调度
Scheduling

钮鑫涛
南京大学
2024春

本门课由蒋炎岩和钮鑫涛共同构建

你更加愿意排在哪个队后面?

人更少?

单人服务时间少?

调度（scheduling）

吞吐量(throughput)
等待时间(wait time)
响应时间(response time)
公平(faireness)
...

CPU
网络连接
...

- 为了满足既定目标，对计算任务进行资源分配的行为

进程(process)
线程(thread)
数据流(data flow)
...

调度指标

- CPU利用率（CPU utilization)：CPU被进程所使用的时间在所有CPU时间的占比（越大越好）
- 公平 (fairness) : 同等优先级下的进程获得的CPU使用时间应该尽可能相等
- 吞吐量（throughput）： 单位时间内完成执行的进程数 （越大越好）
- 周转时间（turnaround time）： 某个进程需要完成的时间（越小越好）
- 等待时间（waiting time）： 某个进程在就绪队列的时间（越小越好）
  - 注: 等待时间 = 周转时间 - 获得CPU执行的时间
  - 一个进程可能会多次放到就绪队列中，等待时间是这些等待的总和
- 响应时间：从发出申请执行到**第一次**获得响应执行的时间（越小越好）

调度的时机

- CPU回到操作系统的掌控之中
  - 发生系统调用：比如 fork() 、 exit()
  - 某个运行的进程阻塞了（比如wait子进程或者等待一个I/O完成）
  - 发生中断，比如I/O interrupt，clock interrupt
- 这些事件的发生都意味着系统的状态发生了变化，可能和既定的目标发生了偏离，需要调整（调度）来让系统往既定的目标靠近

机制与策略(mechanism and policy)

- 操作系统中的一个重要设计思想：机制与策略的分离（Separation of mechanism and policy），这种设计是一个典型的模块化思想，可以有效降低系统的复杂度
  - 策略表示“可以做什么” 
  - 机制表示“怎么做” 
- 在一个已有的机制上，可以考虑的问题是，有哪些策略？有没有“最优”的策略?
- 在给定一个策略的基础上，可以考虑的问题是，实现这个策略需要什么样的机制?已有机制是否具备这个能力？实现效果是否高效?

机制与策略(mechanism and policy)

Mechanism

```
/** Context-switch Mechanism with a job queue*/
Task *current; //当前运行task
Task ptable[] = { {...}, {...}, {...}}; //系统中所有的进程

Context *on_interrupt(Event ev, Context *ctx) {
  if (!current) {
    current = &ptable[0];  // First trap
  } else {
    current->context = ctx; // saving context
    current = selectone(ptable); //selecting a new current
  }
  return current->context; //restoring the new task to be executed
}
```

Policy

```
Task* select_one(Task pt[]){
    Task* result;
    /* specific scheduling policy*/
    return result;
}
```

机制与策略(mechanism and policy)

- 比如我们有了context switch的机制，再维护一个PCB（TCB）的队列，当发生一个时间中断时，就可以从这个队列中挑选一个进行执行
- 不同的挑选的策略都可以在这个机制上运行
  - 随机的调度
  - 按照队列的先后顺序依次调度，执行完之后再放回队尾
  - 我们还可以按照队列的反向顺序调度（类似栈）...

一个问题就是在这个机制下，最好的调度策略是什么？（基于某个评价指标）

机制与策略(mechanism and policy)

- 但如果我们想实现一个按照进程的执行所需时间进行调度的策略，之前的机制就不行了
  - 我门没有每个进程所需要的执行时间
  - 因此，需要一个新的机制，每个进程维护一个其所需执行时间的变量，使得内核可以访问
    - 显然不现实！
    - 更现实的机制是什么？

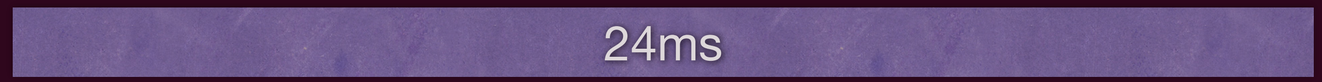
调度策略

- 调度策略大体可以分为如下几类任务
  - 批处理任务的调度
  - 交互性任务的调度
  - 实时任务的调度

批处理任务的调度

先来先服务（First Come First Serve, FCFS）

- 按照到达系统（就绪）的先后顺序进行调度，也叫先进先出(First In First Out, FIFO)
- 例子：
  - P1
  - P2
  - P3

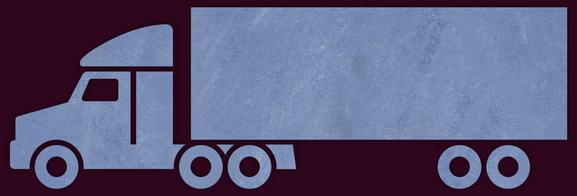


调度结果：

等待时间： P1 = 0， P2 = 24ms, P3 = 27ms。 平均等待时间： 17ms

先来先服务（First Come First Serve, FCFS）

- 护航效应(convoy effect)：短运行时间的进程排在长运行时间的后面，导致平均等待时间过长的现象。



运行时间长的任务

运行时间短的任务

先来先服务-护航效应

- P1 24ms
- P2 3ms
- P3 3ms

平均等待时间： 17ms

- P1 3ms
- P2 24ms
- P3 3ms

平均等待时间： 10ms

最短任务优先（Shortest Job First，SJF）

- 将每个任务与其需要运行时间关联
- 需要运行时间最短的优先被调度

最短任务优先（Shortest Job First，SJF）

- 例子：
  - P1: 6ms
  - P2: 8ms
  - P3: 7ms
  - P4: 3ms

调度结果：

等待时间： P4 = 0， P1 = 3ms, P3 = 9ms，P4 = 16ms。 平均等待时间： 7ms

命题：就平均等待时间而言，SJF是最优的

命题：就平均等待时间而言，SJF是最优的

- 给定：
  - 有 N 个进程
  - t_k 是进程k的执行时间
  - W_k是进程k的等待时间

命题：就平均等待时间而言，SJF是最优的

- 证明思路:
  - 给定一任意进程排列顺序S的平均等待时间
  - 给出一个给出将该序列变为按执行时间降序排列的过程
  - 证明该过程是单调的，即每一步都会导致平均等待时间变长

命题：就平均等待时间而言，SJF是最优的

- 证明:

1

  - 假定任意序列$S = (1, 2, 3, 4, \ldots, N)$
  - 由于进程k在进程$k-1$之后执行，我们可以得出：$W_k = W_{k-1} + t_{k-1}$ (Eq_1)
  - 第1个进程的等待时间为0，即$W_1 = 0$
  - 序列S中所有的进程的平均等待时间即为：
    - $W_S = \dfrac{W_1 + W_2 + \ldots + W_N}{N}$ (Eq_2)
  - 将Eq_1带入Eq_2得：$W_S = \dfrac{(N-1)t_1 + (N-2)t_2 + \ldots + (N-k)t_k + \ldots + t_{N-1}}{N}$ (Eq_3)

命题：就平均等待时间而言，SJF是最优的

- 证明:

2

- 考虑如下过程（冒泡排序）：
  - 遍历序列S中的进程:
    - 依次比较两个相邻进程的执行时间。
    - 如果两个进程中前一个的执行大于后一个的进程时间,则调换这两个进程的调度顺序。
  - 反复迭代这个遍历过程直到没有交换发生停止。

每遍历一遍，执行时间最大的会排到序列最后

该迭代最终生成一个和SJF顺序一致的调度序列

命题：就平均等待时间而言，SJF是最优的

- 证明:

- 对于进程序列$S = (1, 2, 3, 4, \ldots, N)$，任意挑选两个相邻进程，如$k-1$和$k$，假设$t_{k-1} > t_k$，调换这两个进程的调度顺序，则：

原来 $$W_S = \frac{(N-1)t_1 + \ldots + (N-(k-1))t_{k-1} + (N-k)t_k + \ldots + t_{N-1}}{N} \qquad (Eq_3)$$

转变为

现在 $$W_{S'} = \frac{(N-1)t_1 + \ldots + (N-(k-1))t_k + (N-k)t_{k-1} + \ldots + t_{N-1}}{N} \qquad (Eq_4)$$

命题：就平均等待时间而言，SJF是最优的

- 证明:

3

  - 对两个时间做差：

$$W_S - W_{S'} = (N-(k-1))t_{k-1} + (N-k)t_k - (N-(k-1))t_k - (N-k)t_{k-1}$$

$$= t_{k-1} - t_k$$

  - 由于 $t_{k-1} > t_k$，因此 $W_S - W_{S'} > 0$

  - 每调整一步（向SJF靠近），平均等待时间都会减少！

证毕

最短剩余时间优先（Shortest Remaining Time First, SRTF）

- 一个**非抢占式**调度算法选择一个进程来运行，然后就让它一直运行，直到它被阻塞（无论是在I/O操作上还是等待另一个进程），或者自愿释放CPU。
- 一个**抢占式**调度算法选择一个进程，并允许其运行一段固定的最长时间。如果在时间间隔结束时它仍在运行，则被挂起，调度器选择另一个进程来运行。（需要时间中断的机制支持）
- 最短任务优先的可抢占（**Preemptive**）版本
  - 始终选择**剩下**需要时间时间最短的进程进行运行！

最短剩余时间优先（Shortest Remaining Time First, SRTF）

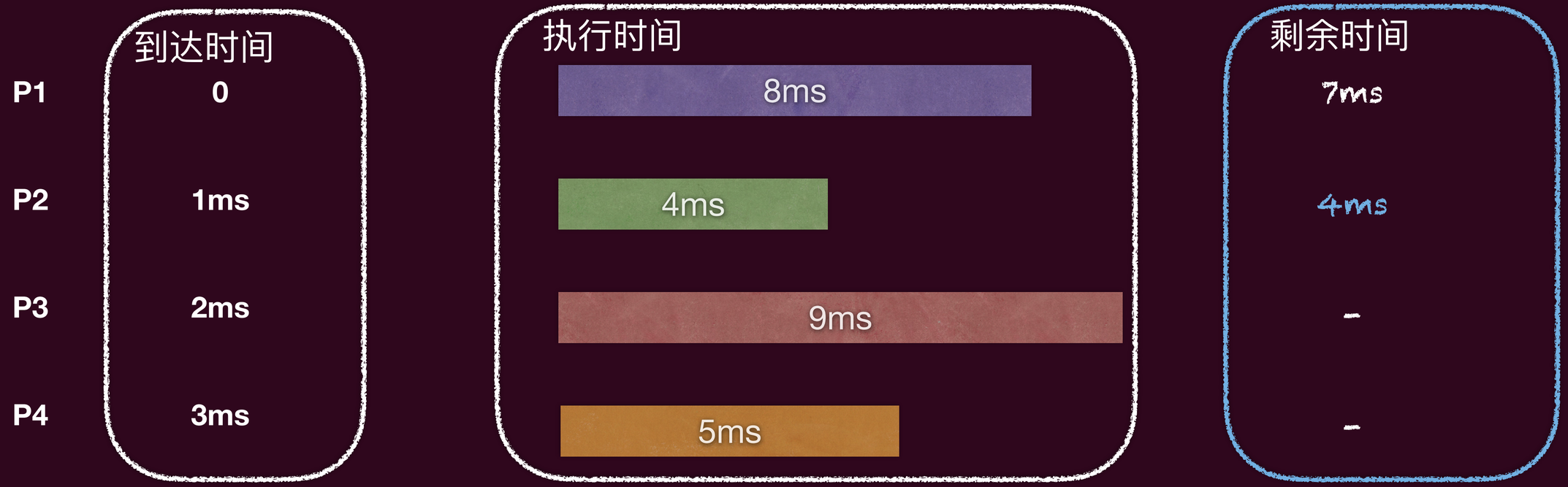
- 例子：

	到达时间	执行时间	剩余时间
P1	0	8ms	8ms
P2	1ms	4ms	-
P3	2ms	9ms	-
P4	3ms	5ms	-

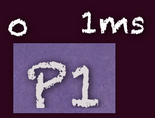
调度结果：

最短剩余时间优先（Shortest Remaining Time First, SRTF）

- 例子：



调度结果：



最短剩余时间优先（Shortest Remaining Time First, SRTF）

- 例子：

	到达时间	执行时间	剩余时间
P1	0	8ms	7ms
P2	1ms	4ms	3ms
P3	2ms	9ms	9ms
P4	3ms	5ms	-

调度结果：

最短剩余时间优先（Shortest Remaining Time First, SRTF）

- 例子：

	到达时间	执行时间	剩余时间
P1	0	8ms	7ms
P2	1ms	4ms	2ms
P3	2ms	9ms	9ms
P4	3ms	5ms	5ms

调度结果：

最短剩余时间优先（Shortest Remaining Time First, SRTF）

- 例子：

	到达时间	执行时间	剩余时间
P1	0	8ms	7ms
P2	1ms	4ms	0ms
P3	2ms	9ms	9ms
P4	3ms	5ms	5ms

调度结果：

最短剩余时间优先（Shortest Remaining Time First, SRTF）

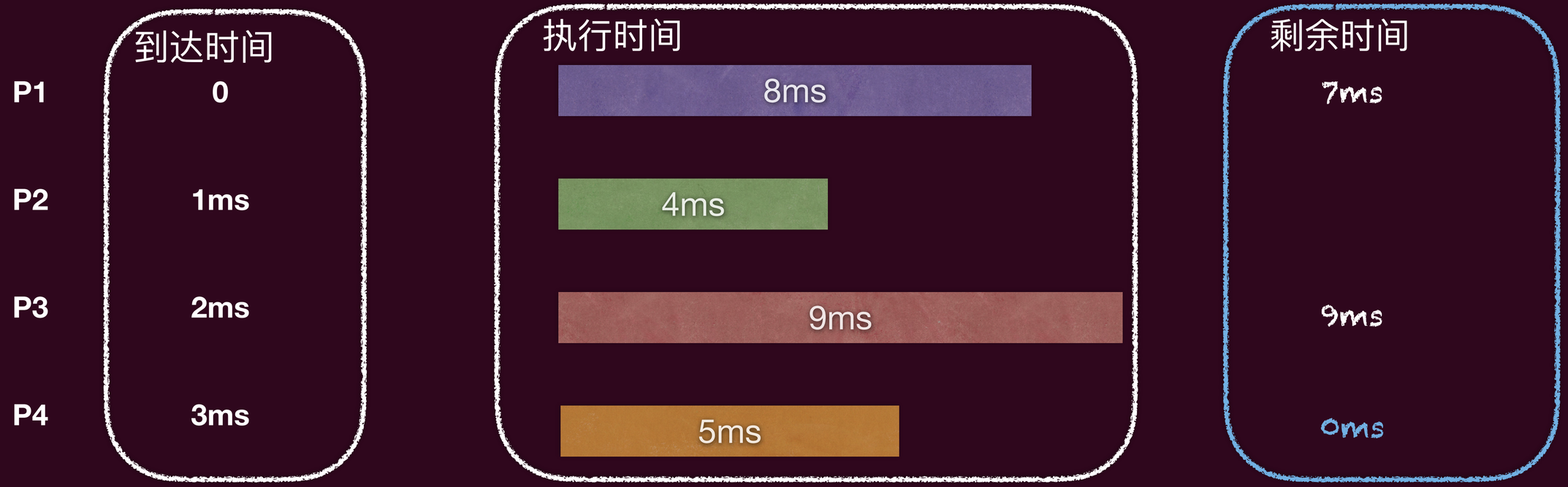
- 例子：

	到达时间	执行时间	剩余时间
P1	0	8ms	7ms
P2	1ms	4ms	
P3	2ms	9ms	9ms
P4	3ms	5ms	5ms

调度结果：

最短剩余时间优先（Shortest Remaining Time First, SRTF）

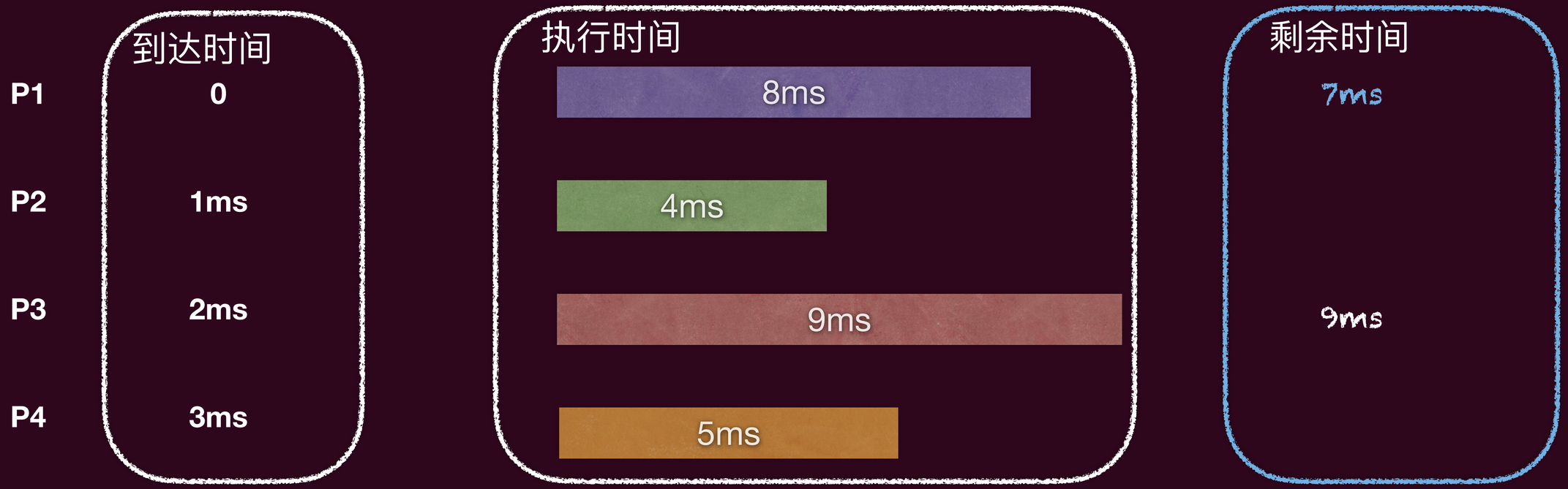
- 例子：



调度结果：

最短剩余时间优先（Shortest Remaining Time First, SRTF）

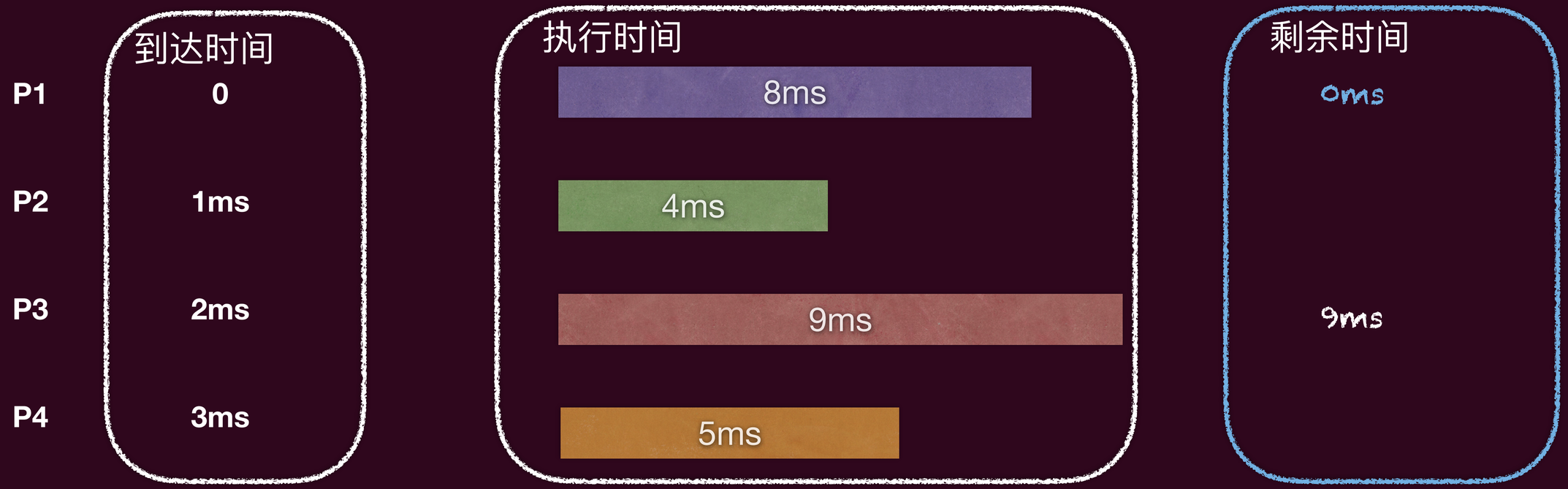
- 例子：



调度结果：

最短剩余时间优先（Shortest Remaining Time First, SRTF）

- 例子：



调度结果：

最短剩余时间优先（Shortest Remaining Time First, SRTF）

- 例子：

	到达时间	执行时间	剩余时间
P1	0	8ms	
P2	1ms	4ms	
P3	2ms	9ms	9ms
P4	3ms	5ms	

调度结果：

最短剩余时间优先（Shortest Remaining Time First, SRTF）

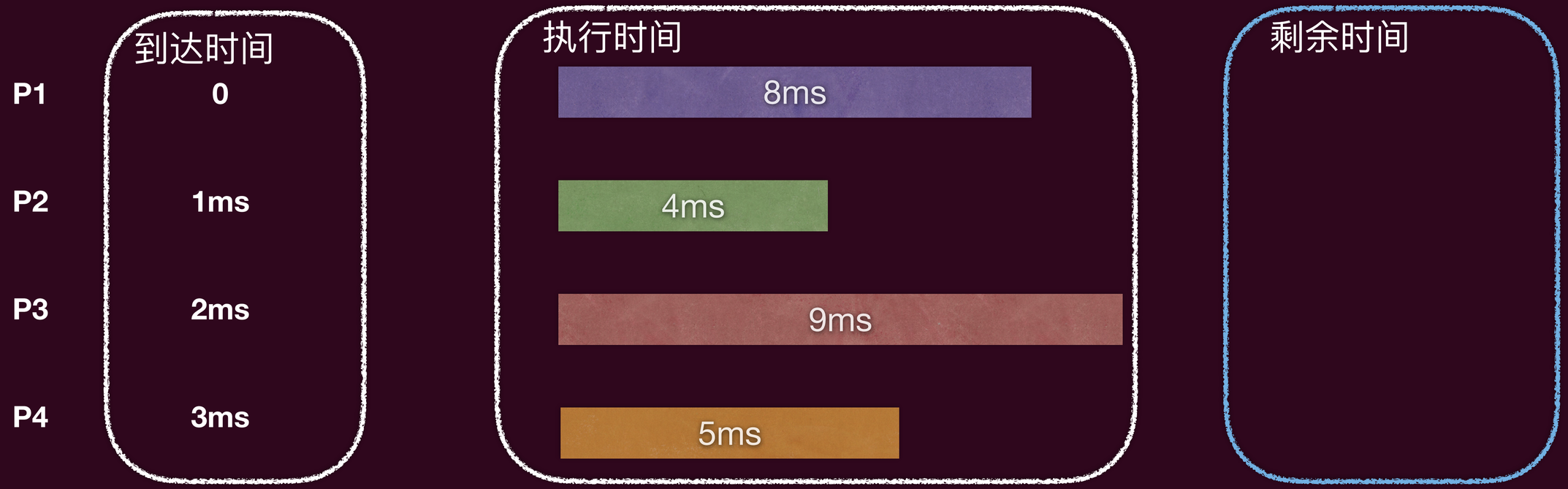
- 例子：

	到达时间	执行时间	剩余时间
P1	0	8ms	
P2	1ms	4ms	
P3	2ms	9ms	0ms
P4	3ms	5ms	

调度结果：

最短剩余时间优先（Shortest Remaining Time First, SRTF）

- 例子：



调度结果：

平均等待时间：$[(10-1)+(1-1)+(5-3)+(17-2)]/4 = 6.5ms$

时间预测算法 (exponential moving average)

- 思想：下一次运行时间应该与之前一次类似
  - t_n ： 第n次实际运行时间
  - τ_{n+1} :进预测下一次（$n+1$）的执行时间
  - α：相关系数
  - $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$

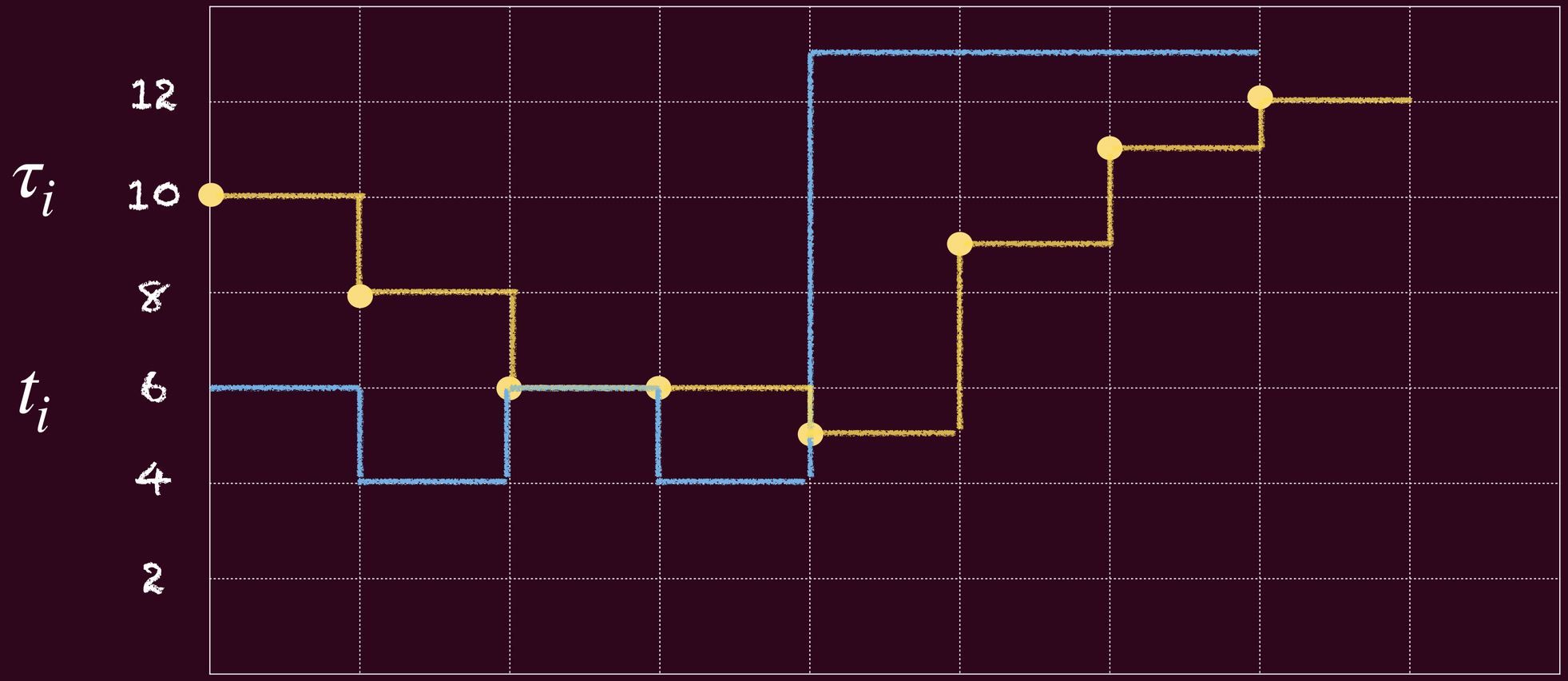
时间预测算法 (exponential moving average)

- 将$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$展开
  - $\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2 \alpha t_{n-2} + \ldots + (1-\alpha)^j \alpha t_{n-j} + \ldots + (1-\alpha)^{n+1}\tau_0$
- 由于α和$1-\alpha$都小于等于1，因此该多项式中每一项都比后一项的权重更大
  - 这也就意味最近的执行时间更加影响当前的预测，而更之前的执行时间则影响较小

时间预测算法 (exponential moving average)

- 两个极限情况：
  - 当$\alpha = 0$ 那么 $\tau_{n+1} = \tau_n = ... = \tau_0$意味着最近的执行历史没有关联
  - 当$\alpha = 1$ 那么 $\tau_{n+1} = \alpha t_n$意味着仅仅和最近的历史相关
- 一般而言，α为0.5

时间预测算法 (exponential moving average)



实际运行	t_i		6	4	6	4	13	13	13
猜测	τ_i	10	8	6	6	5	9	11	12

SJF的一些缺陷

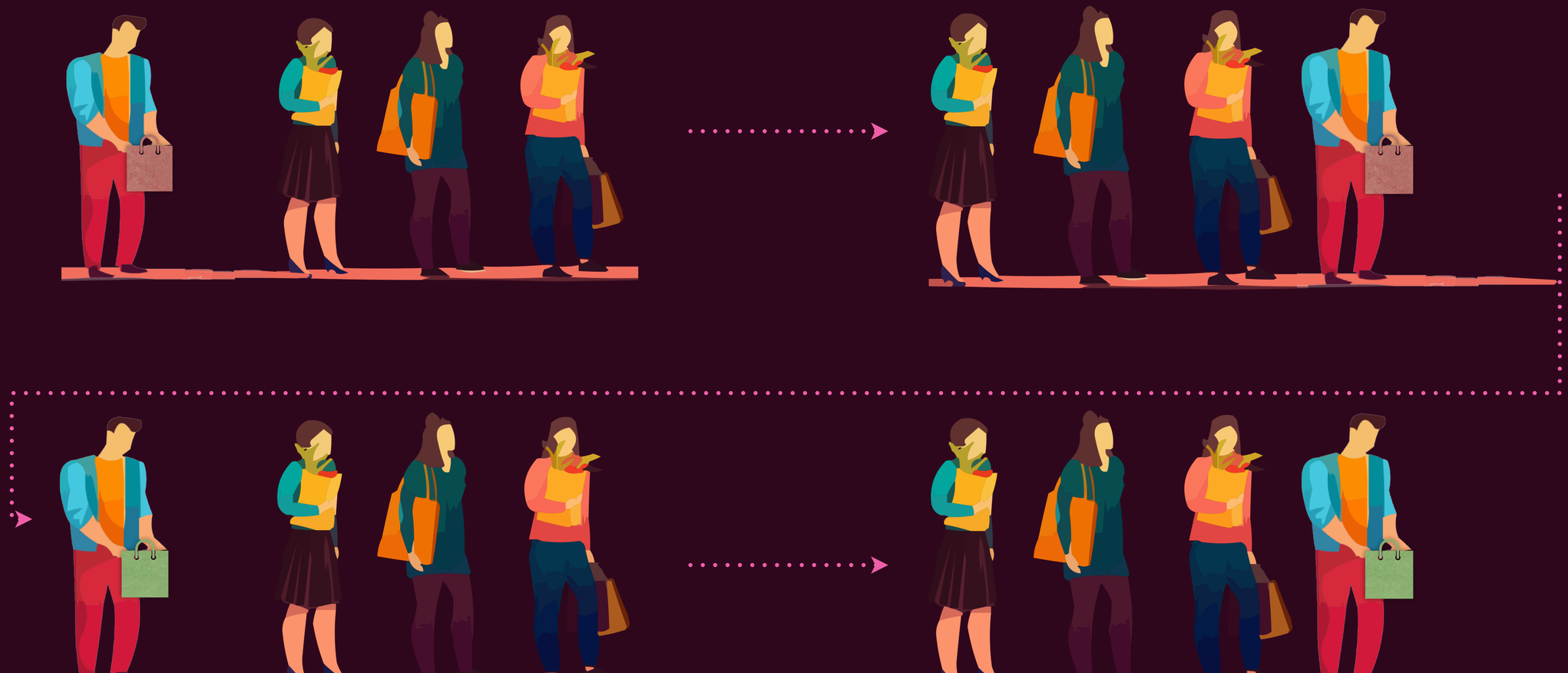
- 并不公平！ 
- 等待时间差异大：进程饿死！ 
- 进程的需要运行时间难以给定！ 
- 以及，可以轻易被愚弄（gaming）！ 

我们可以尝试愚弄（Gaming）调度

- 将自己切割成很多份“非常”短执行时间的小任务，然后再批量运行这些小任务

我们可以尝试愚弄（Gaming）调度

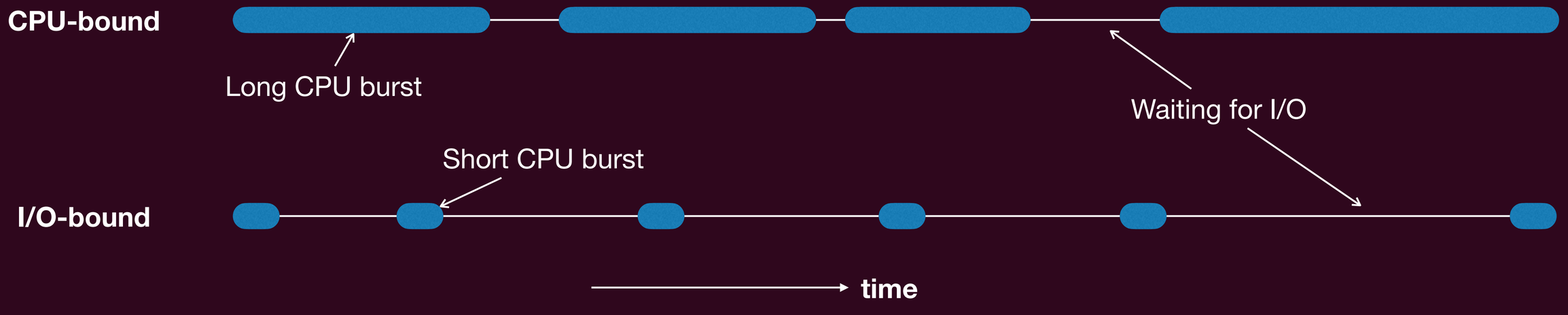
- 将自己切割成很多份“非常”短执行时间的小任务，然后再批量运行这些小任务



交互性任务的调度

计算密集型（CPU bound）和I/O密集型(I/O bound)

- 在调度策略中，有两类进程会被区别对待：计算密集型和I/O密集型 （当然也有混合型）
  - CPU密集型程序主要消耗CPU计算资源，例如数学运算、图形处理或数据分析等任务。这些程序通常会在 CPU 上执行大部分时间的计算和逻辑判断等操作，而不需要等待外部资源（如磁盘读写或网络通信）完成。
  - I/O 密集型指的是系统大部分的时间在等待 I/O（硬盘/内存/键盘）的读取/写入操作（即和外界进行频繁交互的进程），此时 CPU 负载并不高，需要消耗CPU计算的时间很少



时间片轮转调度(Round-Robin, RR)

- 每个任务都会获得一段固定时间的资源（时间片, time-slicing）。
  - 如果任务没有完成，它将重新回到队列中。
- 时间片应该相对于上下文切换时间较大，否则开销会太高！当然时间片也不能过大，否则就蜕变为FCFS调度了
  - 一般来说时间片大概设置为10ms到100ms (context switch一般小于10 microseconds)

RR在时间片为4ms的样例

- 例子：
  - P1: 8ms
  - P2: 5ms
  - P3: 3ms

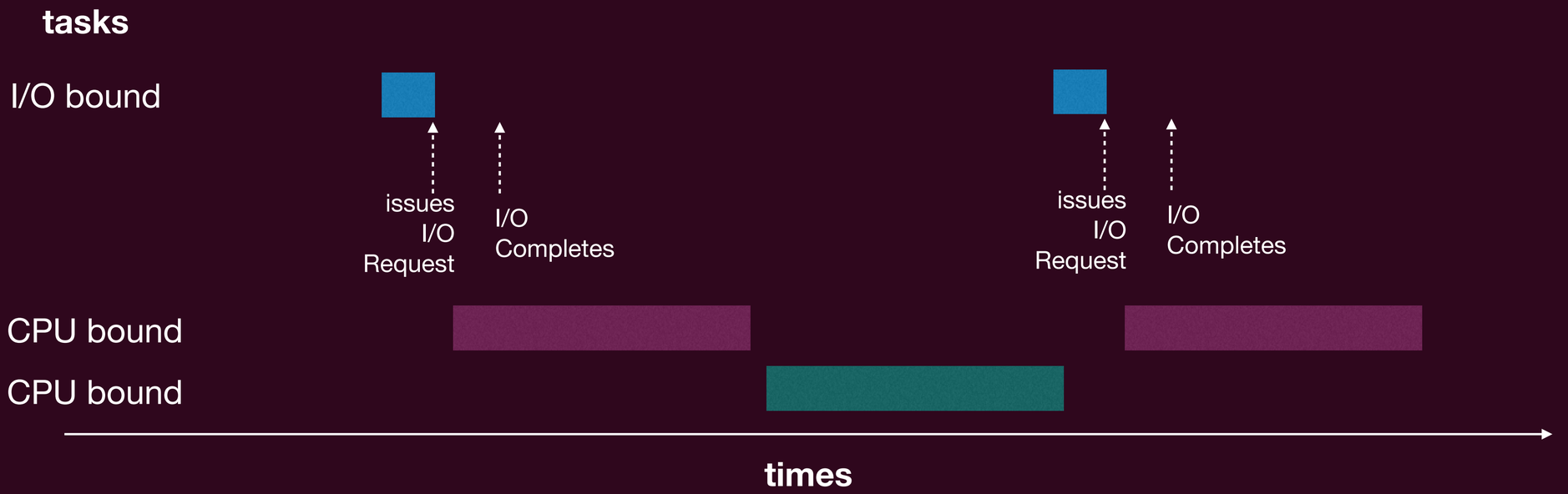
调度结果：

Round Robin vs. FIFO and SJF

- RR: 平均周转时间较长，并且有较高的overhead!

RR的另外一个问题

- 在运行既有 I/O 密集型任务又有计算密集型任务的情况下，当 I/O 密集型任务执行 I/O 操作时，它会让出处理器（由于需要的CPU计算很短，没有用完时间片就让出CPU了）。
  - 这时即使 I/O 操作很快完成（比如你正在用VIM打字），也必须等待重新分配处理器，直到其他计算密集型任务用完他们完整的CPU切片。

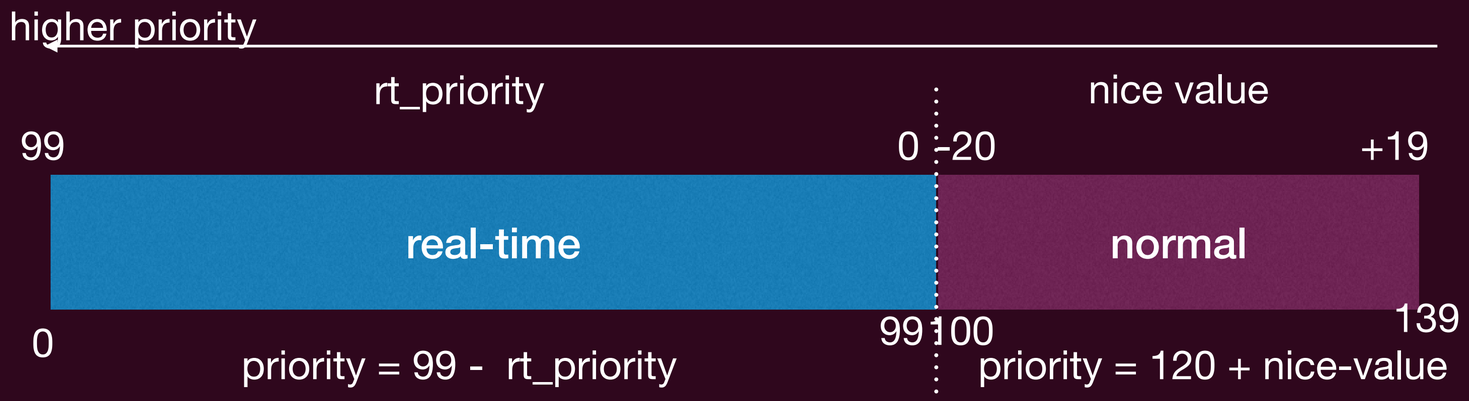


基于优先级的调度（Priority Scheduling）

- I/O-bound和CPU-bound的优先级不同，我门需要基于优先级的调度：
  - 每个进程都关联有一个优先级数（整数）。
  - 每次发生调度时，CPU被分配给具有最高优先级的进程。
- SJF（Shortest Job First）就可以看成是一种优先级调度算法，其中每个进程的优先级与预测其下一个CPU执行时间的倒数成正比。

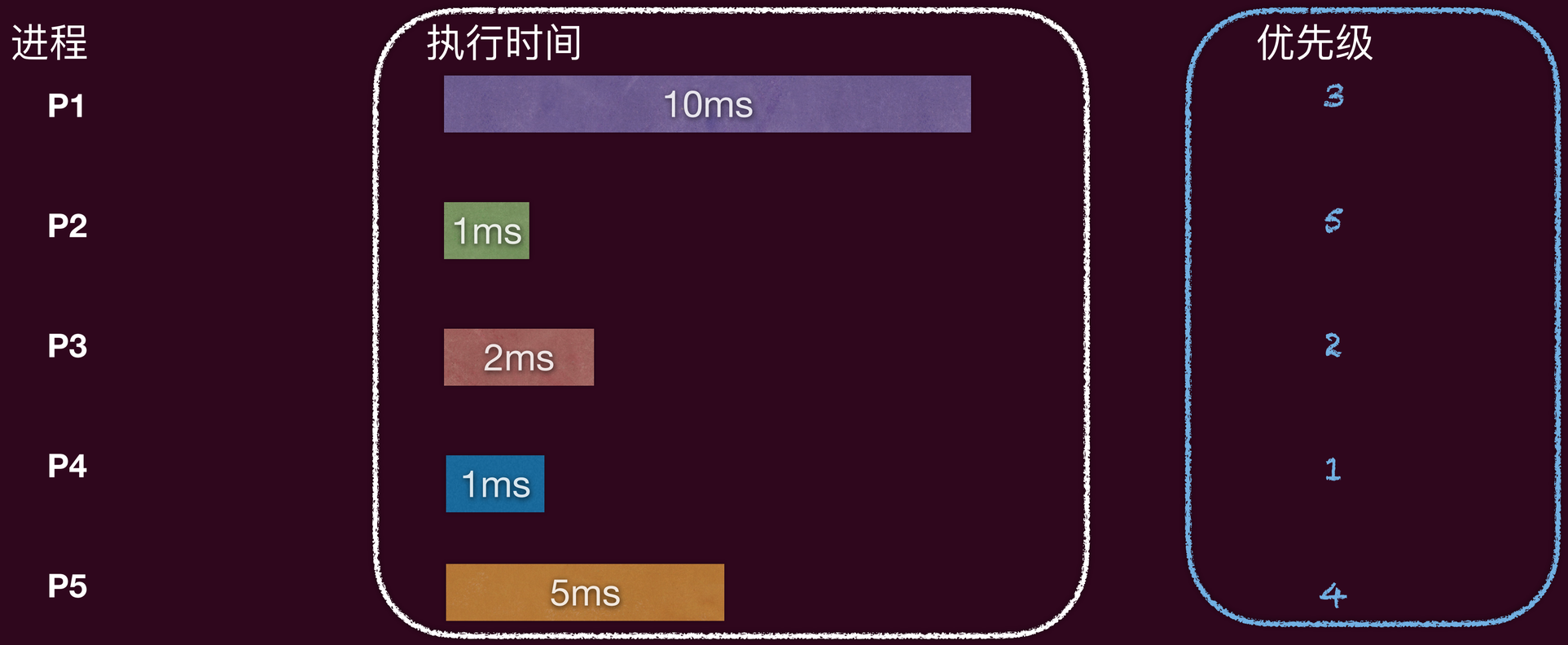
基于优先级的调度（Priority Scheduling）

- Linux下默认的优先级(转换之后)：
  - 实时任务(real time): 高优先级 (0 - 99)
    - 试试`chrt`命令： chrt -r -p priority PID
  - 普通任务(normal ones): 低优先级 (100 - 139)
    - 试试`renice`命令 renice [-n] priority [-g|-p|-u] identifier
    - 试试`nice`命令 nice [option] [command [ARG]…]



优先级的调度例子

- 给定如下workload, 假设优先级数字越大优先级越高



调度结果：

平均等待时间： 8.2ms

优先级调度的问题

- **饿死(Starvation)** – 低优先级的进程可能永远不会执行。
  - 解决方案： 老化（Aging)–随着时间的推移增加进程的优先级。
    - 需要一种能动态改变优先级的策略，如：多级反馈队列（**multilevel feedback queue**）

多级反馈队列（Multilevel Feedback Queue, MFQ）

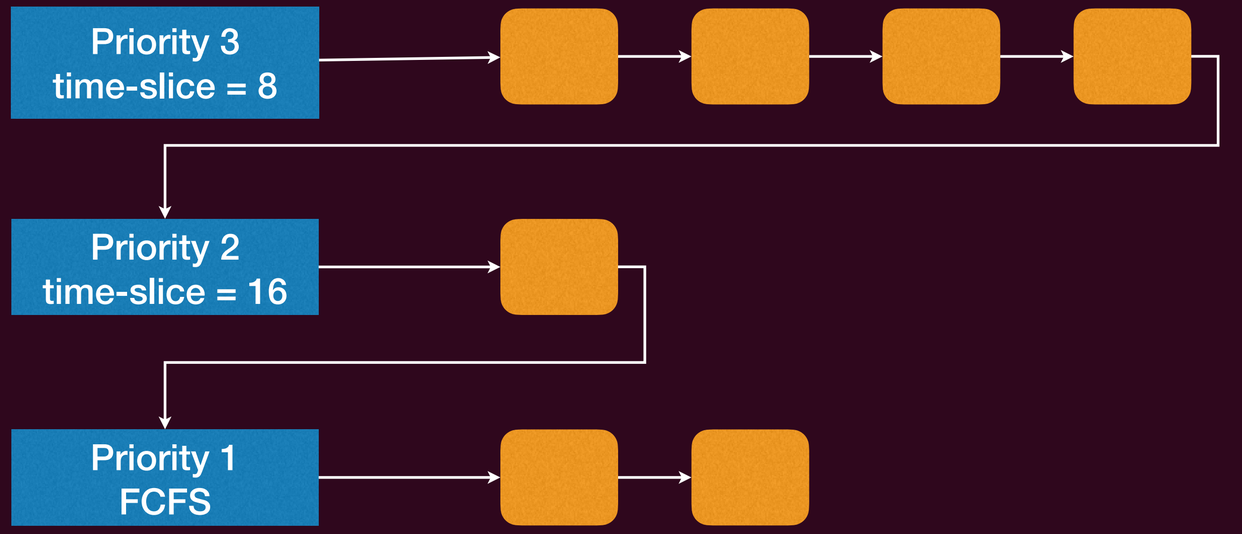
- 一个进程可以在各个队列（代表不同优先级）之间移动。
- 多级反馈队列调度器由以下参数定义：
  - 队列的数量
  - 每个队列的调度算法
  - 确定何时将进程提升优先级的方法
  - 确定何时将进程降优先级的方法
  - 确定当某个进程需要服务时该将进入哪个队列的方法

多级反馈队列（Multilevel Feedback Queue, MFQ）

- 一个典型的多级反馈队列：
  - 一组轮转队列：
    - 每个队列都有单独的优先级。
  - 高优先级队列拥有短的时间片。
    - 低优先级队列拥有长的时间片。
  - 调度器选择最高优先级队列中的第一个进程。
  - 进程加载到内存中时初始在最高优先级队列中。
  - 如果时间片到期，任务会降低一个级别。

多级反馈队列（Multilevel Feedback Queue, MFQ）

- 例子： 三个优先级队列：
  - Priority 3：轮转调度，时间片为8毫秒
  - Priority 2：轮转调度，时间片为16毫秒
  - Priority 1：先进先出（FCFS）可视为时间片无穷大
- 调度过程如下：
  - 新进程进入队列Priority 3（最高优先级），以轮转调度的方式服务。
  - 当获得CPU时，该进程获得8毫秒的时间片。
  - 如果在8毫秒内未完成，则将进程移动到队列Priority 2。
  - 在Priority 2中，作业再次以轮转调度的方式服务，并获得额外的16毫秒。
  - 如果仍然没有完成，则被抢占并移动到队列Priority 3，进行FCFS调度



多级反馈队列（Multilevel Feedback Queue, MFQ）

- 当使用多级反馈队列（MFQ）时：
  - CPU密集型进程将下沉到长时间片的优先级队列。
    - 如果使用完时间片，进程会下降一个优先级。
    - 较大的时间片可以减少上下文切换的开销。
  - I/O 密集型进程将保持在高优先级队列中。
    - 如果一个进程没有完成其时间片（即，它在 I/O 操作上被阻塞），那么它将保持在相同的优先级水平。

多级反馈队列（Multilevel Feedback Queue, MFQ）

- 多级反馈队列仍然存在饥饿问题：
  - 如果有大量交互式进程或者频繁创建新进程，则高优先级队列中始终有可用任务。此时，低优先级队列中的 CPU 密集型进程将永远不会被调度。
- 一个相关的问题是，一个交互式进程可能最终会处于低优先级水平。
  - 如果一个进程的某个时期变得 CPU 密集型，它就会降到低优先级水平，而且注定会永远留在那里。一个例子是一个游戏需要花费大量 CPU时间来初始化，但随着初始化完成，就变为了等待玩家输入的交互式进程 （此调度下的游戏体验可想而知）
- 因此，MFQ需要一个策略（老化）来定期增加进程的优先级，以确保它会被调度运行。一个简单的方法是定期将所有进程提升到最高优先级队列，即重置

愚弄（Game）系统

- MFQ调度器可以在周转时间、低开销和公平性之间取得平衡。 事实上，它被大多数商业和现实操作系统（包括Windows、MacOS和Linux）使用（作为一种策略）
- 但这个MFQ还是可以被愚弄的 
  - 比如一个恶意攻击者了解调度器的工作原理，那么他/她可以编写代码，在时间片到期之前强制系统在某些低延迟的 I/O 操作上阻塞（例如，睡眠几毫秒）
  - 这样，即使进程反复消耗大部分时间片，它也会因为没有使用完时间片而受到奖励（不会被降低优先级），从而始终保持在高优先级水平。

愚弄（Game）系统

- 解决方案 — 追踪：
  - 不仅仅是简单地检查进程是否使用完了它的时间片，调度器会跟踪进程在较长时间间隔（几个时间片）内运行的总时间。
  - 每个优先级队列将有一个与之关联的最大CPU时间分配。
    - 超出实用的分配则降低优先级

乐透（Lottery）调度

- 有的时候不能简单的优先级高的一定先执行，优先级低的一定等优先级高的执行完再执行
  - 更加希望的是：两者获得的CPU时间上呈一定精确的比例！高优先级的占比高一点，低优先级的占比低一点
- 一个灵活的调度算法：乐透算法
  - 给每个作业分配一定数量的彩票票数
  - 然后随机选择一个中奖票（拥有更多票数的作业有更大的中奖机会）。
  - 为了避免饥饿，至少给每个作业分配一张彩票。

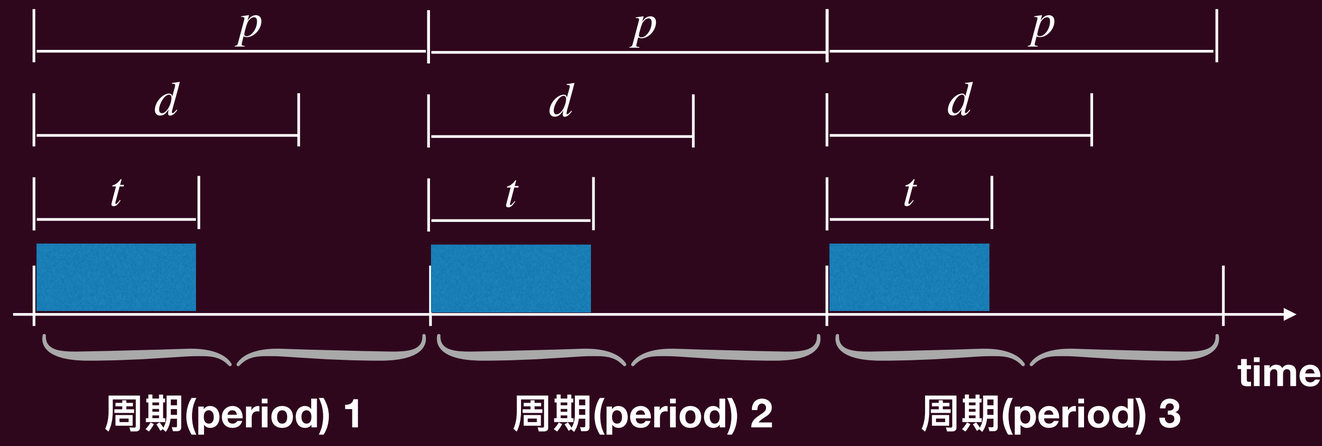
实时任务的调度

实时系统的调度

- 实时系统中时间扮演着至关重要的角色
  - 必须在截止日期（deadline）之前得到服务；
  - 截止日期过期后才得到服务与根本没有服务一样。
- 周期性（Periodic）
  - 截止日期以规律的间隔发生。

实时系统的调度

- 周期性进程的每个周期（本门课只考虑简单的固定周期任务）：
  - 计算时间t, 截止日期d, 时间周期p
    - $0 \leq t \leq d \leq p$
  - 周期性任务的执行速率为$\frac{1}{p}$



- 准入控制（Admission-control):
  - 给定m个周期性进程，每个进程的周期时间为P_i和周期内的计算时间为C_i，那么它满足以下条件时可调度（一般默认截止日期就是周期的时间）：$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$

单调速率(Rate Monotonic)调度

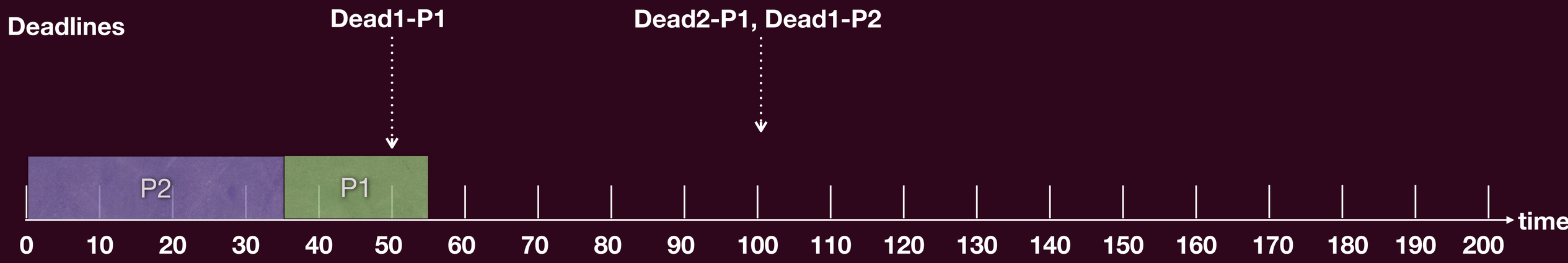
- 根据其速率（即周期的倒数）分配优先级。
  - 周期较短的任务具有较高的优先级，
  - 而周期较长的任务具有较低的优先级。
    - 例如，有两个进程P1和P2，它们的周期分别是50和100，那么P1首先被调度，然后是P2，P1可以抢占P2。
- 这种策略背后的理念是为需要更频繁占用CPU的任务分配更高的优先级。

单调速率(Rate Monotonic)调度

- 例子： 我们有两个进程：P1和P2。
  - P1的周期为50，处理时间为20，
  - P2的周期为100，处理时间为35，
  - 截止期限都为下一个周期之前。
- 首先这个例子满足准入控制 $\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$
  - (20/50 + 35/100) < 1

单调速率(Rate Monotonic)调度

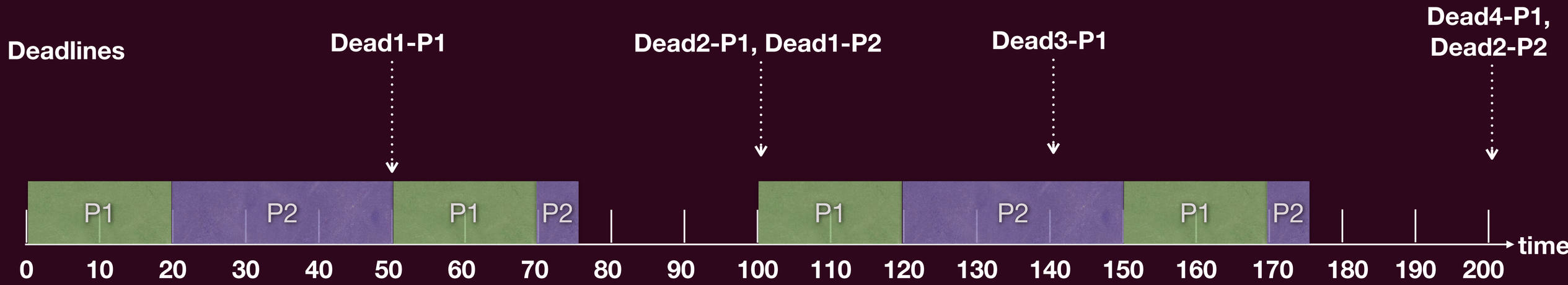
- P1 (计算时间20)和P2 (计算时间35)的速率分别为1/50和1/100， 如果我们不使用率单调调度，即P2首先被调度：



- P1 过了截止日期，调度失败

单调速率(Rate Monotonic)调度

- 如果我们按照单调速率进行调度，P1和P2都会调度成功



Claim: Rate-monotonic scheduling is considered **optimal** in that if a set of processes cannot be scheduled by this algorithm, it cannot be scheduled by any other algorithm that assigns static priorities.

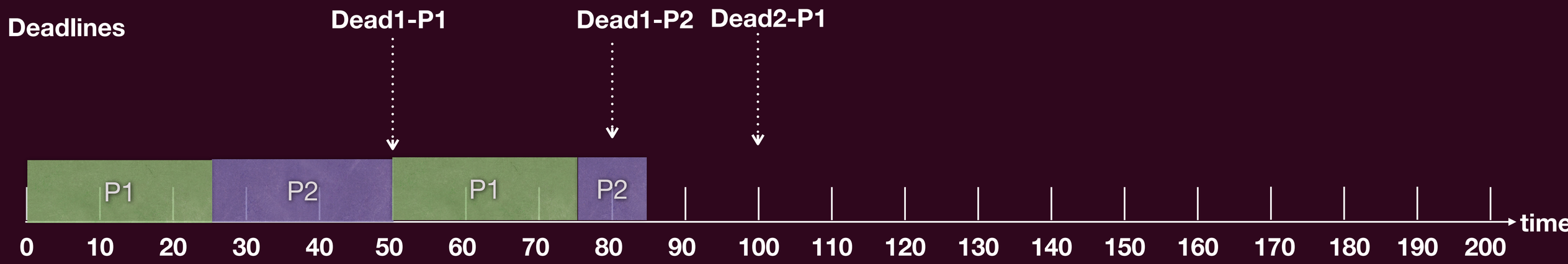
Liu C L, Layland J W. Scheduling algorithms for multiprogramming in a hard-real-time environment[J]. Journal of the ACM (JACM), 1973, 20(1): 46-61.

单调速率的失效

- 我们有两个进程：P1和P2。
  - P1的周期为50，处理时间为25，截止期限为下一个周期之前。
  - P2的周期为80，处理时间为35，截止期限也是在下一个周期之前。
- 首先，这个workload是符合准入控制的 $\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$
  - (25/50 + 35/80) = 0.9375 < 1

单调速率的失效

- 使用率单调调度，由于P1(计算时间25)和P2(计算时间35)的速率分别为1/50和1/80，所以P1（优先级大）首先被调度。



- 但 P2 在时间80处错过了截止日期

Claim: For a set of n real-time tasks with fixed priority, the least upper bound (LUB) to processor utilization factor is $U = n(2^{\frac{1}{n}} - 1)$

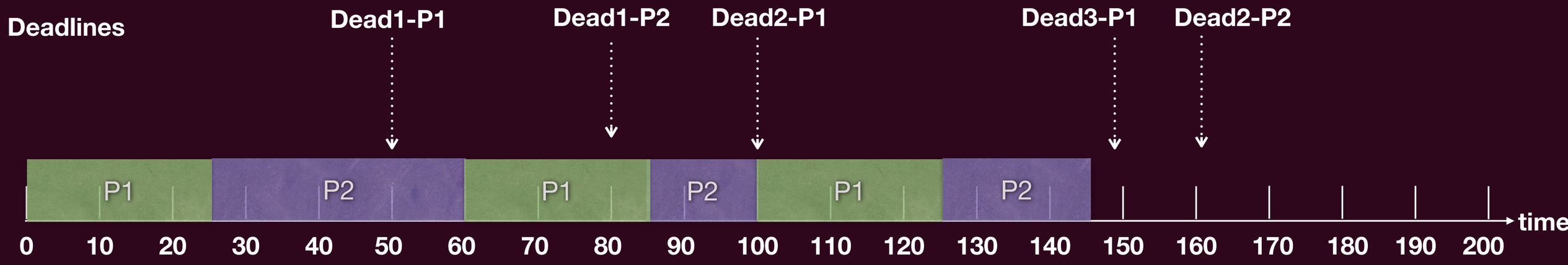
Liu C L, Layland J W. Scheduling algorithms for multiprogramming in a hard-real-time environment[J]. Journal of the ACM (JACM), 1973, 20(1): 46-61.

最早截止日期优先(Earliest Deadline First Scheduling ,EDF)

- 根据截止期限分配优先级：
  - 截止期限越早，优先级越高。
  - 截止期限越晚，优先级越低。
- 注意：这与率单调调度不同，率单调调度中优先级是固定的(周期不变)，而最早截止期限优先调度根据任务的截止期限调整优先级。

最早截止日期优先

- 前面的例子：两个进程P1和P2 周期：P1为50，P2为80， 处理时间：P1为25，P2为35， 截止期限：在下一个周期之前。



Theorem: EDF is an optimal scheduling algorithm on preemptive uniprocessors. With scheduling periodic processes, EDF has a utilization bound of 100%.

Buttazzo, G.C. Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. Springer, New York, 2005

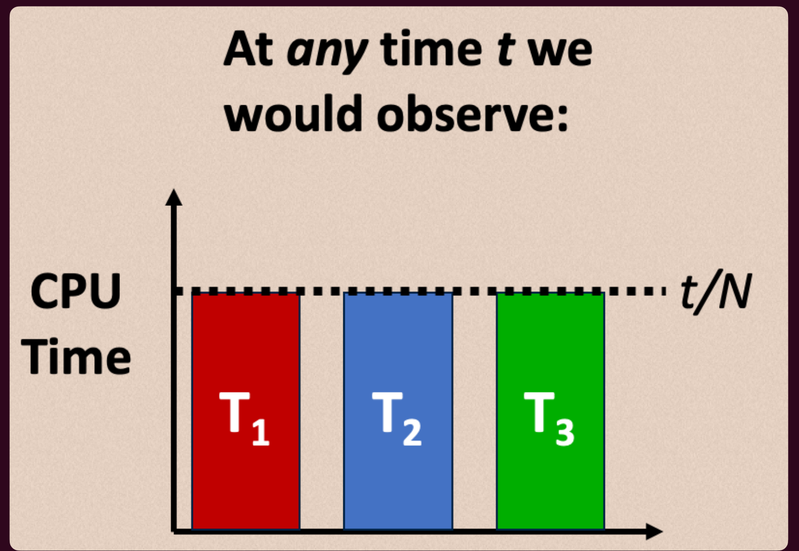
真实操作系统调度器

Linux 2.6.23 之后的调度策略

- 调度类别： 每个类别都有特定的优先级。
- 调度器选择**最高**调度类别中的**最高**优先级任务。
- 包括两个调度类别，其他类别可以添加：
  - 实时类别：FCFS，RR，EDF(在Linux 3.14中合并)
  - 普通类别：完全公平调度器（CFS），IDLE

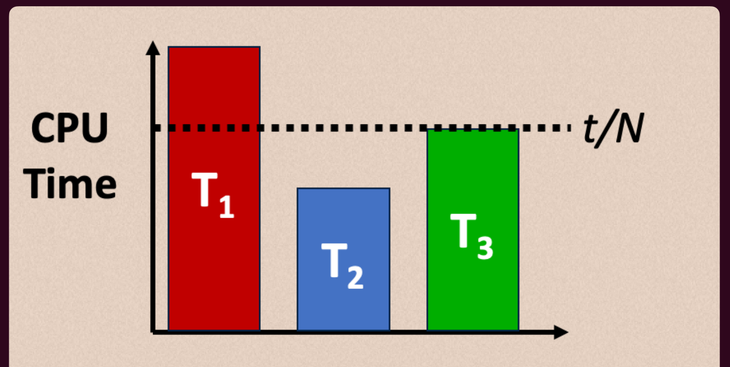
完全公平调度策略（Completely Fair Scheduler, CFS）

- 目标：每个进程获得相等份额的CPU时间（公平性）
  - N个线程下，每个线程在任意时刻获得等份的CPU时间t/N
  - 无法在实际硬件上实现这一点



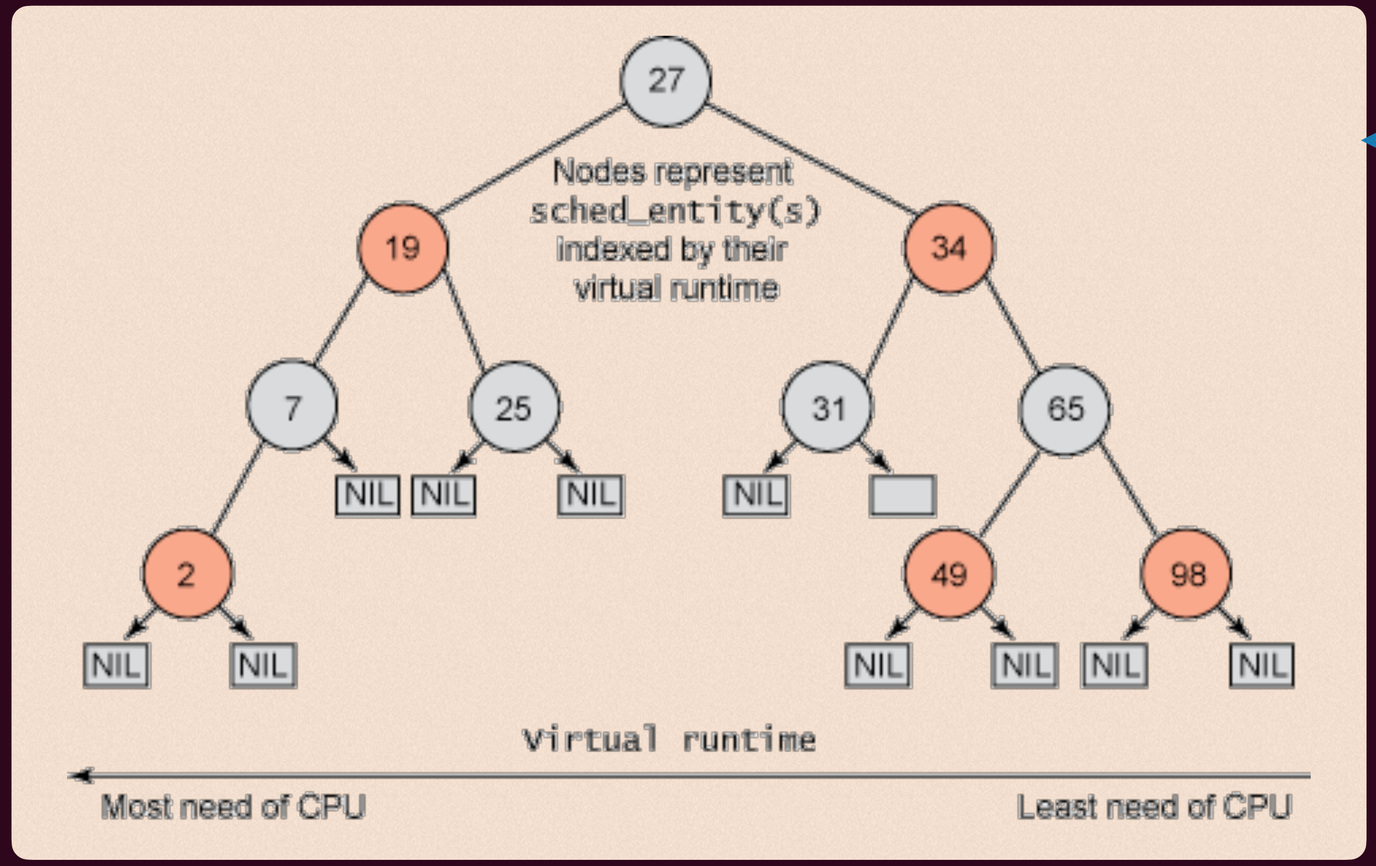
完全公平调度策略

- 更加实际的做法：不断跟踪到目前为止给予进程的CPU时间。
- 动态的根据每个进程当前已经给予的时间来进行调度决策：
  - 每次选择CPU使用时间最少的线程执行
  - 如果线程进入睡眠状态然后重新唤醒，则重置CPU使用时间为当前就绪的红黑树中“最小”的时间
    - 不然其使用时间会远远小于其他进程，那么调度器就会疯狂的给这个进程找补



完全公平调度策略

- 与之前版本的Linux调度器不同，CFS不是维护任务的运行队列，而是维护一个按时间排序的红黑树。

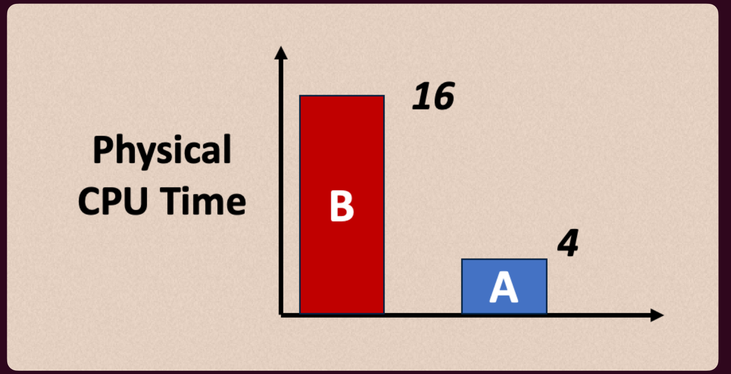


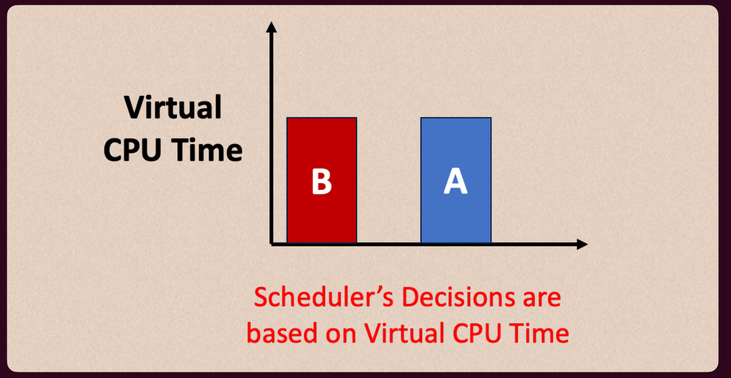
每次选择调度时只挑选数的最左边的叶子结点（时间最少），可以提供一个指针指向这个叶子，因此调度复杂度为$O(1)$

上下文切换时，根据这次执行的时长更新这个current进程的所获得运行CPU时间总长，然后根据其值，重新插入红黑树，时间复杂度$O(\log n)$

完全公平调度策略

- 如果有些进程优先级高，不想和其他进程公平
- 核心想法: 给每个进程i附上其权重w_i
  - Original Basic equal share: $Q = \frac{\text{CPU Time}}{N}$
  - Weighted Share: $Q_i = (\frac{w_i}{\sum_{j=1}^{j=N} w_j}) \times \text{CPU time}$
- 跟踪线程的虚拟运行时间(virtual runtime)而不是其真实的物理运行时间。
  - 较高的权重：虚拟运行时间增长更慢。
  - 较低的权重：虚拟运行时间增长更快。





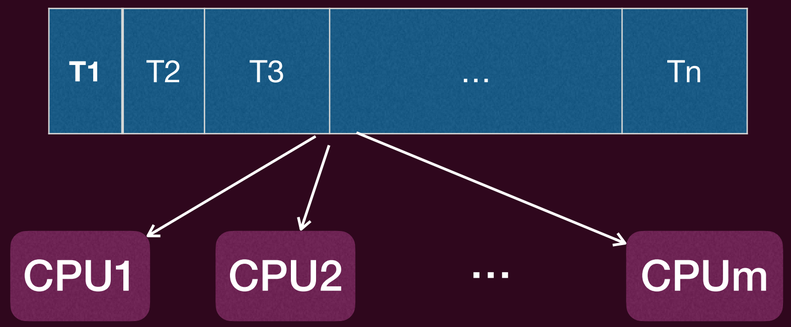
调度的一些其他问题*

多核CPU的调度

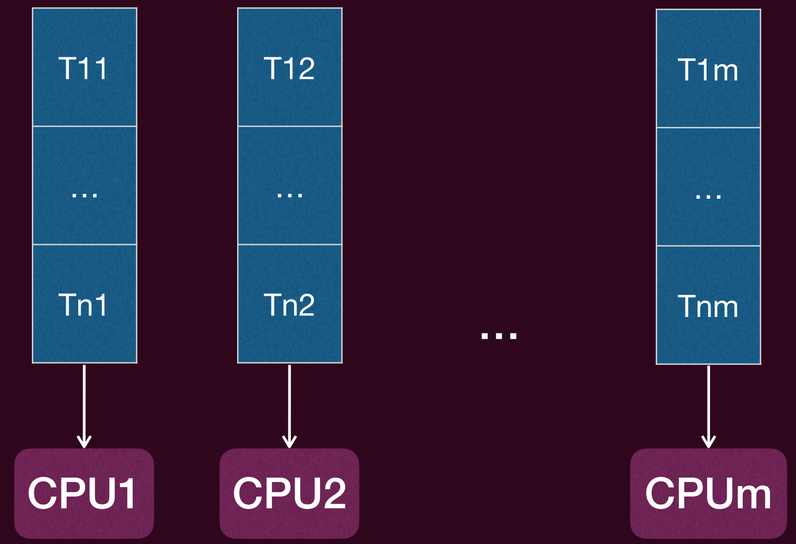
- 在多处理器系统中，CPU调度的一种方法是由单个处理器——主服务器处理所有调度决策，其他处理器仅执行用户代码。
  - 这种不对称多处理是简单的，因为只有一个核心访问系统数据结构，减少了数据共享的需求。
  - 这种方法的缺点是主服务器成为潜在的瓶颈，可能降低整个系统的性能。

多核CPU的调度

- 相反，对称多处理（ symmetric multiprocessing, SMP）是指每个处理器都进行自我调度, 不需要一个中央的调度服务器CPU。
- 几乎所有现代操作系统都支持SMP，包括Windows、Linux和macOS，以及包括Android和iOS在内的移动系统。
- 在多个核心下，对于进程的调度，有两种队列机制：



情况1: 一个统一的队列，多个CPU共享



情况2: 每个CPU维护自己的调度队列

多核CPU的调度

- 在多个核心下的调度底层机制也要做相应的改变

```
/** Context-switch Mechanism with a job queue*/
Task *current[MAXCPU]; //每个cpu都有一个当前运行task
#define current currents[cpu_current()]
Task ptable[MAXCPU][] = { { {...}, {...}, {...} } ,
                          { {...}, {...}, {...} },
                          ...
                        } ; //系统中所有的进程,


Context *on_interrupt(Event ev, Context *ctx) {
  if (!current) {
    current = &ptable[cpu_current()][0];  // First trap
  } else {
    current->context = ctx; // saving context
    current = selectone(ptable[current_cpu()]); //selecting a new current
  }
  return current->context; //restoring the new task to be executed
}
```

多核CPU的调度

- 如果选择第一种选项，我们在共享的就绪队列上存在竞争条件，需要加一把大锁，这显然低效。
- 第二种选项允许每个处理器从其私有的运行队列中调度线程，因此CPU之间不会受到可能的竞争条件问题的影响。
  - 此外，每个处理器都有私有的运行队列，实际上可能会导致对缓存内存的更有效利用（不然会存在多个CPU有重复的缓存，浪费资源）。
  - 因此，在支持SMP的系统中，这是最常见的方法。

负载均衡(Load balancing)

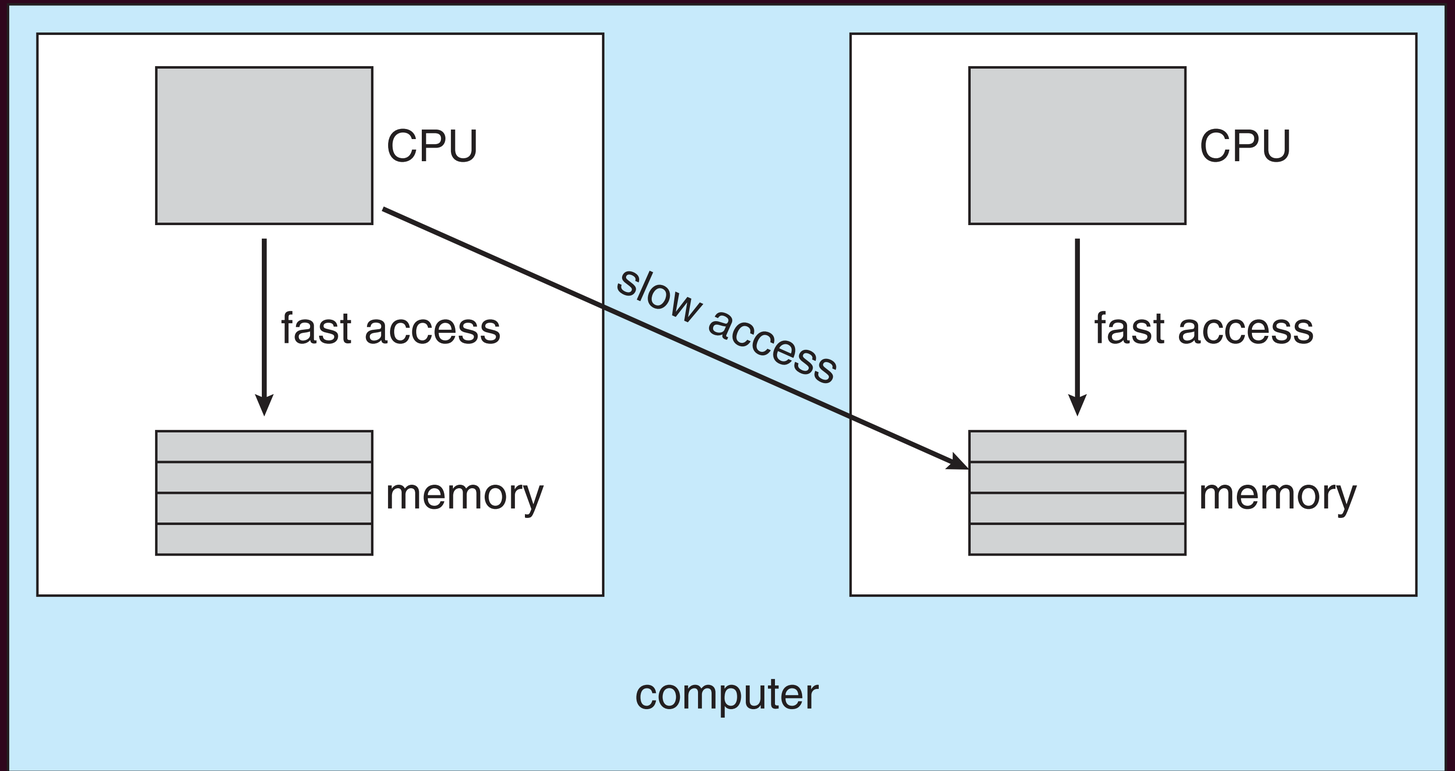
- SMP下需要保持所有CPU负载均衡以提高效率。
- 负载平衡尝试保持工作负载均匀分布，有两种方式：
  - 推迁移（Push migration）：周期性任务检查每个处理器的负载，如果发现过载，则将任务从过载的CPU推送到其他CPU上。
  - 拉迁移（Pull migration）：空闲处理器从繁忙处理器中拉取等待的任务

处理器亲和性(Affinity)

- 当一个线程在一个处理器上运行时，该处理器的缓存内容存储了该线程的内存访问。
  - 我们将这称为线程对处理器具有亲和力，即“处理器亲和性”“processor affinity”）。
- 负载平衡可能会破坏处理器亲和性：
  - 因为线程可能会从一个处理器移动到另一个处理器以平衡负载，但该线程将丢失在移出的处理器的缓存中所拥有的内容。
- 因此想要保持很好的性能，操作系统需要一个保持亲和性的机制，尝试让线程在同一个处理器上运行(往往不能保证)

NUMA和CPU调度

- 如果操作系统是NUMA感知的（采用非统一内存访问架构，比如有两个物理处理器芯片，每个都有自己的CPU和本地内存），那么其应该考虑分配内存给线程所在的CPU最接近的内存。

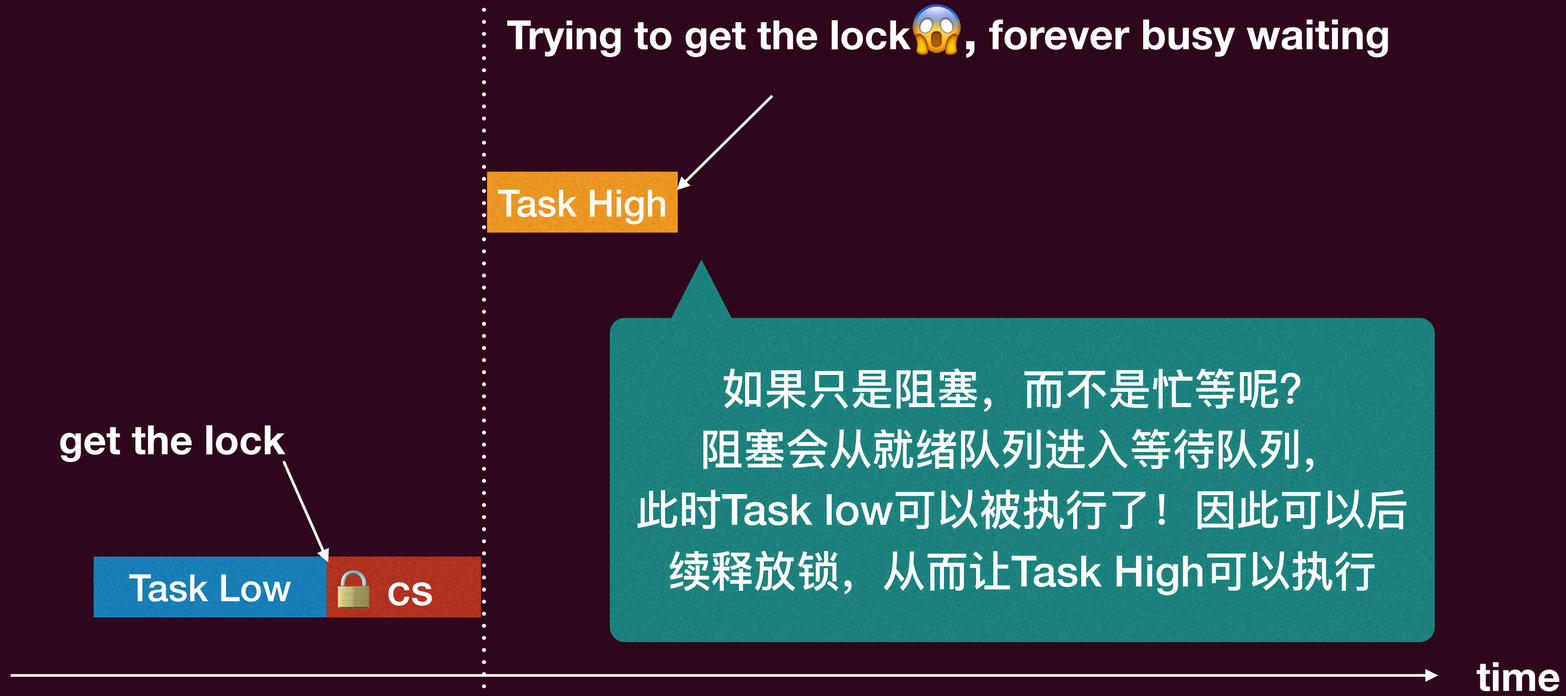


调度和并发的“组合”bug

- 优先级反转问题：
  - 假设系统中有两个线程：T1（低优先级）和T2（高优先级）。 如果两者都是可运行的，调度器将始终优先运行T2。
  - 现在假设T1首先运行（T2此时还不可运行），并获得了一个自旋锁，并进入其临界区。
  - 然后T2开始运行，调度器立即调度T2取代T1运行；
  - 但T2尝试获取锁，并开始忙等待。 由于T2运行时永远不会调度T1，因此T1没有机会离开临界区，而T2则永远循环下去。

优先级反转问题

- 一个高优先级任务反而受制于一个低优先级任务。
  - 只要低优先级任务持有锁，高优先级任务就会被阻塞。

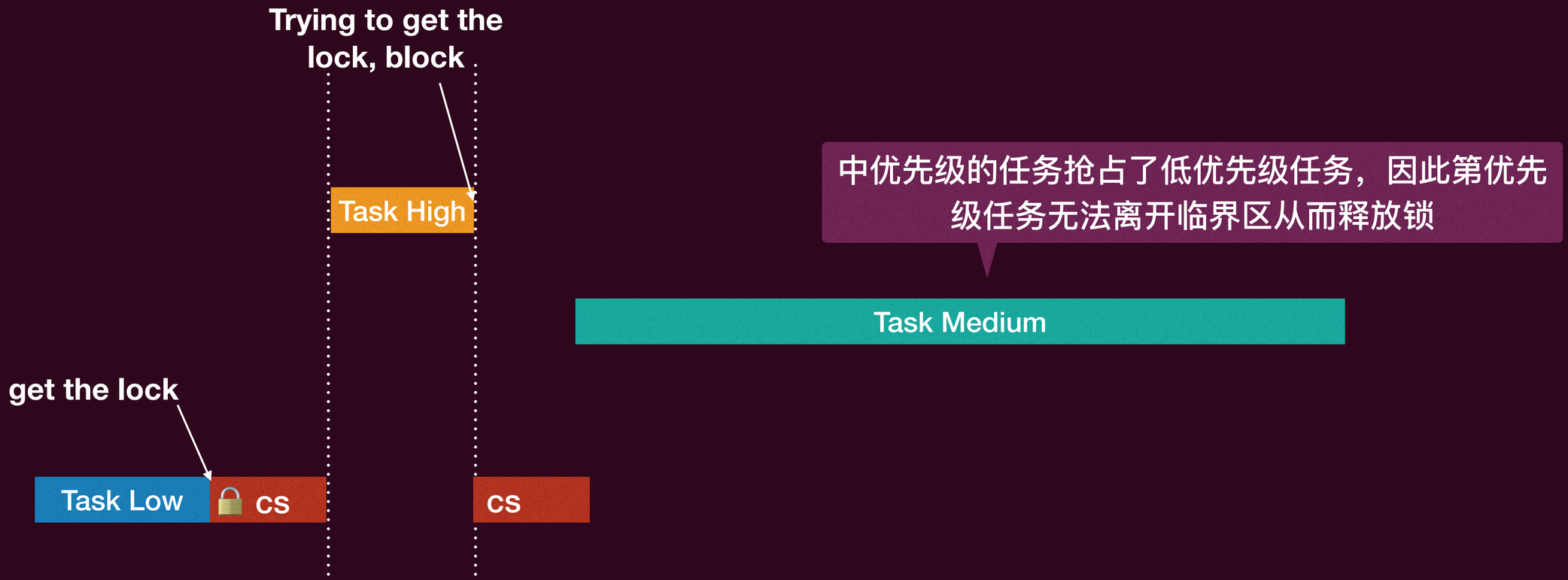


优先级反转问题

- 然而，即使避免使用自旋锁，而是使用futex wait的阻塞也并不能避免优先级反转问题。
- 假设有三个线程，T1、T2和T3，其中T3的优先级最高，T1的优先级最低。
  - T1获取了一个锁，T3开始运行，从而抢占了T1。
  - T3试图获取锁，但被阻塞。
  - T2开始运行，因为它的优先级高于T1，所以它将运行。
  - T3的优先级高于T2，但由于T2正在运行，T3被阻塞在等待T1上，而现在T1可能永远不会运行。

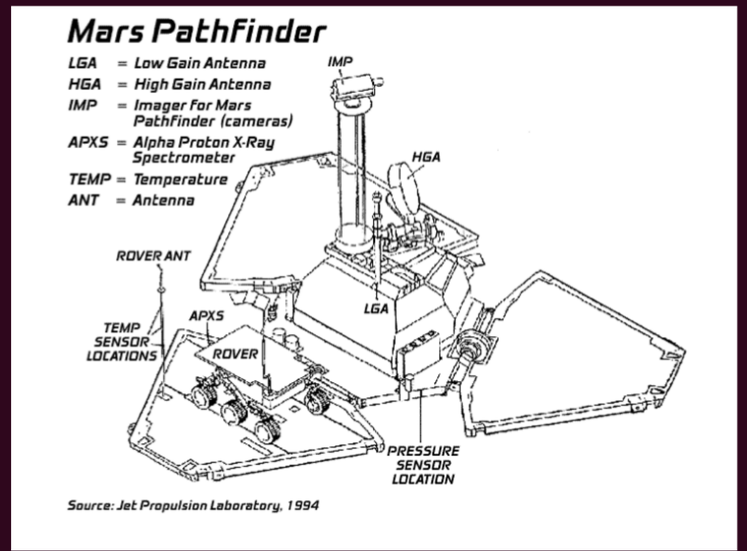
优先级反转问题

- 一个中优先级任务在持有锁时中断了低优先级任务的执行。

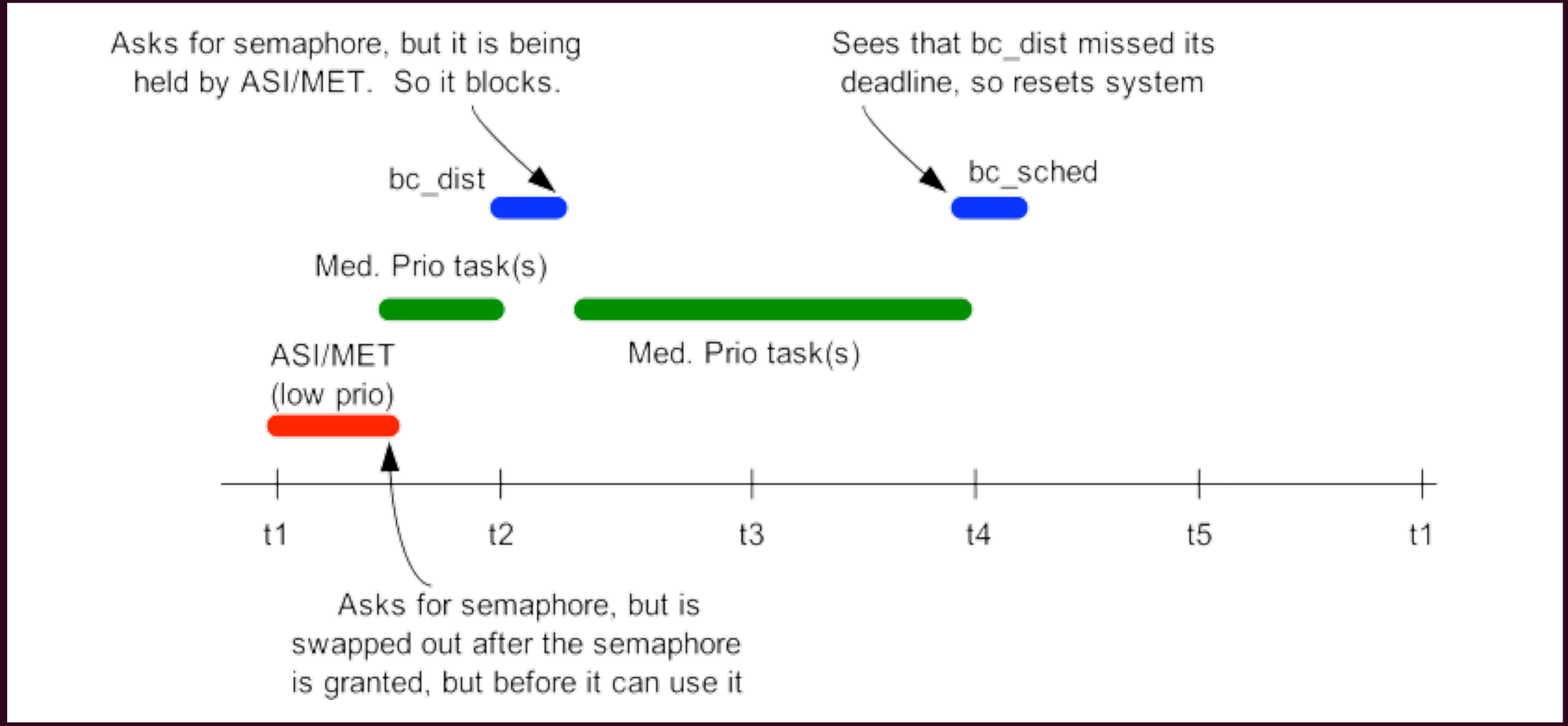


这个bug曾经发生在火星上

- 1997年火星探路者（Mars Pathfinder）在部署探测器后，着陆器会因为间歇性的优先级反转错误而每隔几天就会重新启动
  - 工程师最终找到了这个错误，并向着陆器发送了更新补丁。
  - 这个bug差点导致任务失败



这个bug曾经发生在火星上

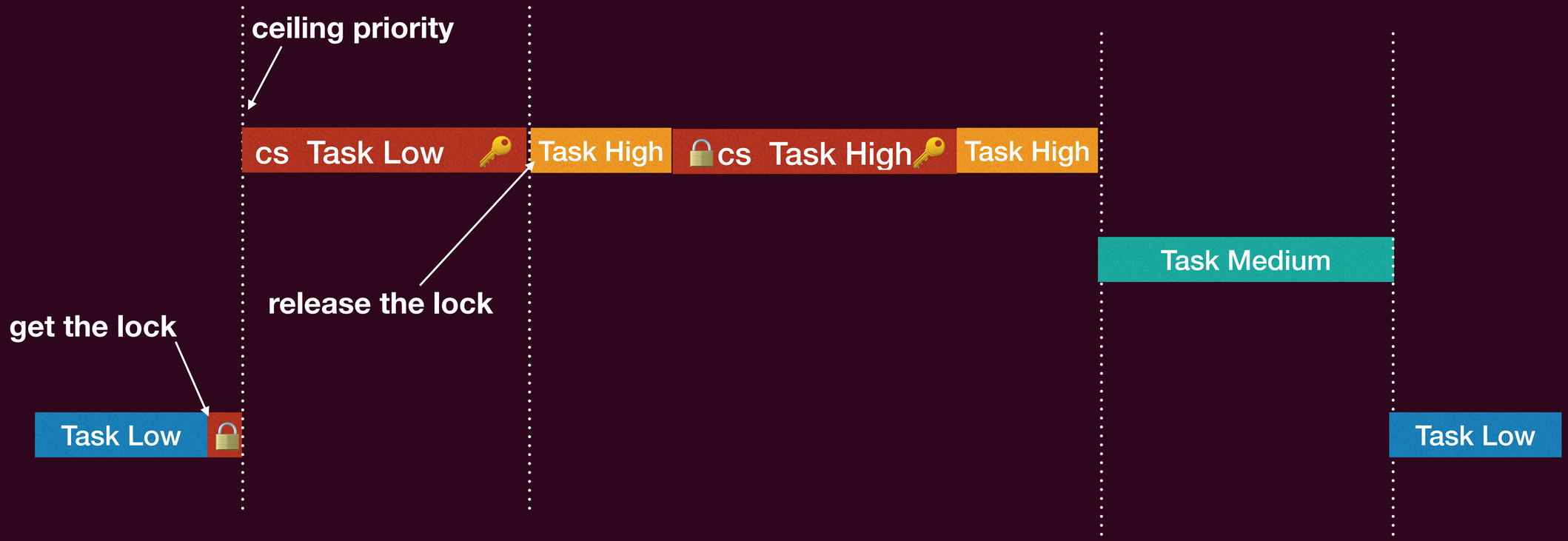


“Even when you think you’ve tested everything that you can possibly imagine, you’re wrong”

— Glenn E. Reeves (Pathfinder’s Software Team Leader)

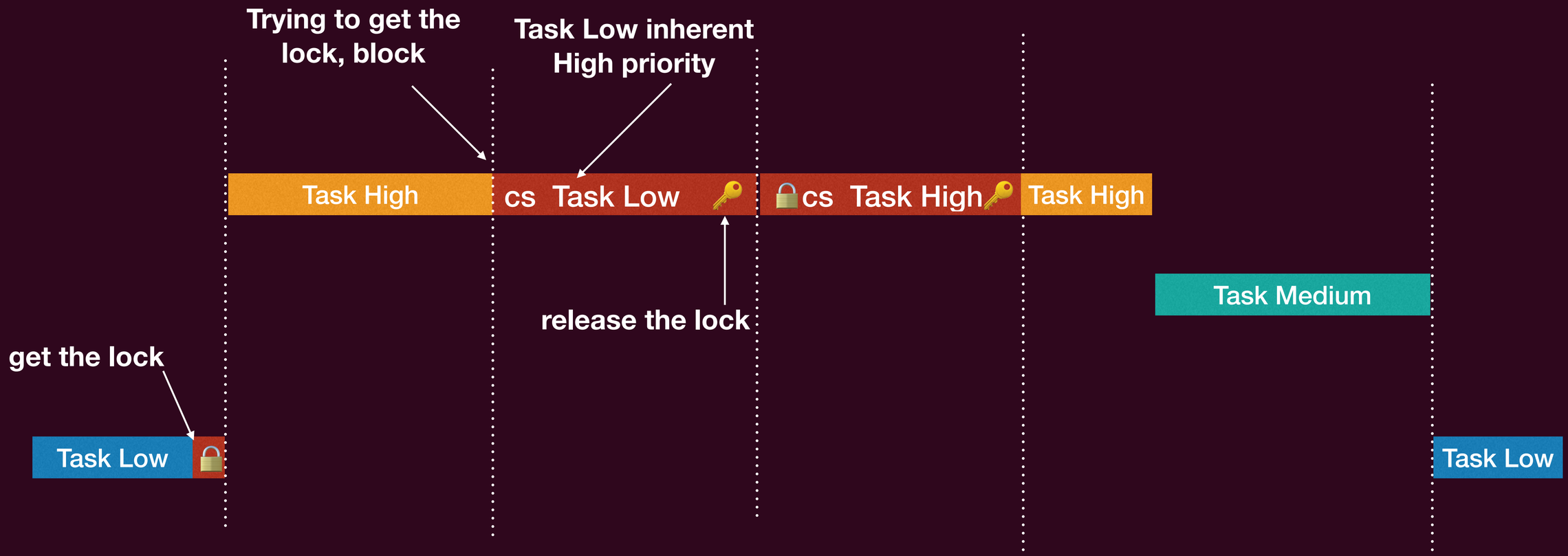
优先权极限（priority ceiling protocol）

- 每当一个任务获取一个锁时，该任务的优先级被提升到与该锁关联的优先级上限相同的优先级。



优先级继承

- 当一个任务持有一个锁时，如果其他（更高优先级的）任务试图获取该锁，那么持有锁的任务的优先级将被提升到那个更高优先级的任务的优先级。



阅读材料

- [OSTEP] 第7、8、9、10章
- [OSC(操作系统概念)] 第5章