操作系统　Operating System

第四章 进程与并发程序设计(5)
——调度

沃天宇
woty@buaa.edu.cn
2021年4月22日

内容提要

- 基本概念
- 设计调度算法要考虑的问题
- 批处理系统的调度算法
- 交互式系统的调度算法
- 实时系统的调度算法
- 多处理机调度
- Linux处理机调度

CPU调度

什么是CPU调度？

CPU 调度的任务是控制、 协调 多个进程对 CPU 的竞争。 也就是按照一定的策略（调度算法），从就绪队列中 选择一个进程，并把 CPU 的控制权交给被选中的进程。

CPU调度的场景

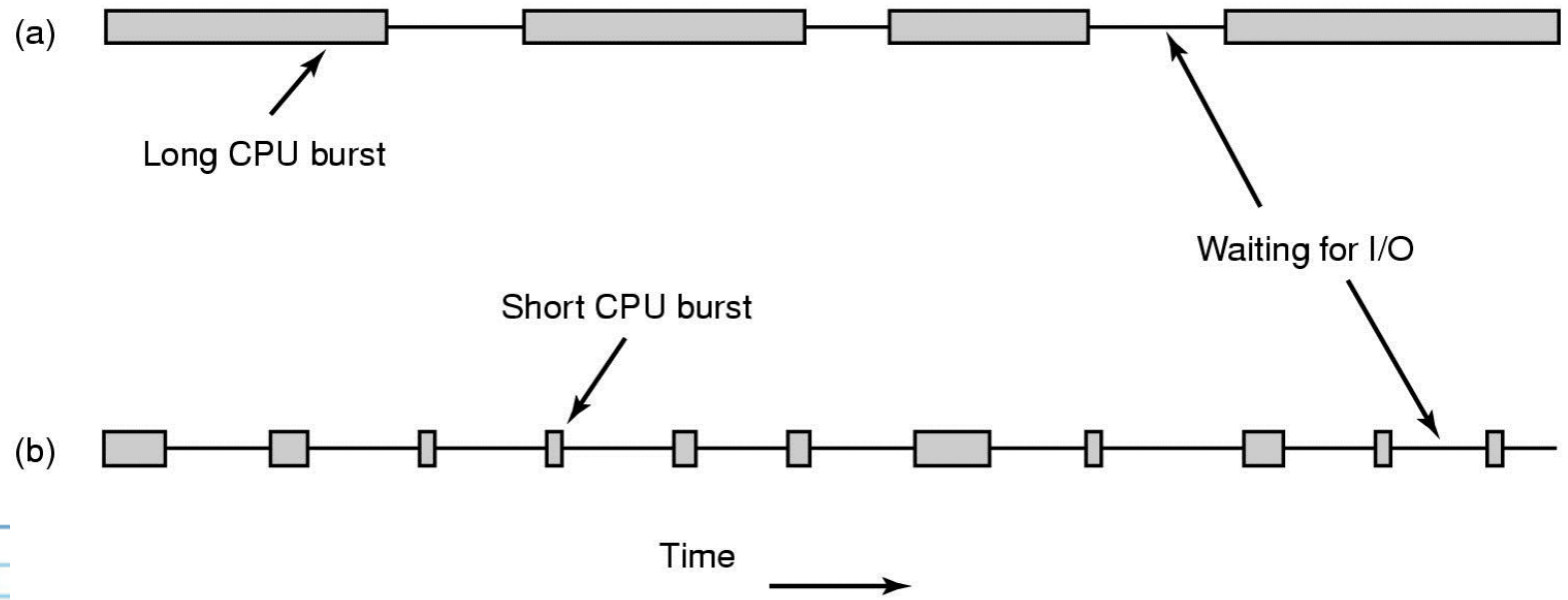
- N 个进程就绪，等待上 CPU 运行
- M个CPU，M≥1
- OS需要决策，给哪个进程分配哪个 CPU 。

要解决的问题

- WHAT：
  - 按什么原则分配CPU—进程调度算法
- WHEN：
  - 何时分配CPU—进程调度的时机
- HOW：
  - 如何分配CPU—CPU切换过程（进程的上下文切换）

问题

- 处理机管理的工作是对CPU资源进行合理的分配使用，以提高处理机利用率，并使各用户公平地得到处理机资源。这里的主要问题是处理机调度算法和调度算法特征分析。



调度的类型

- 高级调度
- 中级调度
- 低级调度

高级调度

- 高级调度：又称为“宏观调度”、“作业调度”。从用户工作流程的角度，一次提交的若干个作业，对每个作业进行调度。时间上通常是分钟、小时或天。

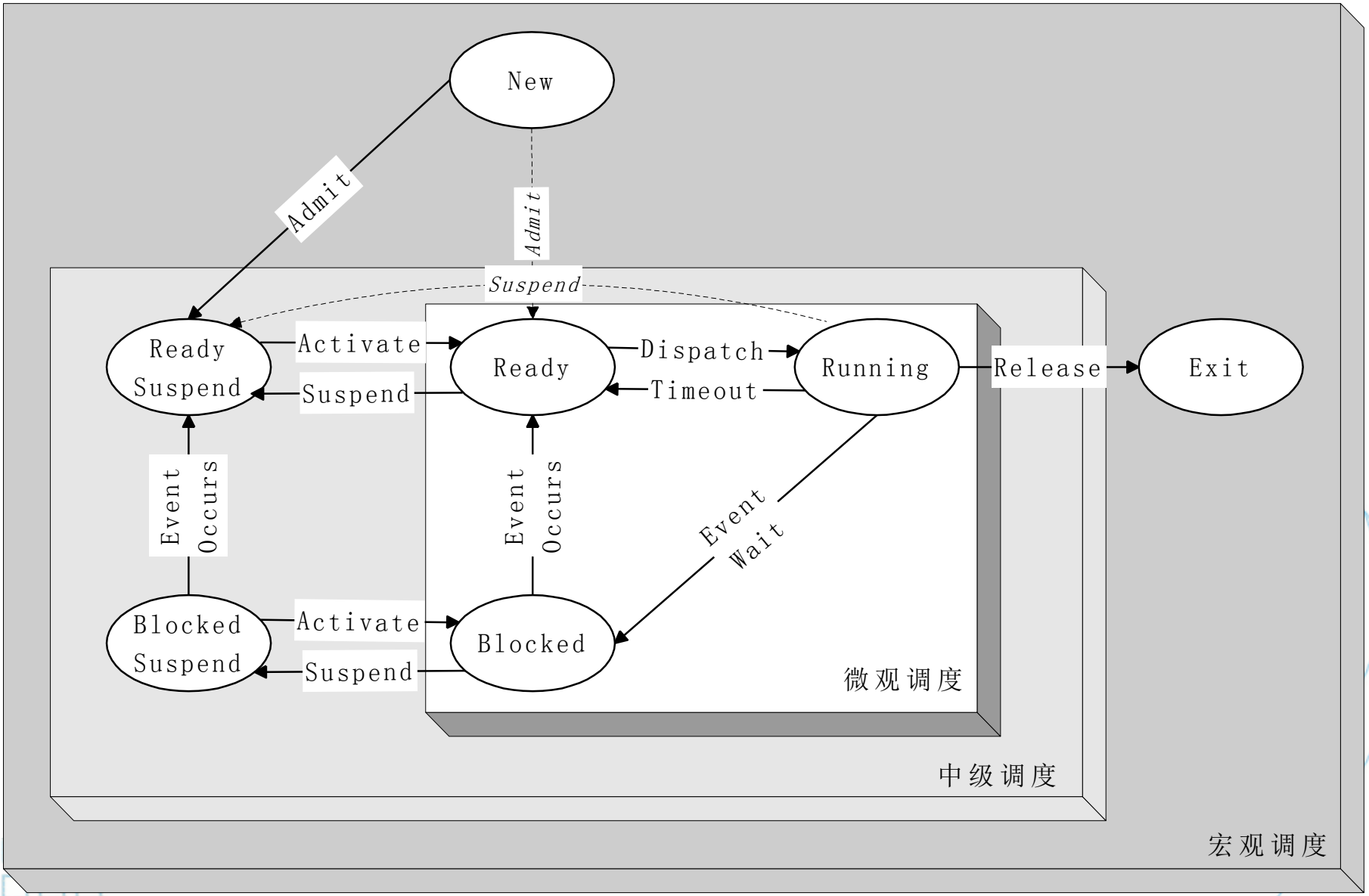
- 接纳多少个作业
- 接纳哪些作业

中级调度

- 内外存交换：又称为“中级调度”。
- 指令和数据必须在内存里才能被CPU直接访问。
- 从存储器资源的角度，将进程的部分或全部换出到外存上，将当前所需部分换入到内存。

低级调度

- 低级调度：又称为“微观调度”、“进程或线程调度”。从CPU资源的角度，执行的单位，时间上通常是毫秒。因为执行频繁，要求在实现时达到高效率。
- 非抢占式
- 抢占式
  - 时间片原则
  - 优先权原则
  - 短作业（进程）优先



何时进行调度

- 当一个新的进程被创建时，是执行新进程还是继续执行父进程？
- 当一个进程运行完毕时；
- 当一个进程由于I/O、信号量或其他的某个原因被阻塞时；
- 当一个I/O中断发生时，表明某个I/O操作已经完成，而等待该I/O操作的进程转入就绪状态；
- 在分时系统中，当一个时钟中断发生时。

何时进行切换

- 只要OS取得对CPU的控制，进程切换就可能发生:
  - **用户调用**：来自程序的显式请求(如：打开文件)，该进程多半会被阻塞
  - **陷阱**：最末一条指令导致出错，会引起进程移至退出状态
  - **中断**：外部因素影响当前指令的执行，控制被转移至中断处理程序

进程切换

- 在进程（上下文）中切换的步骤
  - 保存处理器的上下文，包括程序计数器和其它寄存器
  - 用新状态和其它相关信息更新正在运行进程的PCB
  - 把进程移至合适的队列-就绪、阻塞
  - 选择另一个要执行的进程
  - 更新被选中进程的PCB
  - 从被选中进程中重装入CPU 上下文

调度的性能准则

- 从不同的角度来判断处理机调度算法的性能，如用户的角度、处理机的角度和算法实现的角度。实际的处理机调度算法选择是一个综合的判断结果。

面向用户的调度性能准则1

- **周转时间：**作业从提交到完成（得到结果）所经历的时间。包括：在收容队列中等待，CPU上执行，就绪队列和阻塞队列中等待，结果输出等待——批处理系统（外存等待时间、就绪等待时间、CPU执行时间、I/O操作时间）
  - 平均周转时间
  - 带权平均周转时间（T/Ts)：总周转时间/总服务时间
- **响应时间：**用户输入一个请求（如击键）到系统给出首次响应（如屏幕显示）的时间——分时系统

面向用户的调度性能准则2

- 截止时间：开始截止时间和完成截止时间——实时系统，与周转时间有些相似。
- 优先级：可以使关键任务达到更好的指标。
- 公平性：不因作业或进程本身的特性而使上述指标过分恶化，如长作业等待很长时间。

面向系统的调度性能准则

- 吞吐量：单位时间内所完成的作业数，跟作业本身特性和调度算法都有关系——批处理系统
  - 平均周转时间不是吞吐量的倒数，因为并发执行的作业在时间上可以重叠。如：在2小时内完成4个作业，则吞吐量是2个作业/小时，而平均周转时间可能是0.5小时、1小时、1.25小时、2小时、…
- 处理机利用率：——大中型主机
- 各种资源的均衡利用：如CPU繁忙的作业和I/O繁忙（指次数多，每次时间短）的作业搭配——大中型主机

调度算法本身的调度性能准则

- 易于实现、维护
  - 算法复杂度、实现结构、代码量
- 执行开销比
  - 算法本身执行时间、算法本身占用空间
  - 算法开销占调度后整体任务执行时间比例

内容提要

- 基本概念
- 设计调度算法要考虑的问题（设计要点）
- 批处理系统的调度算法
- 交互式系统的调度算法
- 实时系统的调度算法
- 多处理机调度
- Linux处理机调度

设计调度算法要点

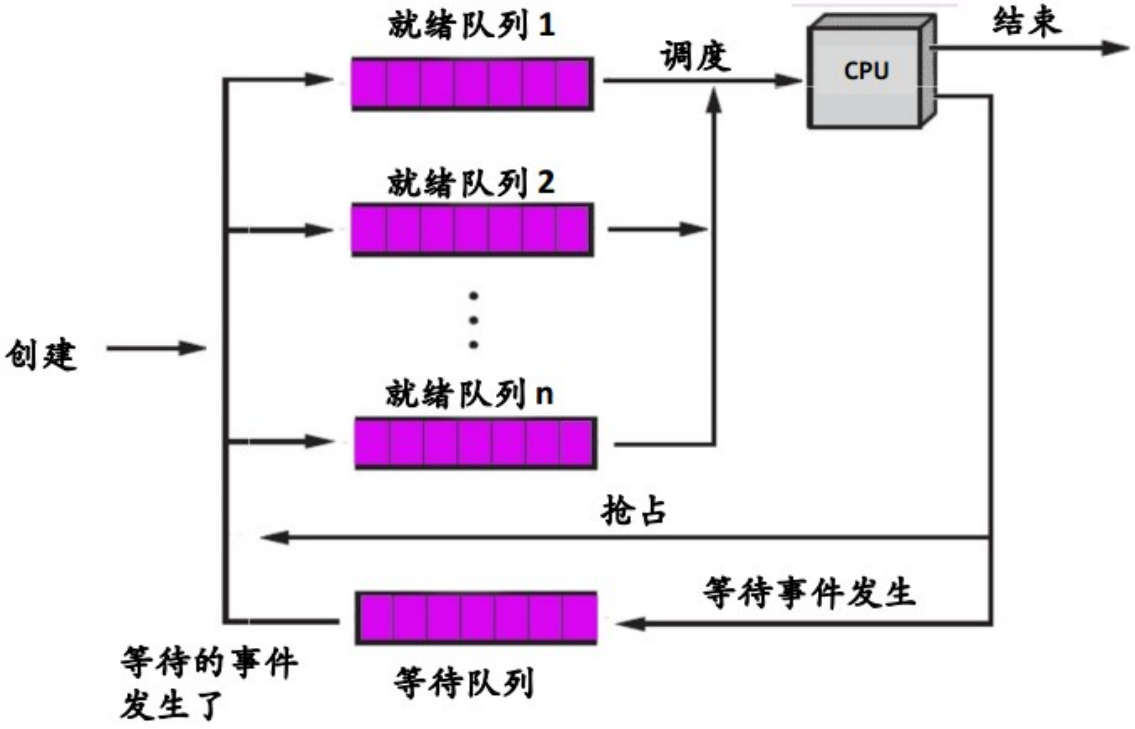
- 进程优先级（数）
- 进程优先级就绪队列的组织
- 抢占式调度与非抢占式调度
- 进程的分类
- 时间片

进程优先级（数）

- 优先级和优先数是不同的，优先级表现了进程的重要性和紧迫性，优先数实际上是一个数值，反映了某个优先级。
- 静态优先级
  - 进程创建时指定，运行过程中不再改变
- 动态优先级
  - 进程创建时指定了一个优先级，运行过程中可以动态变化。如：等待时间较长的进程可提升其优先级。

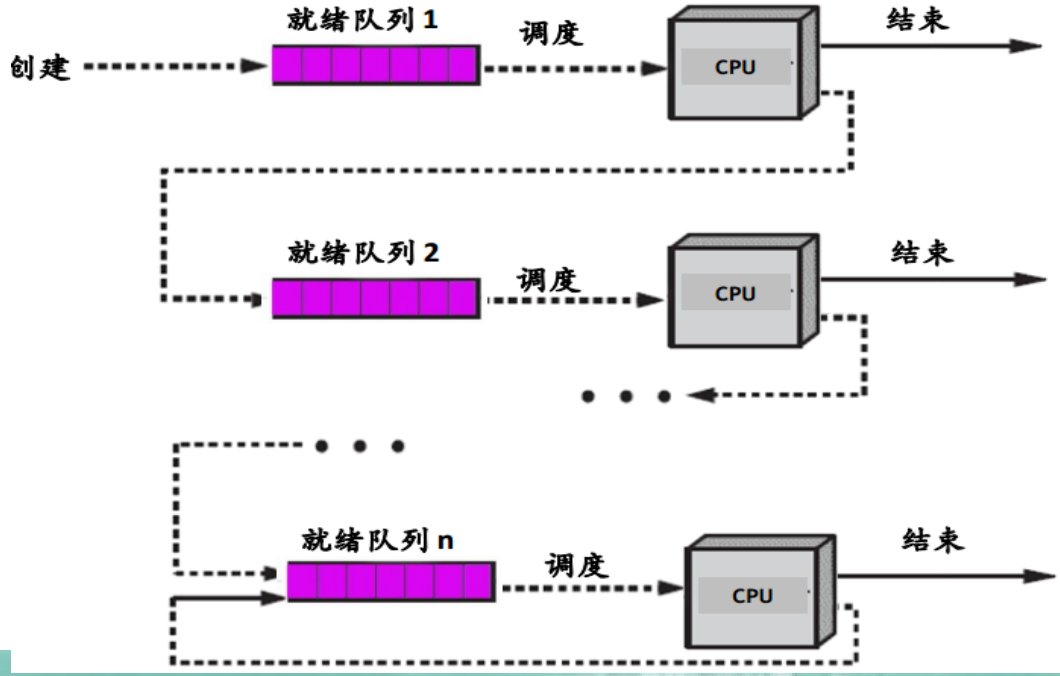
进程就绪队列组织

- 按优先级排队方式
  - 创建多个进程后按照不同的优先级进行排列，CPU调度优先级较高的进程进行执行。



进程就绪队列组织

- 另一种方式
  - 所有进程创建之后都进入到第一级就绪队列，随着进程的运行，可能会降低某些进程的优先级，如某些进程的时间片用完了，那么就会将其降级。

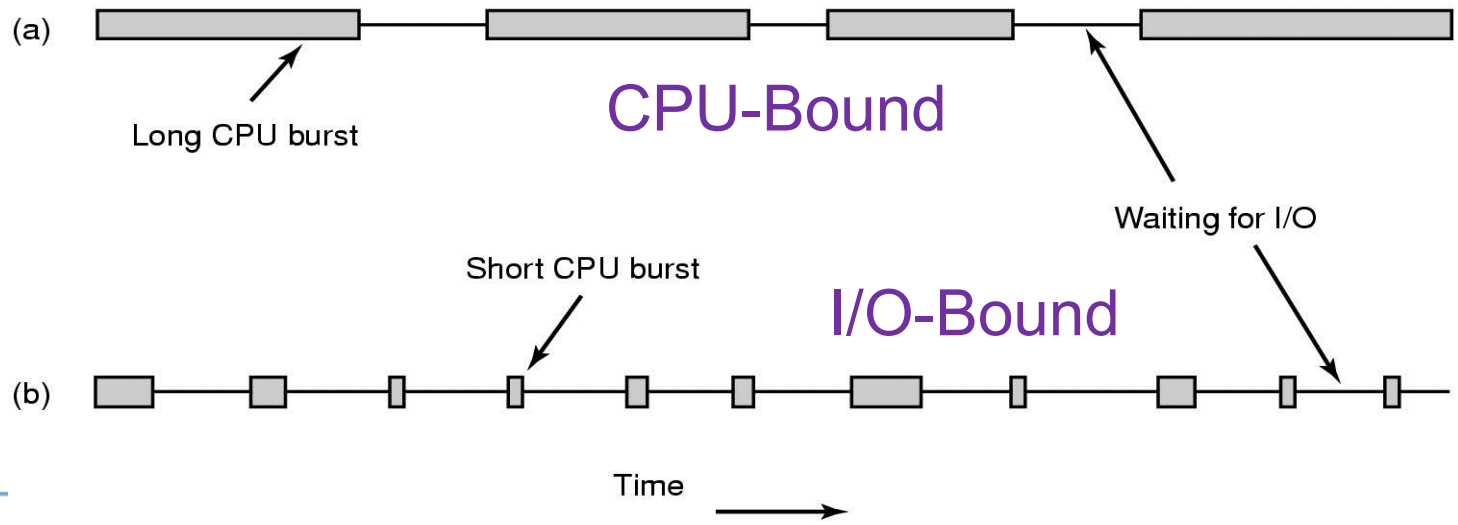


占用CPU的方式

- 不可抢占式方式
  - 一旦处理器分配给一个进程，它就一直占用处理器，直到该进程自己因调用原语操作或等待I/O等原因而进入阻塞状态，或时间片用完时才让出处理器，重新进行
- 抢占式方式
  - 就绪队列中一旦有优先级高于当前运行进程优先级的进程存在时，便立即进行进程调度，把处理器转给优先级高的进程

进程的分类（第一种）

- I/O Bound（I/O密集型）
  - 频繁的进行I/O，通常会花费很多时间等待I/O操作完成
- CPU bound（CPU密集型）
  - 计算量大，需要大量的CPU时间。



进程的分类（第二种）

- 批处理进程（Batch Process)
  - 无需与用户交互，通常在后台运行
  - 不需很快的响应
  - 典型的批处理程序：编译器、科学计算
- 交互式进程（Interactive Process)
  - 与用户交互频繁，因此要花很多时间等待用户输入
  - 响应时间要快，平均延迟要低于50~150ms
  - 典型的交互式进程：Word、触控型GUI
- 实时进程（Real-time Process)
  - 有实时要求，不能被低优先级进程阻塞
  - 响应时间要短且要稳定
  - 典型的实时进程：视频/音频、控制类

时间片（Time slice或quantum）

- 一个时间段，分配给调度上CPU的进程，确定了允许该进程运行的时间长度。那么如何选择时间片？有一下需要考虑的因素：
  - 进程切换的开销
  - 对响应时间的要求
  - 就绪进程个数
  - CPU能力
  - 进程的行为

内容提要

- 基本概念
- 设计调度算法要考虑的问题
- 批处理系统的调度算法
- 交互式系统的调度算法
- 实时系统的调度算法
- 多处理机调度
- Linux处理机调度

吞吐量、平均等待时间和平均周转时间

- 吞吐量 $= \frac{\text{作业数}}{\text{总执行时间}}$，即单位时间CPU完成的作业数量
- 周转时间(Turnover Time) = 完成时刻 - 提交时刻
- 带权周转时间=周转时间/服务时间（执行时间）
- 平均周转时间 $= \frac{\text{作业周转时间之和}}{\text{作业数}}$
- 平均带权周转时间 $= \frac{\text{作业带权周转时间之和}}{\text{作业数}}$

批处理系统中常用的调度算法

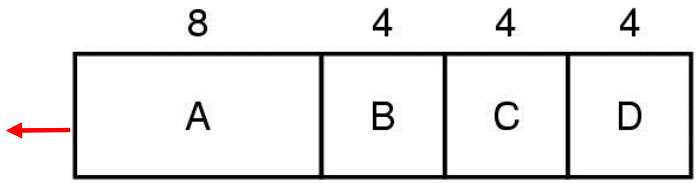
- 先来先服务（FCFS：First Come First Serve)
- 最短作业优先（SJF：Shortest Job First)
- 最短剩余时间优先（SRTF：Shortest Remaining Time First）
- 最高响应比优先（HRRF：Highest Response Ratio First）

先来先服务 (FCFS, First Come First Served)

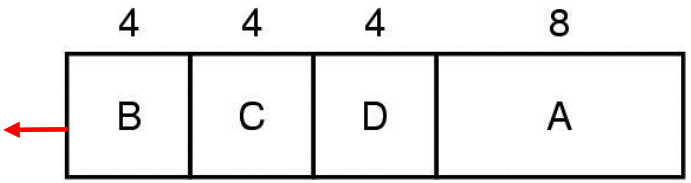
- 这是最简单的调度算法，按先后顺序调度。
  - 按照作业提交或进程变为就绪状态的先后次序，分派CPU；
  - 当前作业或进程占用CPU，直到执行完或阻塞，才出让CPU（非抢占方式）。
  - 在作业或进程唤醒后（如I/O完成），并不立即恢复执行，通常等到当前作业或进程出让CPU。
  - 最简单的算法。
- FCFS的特点
  - 比较有利于长作业，而不利于短作业。
  - 有利于CPU繁忙的作业，不利于I/O繁忙的作业。

短作业优先(SJF, Shortest Job First)

- 又称为“短进程优先”SPN(Shortest Process Next)；这是对FCFS算法的改进，其目标是减少平均周转时间。
  - 对预计执行时间短的作业（进程）优先分派处理机。通常后来的短作业不抢先正在执行的作业。



(a)



(b)

SJF的特点

- 优点：
  - 比FCFS改善平均周转时间和平均带权周转时间，缩短作业的等待时间；
  - 提高系统的吞吐量；
- 缺点：
  - 对长作业非常不利，可能长时间得不到执行；
  - 未能依据作业的紧迫程度来划分执行的优先级；
  - 难以准确估计作业（进程）的执行时间，从而影响调度性能。

示例

有三道作业，它们的提交时间和运行时间见下表

作业号	提交时刻	运行时间/h
1	10:00	2
2	10:10	1
3	10:25	0.25

试给出在下面两种调度算法下，作业的执行顺序、平均周转时间和平均带权周转时间。

（1）先来先服务FCFS调度算法；

（2）短作业优先SJF调度算法。

示例

采用FCFS调度算法时，作业的执行顺序是：作业1 ->作业2 ->作业3。由此可得到运行表

作业号	提交时刻	运行时间/h	开始时刻	完成时刻
1	10:00	2	10:00	12:00
2	10:10	1	12:00	13:00
3	10:25	0.25	13:00	13:15

那么，平均周转时间为

$T=(\sum Ti)/3=[(12-10)+(13-10:10)+(13:15-10:25)]/3$
$=[2+2.83+2.83]/3 =2.55h$

带权平均周转时间为

$W=[\sum(Ti/Tir)]/3=(2/2+2.83/1+2.83/0.25)/3=5.05h$

示例

在SJF调度算法下，作业的执行顺序是作业1 -> 作业3-> 作业2；由此得运行表

作业号	提交时刻	运行时间/h	开始时刻	完成时刻
1	10:00	2	10:00	12.00
2	10:10	1	12:15	13:15
3	10:25	0.25	12:00	12:15

那么，平均周转时间为

T=(∑Ti)/3=[(12-10)+(13:15-10:10)+(12:15-10:25)]/3 =[2+3.08+1.83]/3=2.3h

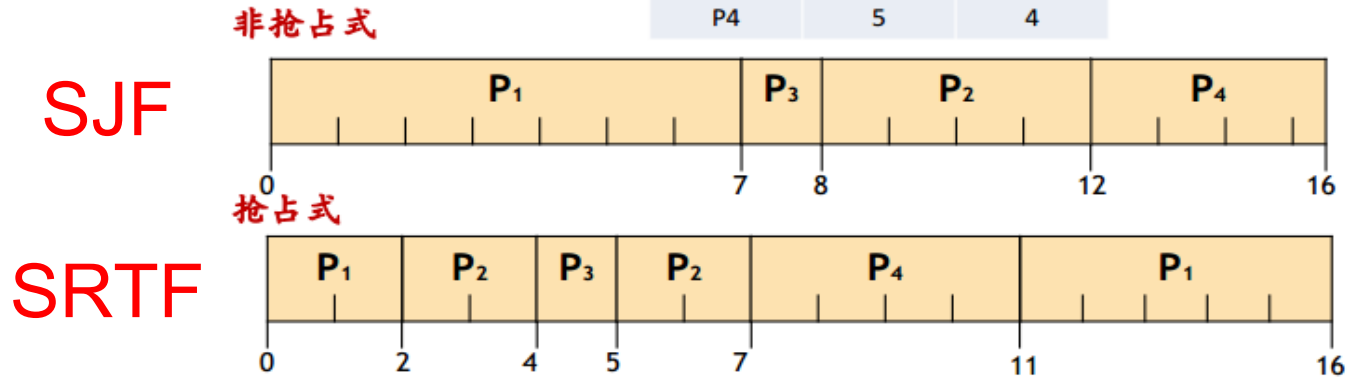
带权平均周转时间为

W=[∑(Ti/Tir)]/3=(2/2+3.08/1+1.83/0.25)/3=3.8h

最短剩余时间优先SRTF

- 将短作业优先进行改进，改进为抢占式，这就是最短剩余时间优先算法了，即一个新就绪的进程比当前运行进程具有更短的完成时间，系统抢占当前进程，选择新就绪的进程执行。

进程	到达时刻	运行时间
P1	0	7
P2	2	4
P3	4	1
P4	5	4



缺点：源源不断的短任务到来，可能使长的任务长时间得不到运行，导致产生“饥饿”现象。

最高响应比优先HRRF

- HRRF算法实际上是FCFS算法和SJF算法的折衷既考虑作业等待时间，又考虑作业的运行时间，既照顾短作业又不使长作业的等待时间过长，改善了调度性能。
- 在每次选择作业投入运行时，先计算后备作业队列中每个作业的响应比RP(相应优先级)，然后选择其值最大的作业投入运行。
- RP定义为：$$RP = \frac{\text{已等待时间} + \text{要求运行时间}}{\text{要求运行时间}} = 1 + \frac{\text{已等待时间}}{\text{要求运行时间}}。$$

最高响应比优先HRRF

- 响应比的计算时机：
  - 每当调度一个作业运行时，都要计算后备作业队列中每个作业的响应比，选择响应比最高者投入运行。
- 响应比最高优先（HRRF）算法效果：
  - 短作业容易得到较高的响应比
  - 长作业等待时间足够长后，也将获得足够高的响应比
  - 饥饿现象不会发生
- 缺点：
  - 每次计算各道作业的响应比会有一定的时间开销，性能比SJF略差。

内容提要

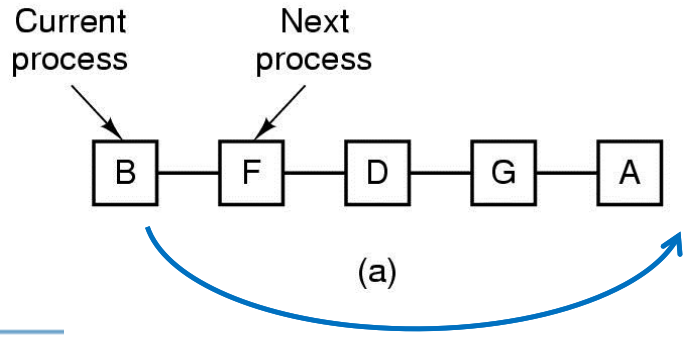
- 基本概念
- 设计调度算法要考虑的问题
- 批处理系统的调度算法
- 交互式系统的调度算法
- 实时系统的调度算法
- 多处理机调度
- Linux处理机调度

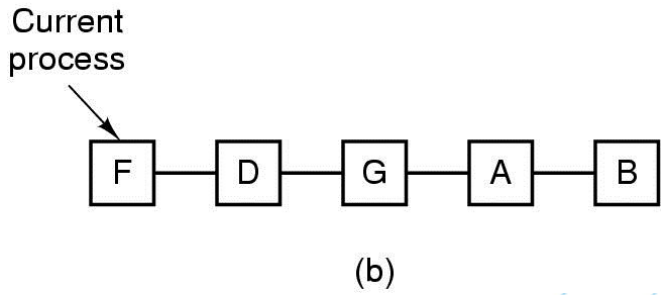
交互式系统的调度算法

- 时间片轮转(RR：Round Robin）
- 多级队列（MQ：Multi-level Queue）
- 多级反馈队列（MFQ： Multi-level Feedback Queue ）

时间片轮转(Round Robin)算法

- 算法主要用于微观调度，设计目标是提高资源利用率。其基本思路是通过时间片轮转，提高进程并发性和响应时间特性，从而提高资源利用率；





时间片轮转算法

- 【排队】将系统中所有的就绪进程按照FCFS原则，排成一个队列。
- 【轮转】每次调度时将CPU分派给队首进程，让其执行一个时间片。时间片的长度从几个ms到几百ms。
- 【中断】在一个时间片结束时，发生时钟中断。
- 【抢占】调度程序据此暂停当前进程的执行，将其送到就绪队列的末尾，并通过上下文切换执行当前的队首进程。
- 【出让】进程可以未使用完一个时间片，就出让CPU（如阻塞）。

时间片长度的确定

- 时间片长度变化的影响
  - 过长—>退化为FCFS算法，进程在一个时间片内都执行完，响应时间长。
  - 过短—>用户的一次请求需要多个时间片才能处理完，上下文切换次数增加，响应时间长。
- 系统的响应时间：

  T(响应时间)=N(进程数目) * q(时间片)
- 就绪进程的数目：数目越多，时间片越小
- 系统的处理能力：应当使用户输入通常在一个时间片内能处理完，否则会使响应时间，平均周转时间和平均带权周转时间延长。

CPU执行速度

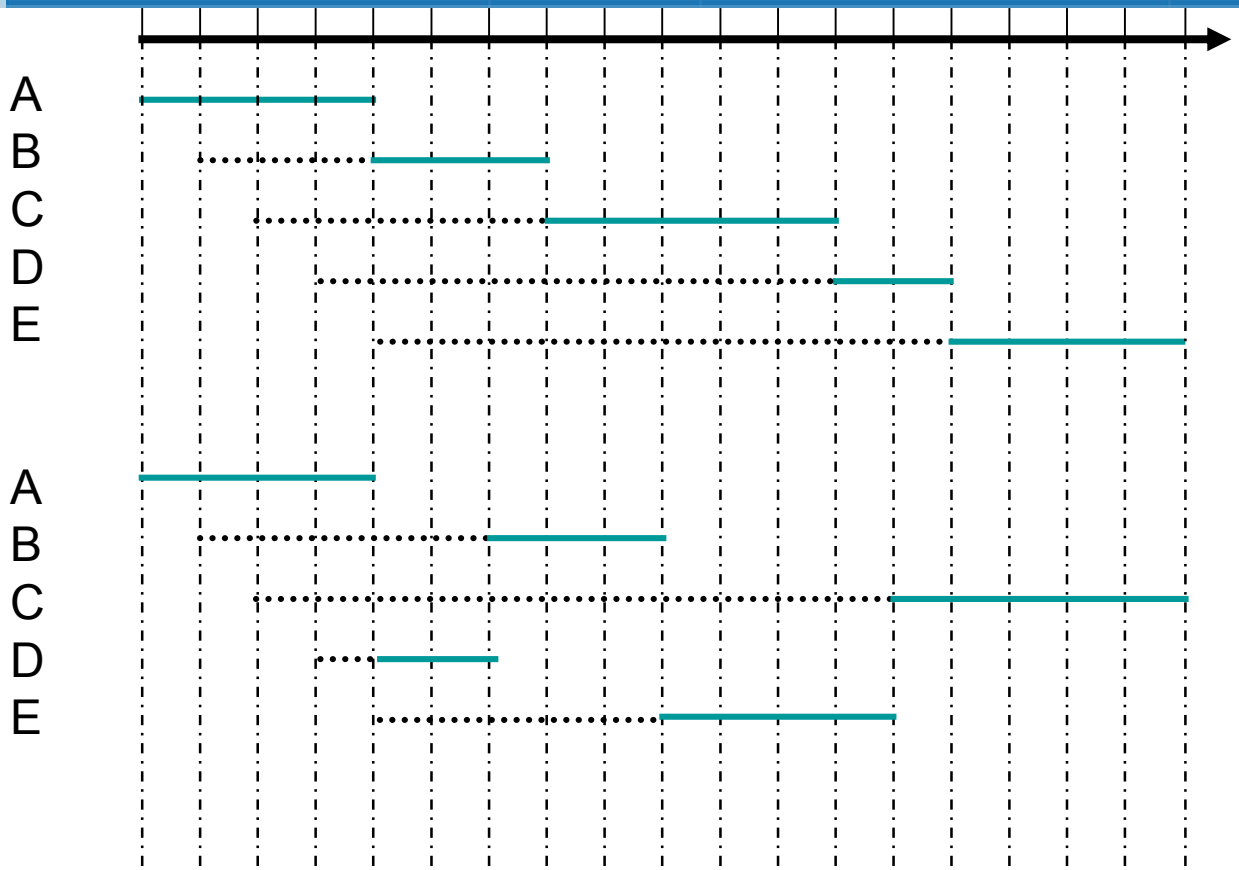
- 10ms能够执行多少条指令？It depends …
  - 主频、外频
  - 流水线、超流水
  - 超标量、乱序执行
  - Hyper-threading
  - 多核、多CPU、SMP、NUMA
- Linux根据用户优先级设置时间片，5-800ms，优先级动态调整
- 现在x86 CPU上Linux 2.6内核上下文切换通常不超过10us。回顾：线程切换和进程切换的开销？

作业调度举例

作业	到达时间	作业时常
A	0	4
B	1	3
C	2	5
D	3	2
E	4	4

	A	B	C	D	E
作业到达时间	0	1	2	3	4
作业执行长度	4	3	5	2	4

先来先服务



平均周转时间＝9

带权平均周转时间＝2.8

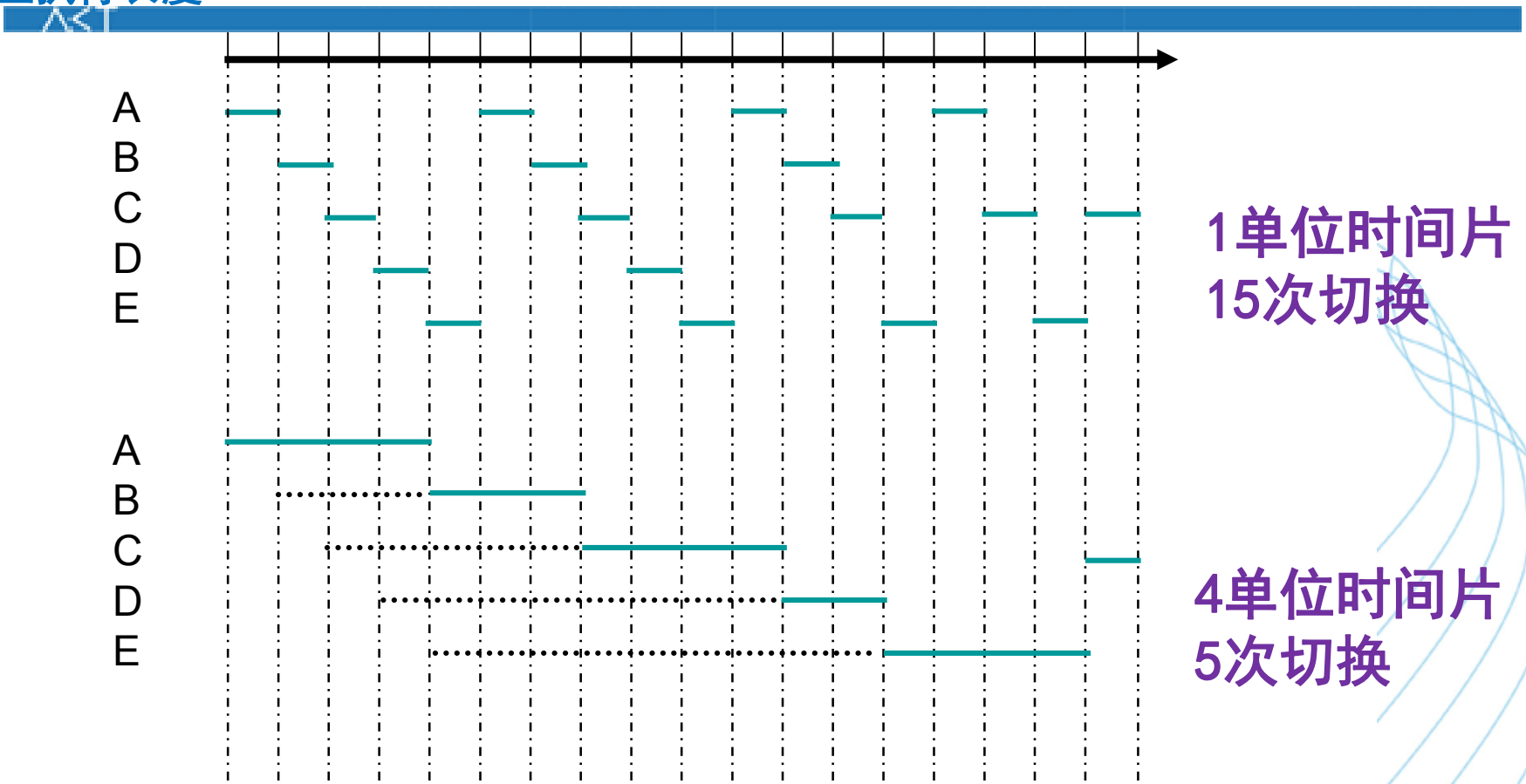
平均周转时间＝8

带权平均周转时间＝2.1

最短作业优先

时间片轮转法

	A	B	C	D	E
作业到达时间	0	1	2	3	4
作业执行长度	4	3	5	2	4

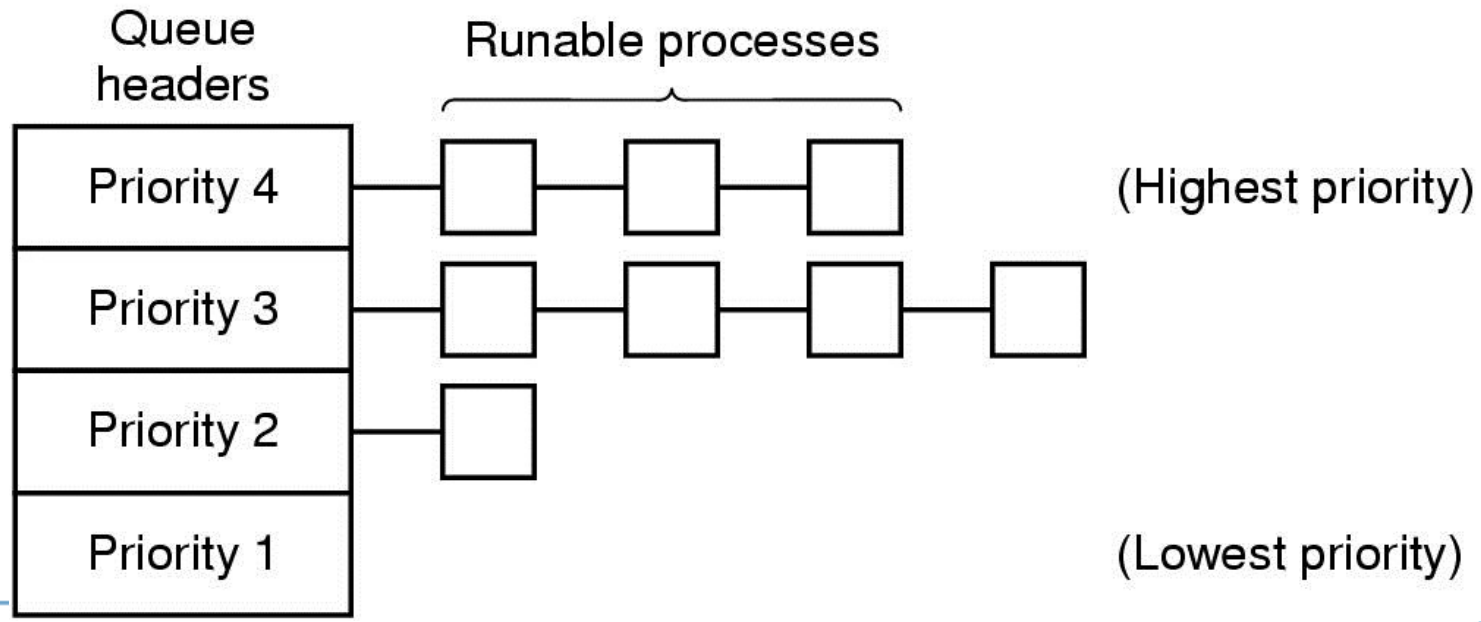


1单位时间片
15次切换

4单位时间片
5次切换

优先级算法(Priority Scheduling)

- 本算法是平衡各进程对响应时间的要求。适用于作业调度和进程调度，可分成抢先式和非抢先式；



静态优先级

- 创建进程时就确定，直到进程终止前都不改变。通常是一个整数。依据：
  - 进程类型（系统进程优先级较高）
  - 对资源的需求（对CPU和内存需求较少的进程，优先级较高）
  - 用户要求（紧迫程度和付费多少）

动态优先级

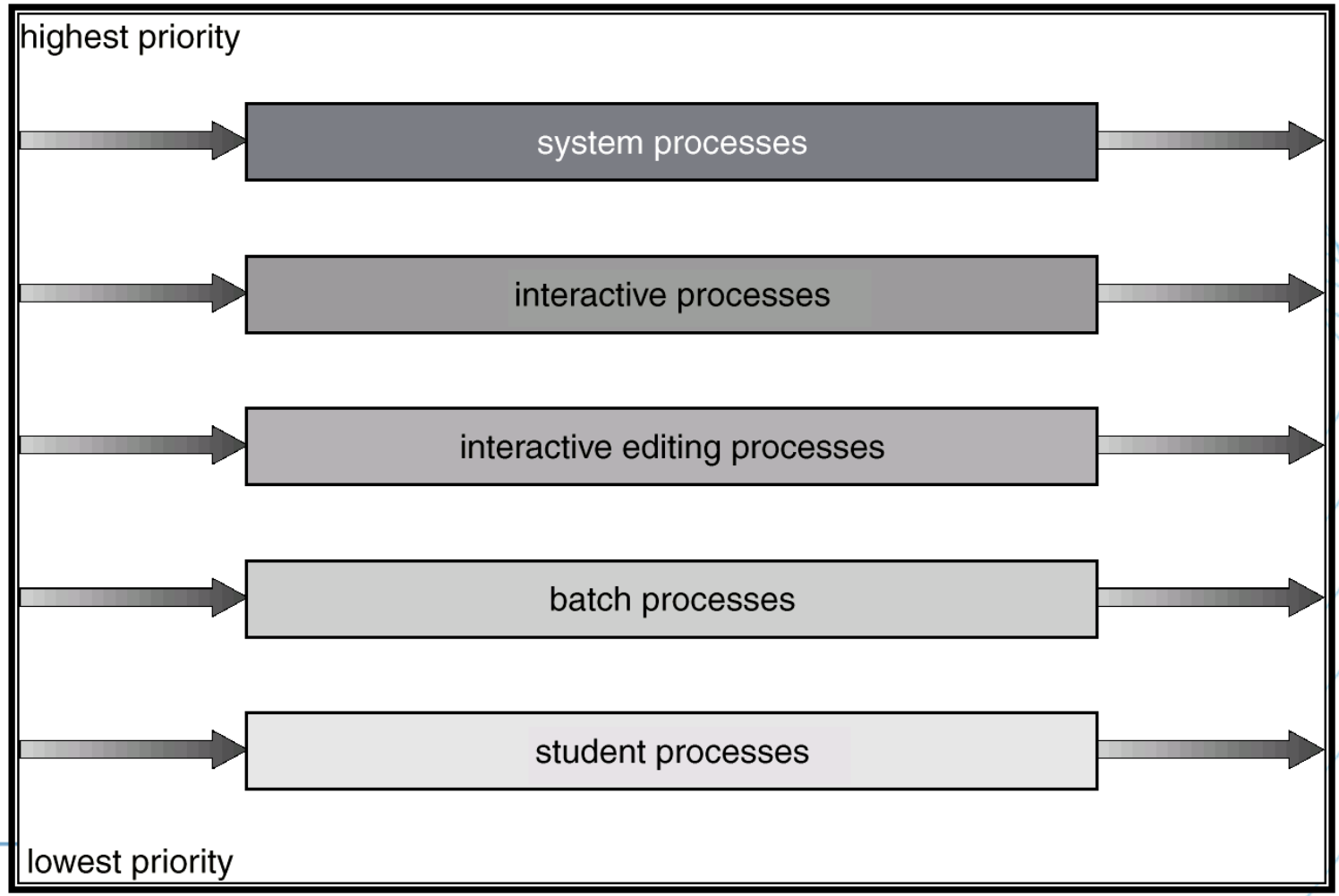
- 在创建进程时赋予的优先级，在进程运行过程中可以自动改变，以便获得更好的调度性能。如：
  - 在就绪队列中，等待时间延长则优先级提高，从而使优先级较低的进程在等待足够的时间后，其优先级提高到可被调度执行；
  - 进程每执行一个时间片，就降低其优先级，从而一个进程持续执行时，其优先级降低到出让CPU。

多级队列算法(Multiple-level Queue)

- 本算法引入多个就绪队列，通过各队列的区别对待，达到一个综合的调度目标；
  - 根据作业或进程的性质或类型的不同，将就绪队列再分为若干个子队列。
  - 每个作业固定归入一个队列。

- 不同队列可有不同的优先级、时间片长度、调度策略等；在运行过程中还可改变进程所在队列。如：系统进程、用户交互进程、批处理进程等。

多级队列算法(Multiple-level Queue)



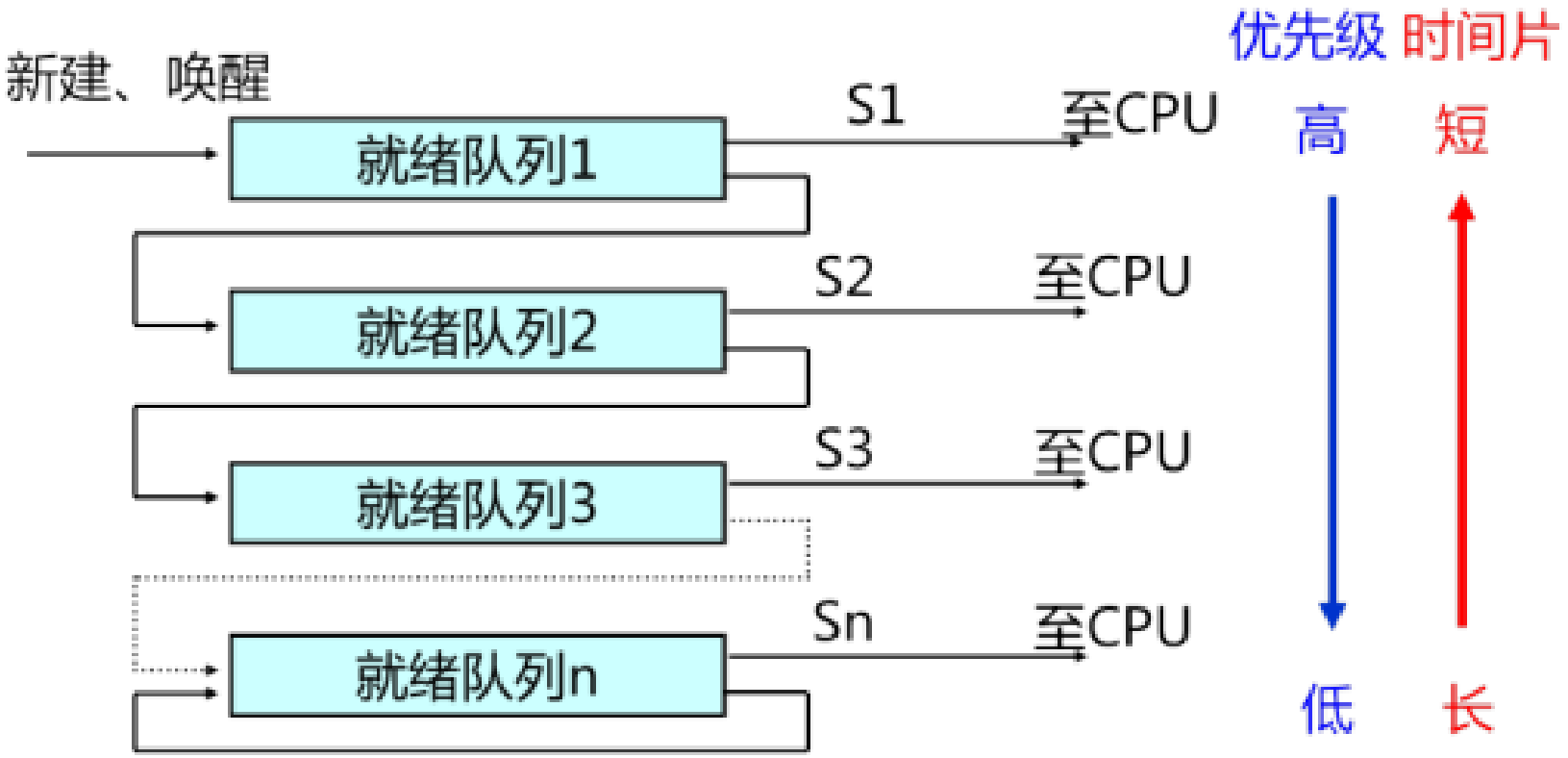
多级反馈队列算法 (Round Robin with Multiple Feedback)

- 多级反馈队列算法：时间片轮转算法和优先级算法的综合和发展。优点：
  - 为提高系统吞吐量和缩短平均周转时间而照顾短进程
  - 为获得较好的I/O设备利用率和缩短响应时间而照顾I/O型进程
  - 不必估计进程的执行时间，动态调节

多级反馈队列算法

- 设置多个就绪队列，分别赋予不同的优先级，如逐级降低，队列1的优先级最高。每个队列执行时间片的长度也不同，规定优先级越低则时间片越长，如逐级加倍
- 新进程进入内存后，先投入队列1的末尾，按“时间片轮转”算法调度；若按队列1一个时间片未能执行完，则降低投入到队列2的末尾，同样按“时间片轮转”算法调度；如此下去，降低到最后的队列，则按“FCFS”算法调度直到完成。
- 仅当较高优先级的队列为空，才调度较低优先级的队列中的进程执行。如果进程执行时有新进程进入较高优先级的队列，则抢先执行新进程，并把被抢先的进程投入原队列的末尾。

多级反馈队列



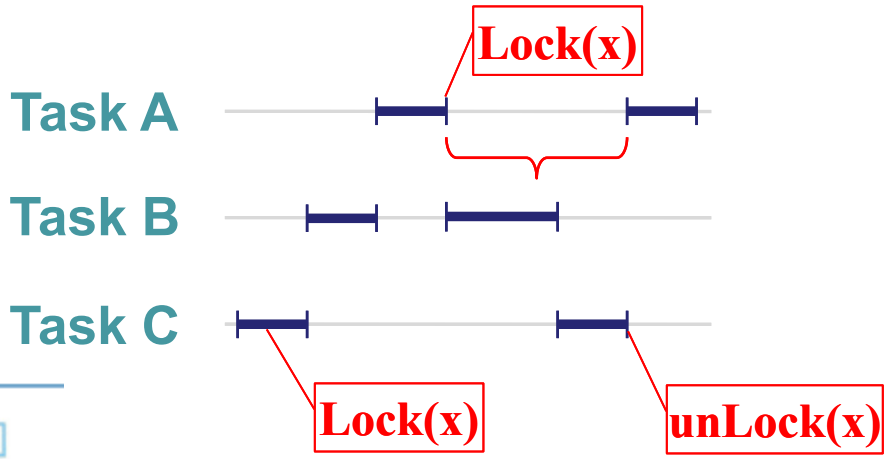
时间片：S1<S2<S3

几点说明

- **I/O型进程：**让其进入最高优先级队列，以及时响应I/O交互。通常执行一个小时间片，要求可处理完一次I/O请求的数据，然后转入到阻塞队列。
- **计算型进程：**每次都执行完时间片，进入更低级队列。最终采用最大时间片来执行，减少调度次数。
- I/O次数不多，而主要是CPU处理的进程，在I/O完成后，放回优先I/O请求时离开的队列，以免每次都回到最高优先级队列后再逐次下降。
- 为适应一个进程在不同时间段的运行特点，I/O完成时，提高优先级；时间片用完时，降低优先级；

优先级倒置

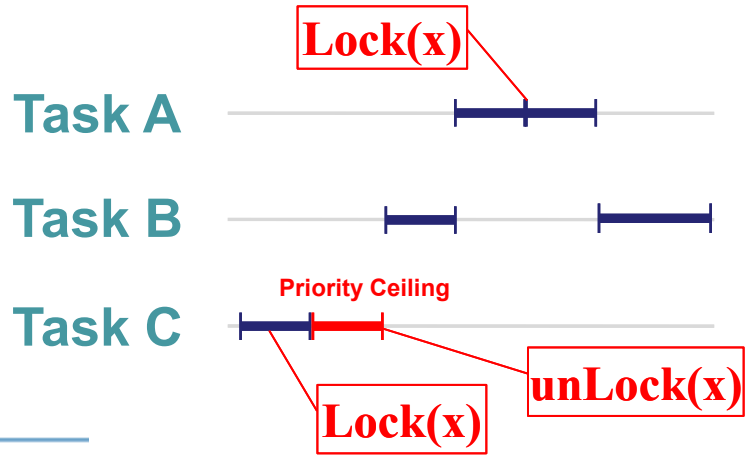
- **优先级倒置现象：**高优先级进程（或线程）被低优先级进程（或线程）延迟或阻塞。
  - 例如：有三个完全独立的进程 Task A、Task B 和 TaskC，Task A 的优先级最高，Task B 次之，Task C 最低。Task A 和 Task C 共享同一个临界资源 X。



Task A 和 Task C 共享同一个临界资源，高优先级进程 Task A 因低优先进程 Task C 被阻塞，又因为低优先进程 Task B 的存在延长了被阻塞的时间。

解决方法——优先级置顶

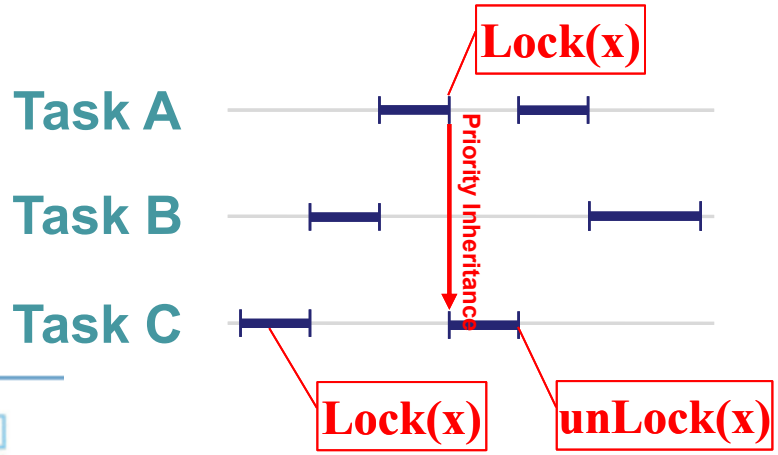
- 优先级置顶（Priority Ceiling）
  - 进程 Task C 在进入临界区后，Task C 所占用的处理机就不允许被抢占。这种情况下，Task C 具有最高优先级（Priority Ceiling）。



如果系统中的临界区都较短且不多，该方法是可行的。反之，如果 Task C 临界区非常长，则高优先级进程Task A 仍会等待很长的时间，其效果无法令人满意。

解决方法——优先级继承

- 优先级继承（Priority Inheritance）
  - 当高优先级进程 Task A 要进入临界区使用临界资源 X时，如果已经有一个低优先级进程 Task C 正在使用该资源，可以采用优先级继（Priority Inheritance）的方法。



此时一方面 Task A 被阻塞，另一方面由 Task C 继承Task A 的优先级，并一直保持到 Task C 退出临界区。

内容提要

- 基本概念
- 设计调度算法要考虑的问题
- 批处理系统的调度算法
- 交互式系统的调度算法
- 实时系统的调度算法
- 多处理机调度
- Linux处理机调度

实时系统

- 实时系统是一种时间起着主导作用的系统。当外部的一种或多种物理设备给了计算机一个刺激，而计算机必须在一个确定的时间范围内恰当地做出反应。对于这种系统来说，正确的但是迟到的应答往往比没有答案还要糟糕。
- 实时系统被分为硬实时系统和软实时系统。硬实时要求绝对满足截止时间要求（如：汽车和飞机的控制系统），而软实时可以偶尔不满足（如：视频/音频程序）。
- 实时系统通常将对不同刺激的响应指派给不同的进程（任务），并且每个进程的行为是可提前预测的。

实时调度

- 要求更详细的调度信息：如，就绪时间、开始或完成截止时间、处理时间、资源要求、绝对或相对优先级（硬实时或软实时）。
- 采用抢先式调度。
- 快速中断响应，在中断处理时（硬件）关中断的时间尽量短。
- 快速任务分派。相应地采用较小的调度单位（如线程）。

实时调度

问题描述：

- 假设一任务集S={t_1,t_2,t_3,..., t_n}，周期分别是T_1,T_2,...,T_n，执行时间为C_1,C_2,..., C_n，截至周期(deadline)为D_1,D_2,..., D_n，通常$D_i = T_i$。CPU利用率：用$U = \sum_1^n (Ci/Ti)$ 表示；
- 前提条件
  - 任务集（S）是已知的；
  - 所有任务都是周期性（T）的，必须在限定的时限（D）内完成；
  - 任务之间都是独立的，每个任务不依赖于其他任务；
  - 每个任务的运行时间（C）是不变的；
  - 调度，任务切换的时间忽略不计。

实时调度算法

- 静态表调度**Static table-driven scheduling**
- 单调速率调度**RMS：Rate Monotonic Scheduling**
  - 任务集可调度，**if**，$\sum_{i=1}^{n} \frac{C_i}{T_i} \le \mathbf{n}(\sqrt[n]{2}-1)$

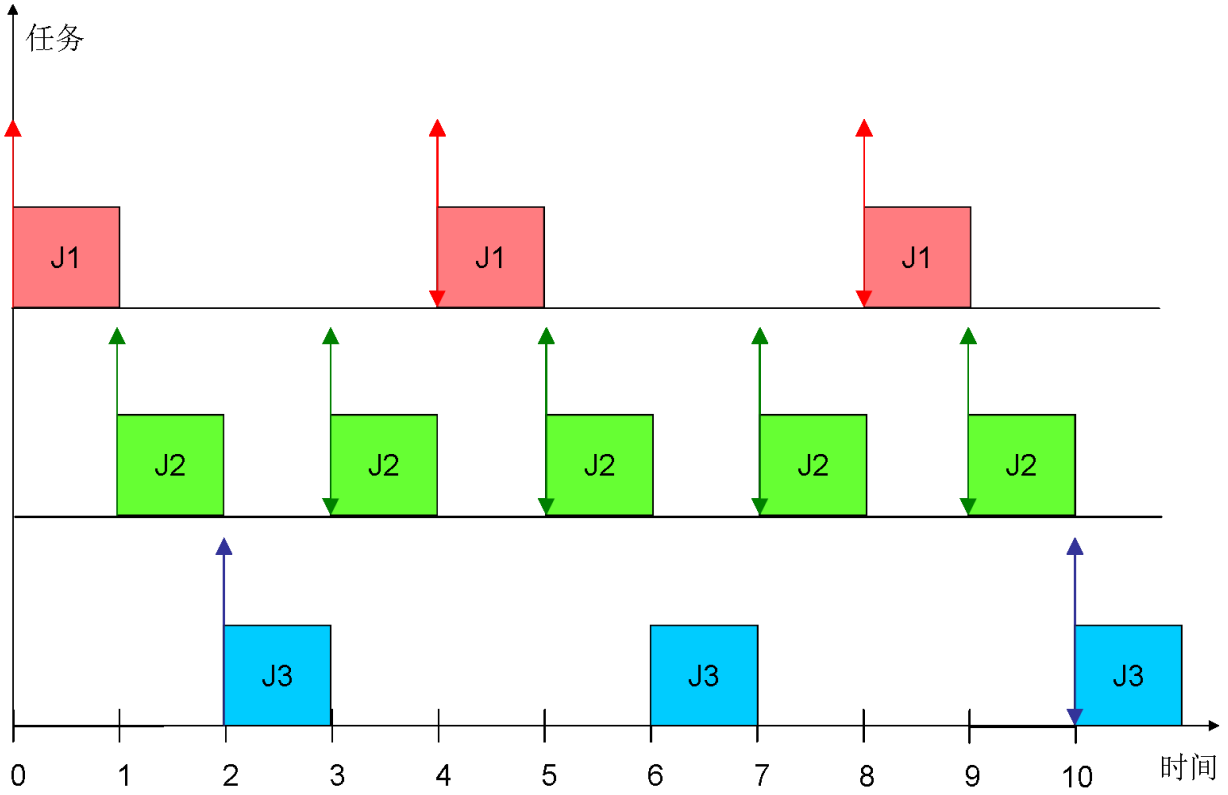
$$\lim_{n\to\infty} n(\sqrt[n]{2}-1) = \ln 2 \approx 0.693147\ldots$$

- 最早截止时间优先算法**EDF：Earliest Deadline First**
  - 任务集可调度，**iff**，$\sum_{i=1}^{n} \frac{C_i}{T_i} \le 1$

静态表调度算法

- 通过对所有周期性任务的分析预测（到达时间、运行时间、结束时间、任务间的优先关系），事先确定一个固定的调度方案。
- 特点：
  - 无任何计算，按固定方案进行，开销最小；
  - 无灵活性，只适用于完全固定的任务场景。

任务Ji	起始时间Si	执行时间Ci	周期Ti	截止时间Di
J1	0	1	4	4
J2	1	1	2	2
J3	2	1	4	4



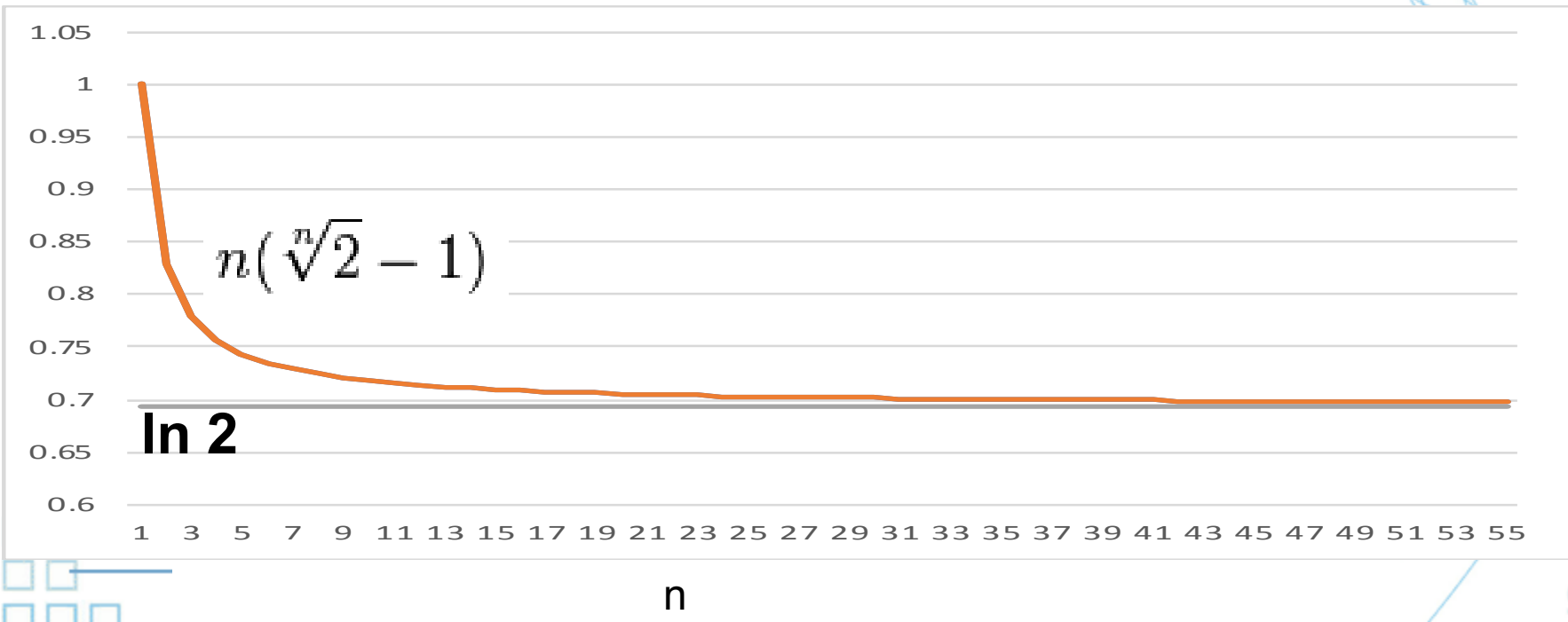
单调速率调度RMS

- RMS是单处理器下的最优静态优先级调度算法。1973年Liu和Layland发表的这篇文章的前半部分首次提出了RM调度算法在静态调度中的最优性。它的一个特点是可通过对系统资源利用率的计算来进行任务可调度性分析，算法简单、有效，便于实现。
- 不仅如此, 他们还把系统的利用系数(utilization factor)和系统可调度性联系起来，推导出用RM调度所能达到的最小系统利用率公式。

单调速率调度RMS

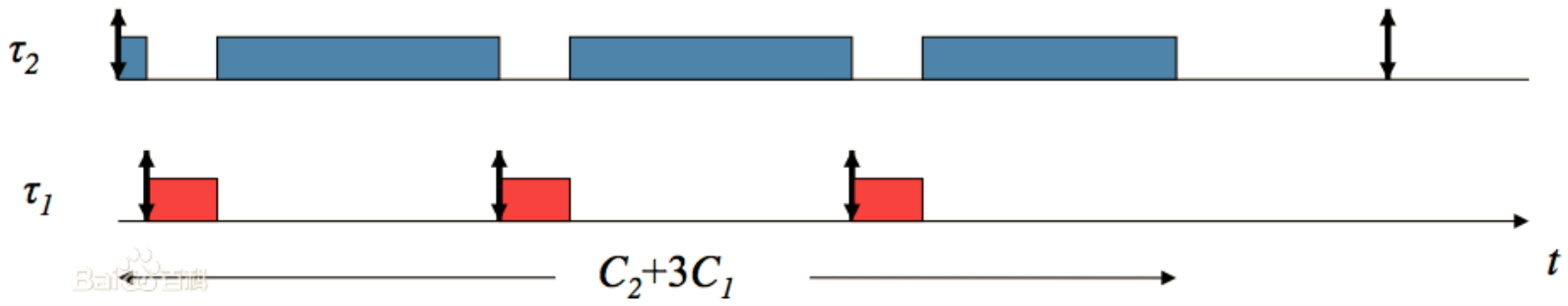
- 任务集可调度，**if**，$\sum_{i=1}^{n} \frac{C_i}{T_i} \le \mathbf{n}(\sqrt[n]{2}-1)$

$$\lim_{n\to\infty} n(\sqrt[n]{2}-1) = \ln 2 \approx 0.693147\ldots$$



单调速率调度RMS

- RMS已被证明是静态最优调度算法, 开销小, 灵活性好, 是实时调度的基础性理论。
- 特点
  - 任务的周期越小，其优先级越高，优先级最高的任务最先被调度
  - 如果两个任务的优先级一样，当调度它们时，RM算法将随机选择一个调度
- 静态、抢先式调度



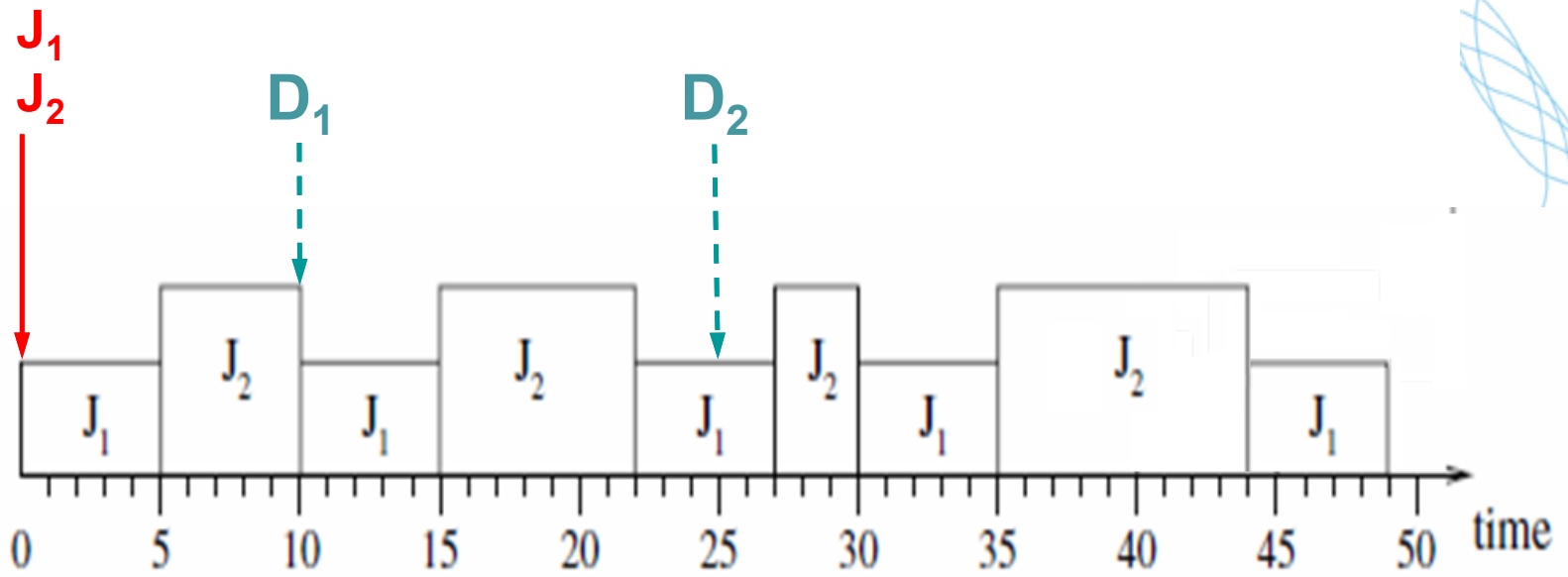
http://dl.acm.org/citation.cfm?doid=321738.321743

最早截止期优先EDF

- 任务的绝对截止时间越早，其优先级越高，优先级最高的任务最先被调度（动态优先级）
- 如果两个任务的优先级一样，当调度它们时，EDF算法将随机选择一个调度

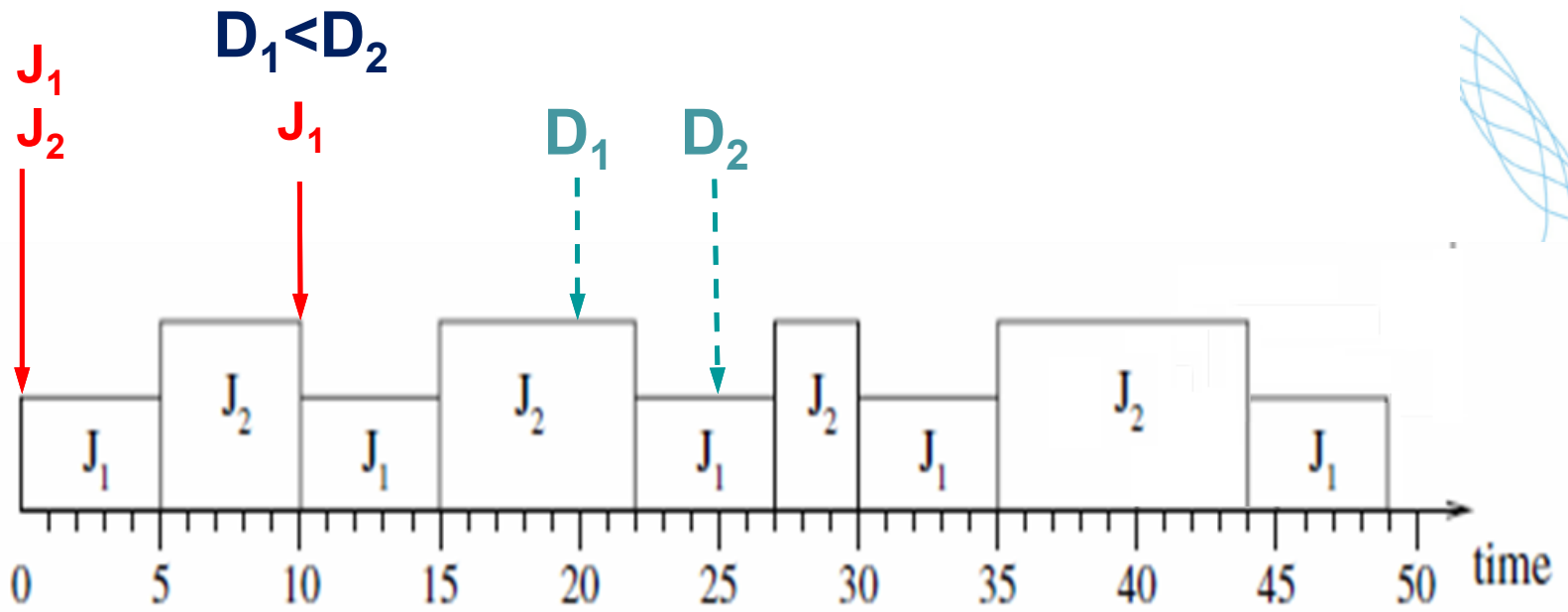
任务Ji	起始时间Si	执行时间Ci	周期Pi	截止时间Di
J1	0	5	10	10
J2	0	12	25	25

$D_1<D_2$



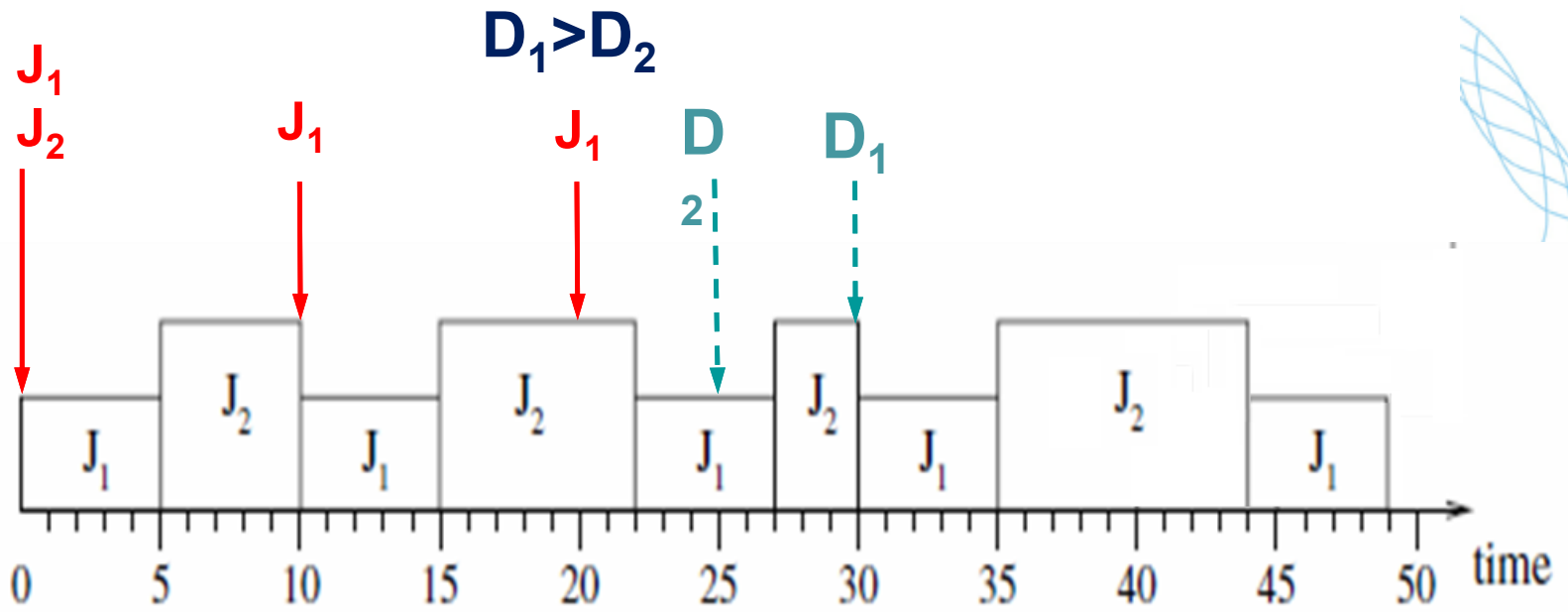
在0时刻，由于$D_1<D_2$选择J_1运行，之后J_2运行。

任务Ji	起始时间Si	执行时间Ci	周期Pi	截止时间Di
J1	0	5	10	10
J2	0	12	25	25



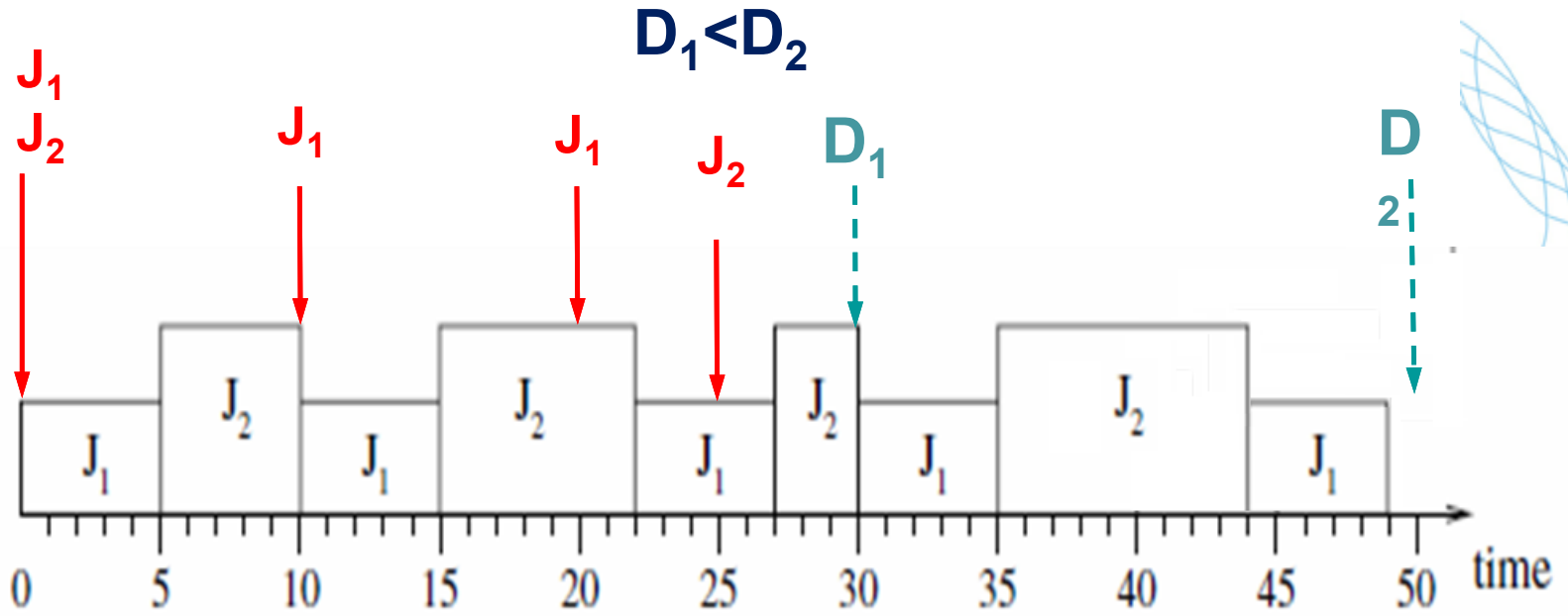
在时刻10，J_1的新周期开始，切换到J_1运行，之后J_2运行。

任务Ji	起始时间Si	执行时间Ci	周期Pi	截止时间Di
J1	0	5	10	10
J2	0	12	25	25



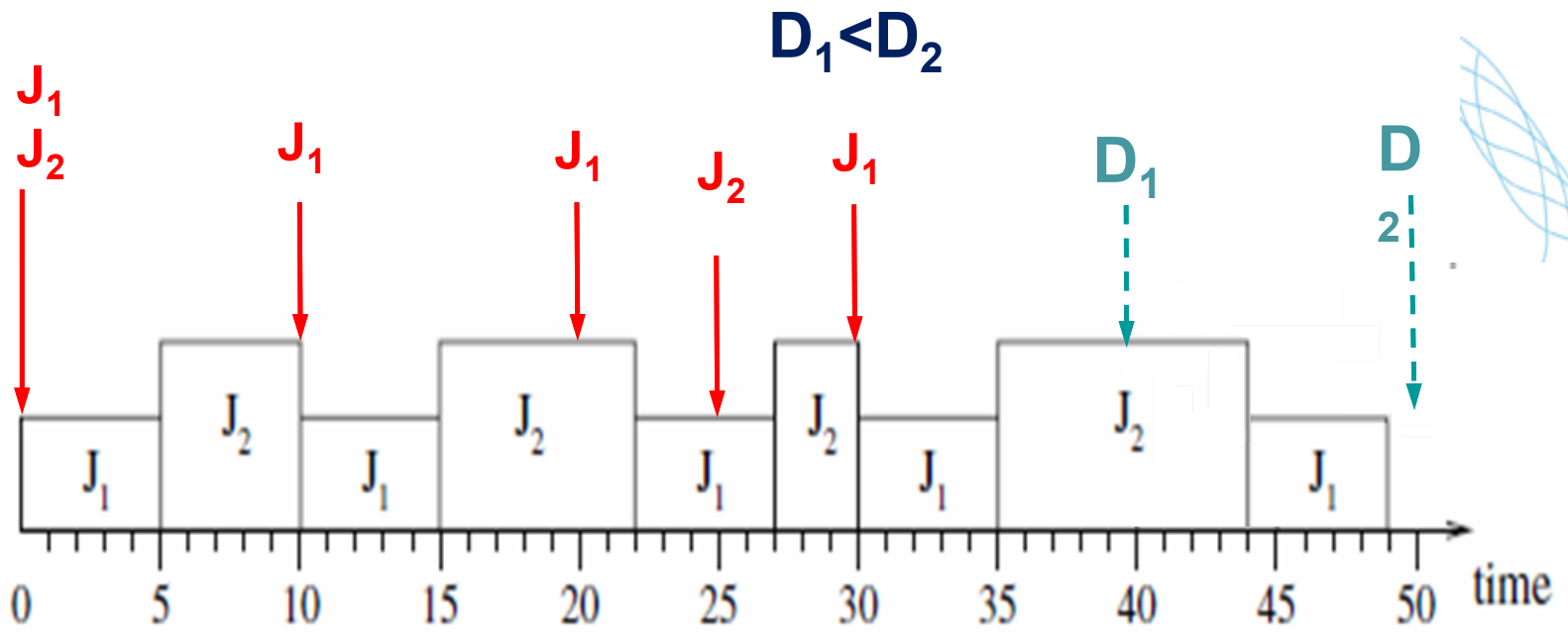
在时刻20，J_1新周期开始，但由于$D_1>D_2$，保持J_2运行，之后J_1运行。

任务Ji	起始时间Si	执行时间Ci	周期Pi	截止时间Di
J1	0	5	10	10
J2	0	12	25	25



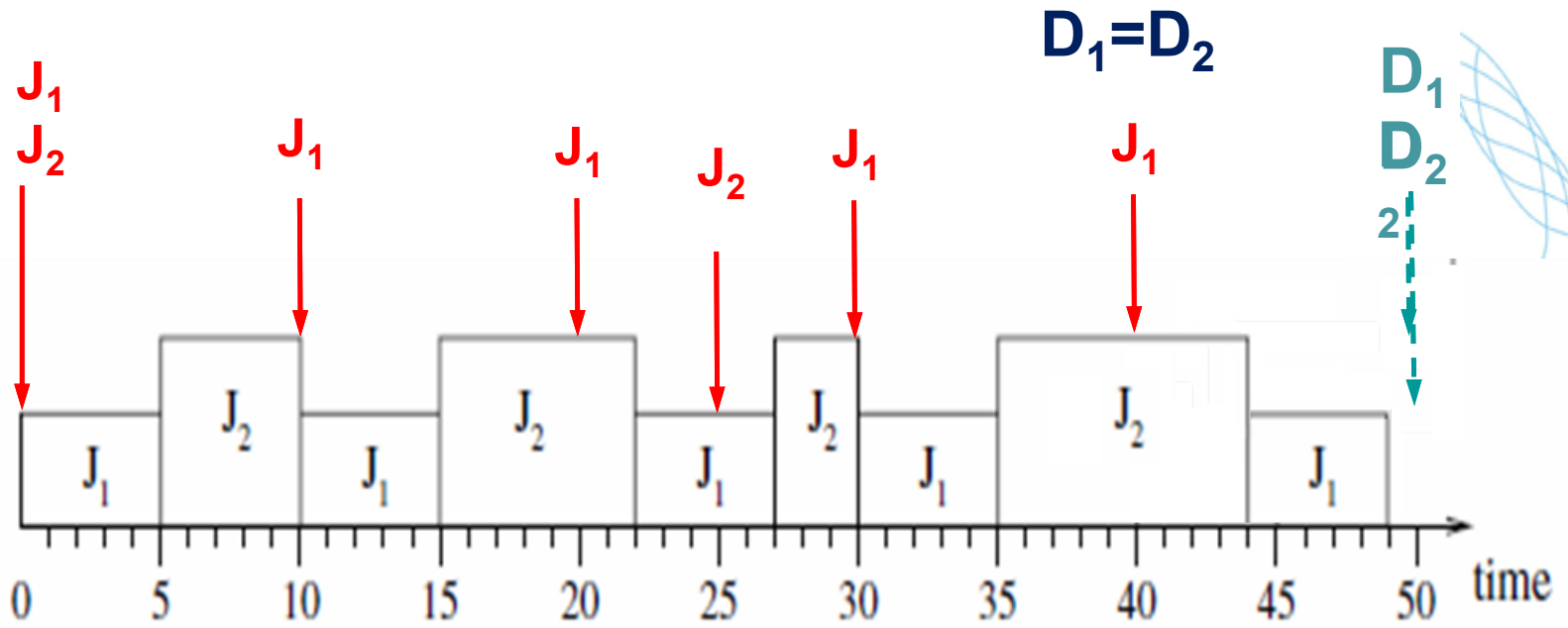
在时刻25，J_2新周期开始，但由于$D_1<D_2$，保持J_1运行，之后J_2运行。

任务Ji	起始时间Si	执行时间Ci	周期Pi	截止时间Di
J1	0	5	10	10
J2	0	12	25	25



在时刻30，J_1的新周期开始，切换到J_1运行，之后J_2运行。

任务Ji	起始时间Si	执行时间Ci	周期Pi	截止时间Di
J1	0	5	10	10
J2	0	12	25	25



在时刻40，J_1新周期开始，$D_1=D_2$，为减少一次切换保持J_2运行，之后J_1运行。

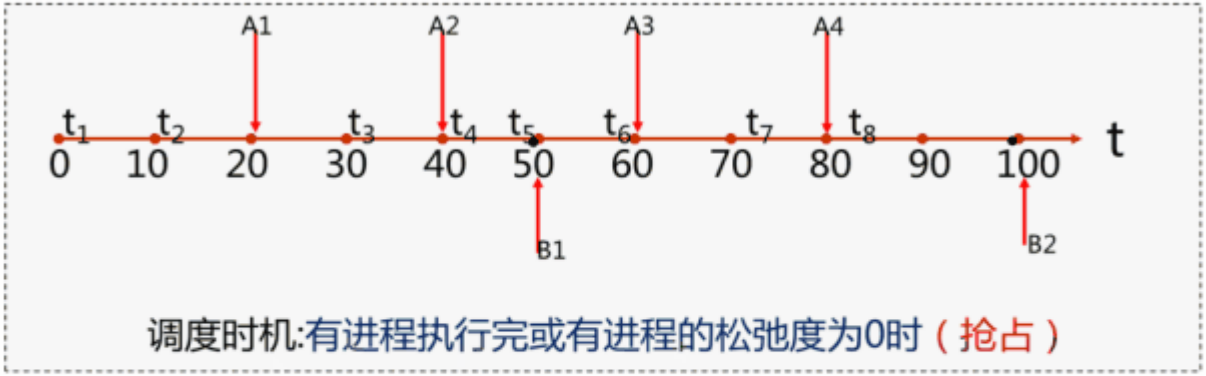
最低松弛度优先算法LLF

LLF（Least Laxity First）

- LLF算法是根据任务紧急（或松弛）的程度，来确定任务的优先级。任务的紧急度越高，其优先级越高，并使之优先执行。
- 松弛度（Laxity）
  = 进程截至时间 - 本身剩余运行时间 - 当前时间
- 调度时机：有进程执行完或有进程的Laxity为0时（抢占）。
- 任务集可调度，**iff**，$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$

LLF示例（自学）

- 在一个实时系统中，有两个周期性实时任务 A 和 B，任务 A 要求每 20ms 执行一次，执行时间为 10ms；任务B 只要求每 50ms 执行，执行时间为 25ms。试按最低松弛度优先算法进行调度。(Ai, Bi分别表示A, B的第i次运行）



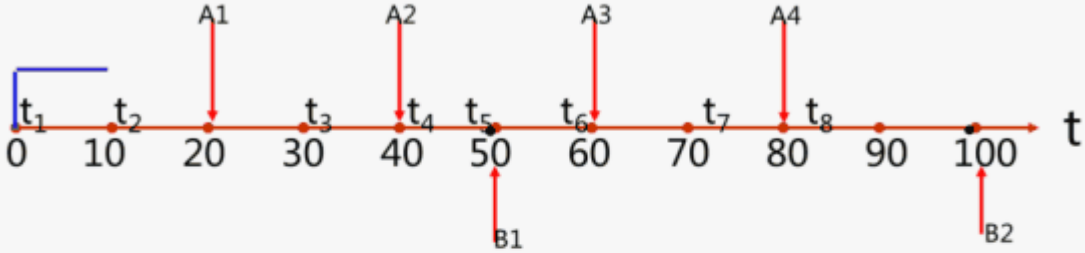
A每次10ms
B每次25ms

初始：t=0时刻，计算A，B松弛度：

$A_1 = 20 - 10 - 0 = 10$

$B_1 = 50 - 25 - 0 = 25$

松弛度=必须完成的时间点 - 本身剩余运行时间 - 当前时间

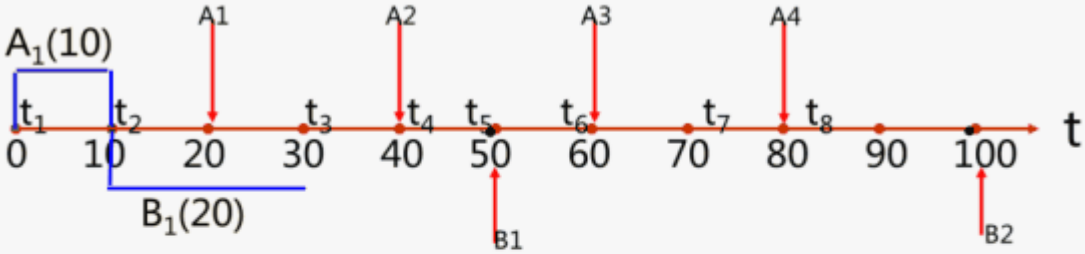


调度时机:有进程执行完或有进程的松弛度为0时（抢占）

A每次10ms
B每次25ms

t=10时刻，计算A，B松弛度：
$A_2 = 40 - 10 - 10 = 20$
$B_1 = 50 - 25 - 10 = 15$

松弛度=必须完成的时间点 - 本身剩余运行时间 - 当前时间

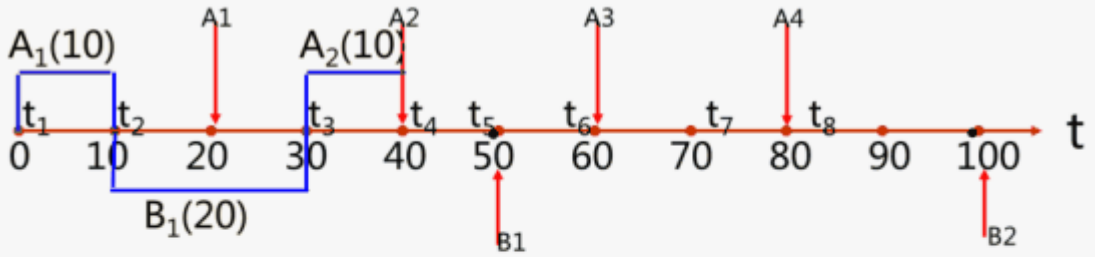


调度时机:有进程执行完或有进程的松弛度为0时（抢占）

A每次10ms
B每次25ms

t=30时刻，计算A，B松弛度：
$A_2 = 40 - 10 - 30 = 0$
$B_1 = 50 - 5 - 30 = 15$

松弛度=必须完成的时间点 - 本身剩余运行时间 - 当前时间

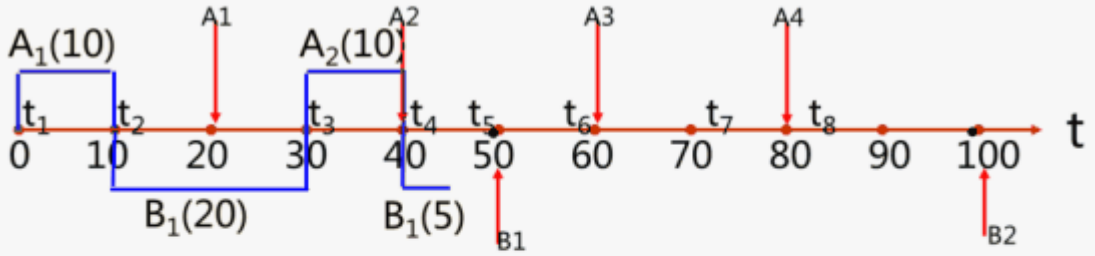


调度时机:有进程执行完或有进程的松弛度为0时（抢占）

A每次10ms
B每次25ms

t=40时刻，计算A，B松弛度：
$A_3 = 60 - 10 - 40 = 10$
$B_1 = 50 - 5 - 40 = 5$

松弛度=必须完成的时间点 - 本身剩余运行时间 - 当前时间

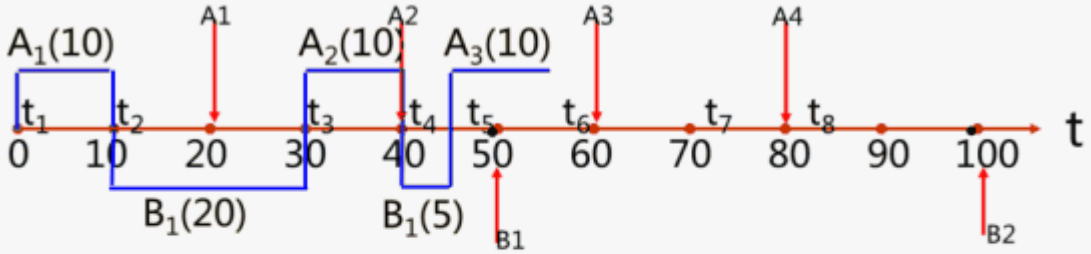


调度时机:有进程执行完或有进程的松弛度为0时（抢占）

A每次10ms
B每次25ms

t=45时刻，计算A，B松弛度：
$B_2 = 100 - 25 - 45 = 30$
$A_3 = 60 - 10 - 45 = 5$

松弛度=必须完成的时间点 - 本身剩余运行时间 - 当前时间



调度时机:有进程执行完或有进程的松弛度为0时（抢占）

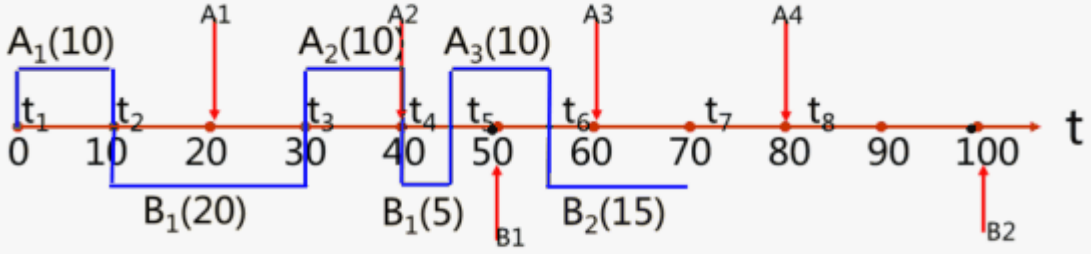
A每次10ms
B每次25ms

t=55时刻，计算A，B松弛度：

$A_4 = 80 - 10 - 55 = 35$

$B_2 = 100 - 25 - 55 = 20$

松弛度=必须完成的时间点 - 本身剩余运行时间 - 当前时间



调度时机:有进程执行完或有进程的松弛度为0时（抢占）

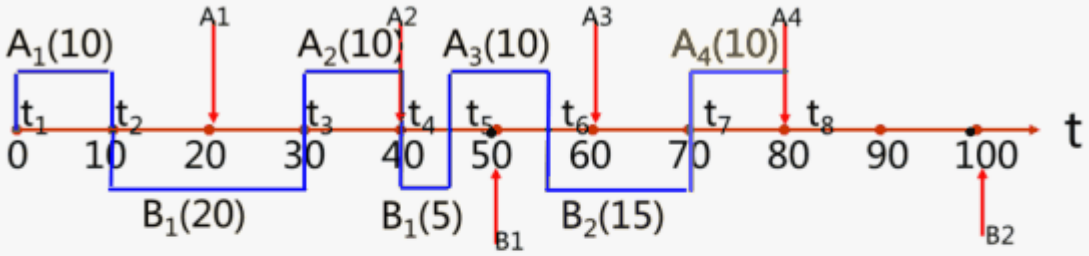
A每次10ms
B每次25ms

t=70时刻，计算A，B松弛度：

$A_4 = 80 - 10 - 70 = 0$

$B_2 = 100 - 10 - 70 = 20$

松弛度=必须完成的时间点 - 本身剩余运行时间 - 当前时间



调度时机:有进程执行完或有进程的松弛度为0时（抢占）

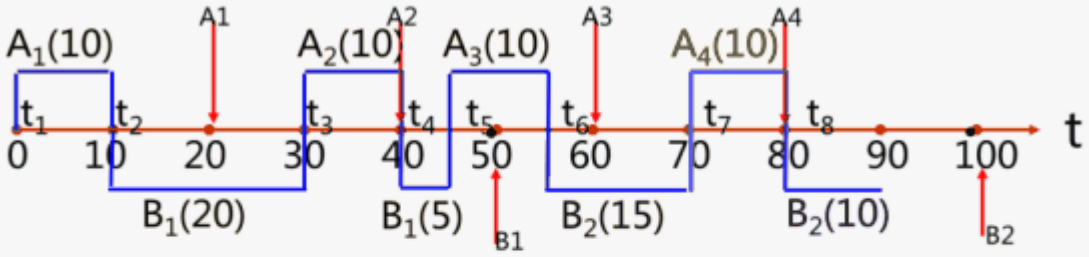
A每次10ms
B每次25ms

t=80时刻，计算A，B松弛度：

$A_5 = 100 - 10 - 80 = 10$

$B_2 = 100 - 10 - 80 = 10$

松弛度=必须完成的时间点 - 本身剩余运行时间 - 当前时间



调度时机:有进程执行完或有进程的松弛度为0时（抢占）

A每次10ms
B每次25ms

松弛度=必须完成的时间点 - 本身剩余运行时间 - 当前时间

内容提要

- 基本概念
- 设计调度算法要考虑的问题
- 批处理系统的调度算法
- 交互式系统的调度算法
- 实时系统的调度算法
- 多处理机调度
- Linux处理机调度

多处理机调度

- 与单处理机调度的区别：
  - 注重整体运行效率（而不是个别处理机的利用率）
  - 更多样的调度算法
  - 多处理机访问OS数据结构时的互斥（对于共享内存系统）
- 调度单位广泛采用线程

非对称式多处理系统(AMP)

- AMP：Asymmetric Multi-Processor，指多处理器系统中各个处理器的地位不同。
- 主－从处理机系统，由主处理机管理一个公共就绪队列，并分派进程给从处理机执行。
- 各个处理机有固定分工，如执行OS的系统功能，I/O处理，应用程序。
- 有潜在的不可靠性（主机故障造成系统崩溃）。

对称式多处理系统(SMP)

- SMP：Symmetric Multi-Processor，指多处理器系统中，各个处理器的地位相同。
- 按控制方式，SMP调度算法可分为集中控制和分散控制。
  - 集中控制：静态和动态调度
  - 分散控制：自调度

对称式多处理系统(SMP)

- 静态分配(static assignment)：每个CPU设立一个就绪队列，进程从开始执行到完成，都在同一个CPU上。
  - 优点：调度算法开销小。
  - 缺点：容易出现忙闲不均。
- 动态分配(dynamic assignment)：所有CPU采用一个公共就绪队列，队首进程每次分派到当前空闲的CPU上执行。可防止系统中多个处理器 忙闲不均。

自调度（Self Scheduling）

- 自调度(self-scheduling)：
  - 整个系统采用一个公共就绪队列，每个处理机都可以从队列中选择适当进程来执行。
  - 可采用单处理机的（成熟）调度技术，是最常用的算法，实现时易于移植。
  - 变型：Mach OS中局部和全局就绪队列相结合，其中局部就绪队列中的线程优先调度。

自调度（Self Scheduling）

- 优点：
  - 不需要专门的处理机从事任务分派工作。
- 问题：
  - **队列同步开销**：各处理机共享就绪队列
  - **缓存更新开销**：被阻塞的进程重新运行时不一定仍在阻塞前的处理机上运行，那么高速缓存中的内容需要重置
  - **线程协作开销**：由于合作中的几个线程没有同时运行而受阻，进而被切换下来

成组调度(gang scheduling)

- 将一个进程中的一组线程，每次分派时同时到一组处理机上执行，在剥夺处理机时也同时对这一组线程进行。
- 优点
  - 通常这样的一组线程在应用逻辑上相互合作，成组调度提高了这些线程的执行并行度，有利于减少阻塞和加快推进速度，最终提高系统吞吐量。
  - 每次调度可以完成多个线程的分派，在系统内线程总数相同时能够减少调度次数，从而减少调度算法的开销

两种成组调度

	应用程序A	应用程序B
Cpu1	线程1	线程1
Cpu2	线程2	空闲
Cpu3	线程3	空闲
Cpu4	线程4	空闲
时间	1/2	1/2

面向程序：浪费3/8的处理机时间

	应用程序A	应用程序B
Cpu1	线程1	线程1
Cpu2	线程2	空闲
Cpu3	线程3	空闲
Cpu4	线程4	空闲
时间	4/5	1/5

面向线程：浪费3/20的处理机时间

面向所有程序
平分处理机时间

面向所有线程
平分处理机时间

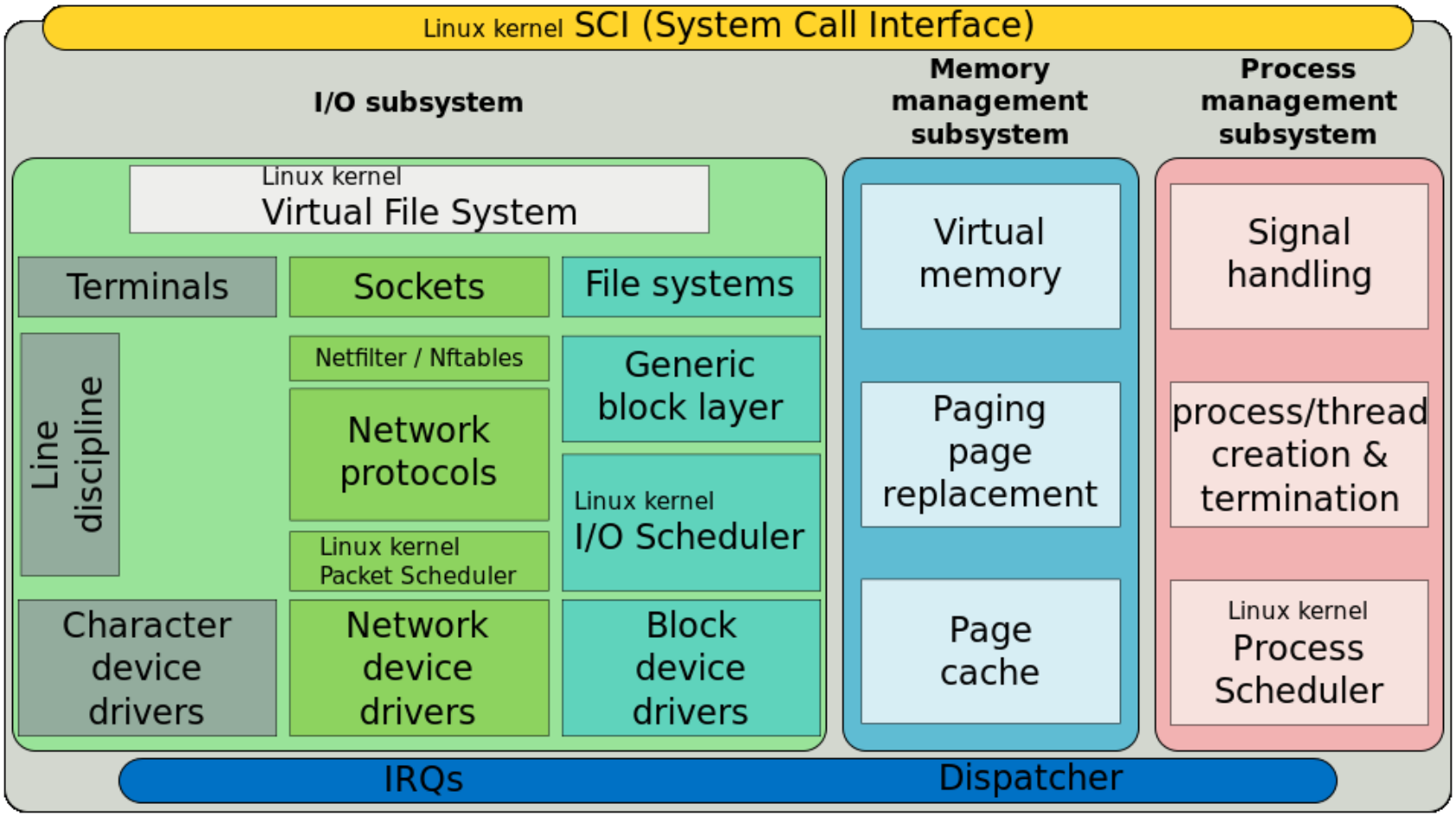
专用处理机调度
(dedicated processor assignment)

- 为进程中的每个线程都固定分配一个CPU，直到该线程执行完成。
- 缺点：线程阻塞时，造成CPU的闲置。
- 优点：线程执行时不需切换，相应的开销可以大大减小，推进速度更快。
- 适用场合：CPU数量众多的高度并行系统，单个CPU利用率已不太重要。

内容提要

- 基本概念
- 设计调度算法要考虑的问题
- 批处理系统的调度算法
- 交互式系统的调度算法
- 实时系统的调度算法
- 多处理机调度
- Linux处理机调度

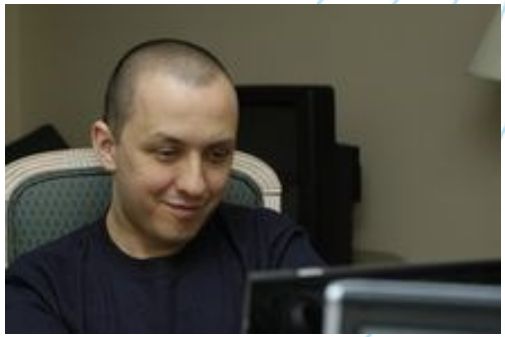
Linux Kernel结构（简化说明）



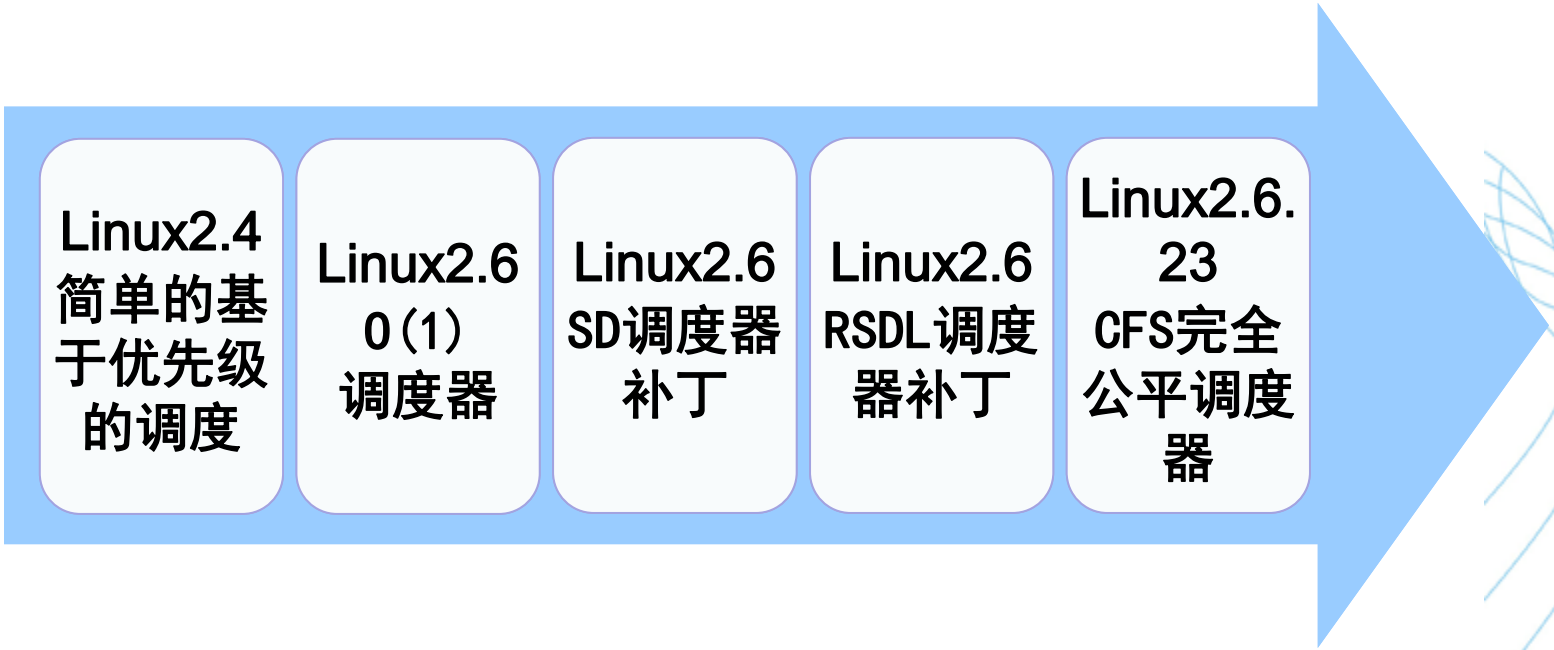
Linux的调度

- Linux 2.4：O(n)调度器 (Linus)
- Linux 2.6.0：O(1)调度器
- Linux 2.6.23：CFS（Completely Fair Scheduler，完全公平调度器）(Ingo Molnár)

- 其他：(Con Kolivas)
  - SD(staircase deadline scheduler)
  - RSDL(The Rotating Staircase Deadline Schedule)
  - BFS



Linux调度算法的发展历史



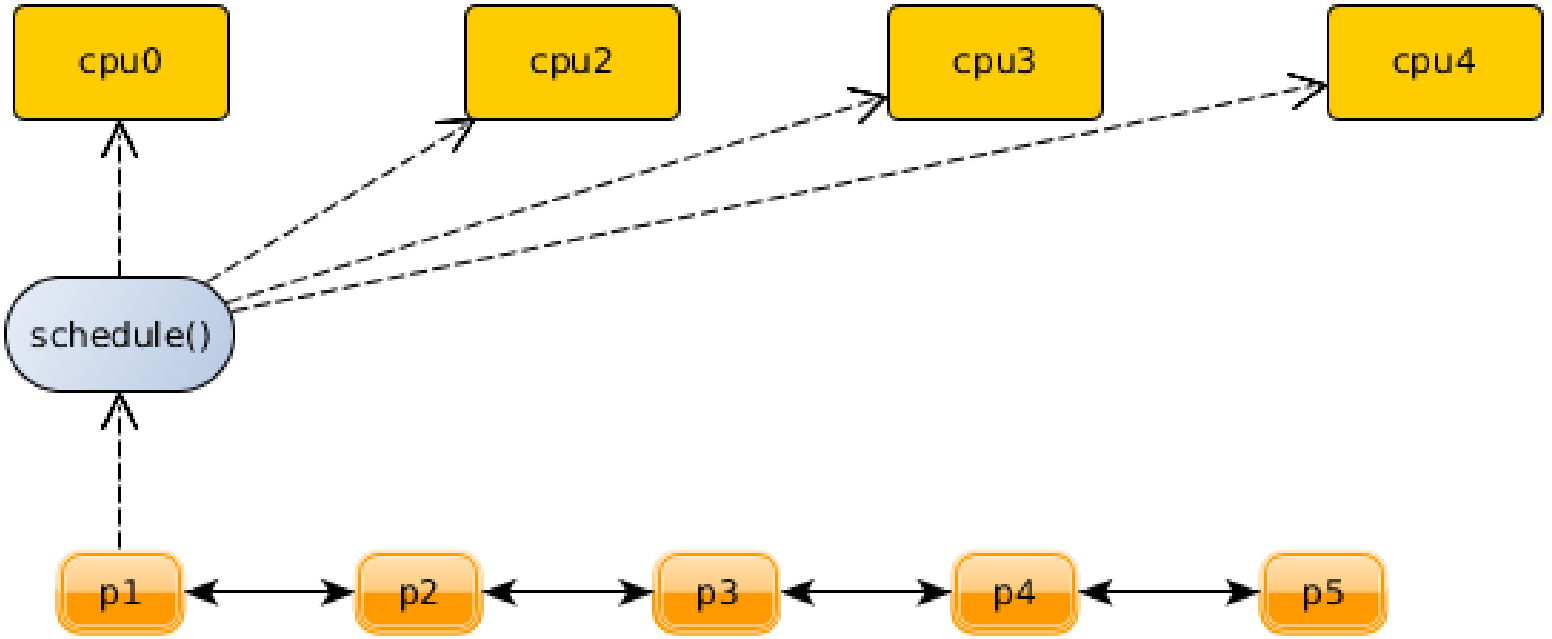
参考网址：
http://abcdxyzk.github.io/blog/2015/01/22/kernel-sched-n1/

Linux2.4调度器

- 2.4的调度算法，将所有的就绪进程组织成一条可运行队列，不管是单核环境还是smp环境，cpu都只从这条可运行队列中循环遍历直到挑选到下一个要运行的进程。如果所有的进程的时间片都用完，就重新计算所有进程的时间片。
- 调度策略：
  - 实时进程可使用FIFO策略：进程会一直占用cpu除非其自动放弃cpu。
  - 实时进程可使用RR策略：当分配给进程的时间片用完后，进程会插入到原来优先级的队列中。
  - 普通进程基于优先级的时间片轮转调度。

Linux2.4调度器

2.4调度：



2.4调度的不足

- 时间复杂度为O(n)，每次都要遍历队列，效率低！
- 在smp环境下多个cpu使用同一条运行队列，进程在多个cpu间切换会使cpu的缓存效率降低，降低系统的性能。
- 不支持内核抢占，内核不能及时响应实时任务，无法满足实时系统的要求（即使linux不是一个硬实时，但同样无法满足软实时性的要求）。
- 更倾向优先执行I/O型进程。
- 负载平衡策略简单，进程迁移比较频繁。

linux 2.6 O(1)调度器

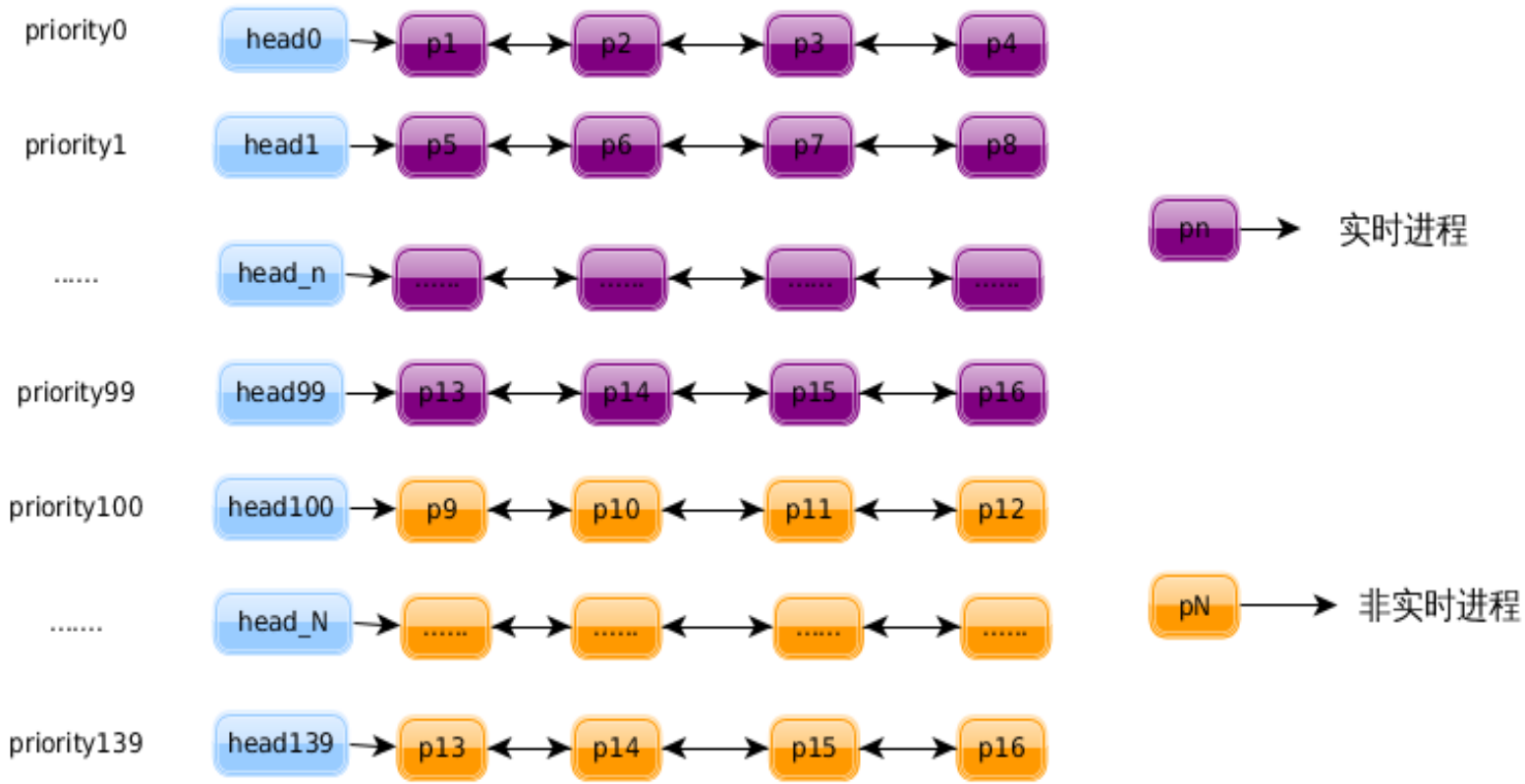
- O(1)调度器每个cpu维护一个自己的运行队列，每个运行队列有自己的active队列与expired队列。
- 当进程的时间片用完后就会被放入expired队列中。当active队列中所有进程的时间片都用完，进程执行完毕后，交换active和expired。这样expired队列就成为了active队列。这样做只需要指针的交换而已。
- 当调度程序要找出下一个要运行的进程时，要根据位图宏来找出优先级最高的且有就绪进程的队列。这样的数据组织下，2.6的调度器的时间复杂度由原来2.4的O(n)提高到O(1)。对smp环境具有较好的伸缩性。

linux 2.6 O(1)调度器

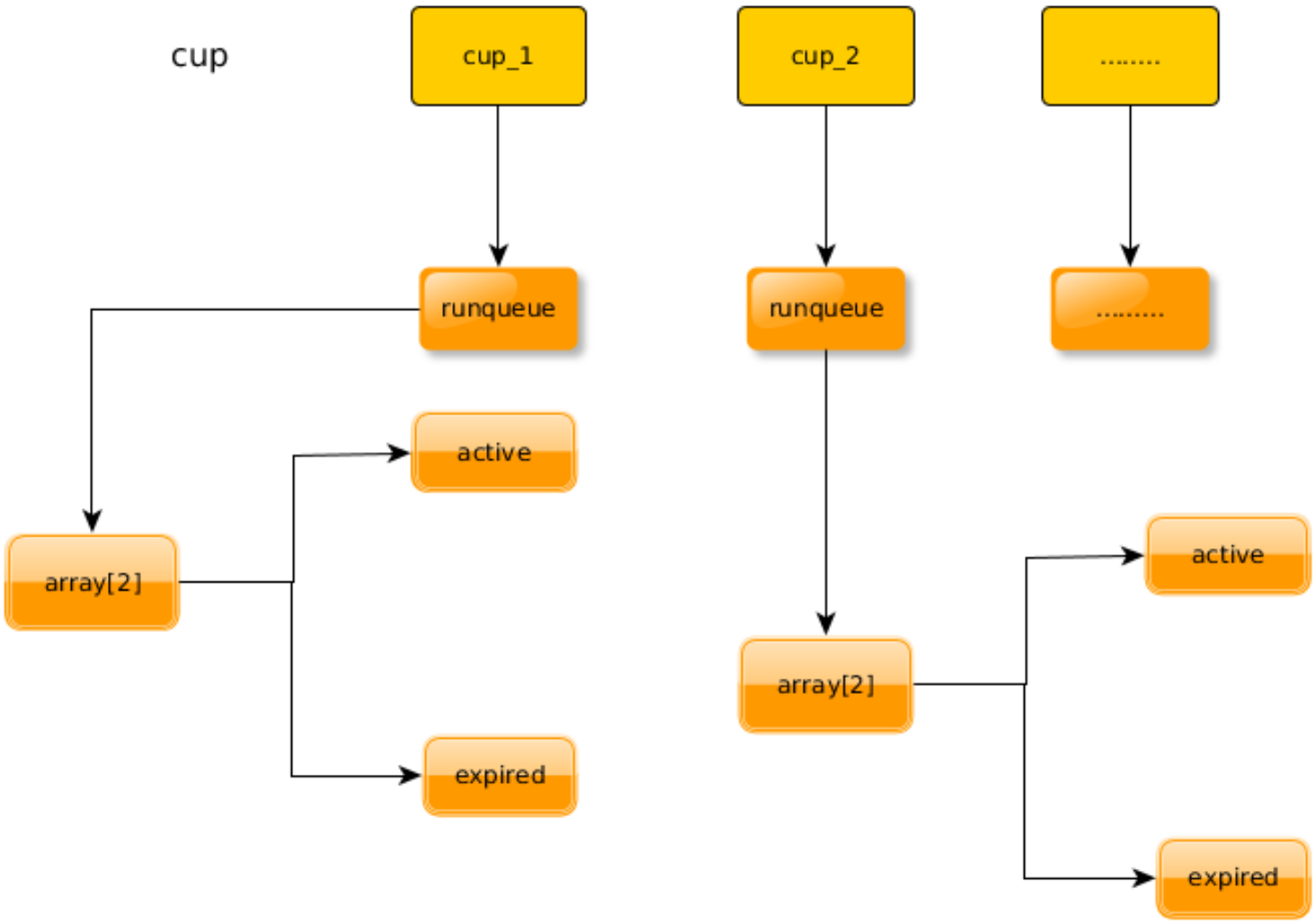
- 调度有140个优先级级别，由0～139, 0～99 为实时优先级，而100～139为非实时优先级。
- 调度策略：与2.4相同。

linux 2.6 O(1)调度器

2.6调度：



linux 2.6 O(1)调度器



2.6 O(1)调度器特点

- 动态优先级是在静态优先级的基础上结合进程的运行状态和进程的交互性来计算。
- 进程优先级越高，它每次执行的时间片就越长。
- 如果一个进程的交互性比较强，那么其执行完自身的时间片后不会移到expired队列中，而是插到原来队列的尾部。这样交互性进程可以快速地响应用户，交互性会提高。

缺陷：

- 负载均衡策略复杂。
- 根据经验公式和平均休眠时间来决定、修改进程的优先级的方法难以理解。

linux 2.6 SD调度器

特点：

- 与O(1)相比，少了expired队列。
- 进程在用完其时间片后放入下一个优先级队列中。当下降到最低一级时，时间片用完，就回到初始优先级队列，重新降级的过程！每一次降级就像下楼梯的过程，所以这被称为楼梯算法。
- 采用粗/细粒度两种时间片。粗粒度由多个细粒度组成，当一个粗粒度时间片被用完，进程就开始降级，一个细粒度用完就进行轮回。这样cpu消耗型的进程在每个优先级上停留的时间都是一样的。而I/O消耗型的进程会在优先级最高的队列上停留比较长的时间，而且不一定会滑落到很低的优先级队列上去。

linux 2.6 SD调度器

- 不会饥饿，代码比O(1)调度简单，最重要的意义在于证明了完全公平的思想的可行性。

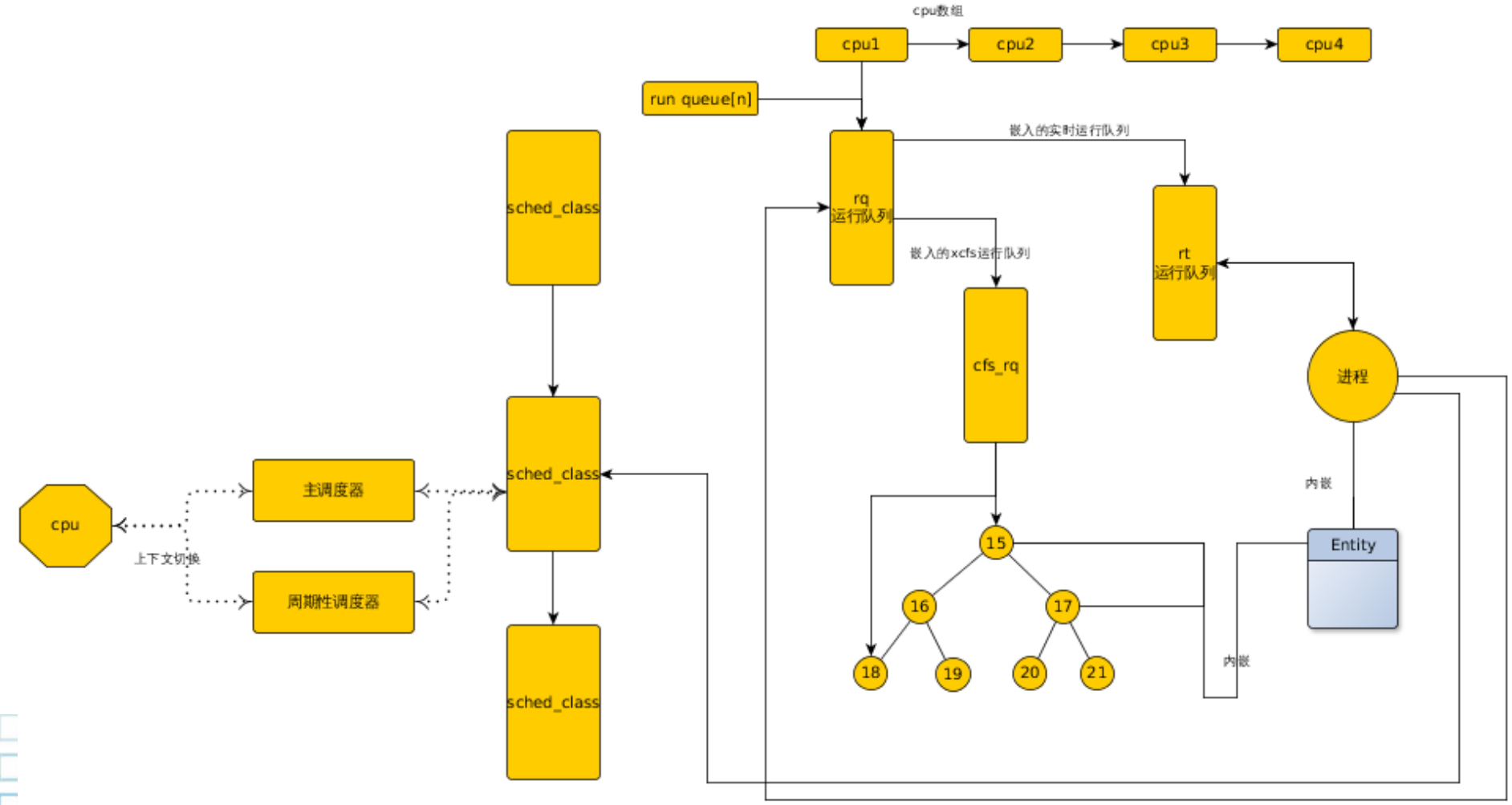
不足：

- 相对与O(1)调度的算法框架还是没有多大的变化。
- 每一次调整优先级的过程都是一次进程的切换过程，细粒度的时间片通常比O(1)调度的时间片短很多。这样不可避免地带来了较大的额外开销，使吞吐量下降的问题。

linux 2.6 RSDL调度器

- 对SD算法的改进，其核心思想是“完全公平”，并且没有复杂的动态优先级的调整策略。
- 引进“组时间配额” → tg 每个优先级队列上所有进程可以使用的总时间，“
- 优先级时间配额” → tp, tp不等于进程的时间片，而是小于进程时间片。
- 当进程的tp用完后就降级。与SD算法相类似。当每个队列的tg用完后不管队列中是否有tp没有用完，该队列的所有进程都会被强制降级。

linux 2.6 RSDL调度器



CFS调度器

- 自内核版本2.6.23开始，Linux就一直使用CFS调度器。
- cfs的 80% 的工作可以用一句话来概括：cfs在真实的硬件上模拟了完全理想的多任务处理器。
- 在完全理想的多任务处理器下，每个进程都能够同时获得cpu的执行时间，当系统中有两个进程时，cpu时间被分成两份，每个进程占50%。

CFS调度器

特点：

- 采用虚拟运行时间vt。进程的 vt 与其实际的运行时间成正比，与其权重成反比。权重是由进程优先级来决定的，而优先级又参照nice（控制优先级的因子，取值范围为-20~+19，初始值为0）值的大小。进程优先级权重越高，在实际运行时间相同时，进程的vt就越小。
- 完全公平的思想。cfs不再跟踪进程的休眠时间、也不再区分交互式进程，其将所有的进程统一对待，在既定的时间内每个进程都可获得公平的cpu占用时间。

CFS调度器

- cfs 引入了模块化、完全公平调度、组调度等一系列特性。
- cfs使用weight 权重来区分进程间不平等的地位，这也是cfs实现公平的依据。权重由优先级来决定，优先级越高，权重越大。但优先级与权重之间的关系并不是简单的线性关系。内核使用一些经验数值来进行转化。
  - 如果有a、b、c 三个进程，权重分别是1、2、3,那么所有的进程的权重和为6, 按照cfs的公平原则来分配，那么a的重要性为1/6, b、c 为2/6, 3/6。这样如果a、b、c 运行一次的总时间为6个时间单位，a占1个，b占2个，c占3个。

小结

- 调度的类型（如调度单位的不同级别，时间周期，不同的OS），性能准则
- 调度算法：FCFS，SJF，RR，多级队列，优先级，多级反馈队列
- 调度算法的性能分析：周转时间和作业长短的关系
- 实时调度：调度算法
- 多处理机调度：自调度，成组调度，专用处理机调度