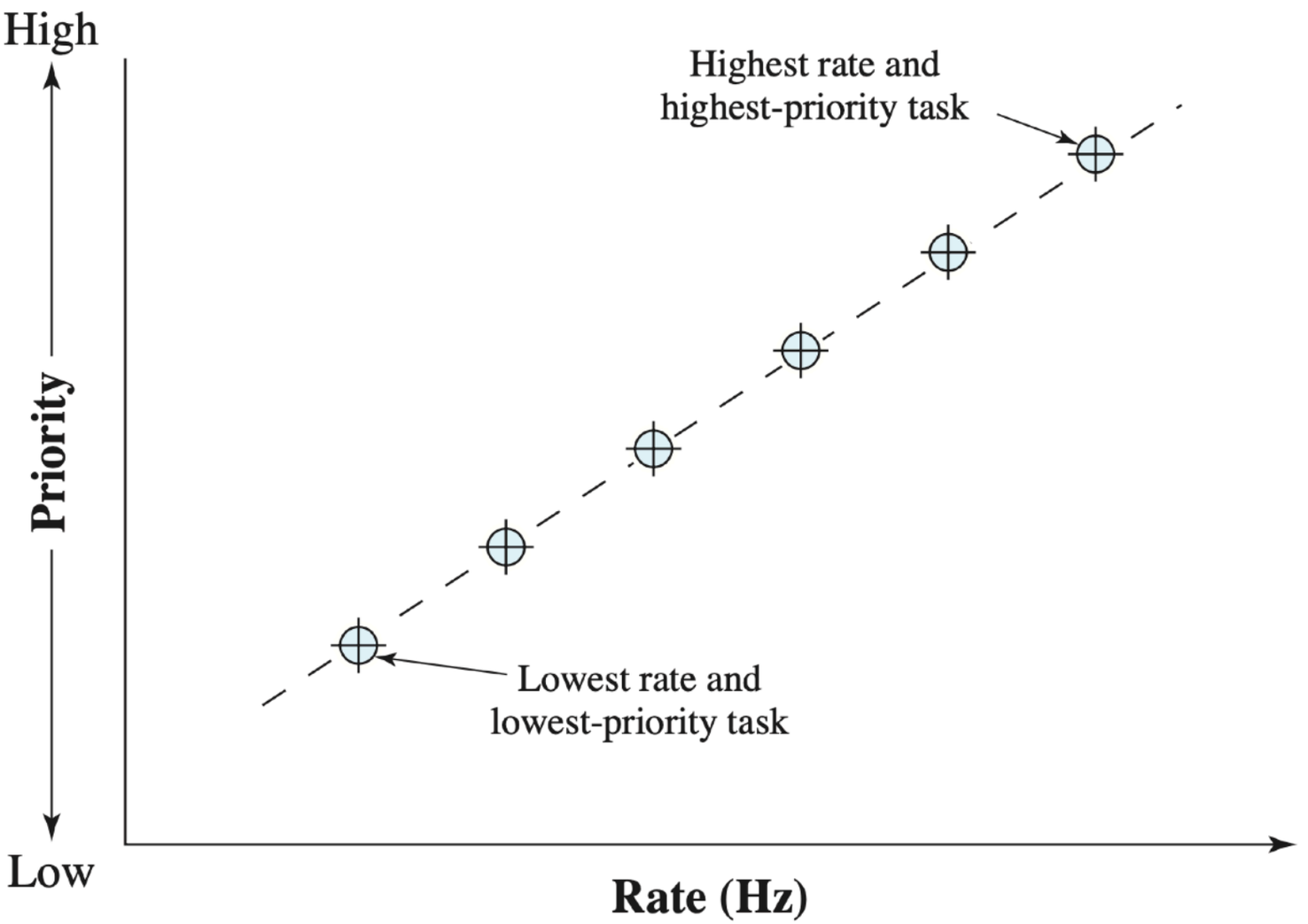
- 每个任务有一个处理时间 (processing time) C 和周期 (period) T
- 准入控制 (admission control)
  - 给定 n 个具有固定处理时间和周期的任务，为了在调度时能满足各任务的截止时间 (schedulable)，则必须有

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ \ldots \ + \frac{C_n}{T_n} \leq 1$$

Rate Monotonic (RM)

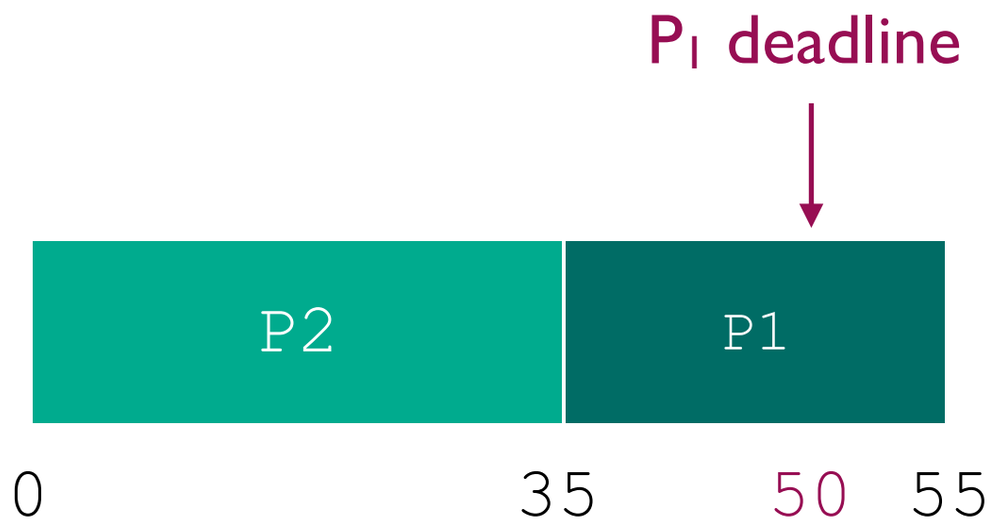
基于静态优先级的可抢占式调度 (static priority)

- 任务周期越短 (任务越频繁)，则优先级越高



Rate Monotonic (RM)

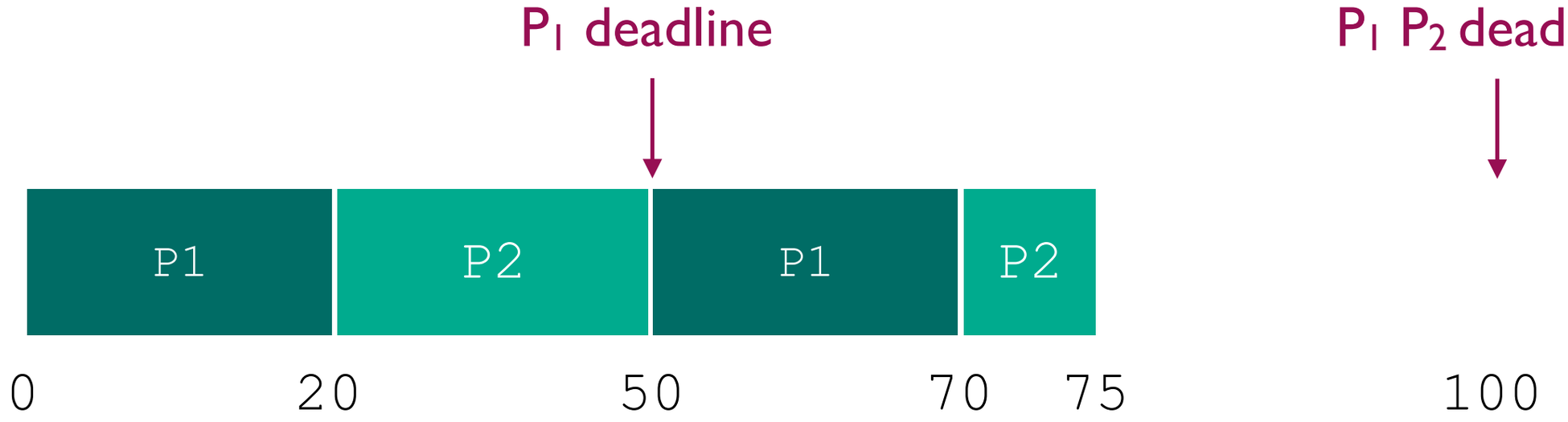
```
P1 period = 50, processing time = 20
P2 period = 100, processing time = 35
Because (20/50 + 35/100) = 0.75 < 1, so schedulable
```



If schedule P_2 first

Rate Monotonic (RM)

```
P1 period = 50, processing time = 20
P2 period = 100, processing time = 35
Because (20/50 + 35/100) = 0.75 < 1, so schedulable
```



Apply RM: schedule P_1 first

Rate Monotonic (RM)

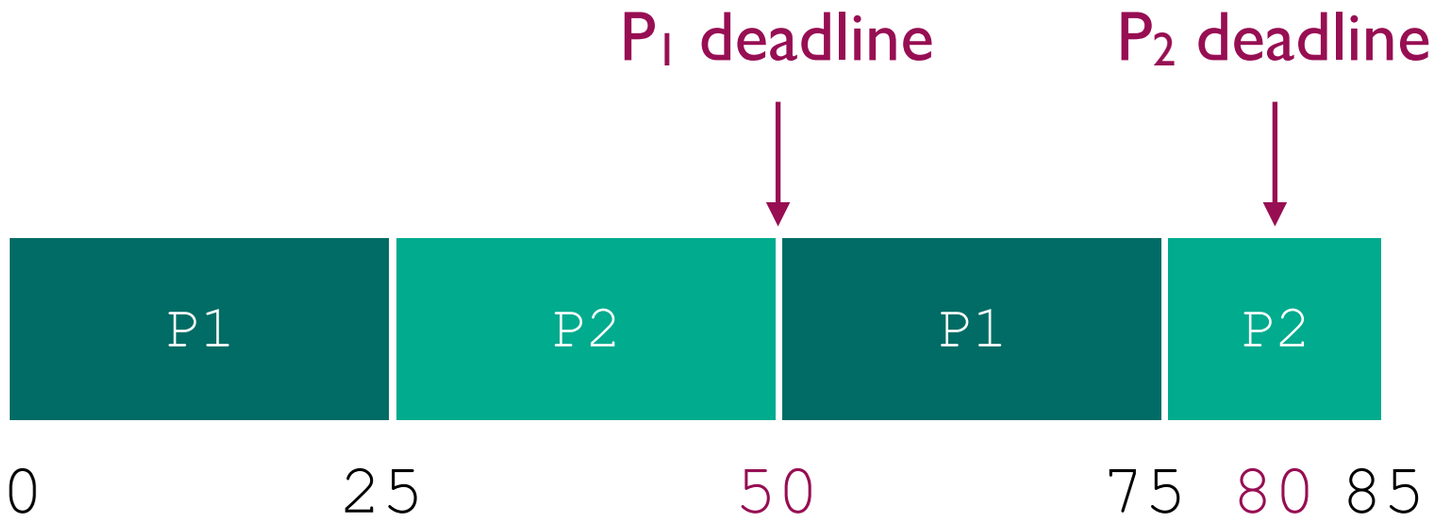
基于静态优先级的可抢占式调度 (static priority)

- 任务周期越短 (任务越频繁)，则优先级越高
- 如果多个任务无法被 RM 调度，则这些任务也无法被任何其它基于静态优先级的调度算法所调度

Liu C L, Layland J W. Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM (JACM), 1973, 20(1): 46-61

Rate Monotonic (RM)

```
P1 period = 50, processing time = 25, rate = 1/50
P2 period = 80, processing time = 35, rate = 1/80
Because (25/50 + 35/80) = 0.9375 < 1, so schedulable
```



Satisfy admission control does not indicate schedulable

Rate Monotonic (RM)

基于静态优先级的可抢占式调度 (static priority)

- 满足准入控制不一定能被 RM 调度 (并不总能最大化 CPU 利用率)
- 对于 n 个具有固定优先级的任务，如果满足以下条件，则一定能用 RM 进行调度：

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_n}{T_n} \leq n(2^{1/n} - 1)$$

When n = 2, about 83%

When n approaches infinity, about 69%

Liu C L, Layland J W. Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM (JACM), 1973, 20(1): 46-61

Earliest Deadline First (EDF)

基于截止时间的可抢占式调度 (dynamic priority)

- 离截止时间 (deadline) 越近的任务，优先级越高
- 任务可以不是周期性的，也不要求任务在每个周期都有相同的处理时间

Earliest Deadline First (EDF)

P1 period = 50, processing time = 25

P2 period = 80, processing time = 35

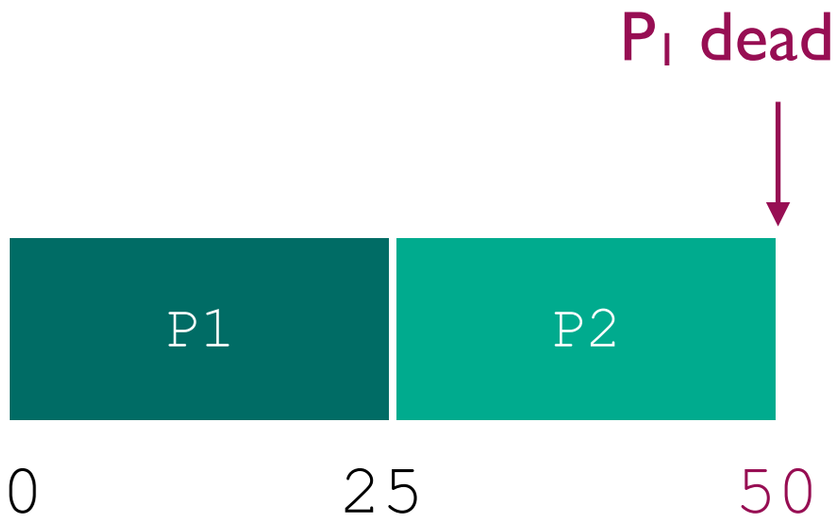
Because (25/50 + 35/80) = 0.9375 < 1, so schedulable

P_1 deadline = 50

P_2 deadline = 80

Earliest Deadline First (EDF)

```
P1 period = 50, processing time = 25
P2 period = 80, processing time = 35
Because (25/50 + 35/80) = 0.9375 < 1, so schedulable
```

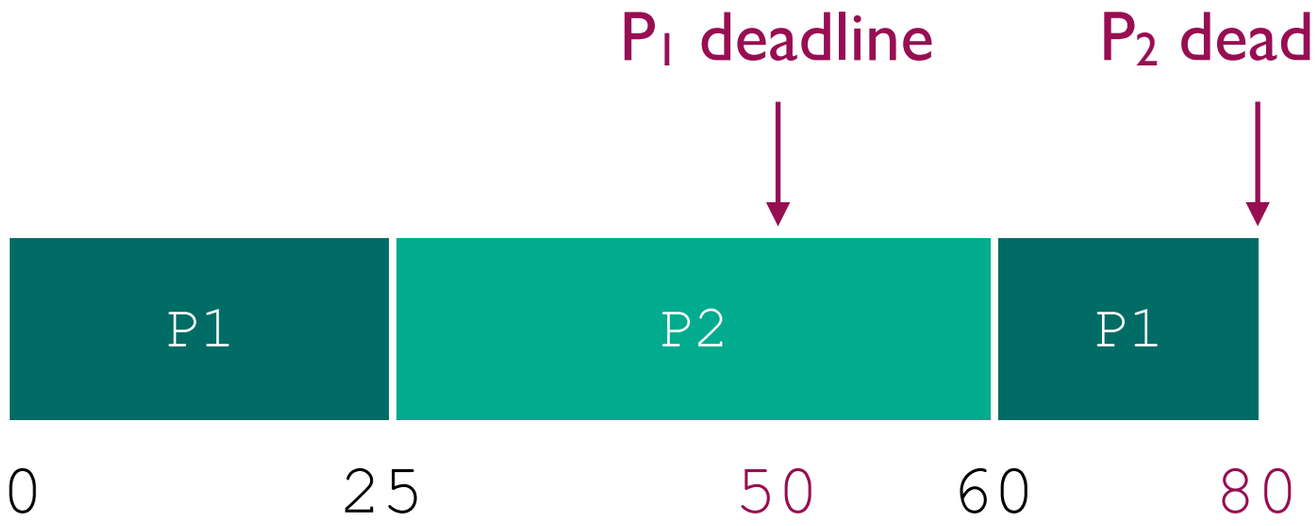


P_1 deadline = 100

P_2 deadline = 80

Earliest Deadline First (EDF)

```
P1 period = 50, processing time = 25
P2 period = 80, processing time = 35
Because (25/50 + 35/80) = 0.9375 < 1, so schedulable
```

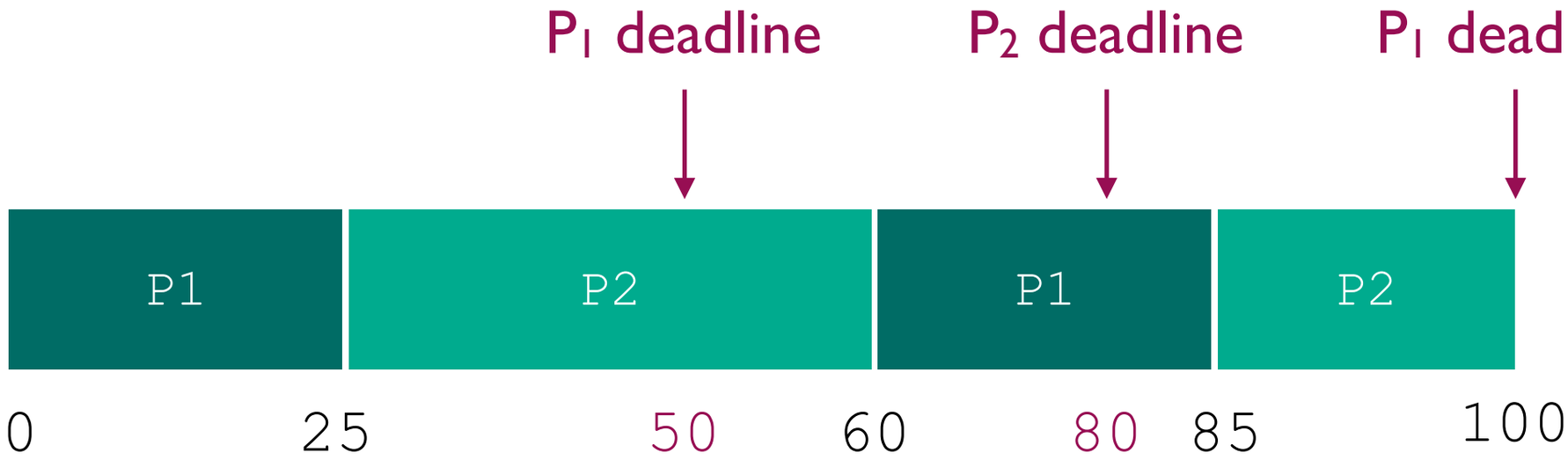


P_1 deadline = 100

P_2 deadline = 160

Earliest Deadline First (EDF)

```
P1 period = 50, processing time = 25
P2 period = 80, processing time = 35
Because (25/50 + 35/80) = 0.9375 < 1, so schedulable
```

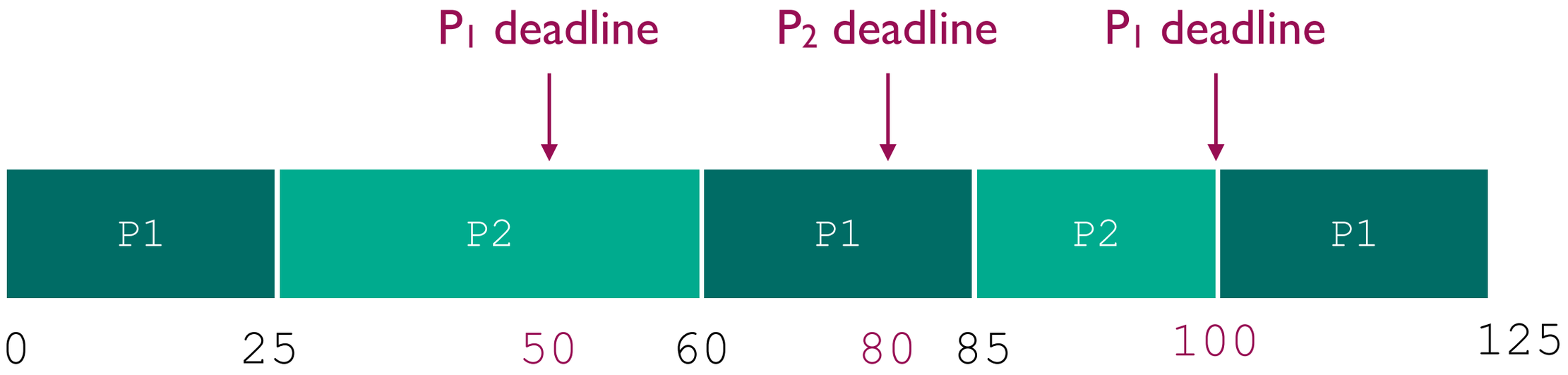


P_1 deadline = 150

P_2 deadline = 160

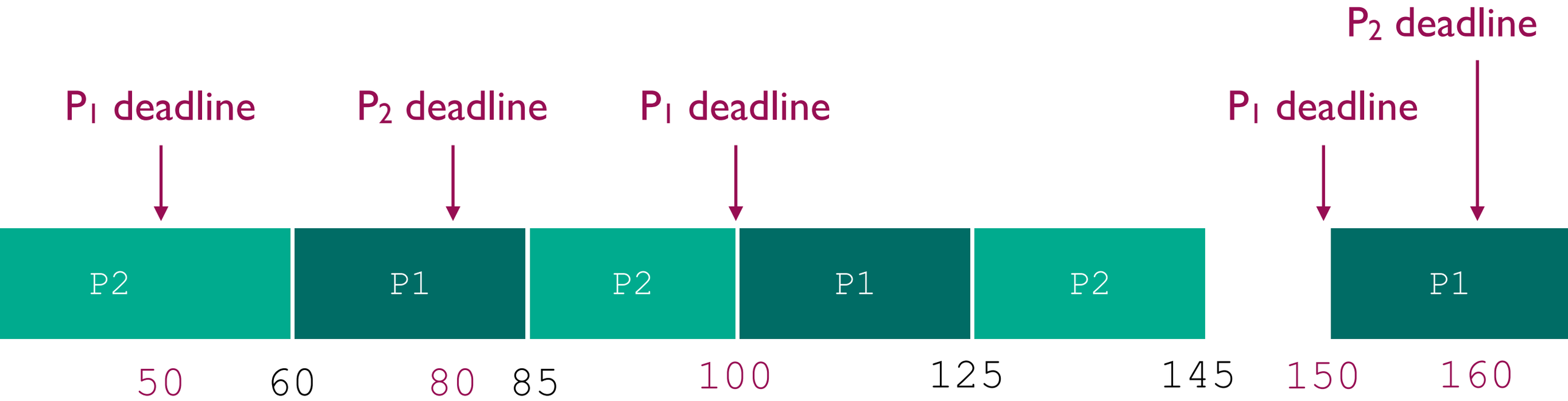
Earliest Deadline First (EDF)

```
P1 period = 50, processing time = 25
P2 period = 80, processing time = 35
Because (25/50 + 35/80) = 0.9375 < 1, so schedulable
```



Earliest Deadline First (EDF)

```
P1 period = 50, processing time = 25
P2 period = 80, processing time = 35
Because (25/50 + 35/80) = 0.9375 < 1, so schedulable
```



Earliest Deadline First (EDF)

基于截止时间的可抢占式调度 (dynamic priority)

- 离截止时间 (deadline) 越近的任务，优先级越高
- 当且仅当满足以下条件时，实时任务能用 EDF 调度：

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ \ldots \ + \frac{C_n}{T_n} \leq 1$$

Liu C L, Layland J W. Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM (JACM), 1973, 20(1): 46-61

实时系统调度

当涉及 CPU-bound 任务和 I/O-bound 任务混合的情况

- 例如，对于同时到达的如下两个任务
  - P_1：1ms 的 CPU 计算和 10ms 的 I/O，截止时间为 12ms
  - P_2：5ms 的 CPU 计算，截止时间为 10ms
    - EDF 将优先调度 P_2，这会导致 P_1 错过截止时间
- 可以将 CPU 计算和 I/O 时间分开考虑
  - 对于 CPU 计算部分，P_1 的真实截止时间应为 2ms
  - 此时，EDF 优先调度 P_1，可满足两个任务的截止时间

总结

- 评价调度策略的指标 
- 针对批处理系统 (batch system) 的调度 
  - FCFS 和 SJF
- 针对交互式系统 (interactive system) 的调度 
  - RR, MLFQ 和 CFS
- 针对实时系统 (real-time system) 的调度
  - RM, EDF