

CPU Scheduling

Operating Systems

郑贵锋 博士
中山大学计算机学院
zhenggf@mail.sysu.edu.cn
https://gitee.com/code_sysu



目录

- 基本概念
- 调度标准
- 简单调度算法
- 高级调度算法
- 多处理器调度
 - 处理器亲和性
 - 负载均衡
 - 多核处理器
- 实时CPU调度
 - 最小化延迟
 - 基于优先级的调度
 - 单调速率调度
 - 最早截止时间优先调度
 - 最小松弛度优先算法
 - 比例分享调度
 - POSIX实时调度
- 算法评估



■ 多处理器调度

- 上一节，我们的讨论集中在单处理器系统中的CPU调度问题上。如果有多个CPU可用，负载分配(load sharing)成为可能，调度问题相应地变得更加复杂。
 - 已经尝试了许多可能性，正如我们在单处理器CPU调度中看到的，没有一个最佳的解决方案。
- 多处理机调度中的几个问题
 - 我们专注于处理器相同(同质)的系统—就其功能而言。
 - 然后，我们可以使用任何可用的处理器来运行队列中的任何进程。
 - 然而，即使使用同质多处理器，有时也存在调度限制。
 - 考虑有一个I/O设备连接到一个处理器的专用总线上的系统。希望使用该设备的进程必须计划在该处理器上运行。



■ 多处理器调度

- 多处理器调度方法
 - 非对称多处理
 - 单处理器（主服务器）处理所有调度决策、I/O处理和其他系统活动。其他处理器只执行用户代码。
 - 这种方法很简单，因为只有一个处理器访问系统数据结构，减少了数据共享的需求。
 - 对称多处理(SMP)
 - 所有进程都可能在一个公共就绪队列中，或者每个处理器都可能有自己的就绪进程专用队列。
 - 如果我们有多个处理器试图访问和更新一个公共数据结构，那么必须仔细编写调度程序。
 - 我们必须确保两个独立的处理器不会选择调度同一个进程，并且进程不会从队列中丢失。
 - 几乎所有现代操作系统都支持SMP，包括Windows、Linux和MacOS。



■ 处理器亲和性

- 考虑进程在特定处理器上运行时缓存内存会发生什么。进程最近访问的数据**填充处理器的缓存**。因此，进程的连续内存访问通常在缓存中得到满足。
- 现在考虑如果进程迁移到另一个处理器会发生什么。必须使第一个处理器的缓存内容无效，并且必须重新填充第二个处理器的缓存。
- 由于缓存失效和重新填充的成本很高，大多数SMP系统试图避免将进程从一个处理器迁移到另一个处理器，而是试图让进程在同一个处理器上运行。这被称为**处理器亲和性**(processor affinity).
 - 处理器亲和性表示进程与**当前运行**它的处理器具有关联性。



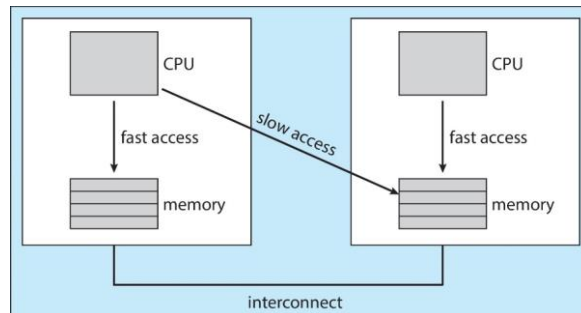
■ 处理器亲和性

- 软亲和性
 - 操作系统将尝试使进程在同一处理器上运行，但进程可能在处理器之间迁移。
- 硬亲和性
 - 某些操作系统提供系统调用，允许进程指定其可以运行的处理器子集。
- 许多系统提供软亲和和硬亲和。
 - Linux实现软亲和，并通过提供系统调用**sched_setaffinity()**来支持硬亲和。



■ 处理器亲和性

- 系统的主存体系结构可能会影响处理器亲和性问题。
 - 例如，在以非一致内存访问（NUMA）为特征的体系结构中，CPU对主存某些部分的访问速度比其他部分快。单板上的CPU访问该单板上的内存的速度比访问系统中其他单板上的内存的速度快。



■ 负载均衡

- **负载均衡（平衡）** 试图使工作负载均匀分布在SMP系统中的所有处理器上，以充分利用拥有多个处理器的好处。
- 需要注意的是，负载均衡通常仅在每个处理器拥有各自私有的可运行进程队列的系统上才有必要。
 - 在大多数支持SMP的现代操作系统中，每个处理器都有一个可运行进程的专用队列。
- 在具有公共运行队列的系统上，负载均衡通常是不必要的，因为一旦处理器空闲，它会立即从公共运行队列中提取可运行的进程。



■ 负载均衡

- 有两种通用的负载均衡方法
 - **推迁移**
 - 特定任务定期检查每个处理器上的负载，如果发现不平衡，则通过将进程从过载移动到空闲或不太繁忙的处理器来均匀分配负载。
 - **拉迁移**
 - 当空闲处理器从繁忙处理器中提取等待的任务时，会发生拉迁移。
- 推迁移和拉迁移不需要相互排斥，事实上通常在负载均衡系统上并行实现。
 - 例如，Linux调度程序和用于FreeBSD系统的ULE调度程序都实现了这两种技术。
- 负载均衡通常会抵消处理器亲和性的好处。
 - 正如系统工程中经常出现的情况一样，关于什么策略是最好的没有绝对规则。



■ 多核处理器

- 多核处理器将多个处理器核（物理）放置在同一芯片上。每个核心都保持其体系结构状态，因此在操作系统看来是一个独立的物理处理器。
- 使用多核处理器的SMP系统比每个处理器都有自己的物理芯片的系统速度更快，功耗更低。
- 内存停顿(存储器停顿)
 - 当处理器访问内存时，它可能会花费大量时间等待数据可用。这种情况称为**内存停顿**。
 - 内存停顿可能由于各种原因而发生，例如缓存未命中（访问不在缓存内存中的数据）。

Multiple-Processor Scheduling

11 / 69

■ 多核处理器

■ 内存停顿的图示。

■ 处理器可能花费高达50%的时间等待数据从内存中可用。

The diagram illustrates the execution of a single thread over time. A horizontal timeline is shown with a series of eight blocks representing execution cycles. The blocks alternate between 'C' (compute cycle) and 'M' (memory stall cycle). The sequence is C, M, C, M, C, M, C, M. Above the timeline, a legend shows a box labeled 'C' with the text 'compute cycle' and a box labeled 'M' with the text 'memory stall cycle'. An arrow labeled 'thread' points to the start of the sequence, and an arrow labeled 'time' points to the right along the timeline.

Multiple-Processor Scheduling

12 / 69

■ 多核处理器

■ 为了纠正这种情况，许多最近的硬件设计都实现了多线程处理器内核，其中每个内核分配两个（或更多）硬件线程。这样，如果一个线程在等待内存时暂停，内核可以切换到另一个线程。

■ 双线程处理器内核的图示

■ 线程0的执行和线程1的执行在其上交错。

The diagram illustrates the execution of two threads, thread₀ and thread₁, on a dual-thread processor core. Two horizontal timelines are shown, one for each thread. The sequence of execution cycles for thread₁ is C, M, C, M, C, M, C. The sequence for thread₀ is C, M, C, M, C, M, C. The timelines are interleaved, showing that the processor can execute both threads simultaneously, switching between them during memory stall cycles. An arrow labeled 'thread₁' points to the start of its sequence, and an arrow labeled 'thread₀' points to the start of its sequence. A horizontal arrow labeled 'time' points to the right at the bottom.



■ 多核处理器

- 从操作系统的角度来看，每个硬件线程都显示为可用于运行软件线程的逻辑处理器。因此，在一个双线程、双核系统上，向操作系统提供了4个逻辑处理器。
 - UltraSPARC T3 CPU每个芯片有16个内核，每个内核有8个硬件线程。从操作系统的角度来看，似乎有128个逻辑处理器。
- 处理器核的多线程方法
 - 两种方法：**粗粒度**(coarse-grained) 和 **细粒度**(fine-grained)。



■ 多核处理器

- 粗粒度多线程
 - 线程在处理器上执行，直到发生长延迟事件（如内存停顿）。由于长延迟事件导致的延迟，处理器必须切换到另一个线程才能开始执行。
 - 在线程之间切换的成本很高，因为在另一个线程开始在处理器内核上执行之前，必须刷新指令管道。一旦这个新线程开始执行，它就开始用它的指令填充管道。
- 细粒度（或交错）多线程
 - 细粒度多线程在更细粒度上切换线程—通常在指令周期的边界。
 - 细粒度系统的体系结构设计包括线程切换逻辑。
 - 在线程之间切换的成本很小。



■ 多核处理器

- 多核处理器多线程的调度级别
 - 这种调度有两个不同的级别：调度决策和硬件线程选择
 - **调度决策**由操作系统做出，选择在每个硬件线程（逻辑处理器）上运行哪个软件线程。操作系统可以选择我们在上一课中讨论过的任何调度算法。
 - 另一级别的调度中，每个处理器核决定运行哪个硬件线程。
 - UltraSPARC T3使用简单的轮转算法将8个硬件线程调度到每个核上。
 - 英特尔Itanium（双核处理器，每个核有两个硬件线程）为每个硬件线程分配了一个动态紧迫值，范围从最低 0 到最高 7。Itanium识别出可能触发线程切换的5种不同事件。当其中一个事件发生时，线程切换逻辑选择具有较高紧迫值的线程在处理器核上执行。



■ 实时CPU调度

- **软实时系统** (soft real-time system)保证关键实时进程优先于非关键进程，但不保证关键实时进程的调度时间。
- **硬实时系统** (hard real-time system)有更严格的要求。任务必须在截止时间前完成；截止时间过后的服务等同于根本没有服务。
- 软实时和硬实时操作系统中与进程调度相关的问题：
 - 最小化延迟
 - 基于优先级的调度
 - 单调速率调度(RMS, Rate-Monotonic Scheduling)
 - 最早截止时间优先调度(EDFS, Earliest-Deadline-First Scheduling)
 - 最小松弛度优先算法(Least Laxity First Algorithm)
 - 比例分享调度(Proportional Share Scheduling)



■ 最小化延迟

- 实时系统是事件驱动的。当事件发生时，系统必须尽快响应和服务它

- 软件事件

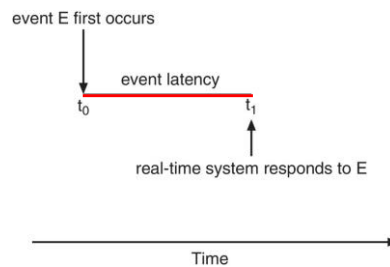
- 例如，计时器超时。

- 硬件事件

- 例如，遥控车辆检测到它正在接近障碍物。

- 事件延迟

- **事件延迟**(Event latency)是从事件发生到得到服务所经过的时间量。



■ 最小化延迟

- 两种类型的延迟会影响实时系统的性能：

- 中断延迟

- 调度延迟

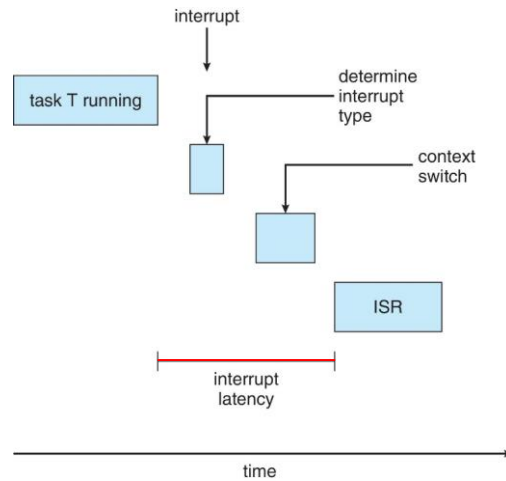
- **中断延迟**

- 是指从中断到达CPU到为中断提供服务的例程开始的时间段。
 - 发生中断时，中断延迟是操作系统执行以下任务所需的总时间：
 - 首先完成它正在执行的指令
 - 确定发生的中断类型
 - 保存当前进程的状态
 - 使用指定中断服务程序(ISR)为中断提供服务
 - 对于硬实时系统，中断延迟必须有界以满足严格的要求
 - 更新内核数据结构时禁用中断的时间量，这一点很重要



■ 最小化延迟

■ 中断延迟



■ 最小化延迟

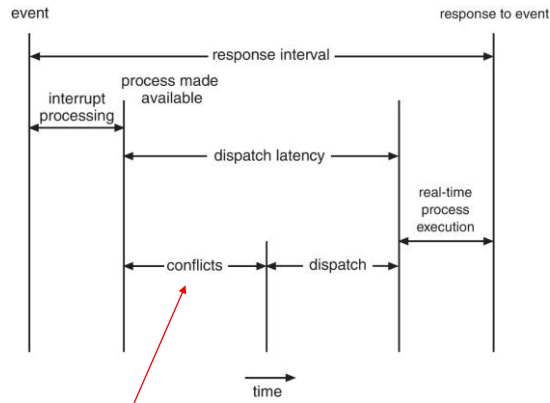
■ 调度延迟

- 是指调度分派程序(Dispatcher)停止一个进程并启动另一个进程所需的时间量。
 - 为实时任务提供对CPU的即时访问要求实时操作系统也将调度延迟降至最低。
 - 保持低调度延迟的最有效技术是提供抢占式内核。
- 调度延迟的构成在下一张幻灯片中说明。调度延迟的**冲突阶段**有两个组成部分：
 - 抢占内核中运行的任何进程
 - 由低优先级进程释放高优先级进程所需的资源
- 例如，在Solaris中，禁用抢占的调度延迟超过100ms。启用抢占后，它将减少到不到一ms。



■ 最小化延迟

■ 调度延迟



■ 调度延迟冲突阶段的两个组成部分：

- 抢占内核中运行的任何进程
- 由低优先级进程释放高优先级进程所需的资源。



■ 基于优先级的调度

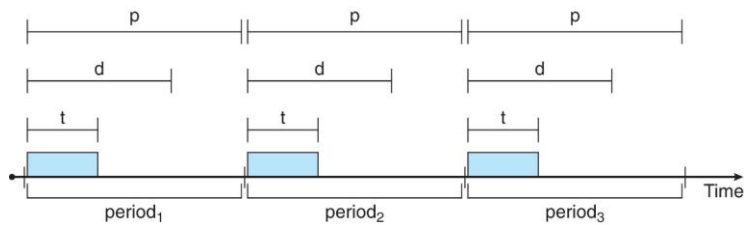
- 实时操作系统的调度程序**必须**支持基于优先级的抢占算法。
 - 实时操作系统最重要的功能是在实时进程需要CPU时立即响应该进程。
- 注意，提供抢占式、基于优先级的调度程序只能保证软实时功能。硬实时系统必须进一步保证实时任务将按照其截止期限要求提供服务，而实现这种保证需要额外的调度功能。
 - 稍后我们将介绍适用于硬实时系统的调度算法。
- Linux、Windows和Solaris为实时进程分配最高的调度优先级。
 - Windows有32种不同的优先级。最高级别的优先级值16至31留给实时进程。
 - Solaris和Linux有类似的优先级划分方案。



■ 单调速率调度

■ 周期性进程

- 如果进程需要CPU以**恒定的**间隔（周期）运行，则该进程是**周期性的**。一旦周期性进程获得了CPU，它就有**一个固定的处理时间** t 、**一个必须由CPU服务的截止期限** d 、**一个周期** p 和**一个周期性任务的速率** $1/p$ 。进程的**CPU利用率**可以用 t/p 来衡量。
- t 、 d 和 p 的关系可以表示为
$$0 \leq t \leq d \leq p$$



■ 单调速率调度

■ 准入控制

- 进程可能必须向调度程序宣布其截止期限要求。使用准入控制算法，调度程序执行以下两项操作之一：
 - 允许**进程进入，保证进程按时完成。
 - 拒绝**不可能的请求。



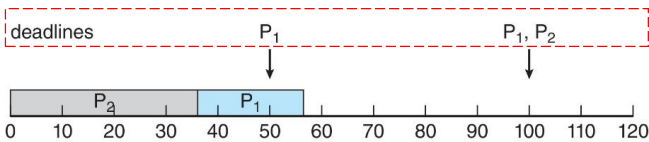
■ 单调速率调度

- 单调速率调度(RMS, *Rate-Monotonic Scheduling*) 算法使用带有抢占的静态优先级策略调度周期性任务。
- 进入系统后, 每个周期性任务根据其速率 (即周期的倒数) 分配优先级。
 - 周期越短 (速率越大), 优先级越高。也就是说, 为更经常需要CPU的任务分配更高的优先级。
- 此外, 单调速率调度假设周期进程的处理时间对于每个CPU执行是相同的。
 - 假设每次进程获取CPU时, 其CPU执行的持续时间都是相同的。



■ 单调速率调度

- 例1
 - 设 P_1 和 P_2 为周期进程, 周期 $p_1=50$, 处理时间 $t_1=20$, $p_2=100$, $t_2=35$. 假设截止期限 $d_i=p_i$, 每个进程的截止期限要求它在下一个周期开始前完成CPU执行。
 - 这两个进程的CPU综合利用率为
$$(20/50)+(35/100) = 0.75$$
因此, 这两个进程可以合理的调度, 且CPU仍有25%的可用时间。
 - 假设我们为 P_2 分配了比 P_1 更高的优先级。这种情况下 P_1 和 P_2 的执行如下所示。
 - P_2 首先开始执行, 并在时间35完成。在这一点上, P_1 开始执行; 它在时间55时完成其CPU执行。但是, P_1 的第一个截止期限是在时间50, 因此调度程序已导致 P_1 错过其截止期限。



Real-Time CPU Scheduling

27 / 69

■ 单调速率调度

■ 例1

■ 现在假设使用单调速率调度：因 $50=P_1<P_2=100$, P_1 的周期更短，给 P_1 分配比 P_2 更高的优先级。此时，进程的执行情况如下：

■ P_1 首先启动，并在时间20完成其CPU执行，满足其第一个截止期限。 P_2 在此点开始运行，直到时间50。此时，它被 P_1 抢占(P_2 的CPU执行中仍有5剩余)。 P_1 在时间70完成其CPU执行，此时调度程序恢复 P_2 。 P_2 在时间75完成其CPU执行，也符合其第一个截止期限。系统处于空闲状态，直到时间100，此时 P_1 再次被调度。

deadlines

P_1

P_1, P_2

P_1

P_1, P_2

0 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200

Real-Time CPU Scheduling

28 / 69

■ 单调速率调度

■ 例2. 错过截止期限的RMS

■ 设 P_1 和 P_2 为周期进程。 $p_1=50, t_1=25, p_2=80, t_2=35$ 。假设 $d_i=p_i$, 每个进程的截止期限也是下一个周期的开始。单调速率调度将为进程 P_1 分配更高的优先级。CPU的综合利用率为

$$(25/50)+(35/80) = 0.94$$

■ 进程 P_1 和 P_2 的调度如下所示。最初， P_1 运行直到时间25完成其CPU执行。然后， 进程 P_2 开始运行并一直运行到时间50，此时它被 P_1 抢占， P_2 的CPU执行仍差10。 进程 P_1 运行到时间75， P_2 在时间85完成其执行，错过了在时间80完成其CPU执行的截止期限

deadlines

P_1

P_2

P_1

P_1, P_2

0 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160

P2错过了80岁的最后期限



■ 单调速率调度

- 单调速率调度被认为是最优的。
 - 如果一组进程不能由RMS调度，那么它也不能由任何其他分配静态优先级的算法调度。
- RMS有一个限制：CPU利用率是有限的，并且并不总是可能完全最大化CPU资源。对于调度N个进程，最坏情况下CPU利用率为

$$N \times (2^{1/N} - 1).$$
 - 当N = 1, $N \times (2^{1/N} - 1) = 1.$
 - 当N = 2, $N \times (2^{1/N} - 1) \approx 0.83.$
 - 当N → ∞, $\lim_{n \rightarrow \infty} (N \times (2^{1/N} - 1)) = \ln(2) \approx 0.69.$
- 在例1中，调度两个进程的综合CPU利用率是75% (< 0.83); 因此，RMS算法保证对他们进行调度，使他们能在截止期限前完成任务。
- 在例2中，综合的CPU利用率大约是94%; 因此，RMS不能保证他们能够被调度以满足他们的截止期限。



■ 单调速率调度

- **定理.** 一组n个周期性硬实时任务 $T = \{\tau_1, \dots, \tau_n\}$ 能在不依赖其开始时间的情况下在截止期限前完成任务，如果

$$U = \sum_i^n \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

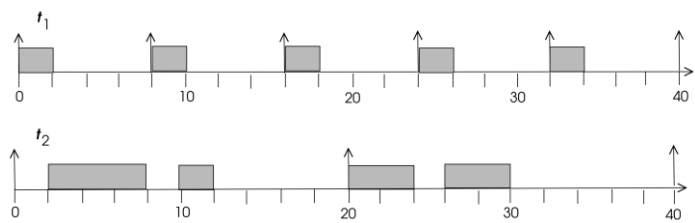
其中 C_i 为最坏情况时间， T_i 为任务 τ_i 的周期。

- 当n>10时，利用率系数U收敛于ln2=0.69. 因此，我们可以使用此不等式测试一个任务集是否可以调度。
- 注意，该测试是充分非必要条件。即，可能有一个任务集的总利用率大于0.69，但在RM调度下，该任务集中的任务仍可能满足其截止期限。



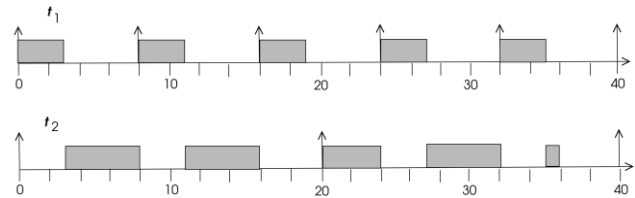
■ 单调速率调度

- 例3.
 - 在RM调度中，考虑 $\tau_i(C_i, T_i)$ 形式的任务集 $\tau_1(2, 8)$ 和 $\tau_2(8, 20)$. 我们可以看到，这两项任务可以从
$$(2/8) + (8/20) = 0.65 < 2(2^{0.5} - 1) \approx 0.83.$$
 - 因此，此任务集可以被允许进入系统。下面描述了这些任务使用RM的调度。由于其周期的LCM（最小公倍数）为40，调度将每40个时间单位重复一次。



■ 单调速率调度

- 例4.
 - 现在让我们增加任务的计算时间，使其具有 $\tau_1(3, 8)$ 和 $\tau_2(10, 20)$. 应用可调度性测试，得
$$(3/8) + (10/20) = 0.875 > 0.83.$$
 - 也就是说，RM算法不能保证该任务集有一个可行的调度。但是，我们仍然可以按如下所示调度这些任务，以满足其截止期限。



- 该方法的优点是实现简单，在所有静态优先级调度算法中是最优的。然而，如上面的示例所示，它可能不能有效地使用处理能力。



■ 最早截止期限优先调度

- 最早截止期限优先(EDF, *Earliest-Deadline-First*) 是一种基于优先级的调度。它根据下一个截止期限动态分配优先级，下一个截止期限越早，优先级越高。
 - 根据EDF策略，当进程变得可运行时，它必须向系统声明其截止期限要求。可能必须调整优先级，以反映新运行进程的截止期限。
- 注意，这与优先级固定的单调速率调度有所不同。



■ 最早截止期限优先调度

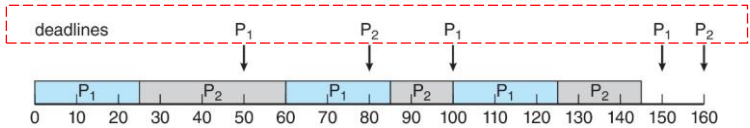
- 例5
 - 考虑例2的进程 P_1 和 P_2 . $p_1=50, t_1=25, p_2=80, t_2=35$, 每个截止期限在其下一个周期开始时($d_i=p_i$). 此例使用RMS将使进程错过截止期限。
 - 这些进程的EDF调度如下所示
-
- 0 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160
- 进程 P_1 的初始优先级高于 P_2 , 因为它的第一个截止期限 $d_1 = 50 < d_2 = 80$ 。
 - 进程 P_2 在 P_1 的CPU执行结束时开始运行。
 - P_2 现在的优先级高于 P_1 , 因为它的下一个截止期限 $d_2 = 80 < d_1 = 100$. EDF调度允许进程 P_2 继续运行。因此, P_1 和 P_2 都满足其第一个截止期限。
 - 然而, 回想一下, RMS允许 P_1 在时间50的下一个周期开始时抢占 P_2 .



■ 最早截止时间优先调度

■ 例5

- 考虑例2的进程 P_1 和 P_2 . $p_1=50, t_1=25, p_2=80, t_2=35$, 每个截止时间在其下一个周期开始时($d_i=p_i$). 此例使用RMS将使进程错过截止时间。
- 这些进程的EDF调度如下所示



- 进程 P_1 在时间60再次开始运行, 并在时间85完成其第二次CPU执行, 也满足其第二个截止时间100. P_2 则从85开始运行。
- 在时间100, P_2 被 P_1 抢占, 因为此时 $d_1 = 150 < d_2 = 160$.
- 在时间125, P_1 完成其CPU执行, P_2 恢复执行, 在时间145完成并满足其最后期限。系统处于空闲状态, 直到时间150, P_1 被调度运行其下一个周期。



■ 最早截止时间优先调度

- 与RMS不同, EDF调度不要求进程是周期性的, 也不要求进程每次执行都需要恒定的CPU时间。
 - 唯一的要求是进程在可运行时向调度器声明其截止时间。
 - 它适用于非周期性和零星的任务调度。
- EDF调度的魅力在于它在理论上是最优的。
 - 理论上, 它可以调度进程, 使每个进程都能满足其截止时间要求, 且CPU利用率可达100%。
 - 然而, 在实践中, 由于进程之间的上下文切换和中断处理的成本, 不可能实现这种CPU利用率。

Real-Time CPU Scheduling

39 / 69

■ 最小松弛度优先算法

■ 例6

■ 当 $t=4$ 时 τ_2 被激活，其松弛度值低于 τ_1 ，从而 τ_1 被抢占。

■ 当 $t=8$ 时 τ_3 变为可用，因 τ_2 在所有3项任务中松弛度最小，它不存在被抢占。

■ 当 $t=10$ 时，有效任务为 τ_1 和 τ_3 ， τ_1 具有较低的松弛度值，因此被调度。

Real-Time CPU Scheduling

40 / 69

■ 比例分享调度

■ 比例分享调度程序通过在所有 n 个应用程序之间分配 T 份时间配额来运行。让第 i 个应用程序接收 N_i 份时间，使得 $T < \sum N_i$ ($i = 1$ to n). 第 i 个应用程序的总处理器时间为 N_i/T .

■ 例7

■ 假设总共 $T=100$ 份时间将分配给A, B, C三个进程. $N_A=50$, $N_B=15$, $N_C=20$ 。比例分享调度程序确保A拥有总处理器时间的50%，B拥有15%，C拥有20%。

■ 比例分享调度程序必须与准入控制策略配合使用，以确保应用程序收到其分配的时间份额。只有在有足够的份额可用时，准入控制策略才会允许请求特定数量份额的进程调度。

- 由于总共100份中的 $50+15+20=85$ 份已分配，如果新进程D请求30份，准入控制程序将拒绝D进入系统。

20



■ POSIX实时调度

- POSIX.1b是用于实时计算的POSIX标准扩展。它为实时线程定义了两个调度类：

SCHED_FIFO
SCHED_RR

- **SCHED_FIFO**使用FIFO队列根据FCFS策略调度线程。在优先级相同的线程之间没有时间切片。位于FIFO队列前端的最高优先级实时线程将被授予CPU，直到其终止或阻塞。
- **SCHED_RR**使用RR策略。它与SCHED_FIFO类似，不同之处在于它在优先级相同的线程之间提供时间切片。
- **SCHED_OTHER**是POSIX提供的一个附加调度类，但它的实现是未定义的和特定于系统的；它在不同的系统上可能表现不同。



■ POSIX实时调度

- POSIX API为获取和设置调度策略指定了以下两个函数：

```
pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)
pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)
```

- 两个函数的第1个参数是线程属性指针。第2个参数在
 - (1) pthread_attr_getsched_policy()函数里，是指向设置为当前调度策略的整数的指针；
 - (2) pthread_attr_setsched_policy()函数里，是整数值 SCHED_FIFO, SCHED_RR, 或SCHED_OTHER.

- 如果发生错误，这两个函数都返回非零值。

- 下一张幻灯片将演示使用此API的POSIX Pthread实时调度程序。该程序首先确定当前调度策略，然后将调度算法设置为SCHED_FIFO。



■ POSIX实时调度

■ 算法.20-1-pthread-scheduling-1.c: 调度策略与优先级(1)

```

/* compiling with -lpthread option */
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS 5

void *runner(void*);

int main(int argc, char *argv[])
{
    int i, policy;
    pthread_t tid[NUM_THREADS];
    struct sched_param param;
    pthread_attr_t attr;

    /* get the default attributes */
    pthread_attr_init(&attr);

```



■ POSIX实时调度

■ 算法.20-1-pthread-scheduling-1.c: 调度策略与优先级(2)

```

/* get the current scheduling policy */
if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
    printf("Unable to get policy.\n");
else {
    printf("The current scheduling policy is ");
    if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
    else if (policy == SCHED_RR) printf("SCHED_RR\n");
    else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");
}

/* get the current priority */
if (pthread_attr_getschedparam(&attr, &param) != 0)
    printf("Unable to get priority from SCHED_OTHER.\n");
else
    printf("current sched_priority = %d\n", param.sched_priority);
    printf("priority_min of OTHER is %d, max is %d\n",
        sched_get_priority_min(SCHED_OTHER), sched_get_priority_max(SCHED_OTHER));
    param.sched_priority = 10; /* set the priority to 10 */
    if (pthread_attr_setschedparam(&attr, &param) != 0)
        printf("Unable to set priority to 10.\n");

/* get the current priority */
if (pthread_attr_getschedparam(&attr, &param) != 0)
    printf("Unable to get priority from SCHED_RR.\n");
else
    printf("The new sched_priority = %d\n", param.sched_priority);

```



■ POSIX实时调度

■ 算法.20-1-pthread-scheduling-1.c: 调度策略与优先级(3)

```

/* set the scheduling policy to RR */
if (pthread_attr_setschedpolicy(&attr, SCHED_RR) != 0)
    printf("Unable to set policy to SCHED_RR.\n");

/* get the current scheduling policy */
if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
    printf("Unable to get policy.\n");
else {
    printf("The new scheduling policy is ");
    if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
    else if (policy == SCHED_RR) printf("SCHED_RR\n");
    else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");
}

/* get the current priority */
if (pthread_attr_getschedparam(&attr, &param) != 0)
    printf("Unable to get priority from SCHED_RR.\n");
else
    printf("current sched_priority = %d\n", param.sched_priority);
printf("priority_min of RR is %d, max is %d\n",
    sched_get_priority_min(SCHED_RR), sched_get_priority_max(SCHED_RR));

```



■ POSIX实时调度

■ 算法.20-1-pthread-scheduling-1.c: 调度策略与优先级(4)

```

/* set the priority to 10 */
param.sched_priority = 10;
if (pthread_attr_setschedparam(&attr, &param) != 0)
    printf("Unable to set priority.\n");

/* get the current priority */
if (pthread_attr_getschedparam(&attr, &param) != 0)
    printf("Unable to get priority from SCHED_RR.\n");
else
    printf("The new sched_priority = %d\n", param.sched_priority);

/* set the scheduling policy to FIFO */
if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
    printf("Unable to set policy to SCHED_FIFO.\n");

/* get the current scheduling policy */
if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
    printf("Unable to get policy.\n");
else {
    printf("The new scheduling policy is ");
    if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
    else if (policy == SCHED_RR) printf("SCHED_RR\n");
    else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");
}

```



■ POSIX实时调度

■ 算法.20-1-pthread-scheduling-1.c: 调度策略与优先级(5)

```

/* get the current priority */
if (pthread_attr_getschedparam(&attr, &param) != 0)
    printf("Unable to get priority from SCHED_FIFO.\n");
else
    printf("current sched_priority = %d\n", param.sched_priority);
printf("priority_min of FIFO is %d, max is %d\n",
    sched_get_priority_min(SCHED_FIFO), sched_get_priority_max(SCHED_FIFO));

/* set the priority to 50 */
param.sched_priority = 50;
if (pthread_attr_setschedparam(&attr, &param) != 0)
    printf("Unable to set priority.\n");

/* get the current priority */
if (pthread_attr_getschedparam(&attr, &param) != 0)
    printf("Unable to get priority from SCHED_FIFO\n");
else
    printf("The new sched_priority = %d\n", param.sched_priority);

```



■ POSIX实时调度

■ 算法.20-1-pthread-scheduling-1.c: 调度策略与优先级(6)

```

/* create the threads */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], &attr, &runner, NULL);

/* now join on each thread */
for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}

```




Real-Time CPU Scheduling

49 / 69

■ POSIX实时调度

■ 算法.20-1-pthread-scheduling-1.c: 调度策略与优先级(6)

```

/* create the threads */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], &attr, &runner, NULL);

isscg@ubuntu:/mnt/os-2020$ gcc alg.20-1-pthread-scheduling-1.c -pthread
isscg@ubuntu:/mnt/os-2020$ ./a.out
The current scheduling policy is SCHED_OTHER
current sched_priority = 0
priority_min of OTHER is 0, max is 0
Unable to set priority to 10.
The new sched_priority = 0
The new scheduling policy is SCHED_RR
current sched_priority = 0
priority_min of RR is 1, max is 99
The new sched_priority = 10
The new scheduling policy is SCHED_FIFO
current sched_priority = 10
priority_min of FIFO is 1, max is 99
The new sched_priority = 20
isscg@ubuntu:/mnt/os-2020$

```



Example: Linux Scheduling

50 / 69

■ Linux调度

- 在内核版本2.5之前
 - 未使用SMP系统设计，也未充分支持具有多个处理器的系统。
 - 导致具有大量可运行进程的系统性能低下。
- 内核2.5
 - 包括一个O(1)调度算法
 - 无论系统中有多少任务，它都以固定时间运行。
 - 增加对SMP系统的支持，包括处理器亲和性和处理器之间的负载均衡。
 - 导致交互进程的响应时间较差。
- 内核2.6
 - 在内核的2.6.23版本中，**完全公平调度程序(CFS, Completely Fair Scheduler)**成为默认的Linux调度算法。



■ 调度类

- Linux系统中的调度基于调度类。
 - 每个类都被分配了一个特定的优先级。
- 不同的调度类有其特定的调度算法
 - 例如，Linux服务器的调度标准可能不同于运行Linux的移动设备的调度标准。
- 调度程序选择属于最高优先级调度类的最高优先级任务。
- 标准Linux内核实现两个调度类：
 - 使用CFS调度算法的默认调度类。
 - 一个实时调度类。
- 当然，可以添加新的调度类。



■ 完全公平调度

- nice友好度值
 - 完全公平调度程序(CFS)为每个进程（任务）分配一定比例的CPU处理时间。该比例是根据分配给每个任务的nice值计算的。
 - nice一词来自这样一种想法：如果一个任务增加了其nice值，那么它通过降低其相对优先级来对系统中的其他任务表示友好。
 - nice值默认为0，范围为-20到+19，其中nice值越低表示相对优先级越高。
 - 具有较低nice值的任务将获得较高比例的CPU处理时间。
 - CFS不使用离散的时间片值，而是标定一定的目标延迟，即每个可运行任务至少运行一次的时间间隔。CPU时间的比例是根据目标延迟值分配的。除了具有默认值和最小值外，如果系统中活动任务的数量增长超过某个阈值，目标延迟也会增加。



■ 完全公平调度

■ 虚拟运行时间

- CFS不直接分配优先级。相反，它通过使用每个任务的变量 `vruntime`，来维护每个任务的虚拟运行时间，来记录每个任务已运行的时间。
- 虚拟运行时间与基于任务优先级的**衰减因子**相关联：优先级较低的任务比优先级较高的任务具有更高的衰减率。
- 对于具有正常优先级的任务（`nice`值为0），虚拟运行时间与实际物理运行时间相同。
 - 例如，如果具有默认优先级的任务运行200ms，则其 `vruntime`也将为200ms。
- 对于优先级较低的任务，其虚拟运行时间将高于其实际运行时间。同样，高优先级任务的`vruntime`将小于其实际运行时间。



■ 完全公平调度

■ 虚拟运行时

- 假设两个任务具有相同的`nice`值。一个任务是I/O密集的，另一个是CPU密集的。然后，I/O密集任务的`vruntime`值最终将低于CPU密集任务，从而使I/O密集任务的优先级高于CPU密集任务。此时，如果在I/O密集任务可运行时CPU密集任务正在执行，则I/O密集任务将抢占CPU密集任务。

Example: Linux Scheduling

55 / 69

■ 实时调度

■ Linux还使用POSIX标准实现实时调度，如下所述。使用SCHED_FIFO或SCHED_RR实时策略调度的任何任务的优先级都高于正常（非实时）任务。

■ Linux使用两个独立的优先级范围，一个用于实时任务，另一个用于正常任务。

● 实时任务分配的静态优先级在0到99之间，正常任务分配的优先级在100到139之间。

■ 这两个范围映射成一个全局优先级方案，其中数值越低表示相对优先级越高。正常任务根据其nice值分配优先级，其中nice值-20表示优先级100，nice值+19表示优先级139。

real-time

normal

0

99

100

139

← higher

priority

lower →

Example: Linux Scheduling

56 / 69

■ 实时Linux (RTLinux)

■ RTLinux (1997) 作为Linux操作系统的扩展而开发，它增加了实时功能，以使其可预测。Linux操作系统中不可预测性的主要来源是调度程序、中断处理和虚拟内存管理，它们针对最佳吞吐量而不是可预测性进行了优化。

■ RTLinux被构造为一个在Linux下运行的小型实时内核，因此，它的优先级高于所描述的Linux内核。

Linux Tasks

Linux

RT-Linux

ISRs and Hardware

Real-time FIFOs

Real-time tasks

28



Example: Linux Scheduling

57 / 69

■ 实时Linux (RTLinux)

- 中断首先交给RTLinux内核，当实时任务可用时，Linux内核被抢占。
- 当中断发生时，它首先由RTLinux的ISR处理，该ISR激活实时任务，从而调用调度程序。
 - 不同版本的RTLinux中，RTLinux和Linux之间的中断传递有不同的处理方式。
- Linux操作系统处理设备初始化、任何阻塞的动态资源分配，并安装RTLinux的组件。
- 调度程序和实时FIFO是RTLinux的两个核心模块，RTLinux提供了单调速率和最早截止时间优先的调度策略。
- 应用程序接口包括中断和任务管理的系统调用。




Algorithm Evaluation

58 / 69

■ 算法评估

- 为特定系统选择CPU调度算法可能很困难。答案取决于太多的因素，无法给出任何。。。
 - 系统工作负载（变化无常）。
 - 调度程序的硬件支持。
 - 性能标准的相对权重（响应时间、CPU利用率、吞吐量...）。
 - 在最大响应时间为1秒的约束下最大化CPU利用率。
 - 最大化吞吐量，使（平均）周转时间与总执行时间成线性比例。
 - 使用的评估方法（每种方法都有其局限性...）。
- 评估方法可能包括
 - 确定性建模
 - 排队模型
 - 仿真
 - 实现



Algorithm Evaluation

59 / 69

■ 确定性建模


■ 评估方法的一大类是分析性评估(Analytic evaluation). 使用给定的算法和系统工作负载生成公式或数字，以评估该工作负载的算法性能

■ 确定性建模 (Deterministic modeling)是一种分析性评估。它使用特定的预定义工作负载，在该工作负载下定义每个算法的性能。

■ 假设工作负载如下所示。所有5个进程都以给定的顺序在时间0到达，CPU执行时间以ms为单位。考虑这组进程的FCFS、SJF和RR（时间量 = 10ms）调度算法。

■ 哪种算法会给出最小平均等待时间？

进程	运行时间
P ₁	10
P ₂	29
P ₃	3
P ₄	7
P ₅	12



Algorithm Evaluation

60 / 69

■ 确定性建模

进程	运行时间
P ₁	10
P ₂	29
P ₃	3
P ₄	7
P ₅	12

■ 对于FCFS算法，我们将按照

P ₁	P ₂	P ₃	P ₄	P ₅	
0	10	39	42	49	61

■ 进程P₁的等待时间为0ms，P₂为10ms，P₃为39ms，P₄为42ms，P₅为49ms。因此，平均等待时间为

$$(0 + 10 + 39 + 42 + 49)/5 = 28\text{ms}$$

30

Algorithm Evaluation

61 / 69

■ 确定性建模

进程	运行时间
P ₁	10
P ₂	29
P ₃	3
P ₄	7
P ₅	12

■ 使用非抢占式SJF调度，我们按照以下方式执行进程：

P₃

P₄

P₁

P₅

P₂

0310203261

■ 进程P₁的等待时间为10ms，P₂为32ms，P₃为0ms，P₄为3ms，P₅为20ms。因此，平均等待时间为

$$(10 + 32 + 0 + 3 + 20)/5 = 13\text{ms}$$

Algorithm Evaluation

62 / 69

■ 确定性建模

进程	运行时间
P ₁	10
P ₂	29
P ₃	3
P ₄	7
P ₅	12

■ 使用RR算法，我们按照以下方式执行进程：

P₁

P₂

P₃

P₄

P₅

P₂

P₅

P₂

01020233040505261

■ P₁的等待时间为0ms，P₂为32ms，P₃为20ms，P₄为23ms，P₅为40ms。因此，平均等待时间为

$$(0 + 32 + 20 + 23 + 40)/5 = 23\text{ms}$$



■ 确定性建模

- 此例中，SJF策略获得的平均等待时间小于FCFS调度的一半，RR算法居中。
- 确定性建模简单快速，给出精确的数字以比较算法。
 - 然而，它需要输入准确的数字，且其答案只适用于这些输入。
- 确定性建模的主要用途是**描述调度算法**和**提供示例**。
 - 如果我们一次又一次地运行同一个程序，并且能够准确地测量程序的处理需求，那么我们可以使用确定性建模来选择调度算法。此外，在一组示例中，确定性建模可能表明可用于单独分析和证明的趋势。
 - 例如，可以证明，对于所描述的环境（所有进程及其时间在时间0可用），SJF策略将始终获得最小等待时间。



■ 排队模型

- 在许多系统上，可以确定的是CPU和I/O执行的分布。这些分布可以测量，然后近似或简单估计。
- 可从以下两个分布中计算大多数算法的平均吞吐量、利用率、等待时间等：
 - 特定CPU执行的概率分布，通常描述为其平均值的指数分布。
 - 进程到达系统的时间分布（到达时间分布）。
- 排队网络分析
 - 计算机系统被描述为具有就绪队列的服务器网络，或为具有设备队列的I/O系统。
 - 知道到达率和服务率后，我们可以计算利用率、平均队列长度、平均等待时间等。
 - 这一研究领域称为**排队网络分析**(queueing-network analysis)。



■ 排队模型

■ 排队网络分析

- 设 n 为平均队列长度（不包括正在服务的进程），设 w 为队列中的平均等待时间，设 λ 是队列中新进程的平均到达率（例如每秒3个进程）。我们希望在进程等待的时间内， $\lambda \times w$ 个新进程将到达队列中。如果系统处于稳定状态，则离开队列的进程数必须等于到达的进程数。因此

$$n = \lambda \times w$$

- 这个方程式被称为利特尔公式 (*Little's formula*, John Little, 1954. 或称利特尔法则), 特别有用, 因为它适用于任何调度算法和到达分布。
 - 排队论和随机系统中最著名和最有用的守恒定律之一。
 - 它指出, 系统中单位的时间平均数 = 单位到达率 \times 每个单位在系统中的平均时间。



■ 排队模型

■ 排队网络分析

- 例如, 让 $\lambda = 7$ (平均每秒有7个进程到达), $n=14$ (队列中通常有14个进程), 然后根据 **Little的公式**, 我们得到 $w = n / \lambda = 2$ (每个进程的平均等待时间为2秒)。
- 排队分析在比较调度算法时很有用, 但也有局限性。
 - 可以处理的算法和分布类相当有限。可能很难处理使用复杂数学的算法和分布。
 - 到达和服务分布是以数学上易于处理但不现实的方式定义的。通常还需要做出一些独立的假设, 这些假设可能不准确。
- 排队模型通常只是真实系统的近似值, 计算结果的准确性可能有问题。



■ 仿真

■ 计算机系统仿真器

- 软件数据结构代表了计算机系统的主要组成部分。
- 系统时钟由一个变量表示。
 - 当该变量的值增加时，仿真器修改系统状态以反映设备、进程和调度程序的活动。
- 在仿真执行时，将收集并打印指示算法性能的统计信息。

■ 数据驱动仿真

- 最常用的方法是使用一个随机数生成器，该生成器根据**概率分布**通过编程生成进程、CPU执行时间、到达、离开等。
- 分布可以用数学方法定义，例如均匀分布、指数分布、泊松分布。
- 也可以基于经验定义分布，我们需要对正在研究的实际系统进行测量。



■ 仿真

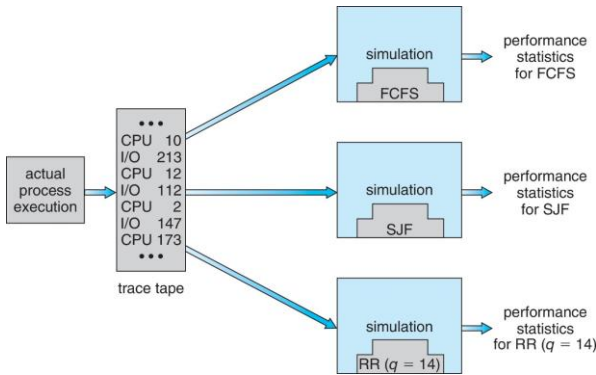
■ 跟踪磁带Trace Tape

- 分布驱动的仿真可能不准确，因为频率分布仅指示每个事件发生的次数；它并没有说明它们发生的顺序。
- 跟踪磁带用于监控真实系统并记录实际事件的顺序。然后使用跟踪磁带驱动仿真。
- 跟踪磁带提供了一种很好的方法，可以在完全相同的一组实际输入上比较两种算法。该方法可以为其输入生成准确的结果。



■ 仿真

■ CPU调度的仿真评估



- 仿真可能很昂贵，通常需要数小时的计算机时间。此外，跟踪磁带可能需要大量存储空间。最后，仿真器的设计、编码和调试是一项主要任务。