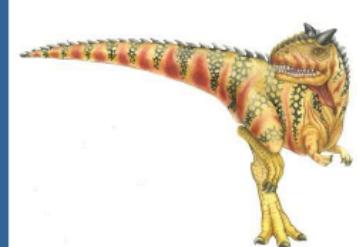
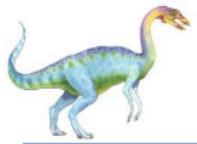


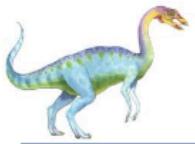
第 5 章：CPU 调度





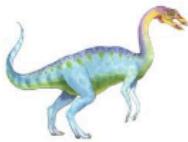
第 5 章：CPU 调度

- 基本概念 调度标准 调
- 度算法 线程调度 多处
- 理器调度 实时调度算法
- 评估
-
-
-
-



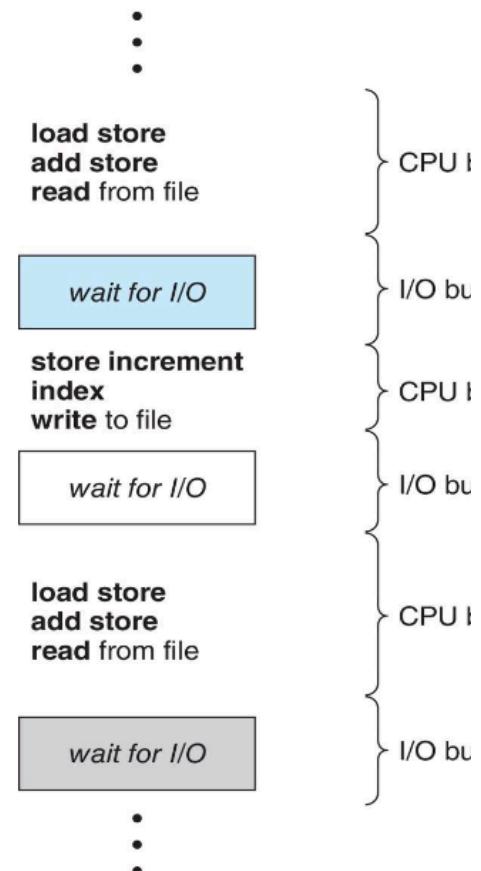
目标

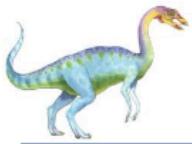
- 描述各种CPU调度算法 根据调度标准评估CPU调度算法 解释与多处理器和多核调度相关的问题 描述实时调度算法
-
-
- 应用建模和仿真来评估 CPU 调度算法



基本概念

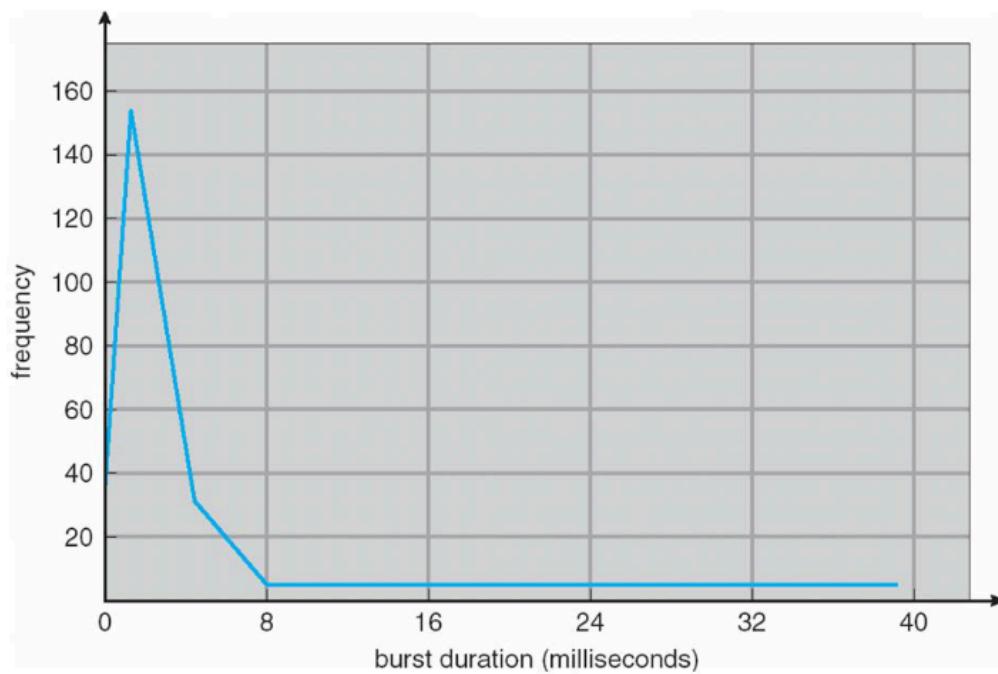
- 多道程序设计的目标是尝试让一个进程始终运行 - 最大化 CPU 利用率 进程执行由 CPU 执行和 I/O 等待周期组成 - 称为 CPU 窃取和 I/O 窃取
- (当不在 CPU 上运行时) CPU 空闲, OS 尝试选择就绪队列上的进程之一来执行, 除非就绪队列为空, 即没有进程处于就绪状态
- 进程的选择是由 CPU 调度器完成的, 也称为进程调度器、短期调度器





CPU 突发时间直方图

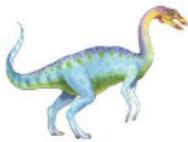
- 多年来，人们对 CPU 突发的持续时间进行了广泛的测量。频率曲线类似下图
- 有很多短CPU突发和少量长CPU突发（长尾分布）。I/O 密集型程序通常具有许多 CPU 突发，而 CPU 密集型程序可能具有一些长 CPU 突发。
- 这种分布对于设计 CPU 调度算法很重要





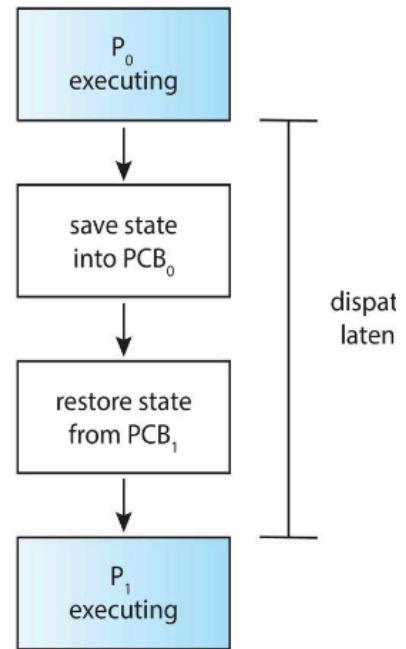
CPU调度器

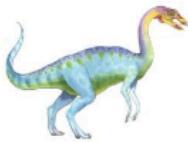
- CPU调度器从就绪队列中的进程中选择一个进程或线程，并将CPU核心分配给选定的进程
 - 队列可以以多种方式排序，单个或多个队列
- CPU调度可能在以下四种情况下发生：
 1. 从运行状态切换到等待状态，例如，I/O 请求或 wait()
 2. 进程终止
 3. 从运行状态切换到就绪状态，例如中断
 4. 从等待切换到就绪，例如I/O完成或从新进程切换到就绪，新进程以更高的优先级就绪队列
- 1和2下的调度是非抢占式的，其中进程自行放弃CPU
- 3和4以下的调度是抢占式的
 - 考虑访问共享数据（第6章中讨论）考虑在内核模式下的抢占
 - 考虑关键操作系统活动期间发生的中断



调度员

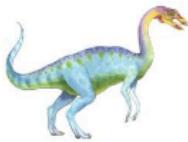
- Dispatcher模块将CPU分配给CPU调度器选择的进程；这涉及：
 - 将上下文从一个进程切换到另一个进程
 - 切换到用户模式跳转到用户程序中的正确位置以重新启动该程序
- 调度延迟 – 调度程序停止一个进程并开始另一个进程运行所需的时间 – 一种开销
- Linux上使用vmstat可以得到上下文切换的次数，一般每秒数百次上下文切换





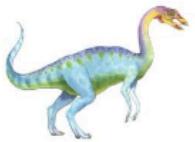
调度标准

- CPU 利用率 – CPU 繁忙时间的一部分 吞吐量 – 每个时间单位完成的进程或作业数
 - 有效利用资源 (CPU、内存、磁盘等) , 有利于短流程
- 等待时间 – 进程在就绪队列中等待的时间 周转时间 – 执行特定进程的时间, 通过 CPU 突发时间、I/O 突发时间和等待时间来衡量
 - 对于单个 CPU 突发, 周转时间 = 等待时间 + CPU 突发时间
- 响应时间——从提交请求到产生第一个响应所花费的时间
 - 在编辑器或游戏中回显击键的时间 这与交互式程序更相关 (通常使用 R 调度) 考虑到单个 CPU 突发, 这是第一个 CPU 时间完成减去该进程加入就绪队列的时间之间的时间
 -
- 公平性
 - CPU 等资源以某种 “公平” 的方式利用



调度标准 (续)

- 希望最大限度地提高 CPU 利用率和吞吐量，并最大限度地减少周转时间、等待时间和响应时间。但这些标准可能是相互冲突的，在实践中不同的考虑因素。在大多数情况下，我们优化平均度量，例如平均等待时间或平均周转时间。然而，在特殊情况下，我们更愿意优化最小值或最大值而不是平均值
 - 考虑到所有用户，我们可能希望最小化最大响应时间
- 对于交互式系统（例如台式机或笔记本电脑），最小化响应时间的差异可能比最小化平均响应时间更重要
 - 具有合理且可预测的响应时间的系统可能被认为比平均速度更快但变化很大的系统更令人满意
- 不同的 CPU 调度算法具有不同的属性。接下来，我们将在仅一个 CPU 核心的情况下描述几种调度算法——系统一次只能运行一个进程或线程



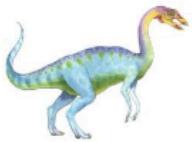
先到先服务 (FCFS) 调度

过程	突发时间
P1	24
P2	3
P3	27

- 假设进程按以下顺序到达： P1、P2、P3 时间表的甘特图为：



- P1的等待时间=0； P2=24； P3 = 27 平均等待时间： $(0 + 24 + 27)/3 = 17$ 平均周转时间： $(24+27+30)/3 = 27$ 在早期的系统中， FCFS 意味着一个程序计划运行直到完成包括所有 I/O
-
-
- 在多道程序设计系统中，这通常意味着进程完成其当前的 CPU 突发时间



FCFS 调度 (续)

假设进程按以下顺序到达：

P2、P3、P1

- 时间表的甘特图是：

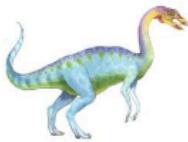


- P1的等待时间=6； P2=0； P3 = 3 平均等待时间：
 $(6 + 0 + 3)/3 = 3$ (短得多！) 平均周转时间：
 $(30+3+6)/3 = 13$
- 护航效应——短流程落后于长流程，对短作业不利，完全取决于到达顺序
- 考虑一个 CPU 密集型进程和多个 I/O 密集型进程，FCFS 还会导致 I/O 设备利用率低
- 在银行等待：存入支票，却无法开立新账户



循环赛 (RR)

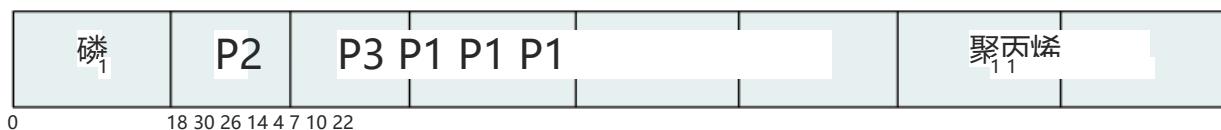
- FCFS调度算法是非抢占式的。一旦CPU核心分配给一个进程，该进程就留CPU直到它释放CPU。因此，FCFS对于交互式系统来说特别麻烦，因
- 在交互式系统中，每个进程定期获得CPU共享非常重要。轮询 (RR) 调度与FCFS调度类似，但增加了抢占，使系统能够在进程之间进行切换。每
- 程获得一个小的CPU时间单位（时间量 q ），通常为10-100毫秒。经过 q 毫秒后，进程将被抢占并添加到就绪队列的末尾（与 FCFS 完全相同）。
-
- 给定 n 个进程，每个进程一次获得最多 q 个时间单位的 $1/n$ 的 CPU 时
– 上下文切换时间被忽略
 - 没有进程等待超过 $(n-1)q$ 时间单位。
- 定时器会中断每个时间片以调度下一个进程，或者当其 CPU 突发时间或其 CPU 突发时间 $< q$ 时，进程会在完成其当前 CPU 突发时间后阻塞



时间量子 = 4 的 RR 示例

过程	突发时间
P1	24
P23	
P33	

- 甘特图是：



$P1=6$ 、 $P2=4$ 、 $P3=7$ 的等待时间 平均等待时间 $(6+4+7)/3 = 5.67$

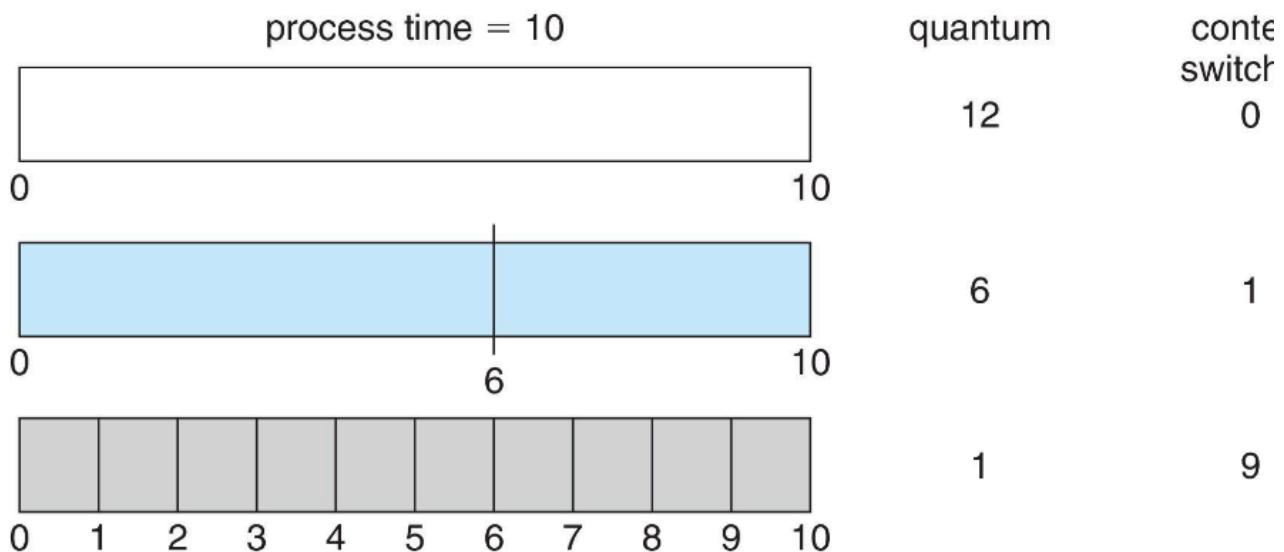
- 平均周转时间： $(30+7+10)/3 = 15.67$
- $P1=6$ 的响应时间
- $P2=7$ 、 $P3=10$ 、平均值=7 RR策略下的平均等待时间可能很长
- 但本质上更“公平”（先进先出顺序），通常比FCFS对于短作业表现更好，并提供更好的平均响应时间——对于交互式工作很重要

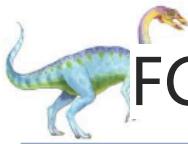


时间量子和上下文切换时间

- RR算法的性能很大程度上取决于时间量子的大小

- q 大 FCFs q 小 交错，但 q 相对于上下文切换时间必须很大（通常 < 10 usec），否则开销太高



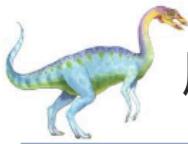


FCFS 与 RR 的比较

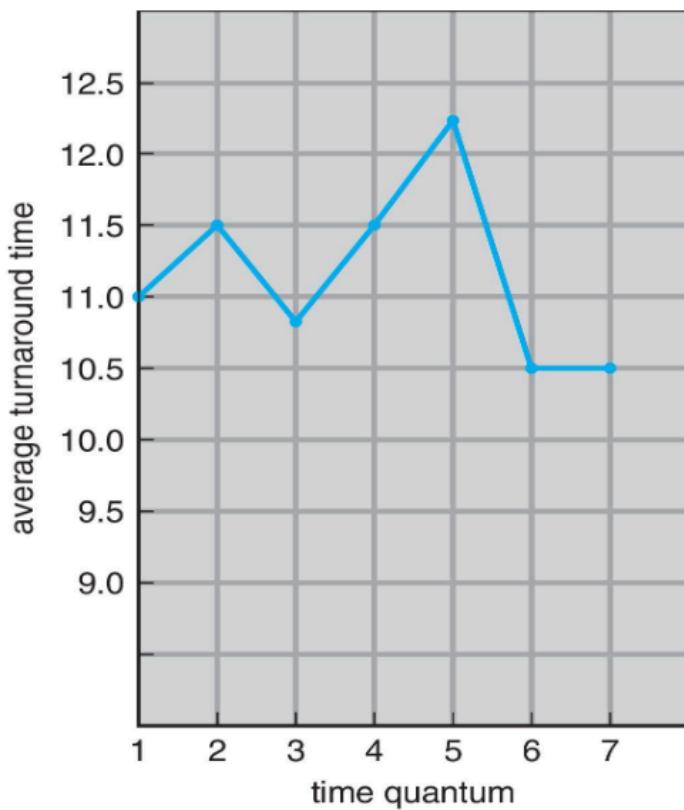
假设上下文切换时间成本为零，RR 是否总是优于 FCFS？一个例子：10个作业同时启动，每个作业占用100秒的CPU时间；RR 调度程序为 1s；

Job #	FIFO	RR
1	100	991
2	200	992
...
9	900	999
10	1000	1000

- RR 下的平均作业周转时间要差得多！
 - 当所有作业都具有相同长度时会很糟糕 □ 平均响应时间要好得多

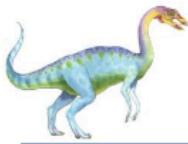


周转时间随时间量子变化



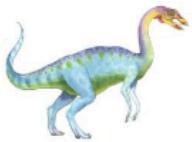
process	time
P_1	6
P_2	3
P_3	1
P_4	7

- 当数量增加时，平均周转时间不一定改善
- 一般来说，如果大多数进程在单个时间量子内完成当前的 CPU 突发，则平均周转时间可以得到改善
- 时间量不能太大，其中 RR 退化为 FCFS。经验法则：80% CPU 突发应该短于时间量 q



最短作业优先 (SJF) 调度

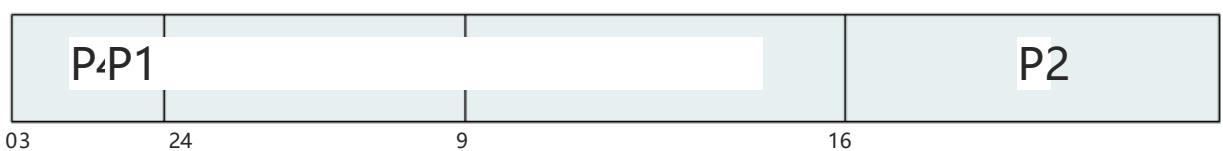
- 注意到在FCFS和RR中，我们在调度时不需要知道每个进程的下一次CPU时间，并且仅根据就绪队列的到达顺序来进行调度
- 如果我们知道未来 - 每个进程的下一个 CPU 突发时间 与每个进程关联其下一个 CPU 突发的长度会怎么样
- 以最短的下一个 CPU 突发来调度进程 最短作业优先或 SJF 调度算法是的 - 它为给定的一组进程生成最短的平均等待时间
- 困难在于知道下一个 CPU 请求的长度 基本思想是尽快将短作业从系统中取出 对短作业影响较大，对长作业影响相对较小 这可以应用于整个程序，或当前的 CPU突发 也许更准确的术语应该是最短下一个 CPU 突发算法，但通常使用最短作业优先或 SJF。
-



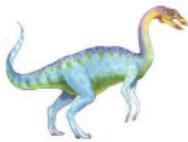
SJF 示例

流程到达时间	突发时间
P1	6
P2	8
P3	7
P4	3

□ SJF调度图



- 平均等待时间 = $(3 + 16 + 9 + 0) / 4 = 7$ 如果到达顺序恰好相同，“最短进程优先”和“先来先服务”执行相同的操作 草图证明：将短流程移到长流程之前会减少流程的平均等待时间
- 时间短进程比长进程增加的等待时间更多。因此，平均等待时间减少
-



确定下一个 CPU 突发的长度

- 如何根据过去的行为估计长度
 - 然后选择预测下一次 CPU 突发时间最短的进程
- 可以通过使用之前CPU突发的长度和指数平均算法来完成

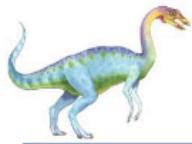
1. t_n

1. t_{n+1} 接下来突发 CPU 为预测值 2.

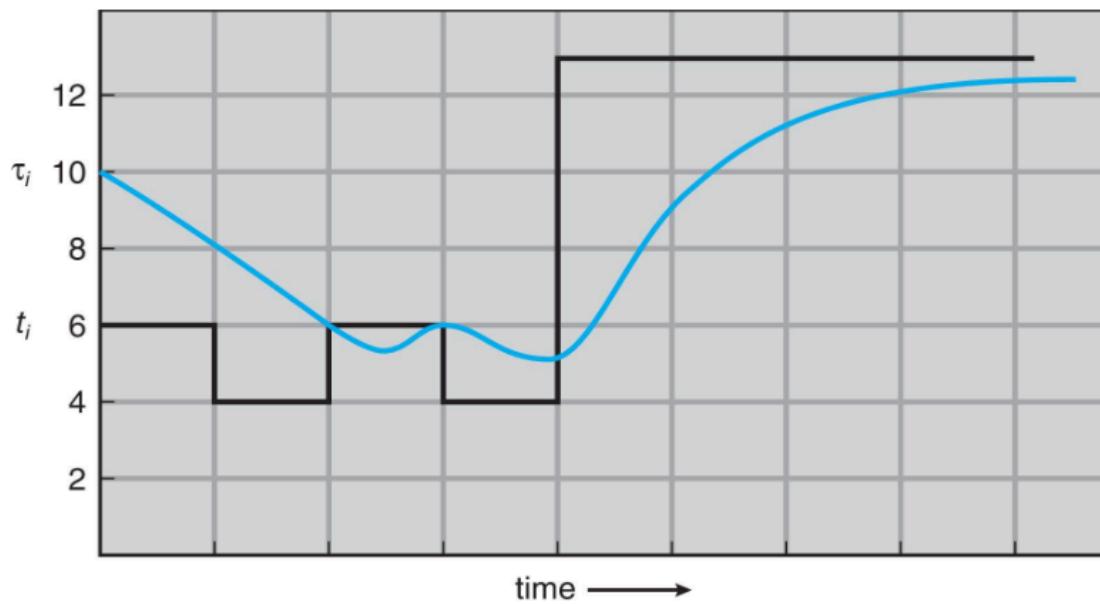
$\alpha = \frac{t_n}{t_{n+1}}$

: 定义4. $t_{n+1} = \alpha t_n + \epsilon$

- 通常, α 设置为 1/2 - 预测中最近和过去历史的相对权重
- 抢占式版本称为最短剩余时间优先 (SRTF)

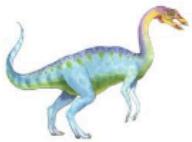


预测下一次 CPU 突发的长度



CPU burst (t_i)	6	4	6	4	13	13	13	...	
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

这显示了 $\alpha = 1/2$ 且 $\tau_0 = 10$ 的指数平均值。



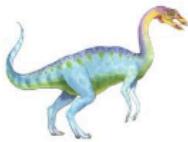
指数平均算法

- 这通常用于根据历史数据估计未来值 - 最近的历史值比过去的历史值更重要

1. t_n = 第 n 个 CPU 突发的实际长度
2. $=$ 下一个 CPU 突发的预测值
3. $\alpha, 0 \leq \alpha \leq 1$
4. 定义: $\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$

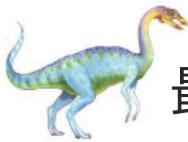
如果我们将公式展开, 我们得到:

$$\begin{aligned}\tau_{n+1} &= \alpha t_n + (1 - \alpha) \tau_n \\&= \alpha t_n + (1 - \alpha) \cdot (\alpha t_{n-1} + (1 - \alpha) \tau_{n-1}) \\&= \alpha t_n + (1 - \alpha) \alpha t_{n-1} + (1 - \alpha)^2 \tau_{n-1} \\&= \alpha t_n + (1 - \alpha) \alpha t_{n-1} + (1 - \alpha)^2 \cdot (\alpha t_{n-2} + (1 - \alpha) \tau_{n-2}) \\&= \alpha t_n + (1 - \alpha) \alpha t_{n-1} + (1 - \alpha)^2 \alpha t_{n-2} + (1 - \alpha)^3 \tau_{n-2} \\&= \dots \\&= \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \dots + (1 - \alpha)^j \alpha t_{n-j} + \dots + (1 - \alpha)^{n-1} \tau_1\end{aligned}$$



指数平均的示例

- $\alpha = 0$
 - $\alpha_{n+1} = \alpha_n$ 最近的历史记录
 - 不计算在内
- $\alpha = 1$
 - $\alpha_{n+1} = \alpha t_n$ 仅实际最后一次 CPU 突发计数
- 如果我们将公式展开，我们得到：
$$\alpha_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots + (1 - \alpha)^j \alpha t_{n-j} + \dots + (1 - \alpha)^n \alpha + 1 - \alpha$$
 - 所有过去的值（即 CPU 突发的长度）都很重要，因为它们 ($t_n, t_{n-1}, t_{n-2}, \dots$) 都包含在公式中
 - 但最近的价值观更重要。过去的价值效应以指数方式快速衰减，每次都会乘以一个分数 ($1 - \alpha$)。例如，如果 $\alpha = 0.5$ ，则 9 轮后的效果为 $(1 - \alpha)^9 + 1 = 1/1024$ ，可以忽略。



最短剩余时间优先的示例

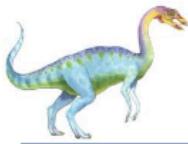
- SJF 算法可以是抢占式的，也可以是非抢占式的。当一个新进程到达就绪队列而另一个进程仍在执行时，就会出现这种选择

过程A	到达	到达时间T	突发时间
P10			8
P21			4
P32			9
P43			5

- 先发制人的 SJF 甘特图

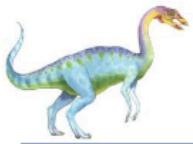


- 平均等待时间 = $[(10-1)+(1-1)+(17-2)+5-3]/4 = 26/4 = 6.5$ 现
- 在需要在到达就绪时调用调度队列（调度条件4）



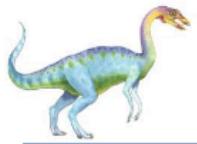
SJF/SRTF 与 FCFS 的比较

- SJF/SRTF 是我们在尽量减少平均等待时间方面所能做的最好的事情。或最平均周转时间
 - 可证明最优 (非抢占式中的 SJF, 抢占式中的 SRTF)
 - SRTF 始终至少与 SJF 一样好
- 如果所有进程具有相同的 CPU 突发时间, SJF/SRTF 的执行效果与 FCFS 相同
- 如果总是有较短的进程加入就绪队列, SJF/SRTF 可能会导致长进程饥饿
 - “公平”无法执行



优先级调度

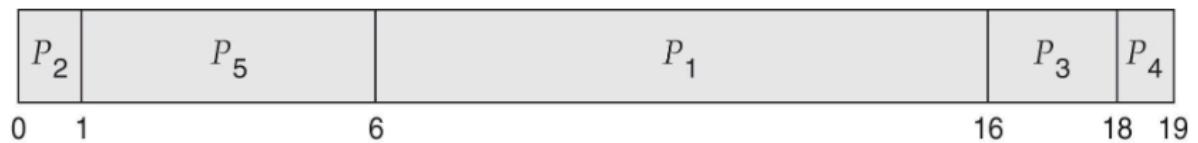
- 优先级编号（例如整数）与每个进程相关联
- CPU被分配给具有最高优先级的进程（最小整数口最高优先级），可以是
 - 抢占式（当更高优先级进程新到达时） 非抢占式
 -
- 等优先级进程按照 FCFS 顺序进行调度 SJF 是一般优先级调度算法的特例，其中优先级是预测的下一个 CPU 突发时间的倒数
- 问题 □ 饥饿 – 低优先级进程可能永远不会执行
- 解决方案□ 老化 – 随着时间的推移，进程的优先级会增加



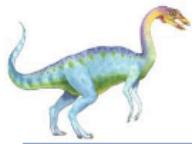
优先级调度示例

处理Arri突发时间T	优先事项
P1	3
P21	1
P32	4
P41	5
P55	2

□ 优先级调度甘特图



□ 平均等待时间 = 8.2 毫秒

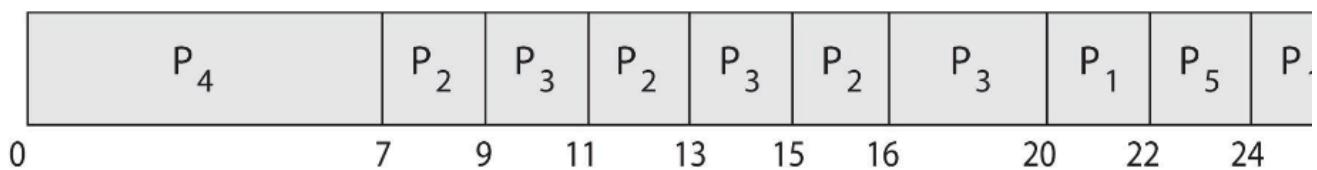


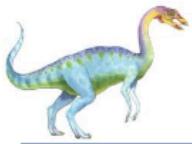
带循环的优先级调度

<u>处理Arri突发时间T</u>	<u>优先事项</u>
P14	3
P25	2
P38	2
P47	1
P53	3

□ 运行具有最高优先级的进程。具有相同优先级的进程循环运行

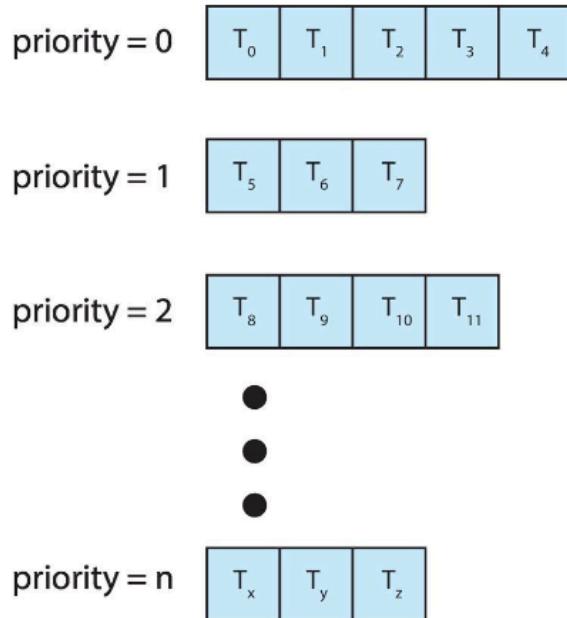
□ 2 毫秒时间量的甘特图

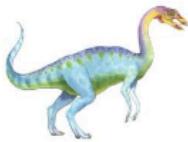




多级队列

- 多级队列调度仍然可以是优先级调度结合循环
- 优先级静态分配给每个进程，并且进程在其运行期间保留在同一队列中

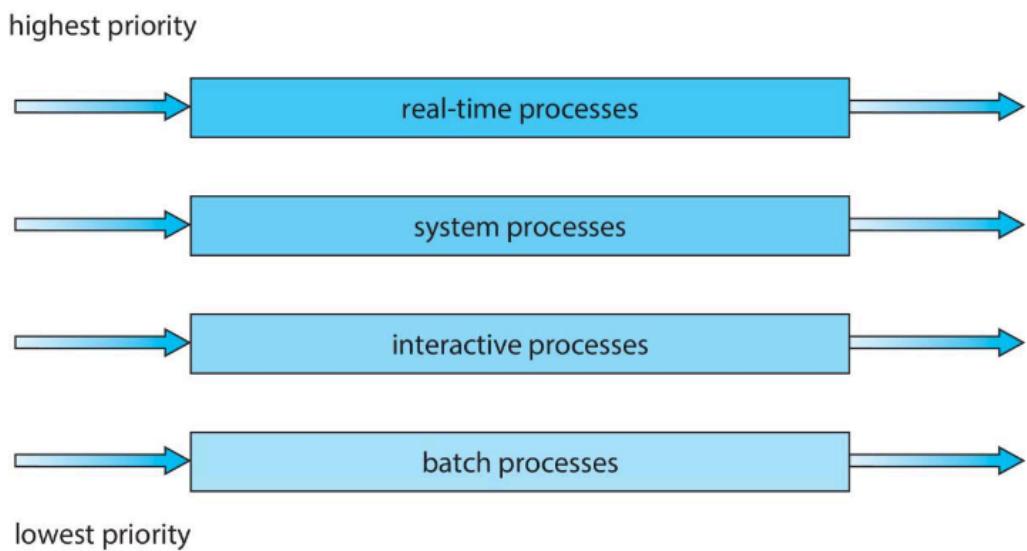


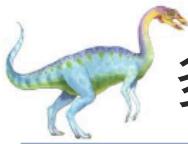


多级队列 (续)

根据进程类型将进程划分到不同的队列中

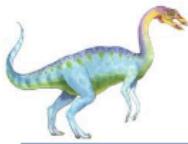
- 每个队列可以根据需要有自己的调度算法 队列之间的调度，通常实现为定优先级抢占式调度或每个队列获得一定量的 CPU 时间 – 时间片（例如 60%、20%、10 %、10%）





多级反馈队列 (MLFQ)

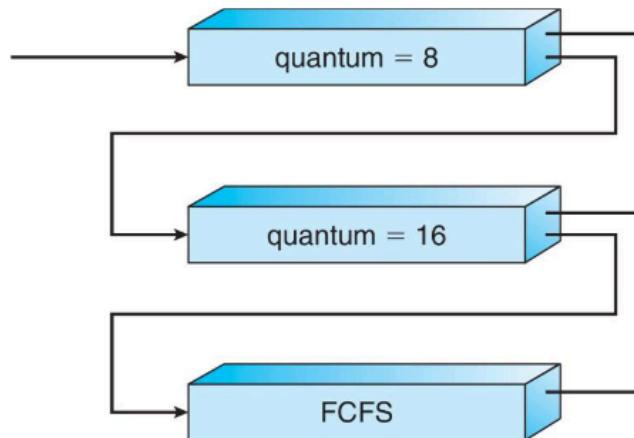
- 一个进程可以在各个队列之间移动；老化可以通过这种方式实现。这提供了灵活性
- 多级反馈队列或 MLFQ 调度程序由以下参数定义：
 - 每个队列方法的队列数调度算法用于确定何时升级进
 - 程方法用于确定何时降级进程方法用于确定进程需要
 - 服务时将进入哪个队列
 -
 -



多级反馈队列示例

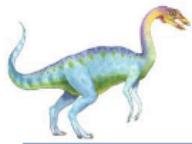
三个队列：

- Q0 – RR 时间量为 8 毫秒 Q1 – RR 时
间量为 16 毫秒 Q2 – FCFS
- 调度
 - 新作业进入由 FCFS 提供服务的队列 Q0，如果当前在 CPU 上运行，还会抢占 Q1 或 Q2 中的作业
 - 当它获得CPU时，作业接收8毫秒如果它没有在8毫秒内完成，作业将移至队列 Q1
 - 在第 1 季度，作业再次由 FCFS 提供服务并额外接收 16 毫秒如果仍未完成，则它被抢占并移动到队列 Q2 如果 Q1 或 Q2 中的作业被 Q0 中的新作业抢占，则它分别加入队列 Q1 或 Q2 的头部

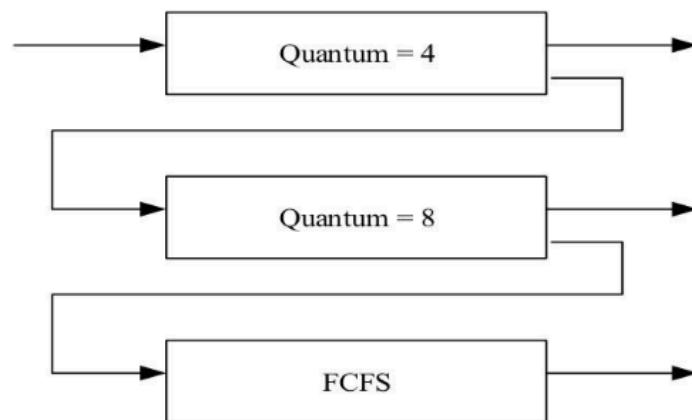


□ 这近似于 SRTF：

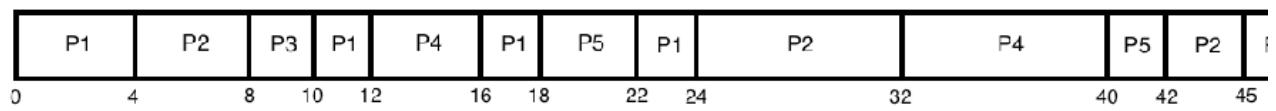
- CPU 密集型作业急剧下降短
间运行的 I/O 密集型作业仍
然顶部

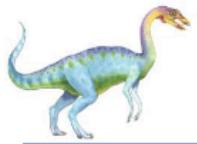


MLFQ 示例



过程	到达时间 (毫秒)	突发时间 (毫秒)
P1	0	10
P2	2	15
P3	5	2
P4	12	14
P5	18	6

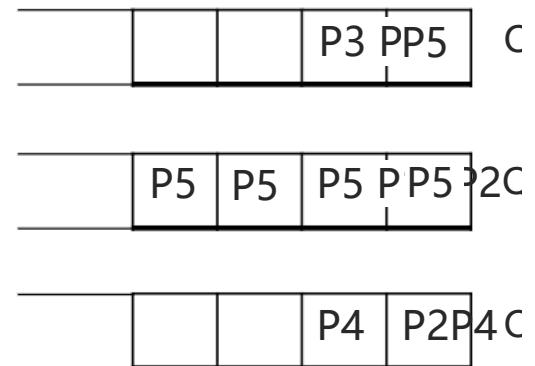


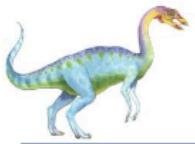


MLFQ 调度：示例



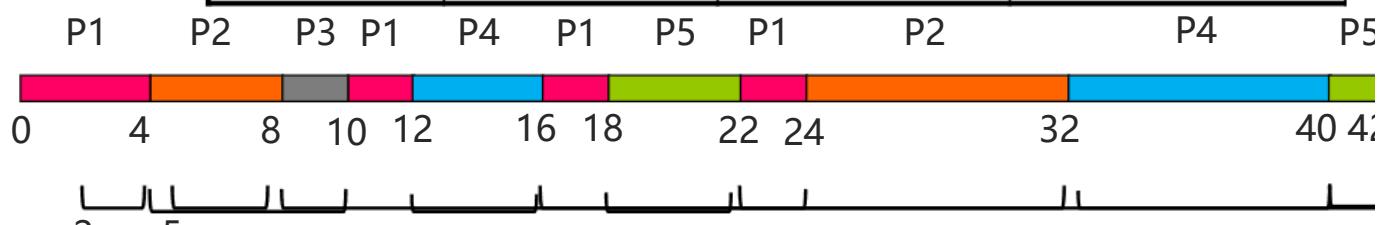
- 时间 47: P4 结束 在第二季度投入使用



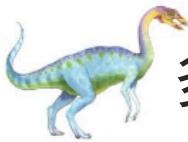


MLFQ 调度：示例

过程	突发时间	到达时间	剩余时间
P1	10	0	0
P2	15	2	0
P3	2	5	0
P4	14	12	0
P5	6	18	0

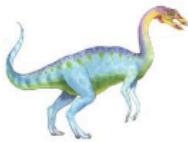


- P1=14、P2=28、P3=3、P4=16+5=21 P1=14、P2=28、P3=3、P4=21、P5=18 平均等待时间: $(14+28+3+21+18)/5=16.8$
- P1=14、P2=28、P3=3、P4=21、P5=18 的等待时间



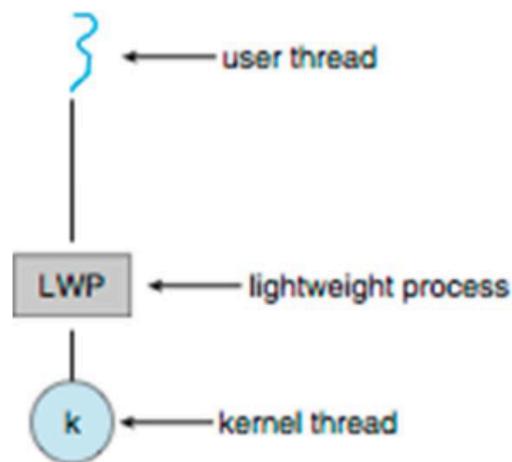
多级反馈队列 (MLFQ)

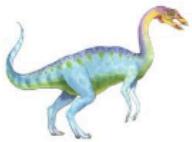
- MLFQ 常用于 BSD Unix、Solaris、Window NT 及后续 Window 操作系统等许多系统中，MLFQ 有几个独特的优点：
- - 它不需要预先知道下一个 CPU 突发时间 它可以通过在响应时间方面提供比 RR 更好的性能来很好地处理交互式作业
 - 它产生与 SJF 或 SRTF 类似的性能 通过在 CPU 密集型作业上取得进展，它也是“公平的”
 -
- 可能的饥饿问题可以通过定期将作业重新洗牌到不同的队列来解决
 - 例如，一段时间后，将所有作业移至顶部队列



线程调度

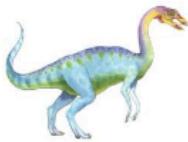
- 在现代操作系统上，内核级线程是被调度的线程，用户级线程由线程库管理
- 操作系统使用用户线程和内核线程之间的中间数据结构，即轻量级进程（LWP）
 - 显示为虚拟处理器，用户线程被安排在其上“运行”每个连接到内核线程的 LWP（一对一）





线程调度

- 在多对一和多对多模型下，线程库“调度”用户级线程在LWP上运行。这称为进程争用范围 (PCS) – 确定哪个用户线程映射到哪个内核线程
- 由于调度竞争发生在属于同一进程的线程之间，通常通过程序员设置的优先级完成
 - 线程库通常无法调整优先级 PCS 通常会抢占当前正在运行的线程，以支持更高优先级的用户级线程
- 调度到可用 CPU 上的内核线程是系统争用范围 (SCS) – 系统中所有线程之间的竞争 – CPU 调度 使用一对一映射模型的系统，例如
 - Windows、Linux 和 Solaris，仅使用 SCS 来调度线程



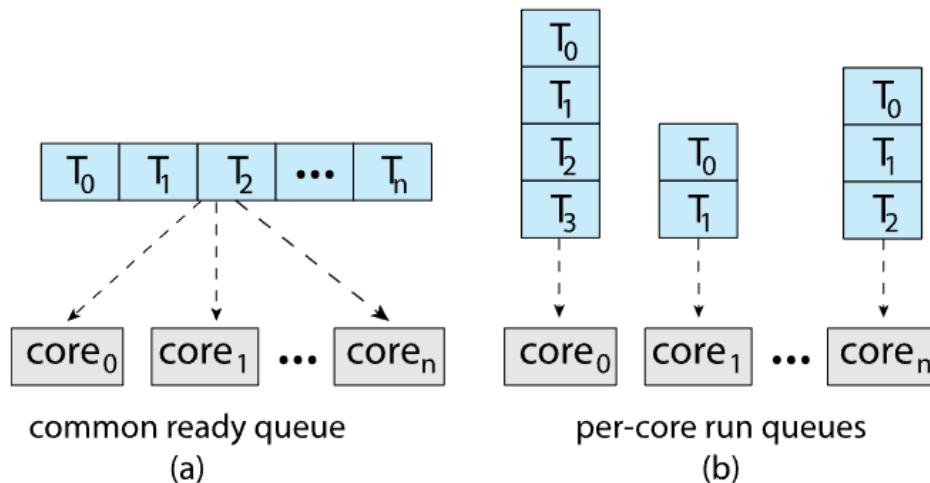
多处理器调度

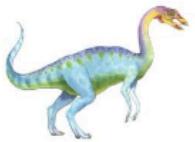
- 具有多个 CPU 的 CPU 调度要复杂得多 - 负载共享 传统上，术语“处理器”是指提供多个物理处理器的系统，其中每个物理处理器芯包含一个单核 CPU
- 多处理器的定义已经发生了很大的演变，在现代计算系统中，多处理器现适用于多核 CPU、多线程核心、NUMA 系统和异构多处理。多处理系统常有两种类型：非对称多处理和对称多处理
 -
 - 非对称多处理——只有一个处理器可以访问内核数据结构，从而减轻了数共享的需要。其他处理器仅执行用户代码
 - 所有调度决策、I/O 处理和其他系统活动均由单个处理器（主服务器）处理 主服务器可能成为潜在的瓶颈
 -



对称多处理 (SMP)

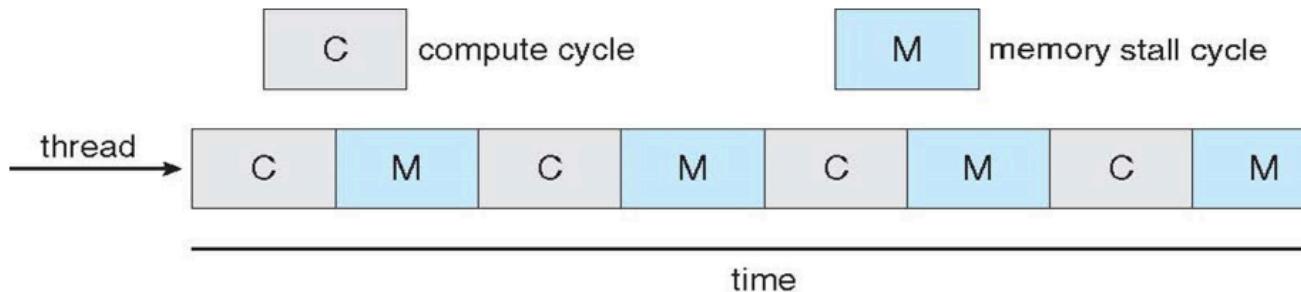
- 对称多处理 (SMP): 每个处理器都是自调度的，具有一个公共就绪队列，或每个处理器都有自己的专用就绪队列
 - 通过让每个处理器的调度程序检查就绪队列并选择要运行的线程来进行调度
 - 确保两个单独的处理器不会选择使用公共就绪队列来调度同一线程 - 可能争条件 (在第 6 章中讨论) 所有现代操作系统都支持 SMP，包括Window、Linux、Mac OS X 以及包括 Android 在内的移动系统和iOS

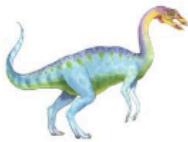




多核处理器

- 最近的趋势是将多个处理器核心放置在同一物理芯片上
 - 速度更快、功耗更低，但调度设计复杂化
- 内存停顿：当处理器访问内存或缓存时，它会花费大量时间等待数据可用。这主要是因为现代处理器的运行速度比内存快得多，尤其是内存。当缓存命中时
- 每个核心多个硬件线程 - 每个硬件线程都有自己的状态、程序计数器 (PC)、寄存器集，显示为逻辑 CPU 来运行软件线程。这称为芯片多线程 (CMT)

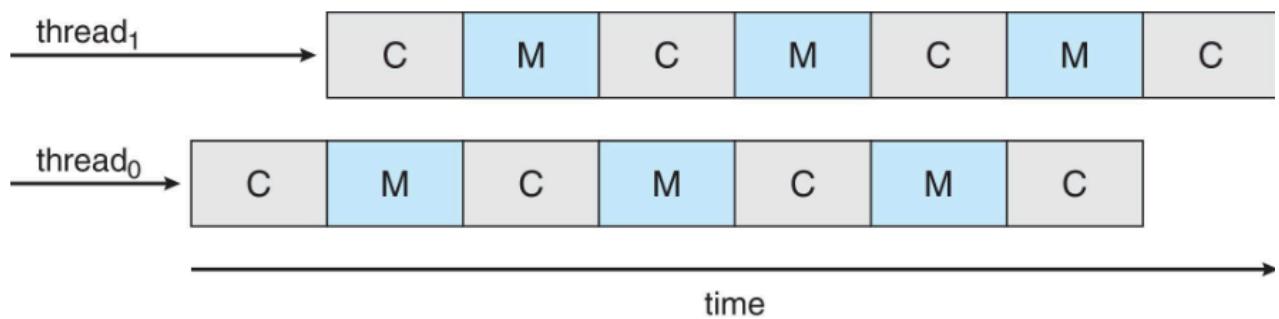


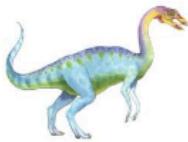


多线程多核系统

调度可以利用内存停顿在另一个硬件线程上取得进展，同时内存检索发生在另一个硬件线程上

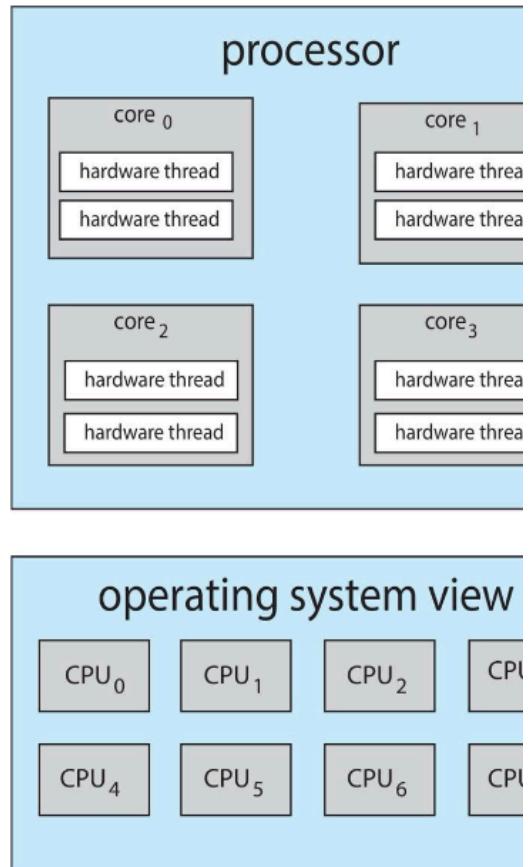
- 如果一个线程在等待内存时停止，核心可以切换到另一个线程。这成为双线程处理器或者类似于两个逻辑处理器。双线程、双核系统向处理器提供四个逻辑处理器。
- 操作系统 UltraSPARC T3 CPU 每个芯片有 16 个内核，每个内核有 8 个线程，从操作系统的角度来看，这似乎是 128 个逻辑处理器

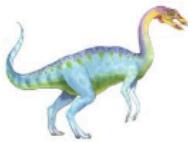




多线程多核系统

- 从操作系统的角度来看，每个硬件线程都维护其体系结构状态，例如指令指针和寄存器集，因此表现为可用于运行软件线程（即内核线程）的逻辑CPU
- 芯片多线程 (CMT) 为每个内核分配多个硬件线程。 (英特尔将此称为超线程)
- 在每个核心有 2 个硬件线程的四核系统上，操作系统有 8 个逻辑处理器。



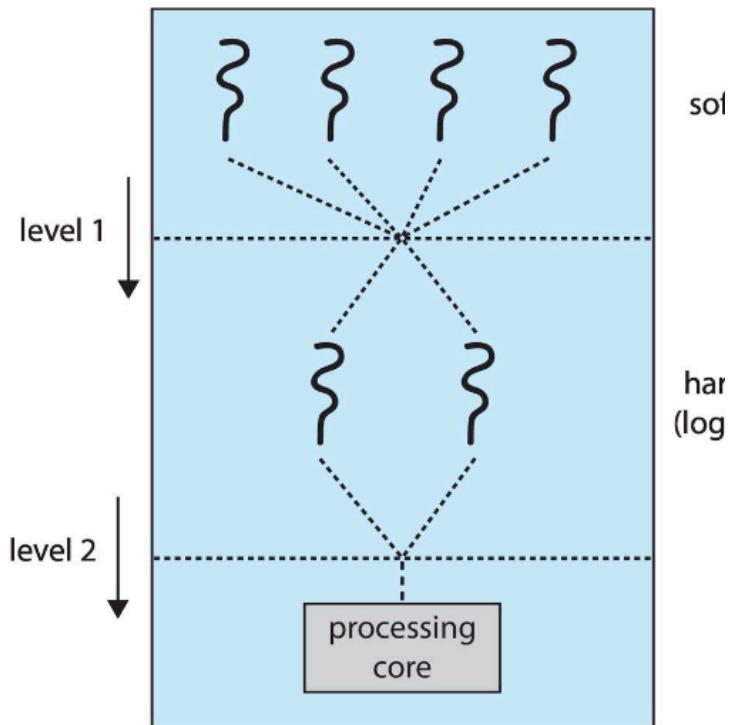


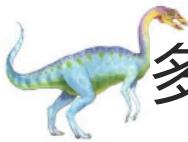
多线程多核系统

□ 两个级别的调度：

1. 操作系统决定在逻辑CPU上运行哪个软件线程（内核线程）——本章介绍的CPU调度

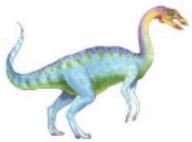
2. 每个核心如何决定在物理核心上运行哪个硬件线程 – 可以使用 RR 调度 (UltraSPARC T3)





多线程多核调度

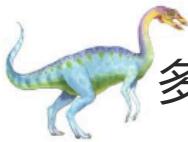
- 用户级线程被调度到 LWP – 内核级线程
 - 在多对一和多对多模型下，线程库通常根据程序员设置的优先级来调度称为进程争用范围（PCS）的用户级线程
- 现在称为软件线程的内核线程被调度到逻辑 CPU（硬件线程）上
 - CPU或进程调度——操作系统
- 硬件线程被安排在CPU核心上运行
 - 每个CPU核决定调度，通常使用RR



处理器亲和力

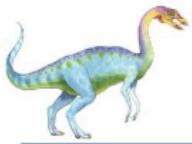
处理器亲和性——进程与其运行的处理器具有亲和性

- 当线程在一个处理器上运行时，该处理器的缓存内容存储该线程对内存访问。我们将其称为对处理器具有亲和力的线程（即“处理器亲和力”）。使缓存无效和重新填充的成本很高，大多数 SMP 系统都会尝试将进程从一个处理器迁移到另一个处理器。就绪队列免费提供处理器亲和力！软亲和力——操作系统尝试保持进程在同一处理器上运行（不保证这一点），并且进程有可能在负载平衡期间在处理器之间迁移
-
-
- 硬关联 – 允许进程指定它可以运行的处理器子集 许多系统同时提供软关联和硬关联
 - 例如，Linux 实现了软关联，但它还提供了系统调用 `sched_setaffinity()`，该调用通过允许线程指定其有资格运行的 CPU 集来支持硬关联



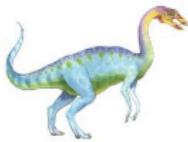
多处理器调度——负载均衡

- 负载平衡尝试保持工作负载均匀分布
 - 在每个处理器都有自己的私有就绪队列来执行合格线程的系统上
- 负载均衡有两种通用方法
 - 推送迁移 – 特定任务定期检查每个处理器上的负载，如果发现不平衡，则将任过载的 CPU 推送到空闲或不太繁忙的 CPU
 - 拉式迁移 – 空闲处理器从繁忙的处理器中拉取等待任务 推式迁移和拉式迁移并不相互排斥，事实上，两者通常在负载平衡系统上并行实现。例如，Linux CFS 实现了这两种技术
- 负载平衡通常会抵消处理器亲和力的好处——负载平衡和最小化内存访问时间的天然张力
 - 因此，现代多核 NUMA 系统的调度算法变得相当复杂



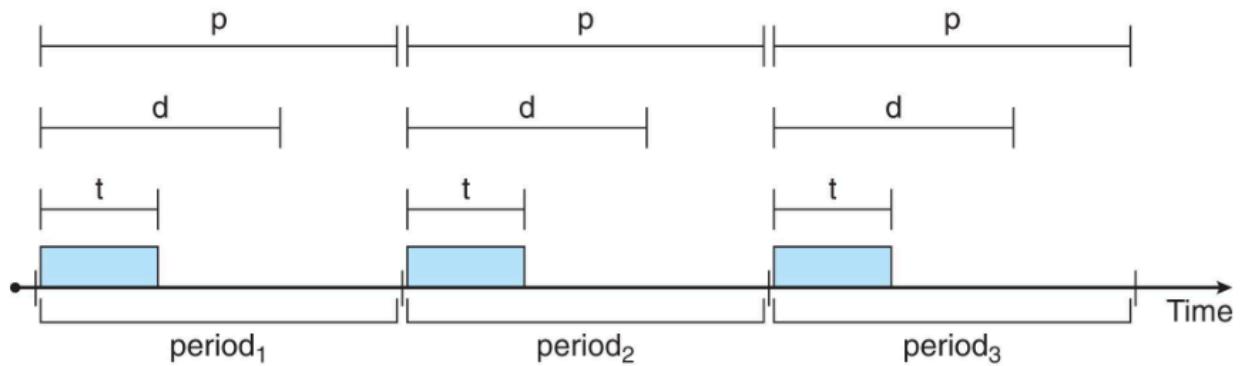
实时CPU调度

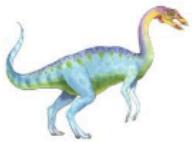
- 实时调度需要性能保证——可预测性 硬实时系统——有更严格的要求。任务必须在截止日期前完成；期限过后的服务与根本没有服务相同
- 软实时系统——不保证何时安排关键的实时进程。它们仅保证实时进程优先于非关键进程
- 实时操作系统的调度程序必须支持基于优先级的抢占算法



基于优先级的调度

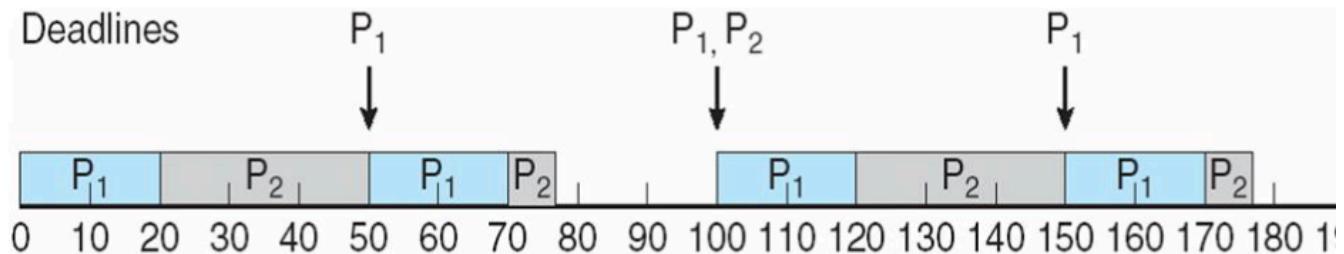
- 请注意，提供抢占式、基于优先级的调度程序只能保证软实时功能。进程具有以下特点：周期性进程需要以恒定的时间间隔（周期）使用 CPU
 - 具有处理时间 t 、截止时间 d 、周期 p ，其中 $0 \leq t \leq d \leq p$
 - p 周期性任务的速率为 $1/p$
 - 进程可能必须向调度程序宣布其截止日期要求。调度程序根据是否能保证进程按时（在截止日期前）来决定是否接纳该进程





速率单调调度

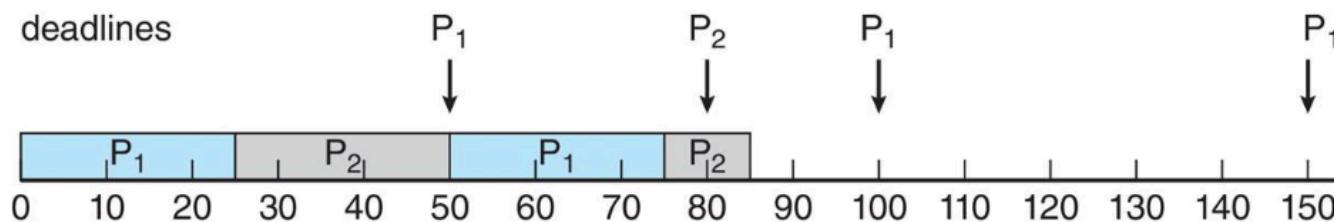
- 静态优先级是根据其周期的倒数分配的
 - 更短（更长）的周期=更高（更低）的优先级；基本原理是为更频繁的任务分配更高的优先级
- 假设 P1 的周期为 50（也是截止时间），处理时间为 20。P2 的周期为 100（也是截止时间），处理时间为 35。
 - 每个进程的最后期限要求它在下一个周期开始前完成其 CPU 突发。由 < 100 , P1 被分配比 P2 更高的优先级 进程 Pi 的 CPU 利用率为其突
 - 周期 t_i/P_i 的比率 P1 的 CPU 利用率为 $20/50 = 0.40$, P2 的 CPU 利用率为 $35/100 = 0.35$, 因此 CPU 总利用率为 75%

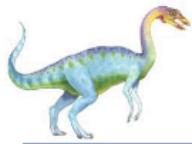




错过了单率调度的最后期限

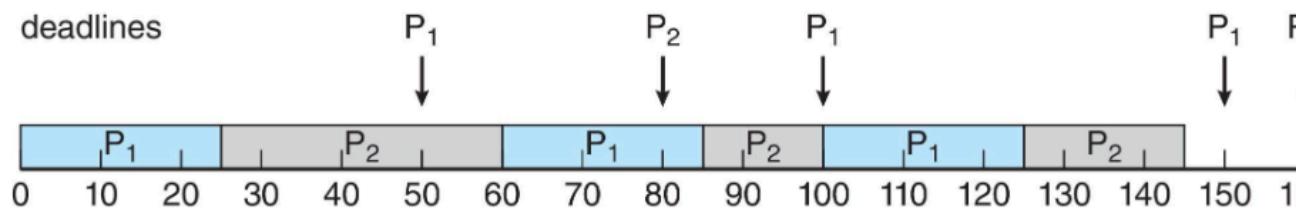
- 假设 P1 的周期为 50 (也是截止时间) , 处理时间为 25。P2 的周期为 8 (也是截止时间) , 处理时间为 35。
 - 由于 $50 < 80$, P1 被分配比 P2 更高的优先级 P1 的 CPU 利用率为 $25/50 = 0.50$, P2 的 CPU 利用率为 $35/80 = 0.44$, 总 CPU 利用率为 94%
- 进程 P2 在时间 85 完成, 错过了截止时间 (80)

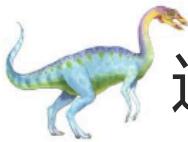




最早截止日期优先调度 (EDF)

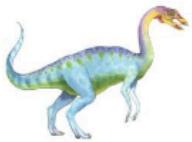
- 最早截止时间优先 (EDF) 调度根据截止时间动态分配优先级
 - 截止日期越早 (晚), 优先级越高 (低)
- 考虑同一示例, 其中 P₁ 的周期为 50 (也是截止时间), 处理时间为 25. P₂ 的周期为 80 (也是截止时间), 处理时间为 35。





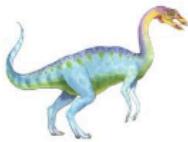
速率单调与 EDF 调度

- 速率单调调度算法使用具有抢占功能的静态优先级策略来调度周期性任务
- 速率单调调度被认为是最优的，因为如果一组进程不能由该算法调度，则也不能由任何其他分配静态优先级的算法调度。
- 与速率单调算法不同，EDF 调度不要求进程是周期性的，也不要求进程每次突发需要恒定的 CPU 时间。唯一的要求是进程在变得可运行时向调度程序布其截止日期
- EDF 调度理论上是最优的 - 它可以调度进程，使每个进程都能满足其截止日期要求，并且 CPU 利用率将达到 100%
 - 但实际上，由于进程之间的上下文切换和中断处理的成本，不可能达到这种级别的 CPU 利用率



算法评估

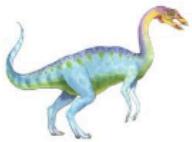
- 在实践中选择 CPU 调度算法可能很困难 - 因为有许多调度算法，每种算法都有自己的一组参数 第一个问题是定义选择算法时使用的标准 - 通常根据 CPU 利用率、响应时间来定义，或吞吐量
- 确定标准——标准可能包括几个具有相对重要性的措施，例如
 - 在最大响应时间300毫秒的约束下最大化CPU利用率
 - 最大化吞吐量，使周转时间（平均）与总执行时间成线性比例



确定性建模

- 确定性建模采用特定的预定工作负载，并定义该工作负载的每种算法的性能
- 确定性建模简单且快速。它为我们提供了准确的数字来比较算法。然而，需要输入精确的数字，并且它的答案仅适用于这些情况
- 流程的运行方式会随时变化，因此没有静态的流程（或时间）集可用于确定性建模
- 考虑 5 个进程在时间 0 到达：

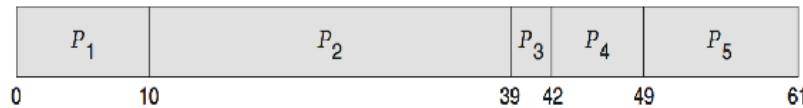
<u>Process</u>	<u>Burst Time</u>
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12



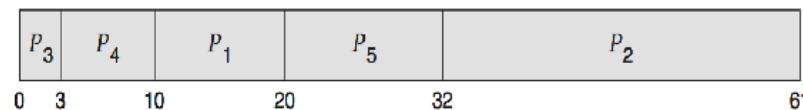
确定性评估

- 对于每种算法，计算平均等待时间 简单快速，但需要输入精确的数字，仅适用于那些输入

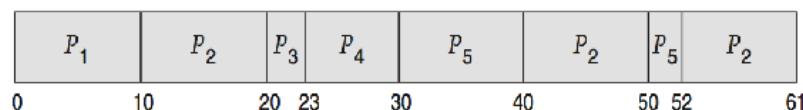
- FCFS 为 28 毫秒：

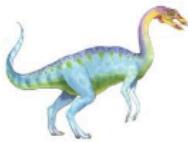


- 非抢占式SJF为13ms：



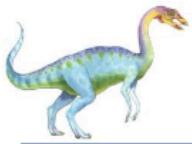
- RR (量子10) 为23ms：





排队分析

- 尽管实际数量（例如，进程到达时间、CPU 或 I/O 突发）随时间（系统）的不同而变化，但 CPU 和 I/O 突发以及进程到达时间的分布是可以测量确定的。然后近似或简单估计
- 计算机系统可以描述为一个服务器网络，每个服务器都有一个等待进程的队列。
 - CPU 是一个服务器，有它的就绪队列，I/O 系统有它的设备队列，通常使用指数分布，并用平均值来描述
 -
- 了解到达率和服务率，我们可以计算利用率、平均队列长度、平均等待时间等。该研究领域称为排队网络分析
-
- 排队分析在比较调度算法时很有用，但可以处理的算法类别和分布非常有限。通常，数学模型易于处理的假设在实践中是不现实的

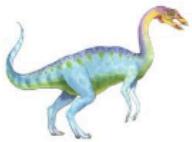


排队模型

- 处理随机工作负载的数学方法 $n = \text{平均队列长度}$
-
- $W = \text{队列中的平均等待时间}$ $\lambda = \text{进入队列的平均到达率 Little's Law}$
- 公式 – 在稳定状态下（有数学假设使其成立，例如，到达率必须小于服务率），离开队列的进程必须等于到达的进程，因此：

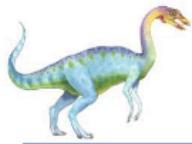
$$n = \lambda \times W \text{ 对于任何调度算法和到达分布都有效}$$

-
- 例如，如果平均每秒有 7 个进程到达，并且队列中通常有 14 个进程，则每个进程的平均等待时间 = 2 秒

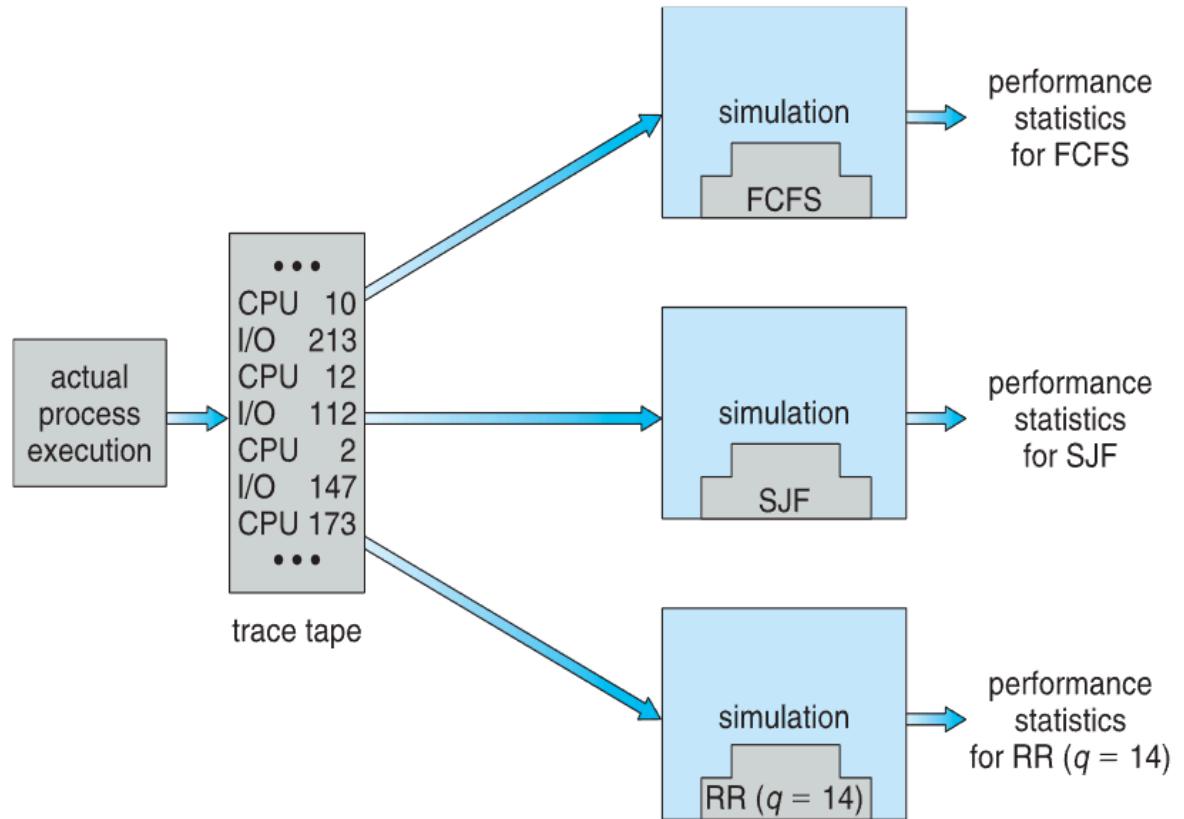


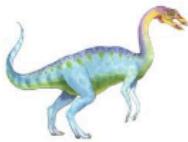
模拟

- 排队模型仅限于一些已知的分布运行模拟涉及对计算机系统的模型进行编程 - 这更准确
- 随着时钟值的增加，模拟器会修改系统状态以反映设备、进程和调度的活动。当模拟执行时，会收集并打印表明算法性能的统计数据。
-
- 驱动模拟的数据可以通过多种方式生成
 - 根据概率分布的随机数生成器 - 可以通过数学方式（均匀分布、指数分布、泊松分布）或经验方式定义分布 跟踪文件来监控真实系统并记录实际事件的序列
 -



通过仿真评估 CPU 调度器





执行

- 即使模拟的准确性也有限构建一个系统，允许实际算法与真实数据一起运行 - 更加灵活和通用。
- 在实际系统中实现新的调度程序和测试有困难：
 - 这会产生高成本（编码新的调度程序）和高风险（例如，可能引入新的错误）
 - 环境也在不断变化
- 最灵活的调度算法是那些可以由系统管理器更改的算法，以便可以针对特定应用程序进行调整
 - 例如，支持图形应用程序或 Web（文件）服务的系统可能有不同的调度需求
 - 许多 UNIX 系统允许系统管理员针对特定系统配置微调调度参数
- API 可用于修改进程或线程的优先级 - 提高特定应用程序的性能，而不是整个应用程序性能
 - Java、POSIX 和 Windows API 提供了此类功能

第五章结束

