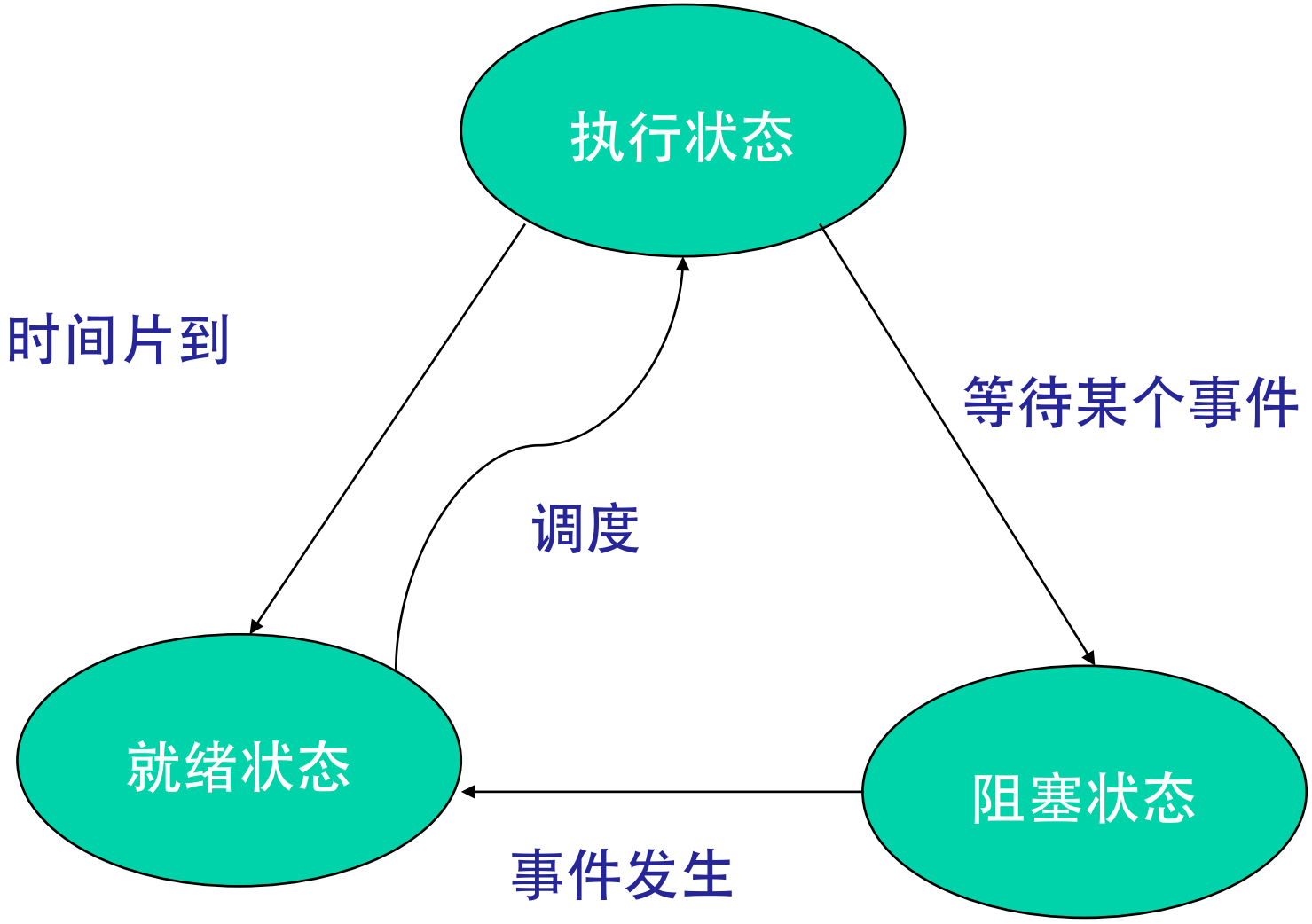
计算机操作系统

- 进程调度

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

进程的状态转换图



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 。

CPU调度

- 什么场景下调度才有价值?
  - 当CPU成为稀缺资源时
- 调度关键的典型场景
  - 支持多道程序的分时系统
  - 结合分时和批处理的大型机
  - 网络服务器
- 调度不那么重要的场景
  - 运行少量程序个人电脑
  - 运行交互式应用为主的高速的个人电脑

要解决的问题

- 调度的时机：
  - 何时分配CPU,进行调度
- 调度的执行：
  - 如何分配CPU —进程的上下文切换
- 调度的策略：
  - 按什么原则选择下一个要执行的进程（分配CPU）— 进程调度算法

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。

何时进行调度

- 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
  - 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
  - 陷阱：最末一条指令导致出错，会引起进程移至退出状态(除0，非法内存访问)

调度的执行-CPU切换过程

- 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；保存内存镜像（页表信息）；
  - 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-就绪或阻塞;
  - 选择另一个要执行的进程，更新进程的PCB;
  - 从被选中进程中重装入CPU 上下文和内存镜像。

调度的执行-CPU切换过程

- 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；保存内存镜像（页表信息）；
  - 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-就绪或阻塞；
  - 选择另一个要执行的进程，更新进程的PCB；
  - 从被选中进程中重装入CPU 上下文和内存镜像。

上下文切换是有成本的- 避免频繁切换

传统调度的类型

- 高级调度
- 中级调度
- 低级调度

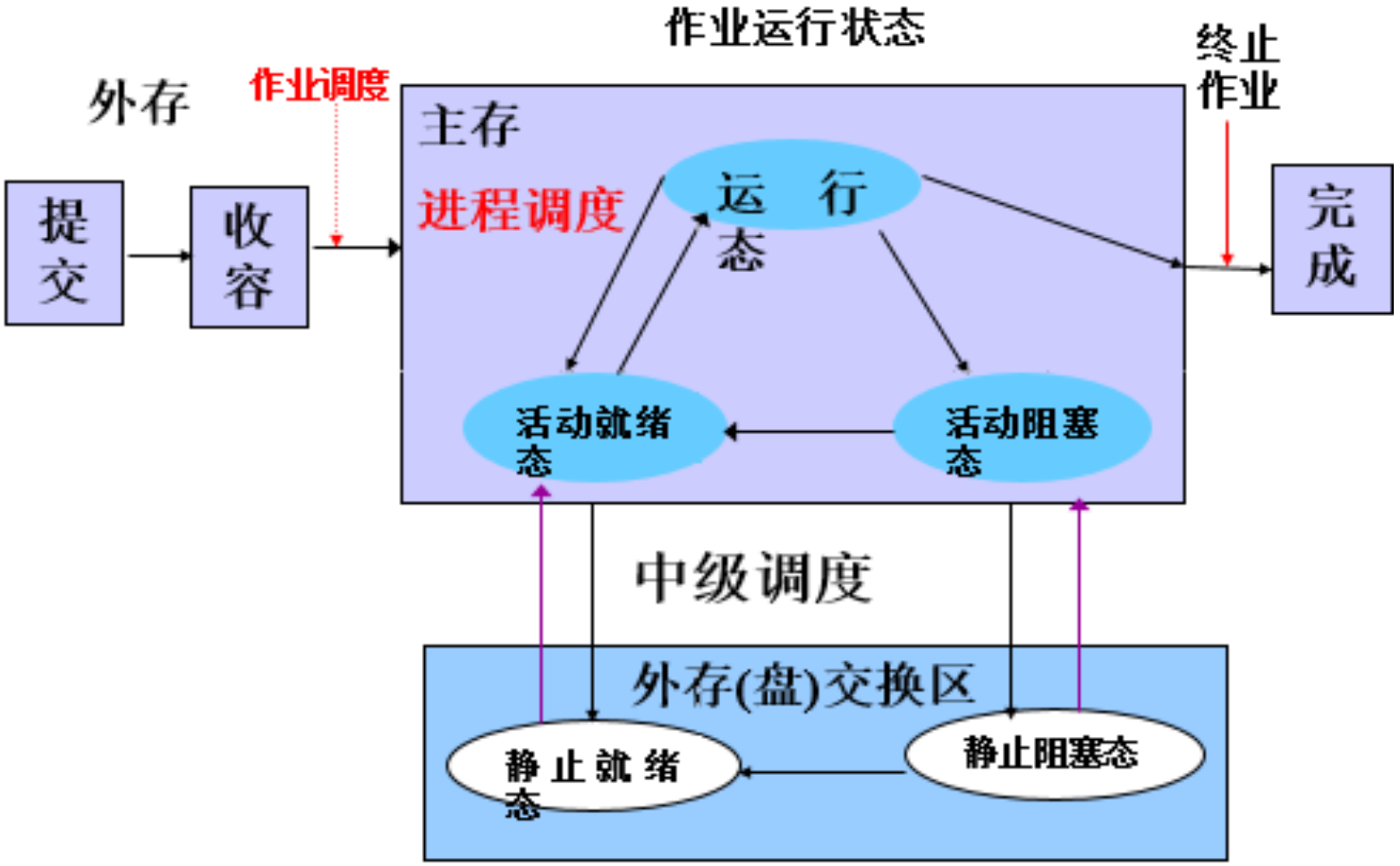
高级调度和中级调度

- 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
  - 接纳那些作业
- 中级调度：又称为“内外存交换”。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。指令和数据必须在内存里才能被CPU直接访问。

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。

CPU三级调度

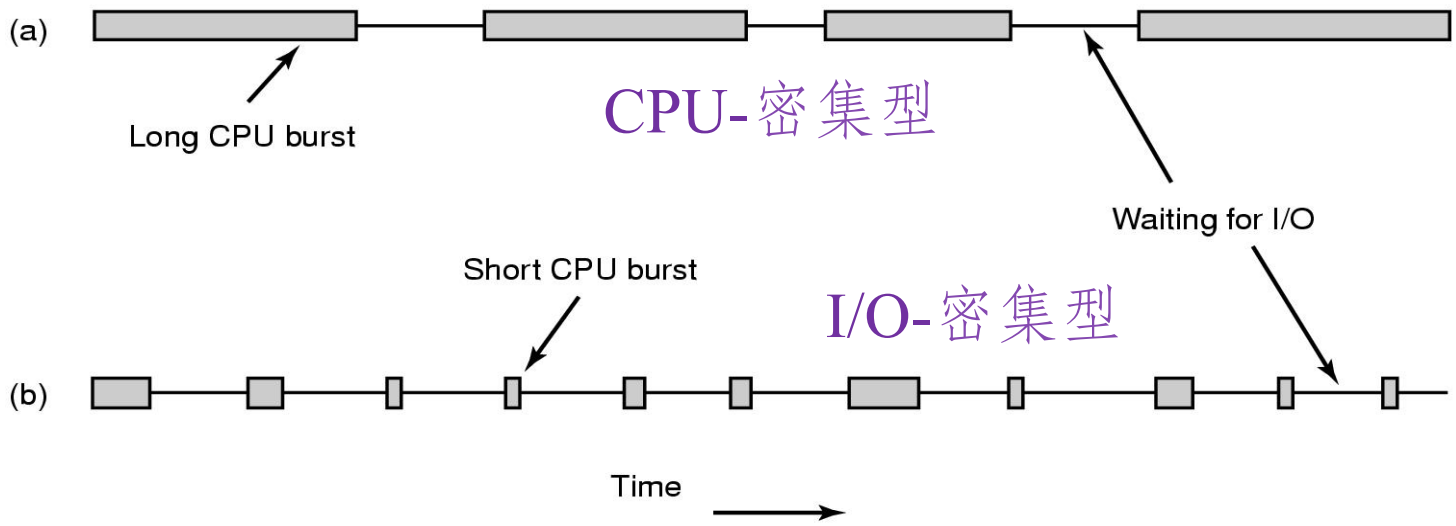

作业运行状态
外存
作业调度
主存
终止
作业
提交
收容
进程调度
运 行
态
完成
活动就绪
态
活动阻塞
态
中级调度
外存(盘)交换区
静止就绪
态
静止阻塞态

内容提要

- 基本概念
- 进程调度要考虑的问题
- 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

进程的特性及其分类

- 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
  - 计算时间相对I/O时间短
- CPU bound（CPU密集型）
  - 计算量大，花费大量的时间在计算上



进程（线程）调度的分类一

- 根据是否在时钟中断作出调度决策
- 非抢占式
  - 被调度进程会一直执行直到阻塞或主动放弃CPU
  - 不会强迫进程挂起
  - 时钟中断时候不会做调度决策
- 抢占式
  - 被调度程序运行一个固定的最大时间段，如果时间结束进程还在执行，挂起进程，切换。
  - 在时间段末，通过时钟中断做调度决策

进程（线程）调度的分类二

- 根据不同的应用领域，优化目标不同
- 批处理系统
  - 交易后清算、结算，索赔处理,利息计算
  - 减少切换改善性能
  - 非抢占式算法或分配长时间周期的抢占算法
- 交互式系统
  - 服务于多个用户的环境，需要避免一个进程霸占CPU
  - 分时系统、手机系统
  - 抢占式算法

进程（线程）调度的分类二

- 根据不同的应用领域，优化目标不同
- 实时系统
  - 机器人控制、飞行控制、多媒体
  - 满足确定性的截止时间需求
  - 抢占式为主，也可能非抢占式

调度算法的目标-通用

- 适用于所有系统的目标
  - 公平
    - 给每个进程公平的CPU份额
    - 相似的进程应该得到相似的CPU份额
    - 不同类型的进程可以有区别
      - » 核反应控制 v.s. 薪水计算

调度算法的目标-通用

- 适用于所有系统的目标
  - 策略的强制执行
    - 必须能够严格执行策略
  - 平衡
    - 尽量让系统的各个部件都忙碌起来
    - 内存中：组合CPU密集型和I/O密集型进程

调度算法的目标-批处理

- 吞吐量：每小时所完成的作业数。
  - 如：在2小时内完成4个作业，则吞吐量是2个作业/小时
- 周转时间：一批作业从提交到完成（得到结果）所经历的统计平均时间。
  - 包括：外存等待时间、就绪等待时间、CPU执行时间、I/O操作时间等
  - 平均周转时间、带权平均周转时间（T/Ts)
- 吞吐量和周转时间不一定负相关的
  - 例如，总是优先运行短作业的调度程序

调度算法的目标-批处理

- CPU利用率
  - 常常被用做评估批处理系统的指标
  - 不如吞度量有效
  - CPU利用率接近100%→需要购买额外的硬件了

调度算法的目标-交互式系统

- 最小化响应时间：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
  - 交互式系统应该首先满足交互式请求
  - 例如：后台的杀毒/邮件不应该影响前台交互
- 等比例变化：任务花费时间随着其复杂度应该线性增长
  - 下载一个2G的电影花费20分钟时间:-)
  - 打开教务处选课系统花费了1分钟时间☹

调度算法的目标-实时系统

- 满足截止时间：满足所有或者大多数应用的截止时间
  - 例如飞控系统需要100ms内完成飞行控制命令
  - 自动驾驶系统需要在0.5s内执行刹车命令
  - 收据收集进程需要及时的收集特定速率的数据
- 可预测性：保证可预测性和规律性
  - 音频播放时可预测性-不规则的音频数据会迅速降低音质

调度算法本身的目标

- 易于实现
- 执行的性能开销小

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

吞吐量、平均等待时间和平均周转时间

- 吞吐量=作业数/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=作业周转时间之和/作业数
- 平均带权周转时间=作业带权周转时间之和/作业数

度量指标的定义

- 吞吐量：作业数/总执行时间
- 周转时间：完成时间-提交时间
- 带权周转时间：周转时间/服务时间（执行时间）
- 平均周转时间：一组作业周转时间之和/作业数
- 平均带权周转时间：一组作业带权周转时间之和/作业数

批处理系统中常用的调度算法

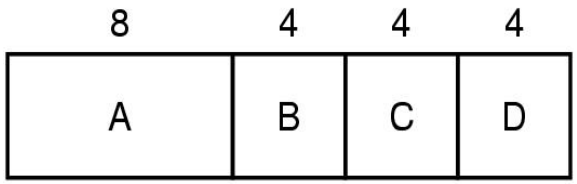
- 先来先服务（FCFS：First Come First Serve)
- 最短作业优先（SJF：Shortest Job First)
- 最短剩余时间优先（SRTN：Shortest Remaining Time Next)
- 最高响应比优先（HRRN：Highest Response Ratio Next)

先来先服务FCFS

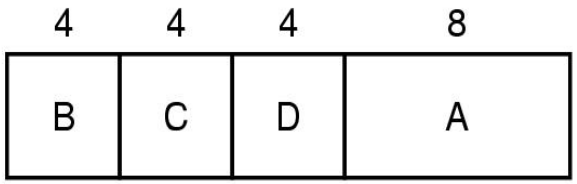
- 最简单的调度算法，按先后顺序调度。
  - 按照作业提交或进程变为就绪状态的先后次序进入就绪队列排队，按队列次序分派CPU
  - 当前作业或进程占用CPU，直到执行完或阻塞，才出让CPU（非抢占）。
  - 在作业或进程唤醒后（如I/O完成），放到队列尾部。
- FCFS的特点
  - 公平，容易理解和实现
  - 长作业>短作业(吞吐量受限)。
  - 有利于CPU繁忙的作业，不利于I/O繁忙的作业。

短作业优先SJF

- Shortest Job First. 又称“短进程优先”SPN(Shortest Process Next);
  - 对预期执行时间短的作业（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。
  - 这是对FCFS算法的改进，其目标是减少平均周转时间，提高吞吐量。



(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；
  - 仅仅当作业同时可运行，周转时间最优
  - 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；
  - 当难以准确估计作业（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	提交时间/时	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/时	运行时间/h	开始时刻/时	完成时刻/时
1	10:00	2	10:00	12:00
2	10:10	1	12:00	13:00
3	10:25	1/4	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/时	运行时间/h	开始时刻/时	完成时刻/时
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

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

带权平均周转时间为

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

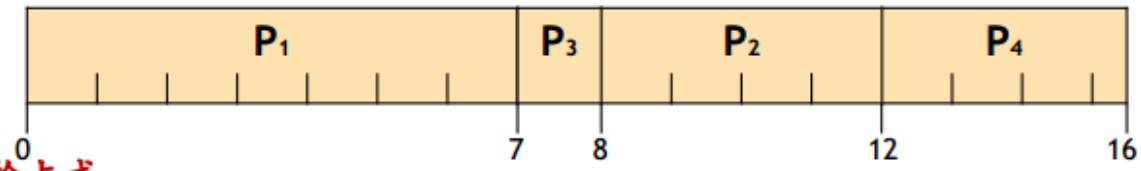
最短剩余时间优先SRTN

- Shortest Remaining Time Next 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。当一个新就绪的进程到达时，如果它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

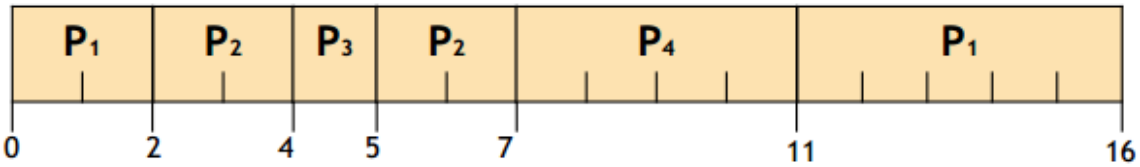
非抢占式

SJF



抢占式

SRTN



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

- HRRN（Highest Response Ratio Next）是FCFS算法和SJF算法的折衷。既考虑作业的运行时间，有考虑作业的等待时间。照顾短作业的同时又避免饿死长作业。
- 每次进行调度时，先计算后备作业队列每个作业的响应优先级，然后优先级最大的作业运行
- Response Priority: RP=1+已等待时间/要求运行时间

最高响应比优先HRRN

- 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（HRRN）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

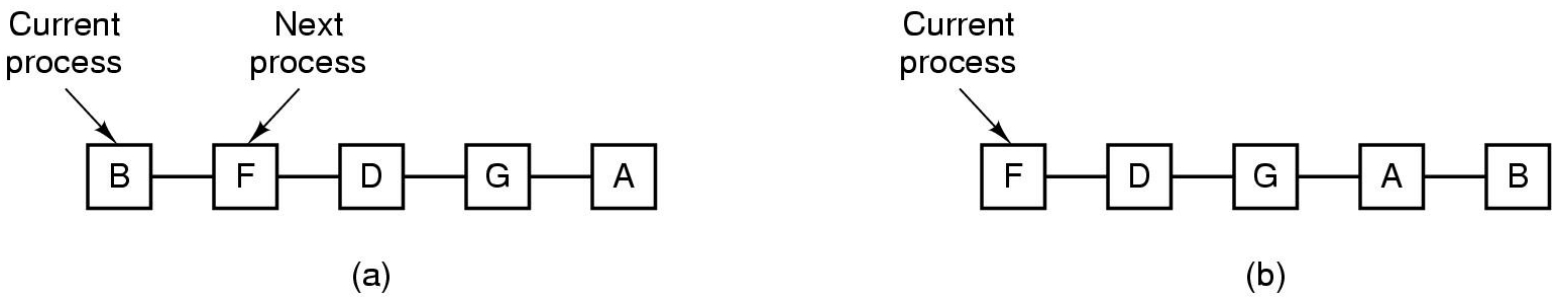
- 基本概念
- 设计调度算法要考虑的问题
- 批处理系统的调度算法
- 交互式系统的调度算法
- 实时系统的调度算法
- 多处理机调度

交互式系统的调度算法

- 时间片轮转(RR：Round Robin)
- 优先级调度(Priority Scheduling)
- 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

- 本算法的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；
- 古老、简单、公平、使用广泛
- 下图：当一个进程用完时间片，会移到队尾。



时间片轮转算法

- 将系统中所有的就绪进程按照FCFS原则，排成一个队列。
- 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。
- 如果进程在时间片结束前阻塞或者结束，会立刻进行CPU切换。
- 在一个时间片结束时，发生时钟中断。
- 如果此时进程还在运行，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。

时间片长度的确定

- 时间片长度变化的影响
  - 过短－>如果时间片过短，切换过于频繁，上下文切换的时间就相对比例较大，切换开销过大。
    - 4ms时间片，上下文切换1ms，切换开销20%
  - 过长－>排在就绪队列后面的进程需要等待的时间过长。
    - 50个进程排队，100ms时间片，最后一个需要5s才被响应
  - 比平均的CPU突发使用（burst）时间长一点，进程会阻塞或者执行完毕，减少切换，提高整体性能。
  - 一般来说20ms~50ms是一个合理的折中值（MOS）

CPU执行速度

10ms能够执行多少条指令？It depends …

- 流水线的工作频率
- 普通流水线、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？

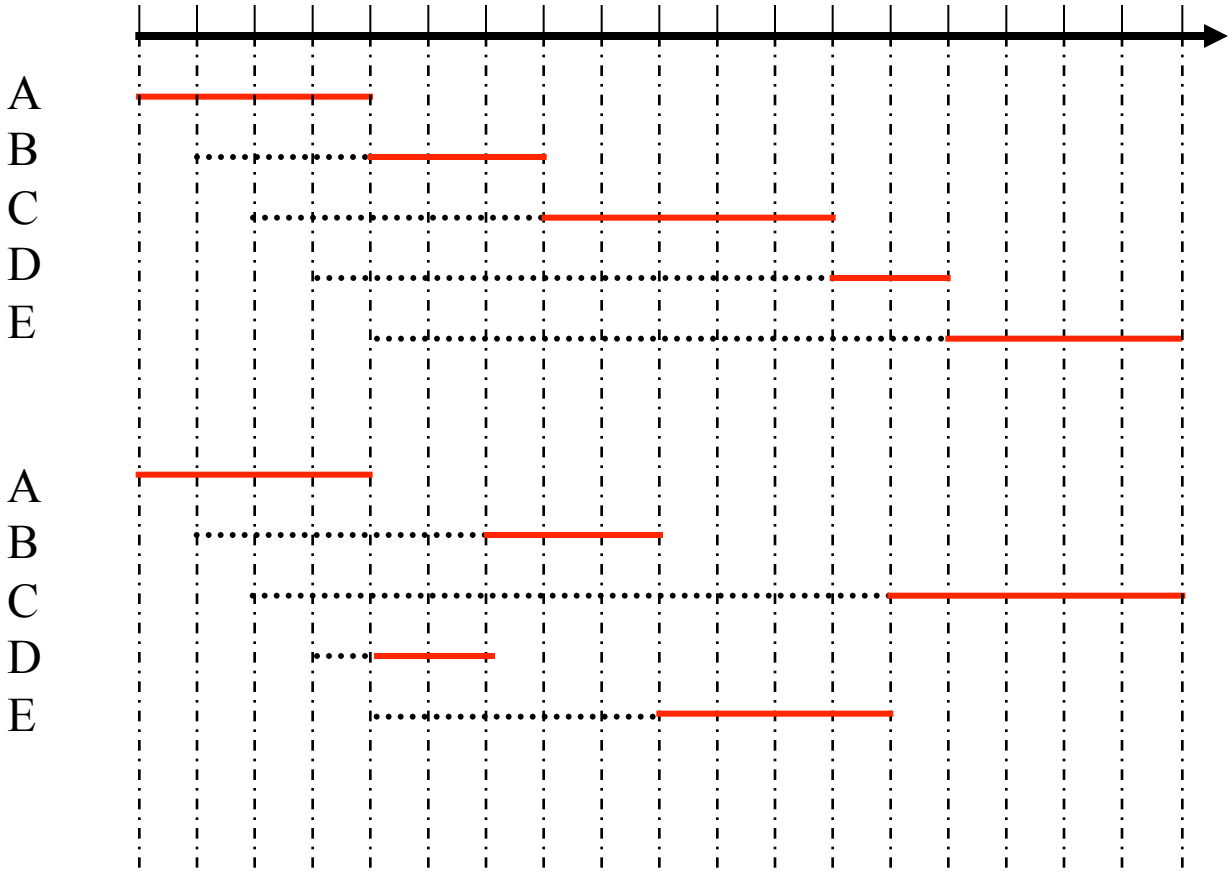
作业调度举例

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

A B C D E
0 1 2 3 4
4 3 5 2 4

作业到达时间

作业执行长度



FCFS

平均周转时间 ＝ 9
带权平均周转时间
＝2.8

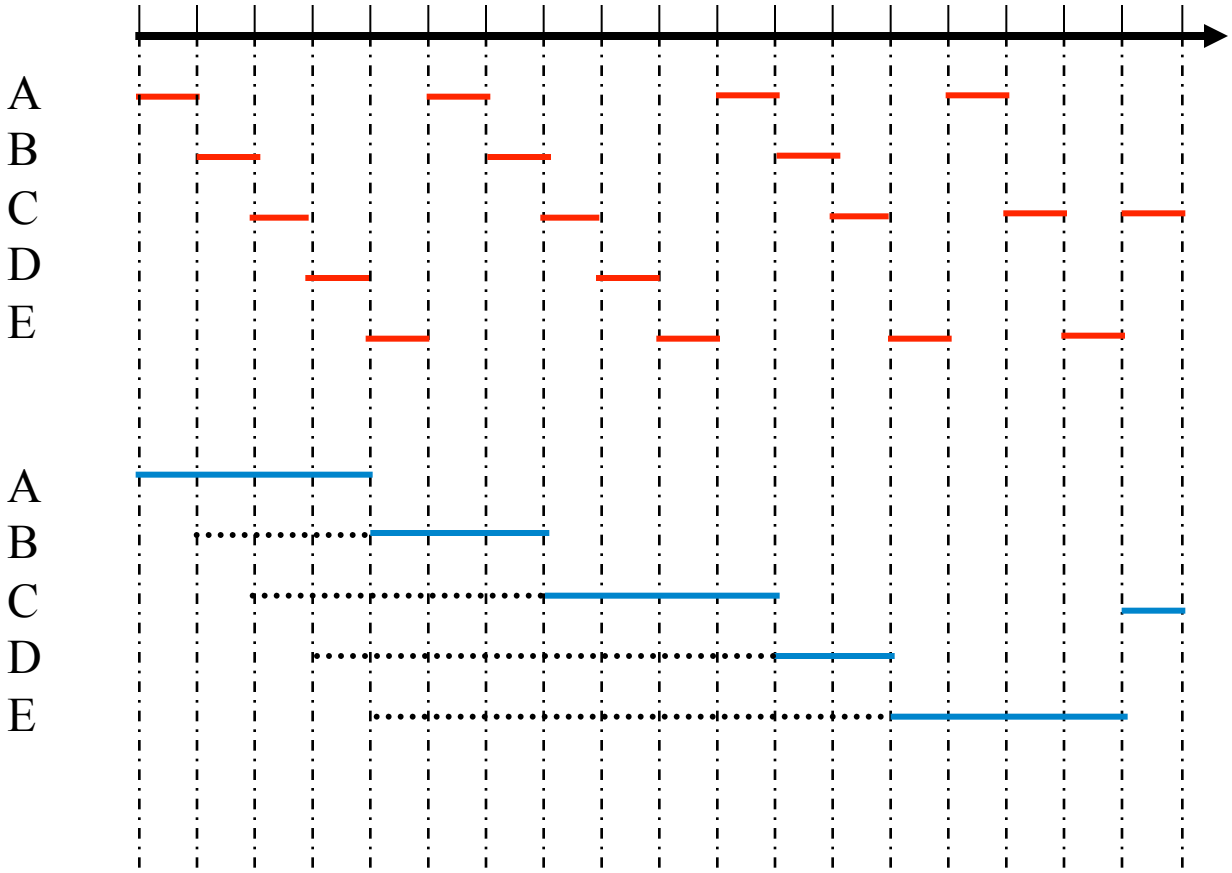
SJF

平均周转时间 ＝ 8
带权平均周转时间
＝2.1

A B C D E
0 1 2 3 4
4 3 5 2 4

作业到达时间

作业执行长度



RR

1单位时间片
15次切换

4单位时间片
5次切换

优先级调度 (Priority Scheduling)

- RR假设所有进程一样重要；
- 优先级调度则赋予每个进程不同优先级，高优先级先运行。
- 可以静态或者动态赋予进程优先级；

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。

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（系统进程和交互进程优先级较高）
  - 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）
    - 高优先级500rmb，中优先级300rmb，低优先级100rmb

静态优先级

- UNIX的nice命令
  - 让进程主动降低地自己的优先级
  - Be nice to other process
  - $ nice -n 19 tar cvzf archive.tgz largefile

动态优先级

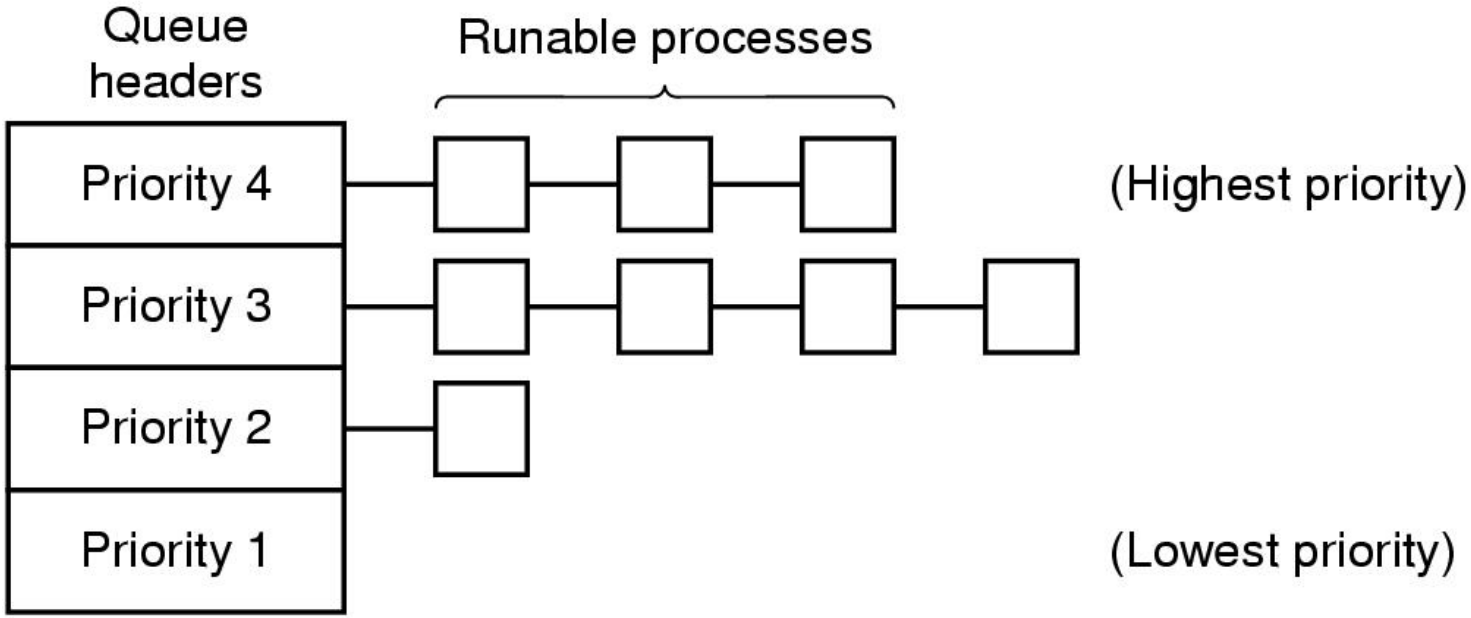
- 在进程运行过程中可以动态改变优先级，以便获得更好的调度性能。如：
  - 在就绪队列中，等待时间越长则优先级提高，从而使优先级较低的进程在等待足够的时间后，其优先级提高到可被调度执行；
  - 进程每执行一个时间片，就降低其优先级，从而一个进程持续执行时，其优先级逐步降低到出让CPU。
  - I/O密集型进程应该设置较高优先级：1/f
    - f为上一个时间片中占用的时间。

多级队列算法MQ

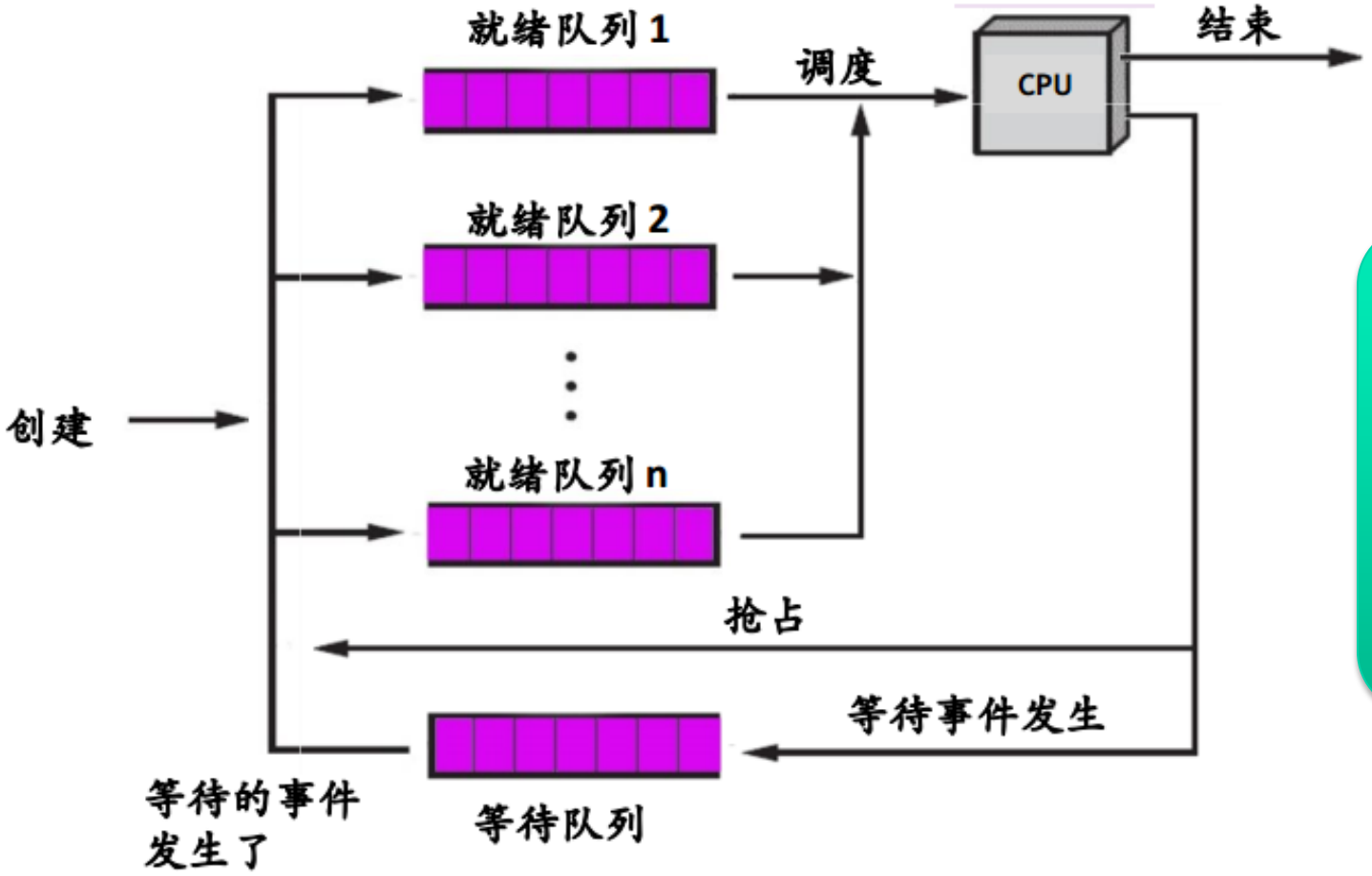
- 本算法引入多个就绪队列，通过各队列的区别对待，达到一个综合的调度目标；
  - 根据作业或进程的性质或类型的不同，分配到不同就绪队列。
  - 每个作业固定归入一个队列。
- 不同队列可有不同的优先级、时间片长度、调度策略等；在运行过程中还可改变进程所在队列。如：系统进程、用户交互进程、批处理进程等。

多级队列算法MQ

- 将进程按照优先级进行分类，每类一个队列。各类之间按照优先级进行调度：队列4空才调度队列3，以此类推。
- 每类之内按照FCFS调度。

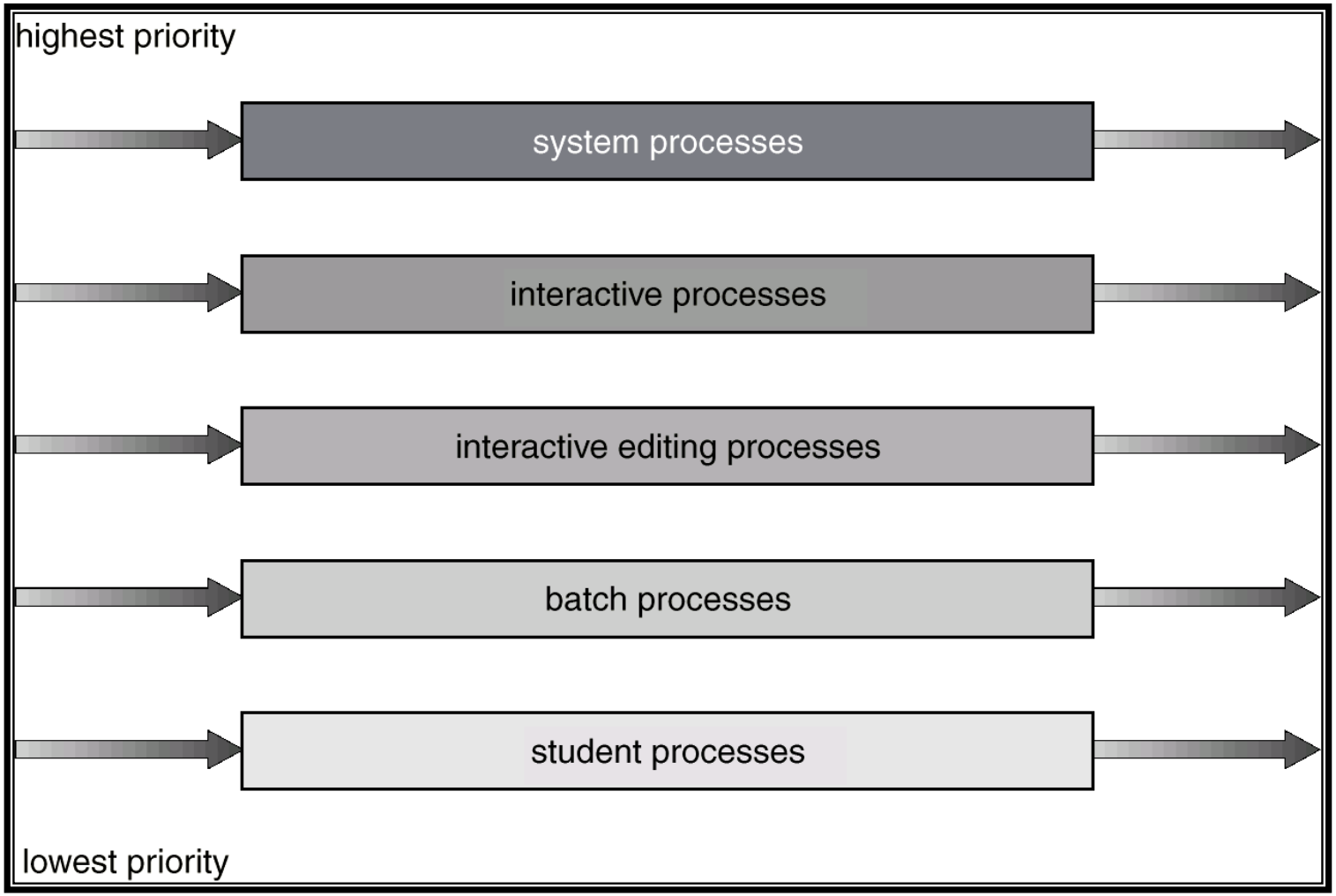


多级队列算法MQ



Berkeley
XDS 940 系统
1. 终端
2. IO
3. 短时间片
4. 长时间片

多级队列算法MQ


highest priority
system processes
interactive processes
interactive editing processes
batch processes
student processes
lowest priority

多级反馈队列算法MFQ

- 问题
  - CPU密集型的任务应该分配较长的时间片连续执行，减少切换开销
  - 但是长时间片进程又会影响对交互进程的响应
  - 预先难以知道进程的性质
- 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。

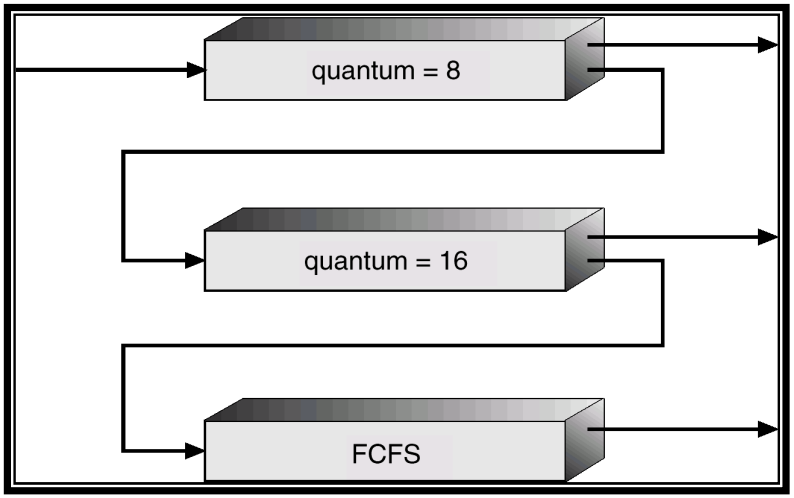
多级队列算法MFQ

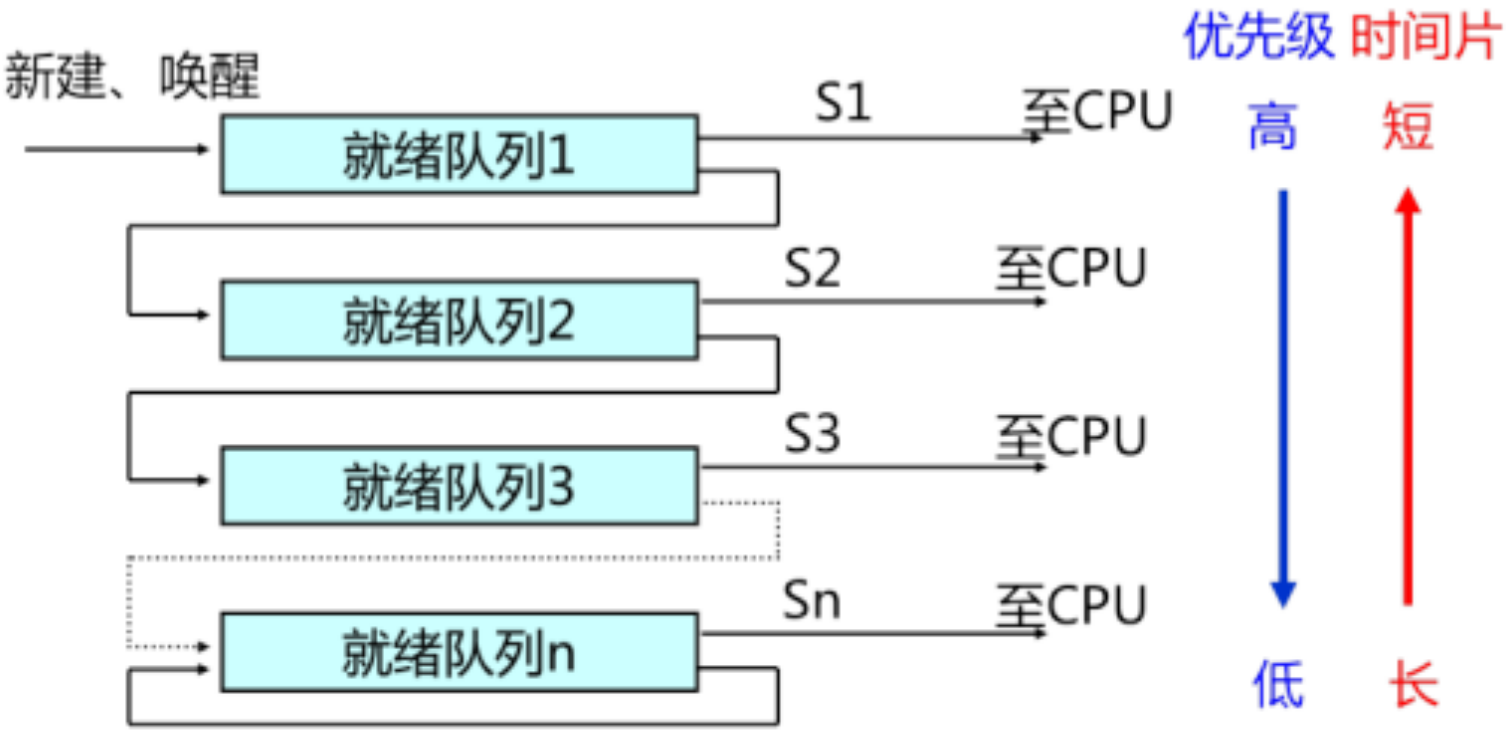
- 方案：引入多级优先队列，队列优先级越高则时间片越短；优先级越低则时间片越长；进程在某个队列运行完时间片后，会被移入下一级队列。队列内部FCFS。队列间优先级调度。
  - 动态判断进程的类型，并调整优先级
  - 交互进程和I/O进程能够及时响应
  - CPU密集型进程能够逐步获得较大时间片，减少切换

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FCFS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FCFS算法调度；如此下去，降低到最后的队列，则按“时间片轮转”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。

多级反馈队列算法





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。
- 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；

彩票调度 lottery scheduling

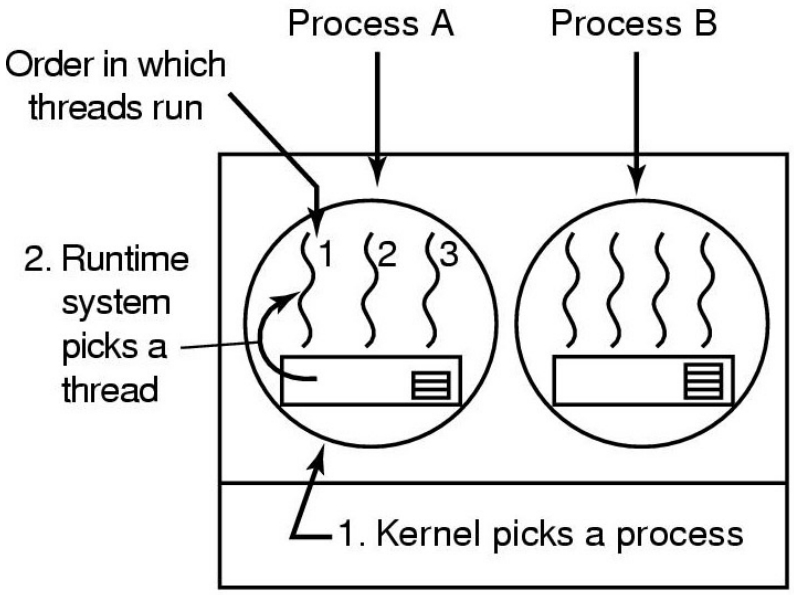
- 将CPU资源的使用权发放为彩票给进程，每次进行随机抽奖，中奖的进程得到资源。
  - 一个系统可以1秒抽50次奖，每次抽奖的幸运进程可以执行20ms
- 通过设定一个进程得到彩票数的比例，可以控制其得到系统资源的比例
  - 比优先级调度更加容易控制
  - 拥有f%比例的进程，会获得f%的系统资源

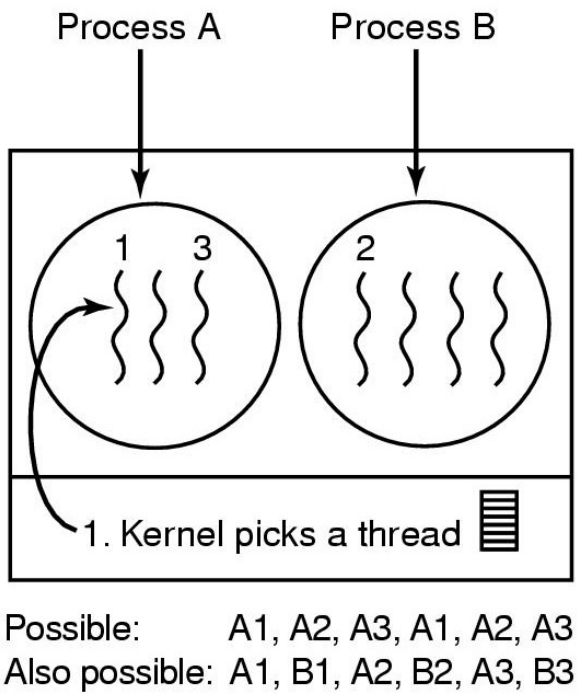
彩票调度 lottery scheduling

- 具有很好地响应性
  - 一旦进程被分配彩票，就能够在下轮抽奖有相应的获奖机会
- 协作的进程可以交换彩票
  - 客户端请求服务器进程，将所有彩票交给服务器进程
- 能够准确定义几个进程获取资源的比例
  - 视频服务器的视频生产进程，10/20/25不同的帧率
  - 通过设置10/20/25张彩票，控制CPU分配比率

线程调度

- 用户级线程：内核调度进程，进程内部调度线程。
- 内核级线程：内核直接调度线程
- 50ms时间片，每次5ms的CPU线程
- 左边用户级线程，右边内核级线程。

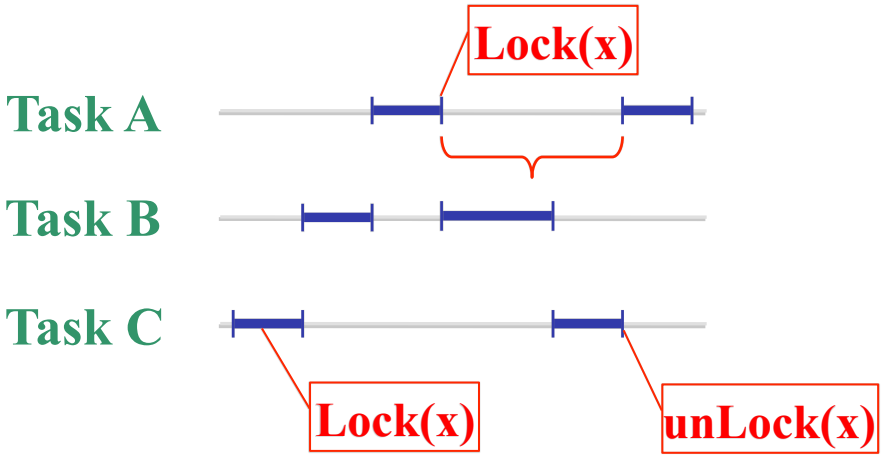




优先级倒置

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。

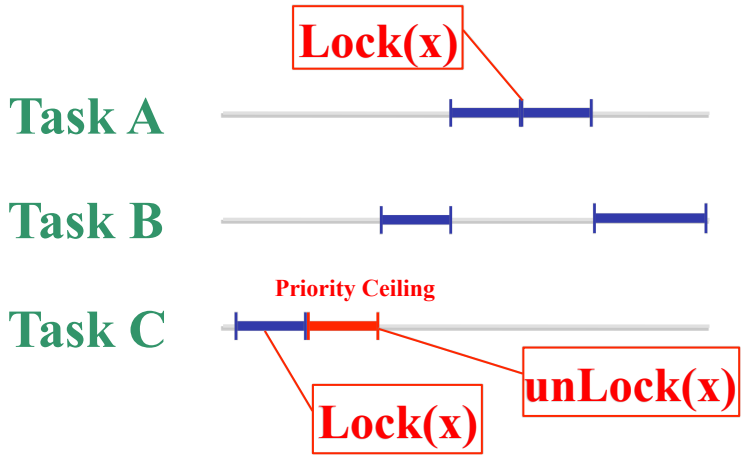


Task A 和 Task C 共享同一个临界资源，高优先级进程 Task A 因低优先进程 Task C 被阻塞，又因为低优先进程 Task B 的存在延长了被阻塞的时间。

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）。

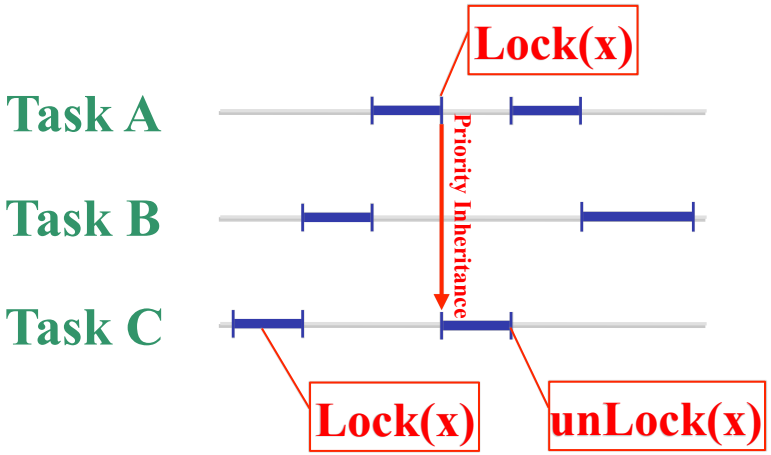


如果系统中的临界区都较短且不多，该方法是可行的。反之，如果 **Task C** 临界区非常长，则高优先级进程 **Task A** 仍会等待很长的时间，其效果无法令人满意。

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

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），而软实时倒是可以偶尔不满足（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

问题描述：

- 假设一任务集S={$t_1,t_2,t_3,\ldots,t_n$}，周期分别是$T_1,T_2,\ldots,T_n$，执行时间为$c_1,c_2,\ldots,c_n$，截至周期(deadline)为$D_1,D_2,\ldots,D_n$，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**，$\sum_{i=1}^{n} \frac{C_i}{T_i} \le 1$

C_i是任务i的执行时间，P_i是任务i的出现周期，m是任务的个数

静态表调度算法

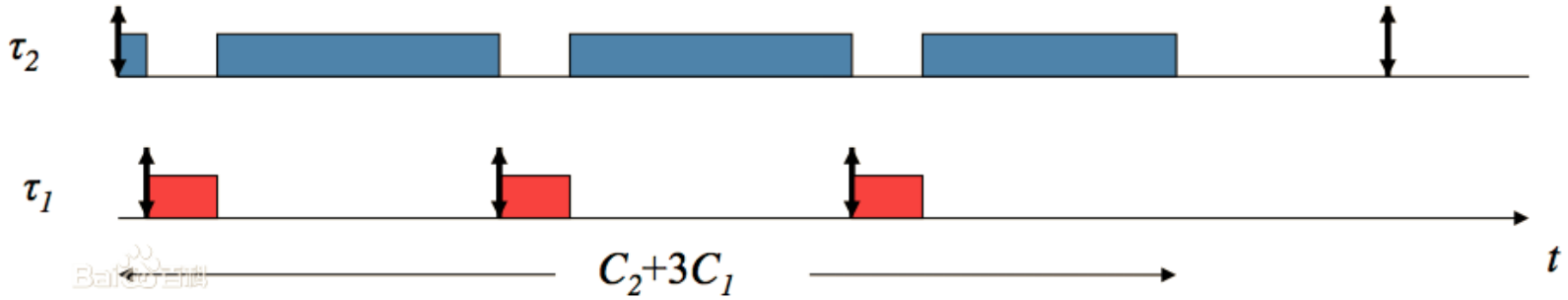
- 通过对所有周期性任务的分析预测（到达时间、运行时间、结束时间、任务间的优先关系），事先确定一个固定的调度方案。
- 特点：
  - 无任何动态计算，按固定方案进行，开销最小；
  - 无灵活性，只适用于完全固定的任务场景。

单调速率调度RMS

- RMS是单处理器下的最优静态调度算法。
- 1973年Liu和Layland首次提出了RMS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。

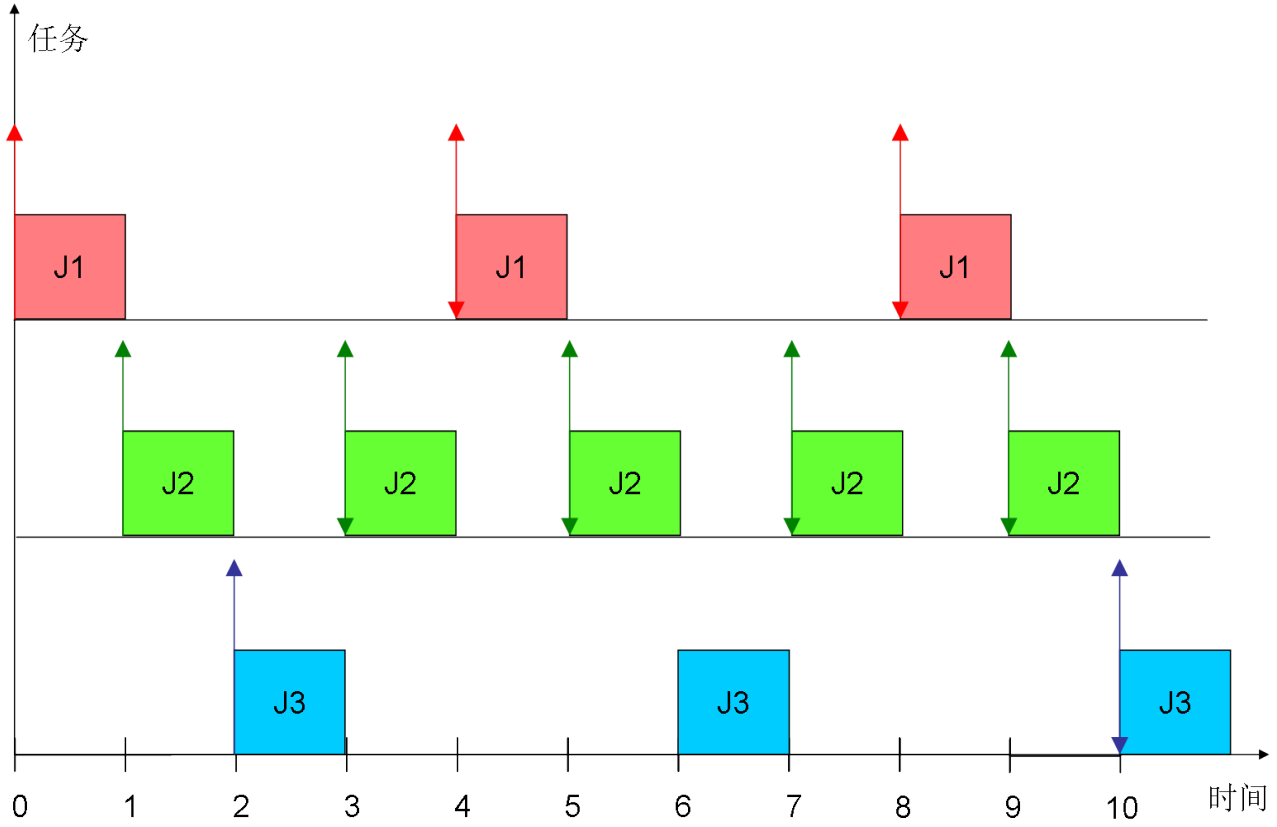
单调速率调度RMS

- 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

任务Ji	起始时间Si	执行时间Ci	周期Pi	截止时间Di
J1	0	1	4	4
J2	1	1	2	2
J3	2	2	8	8

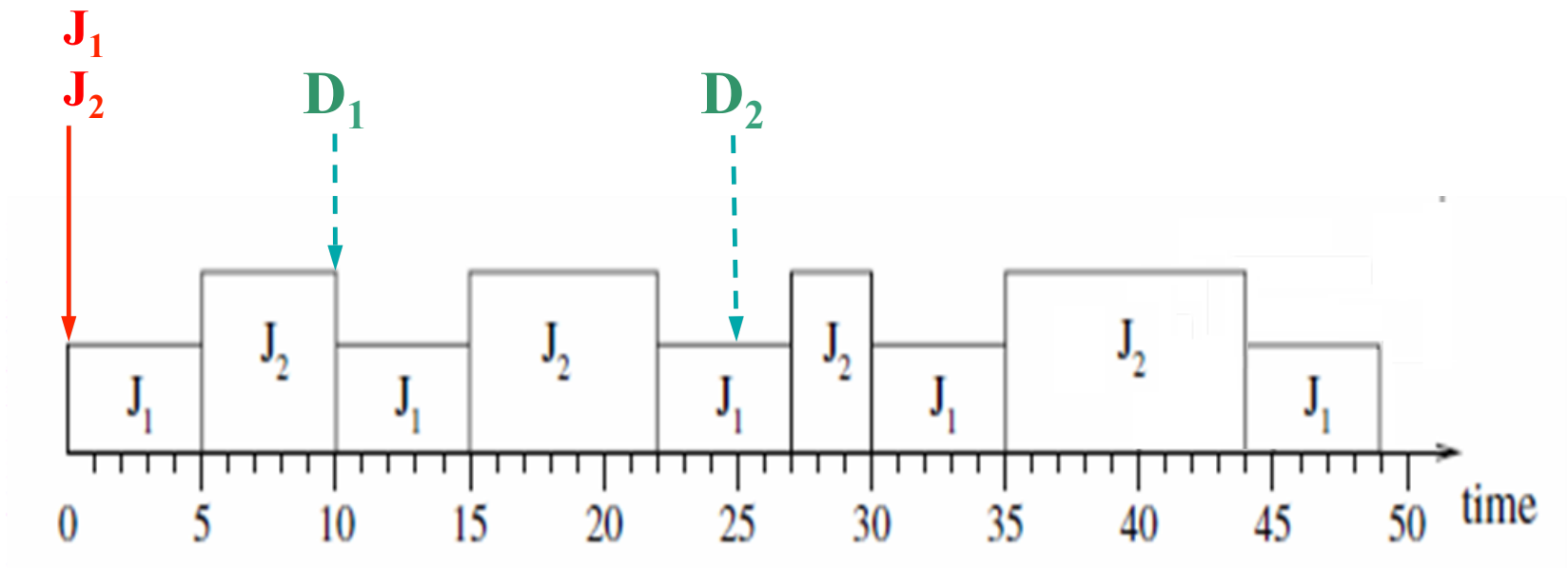


最早截止期优先EDF

- 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
- 组织了一个优先级队列，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
- 调度时机：任务执行时间到、新任务到来（按照周期）

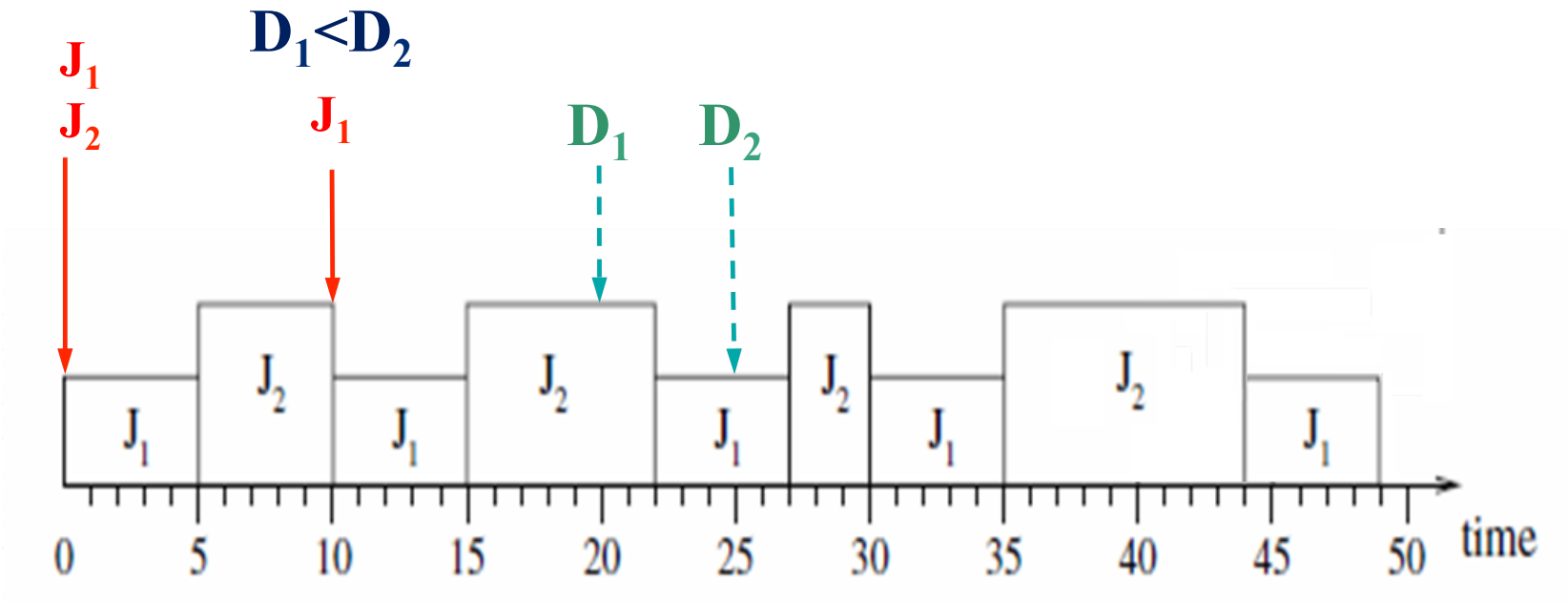
任务Ji	起始时间Si	执行时间Ci	周期Pi	截止时间Di
J1	0	5	10	10
J2	0	12	25	25

$D_1<D_2$



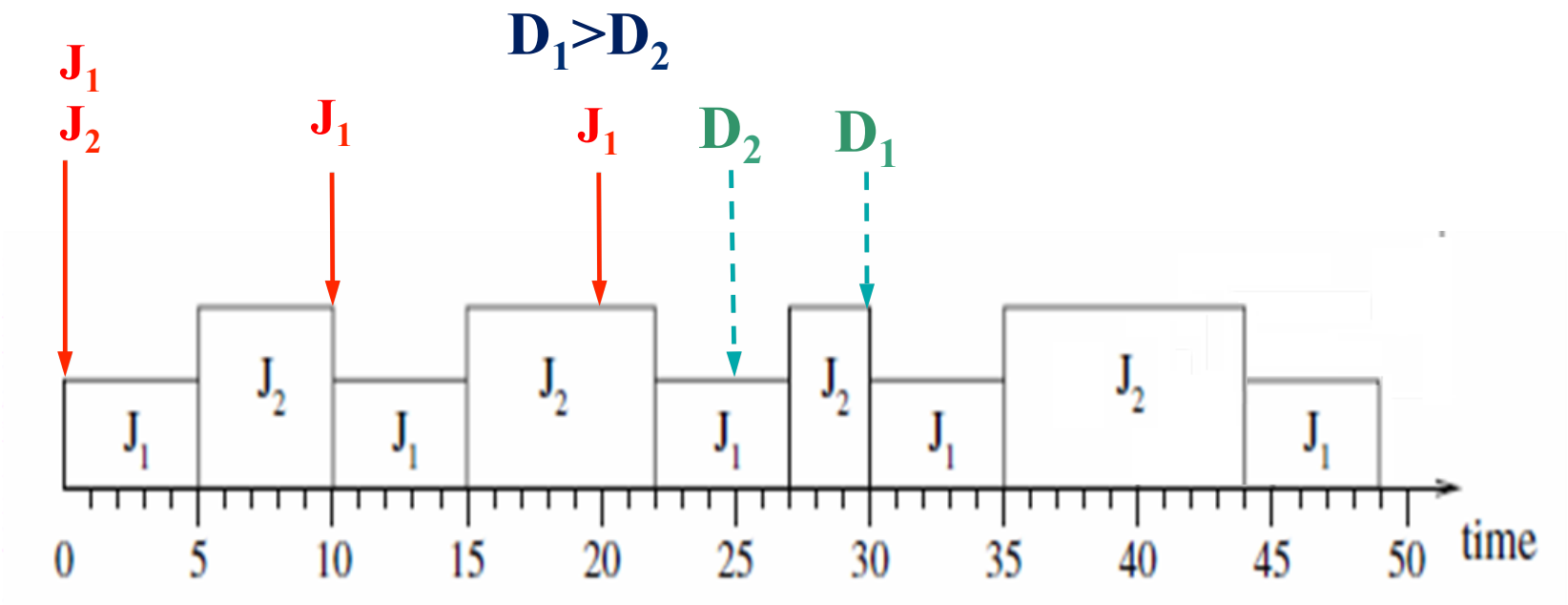
在0时刻，由于$D_1<D_2$选择J_1运行，之后J_2运行。

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



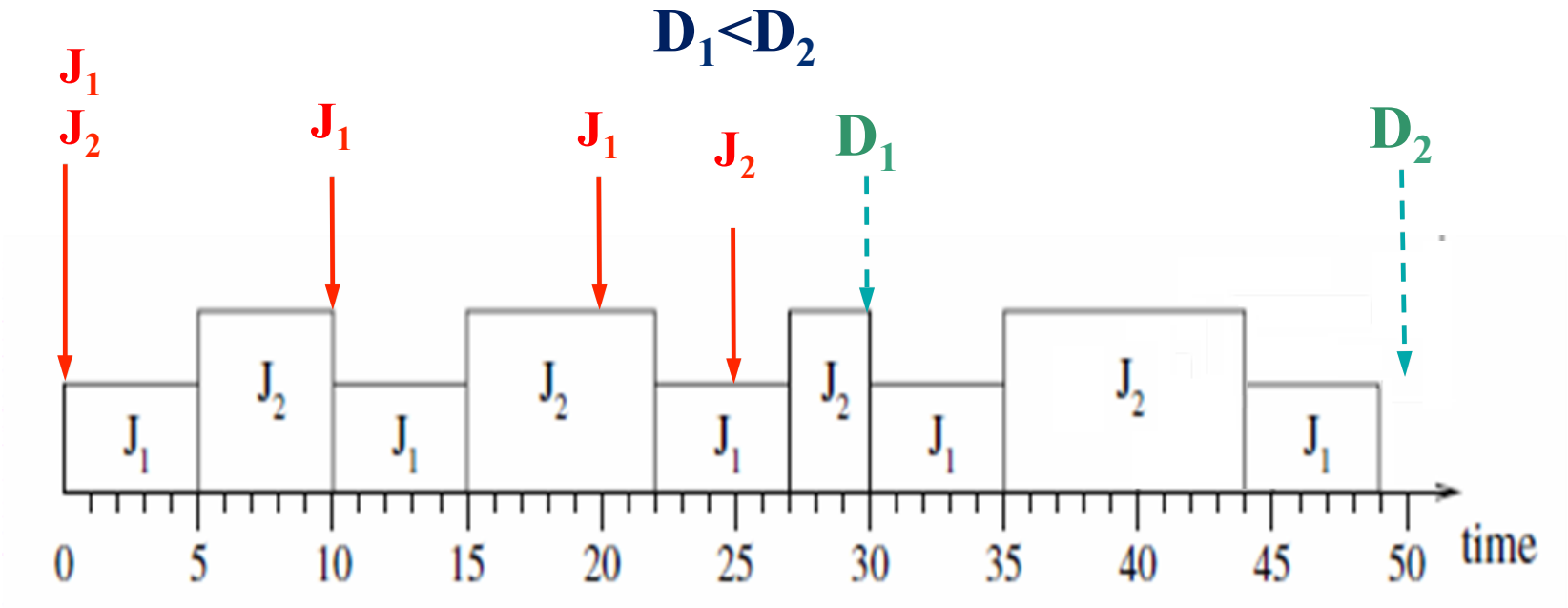
在时刻10，J_1的新周期开始，切换到J_1运行，之后J_2运行。

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



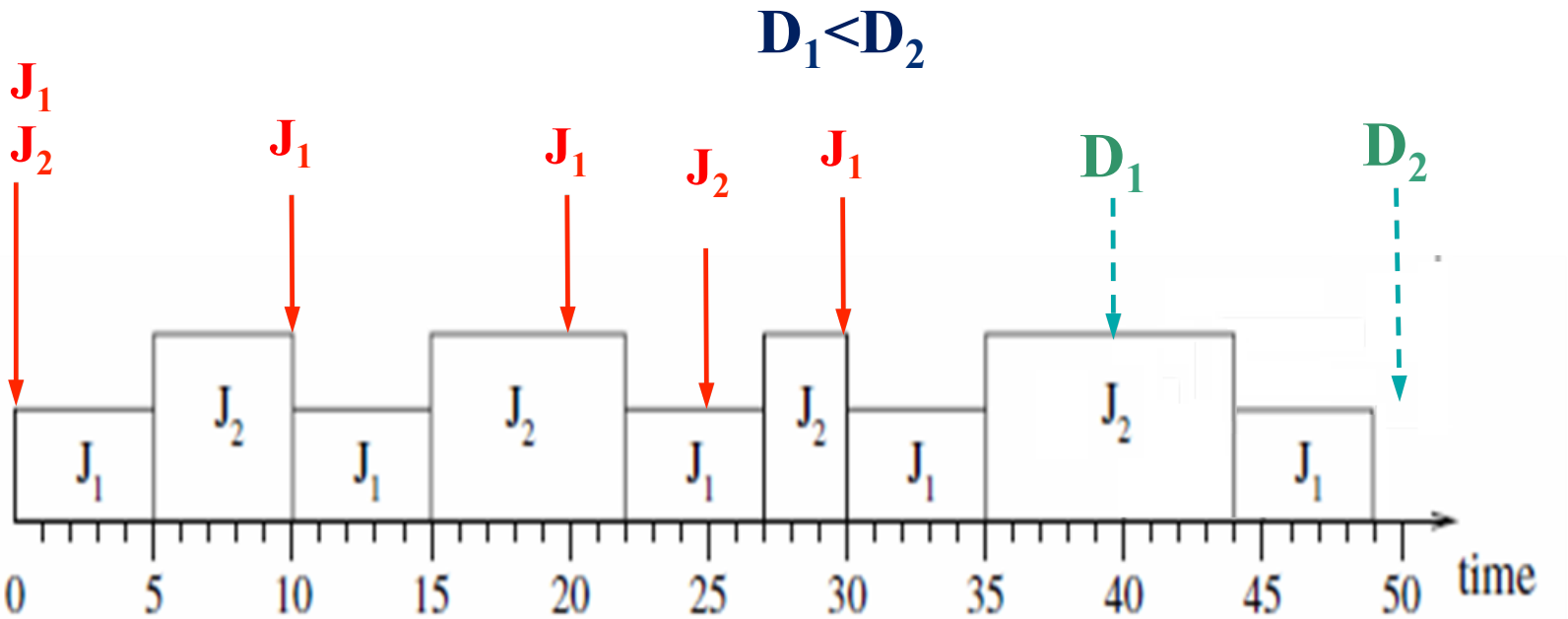
在时刻20，J_1新周期开始，但由于$D_1>D_2$，保持J_2运行，之后J_1运行。

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



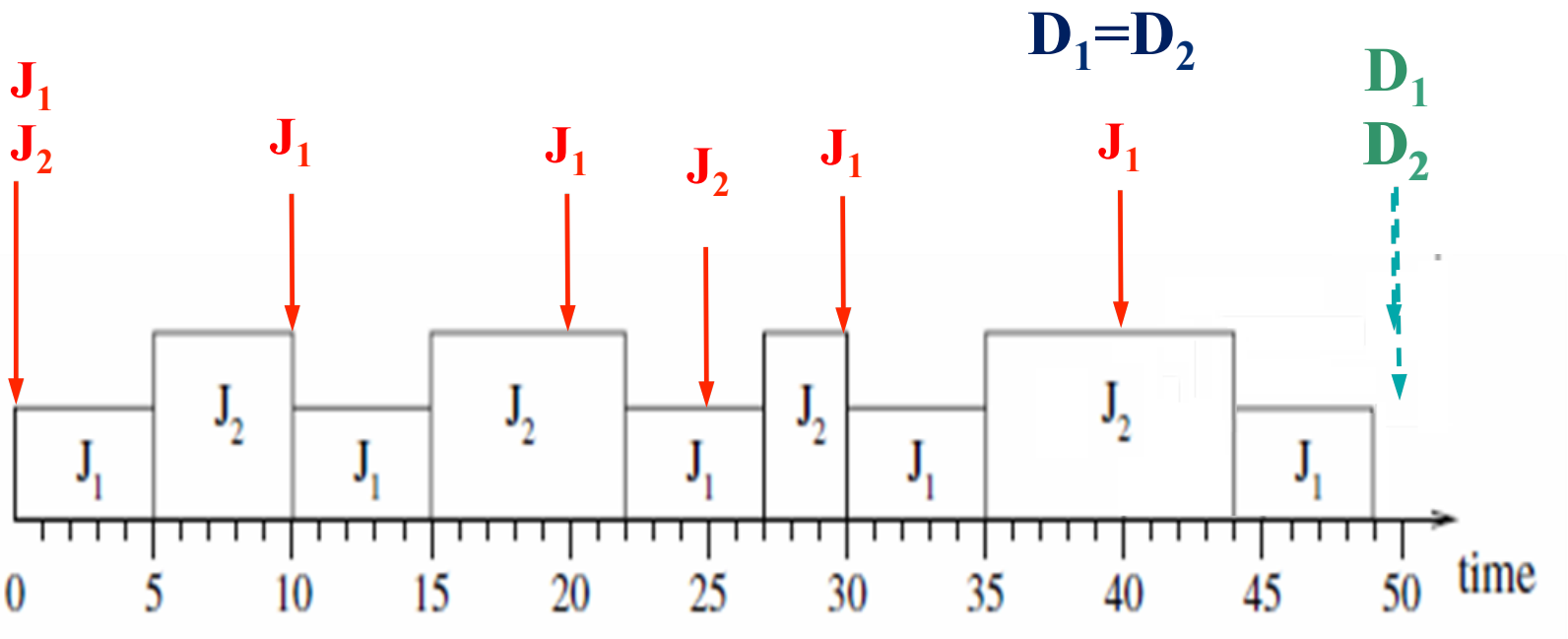
在时刻25，J_2新周期开始，但由于$D_1<D_2$，保持J_1运行，之后J_2运行。

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。

内容提要

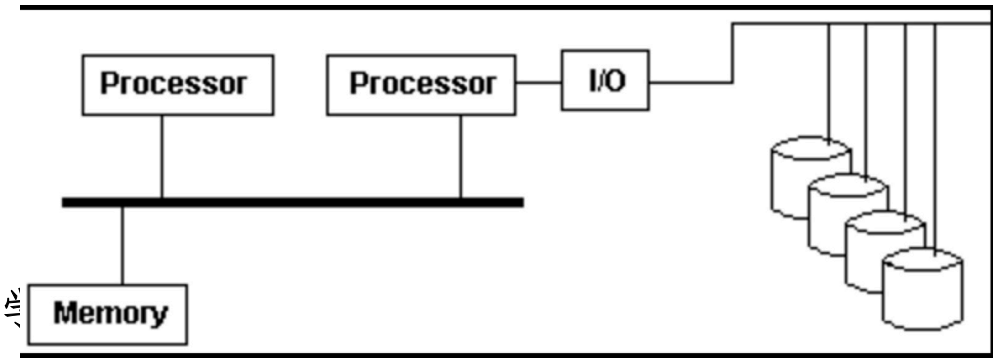
- 基本概念
- 设计调度算法要考虑的问题
- 批处理系统的调度算法
- 交互式系统的调度算法
- 实时系统的调度算法
- 多处理机调度

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，如一个处理机执行OS的系统功能，另一个处理I/O。



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。下面所述静态和动态调度都是集中控制，而自调度是分散控制。

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)：各个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。

对称式多处理系统(SMP)

- 自调度(self-scheduling)：各个CPU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。需要对就绪队列的数据结构进行互斥访问控制。是最常用的算法，实现时易于移植，采用单处理机的调度技术。

自调度（Self Scheduling）

自调度：各个处理机自行在就绪队列中取任务。

- 优点：不需要专门的处理机从事任务分派工作。
- 缺点：当处理机个数较多（如十几个或上百个）时，对就绪队列的访问可能成为系统的瓶颈。
- 低效问题?

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。

Linux调度算法的发展历史

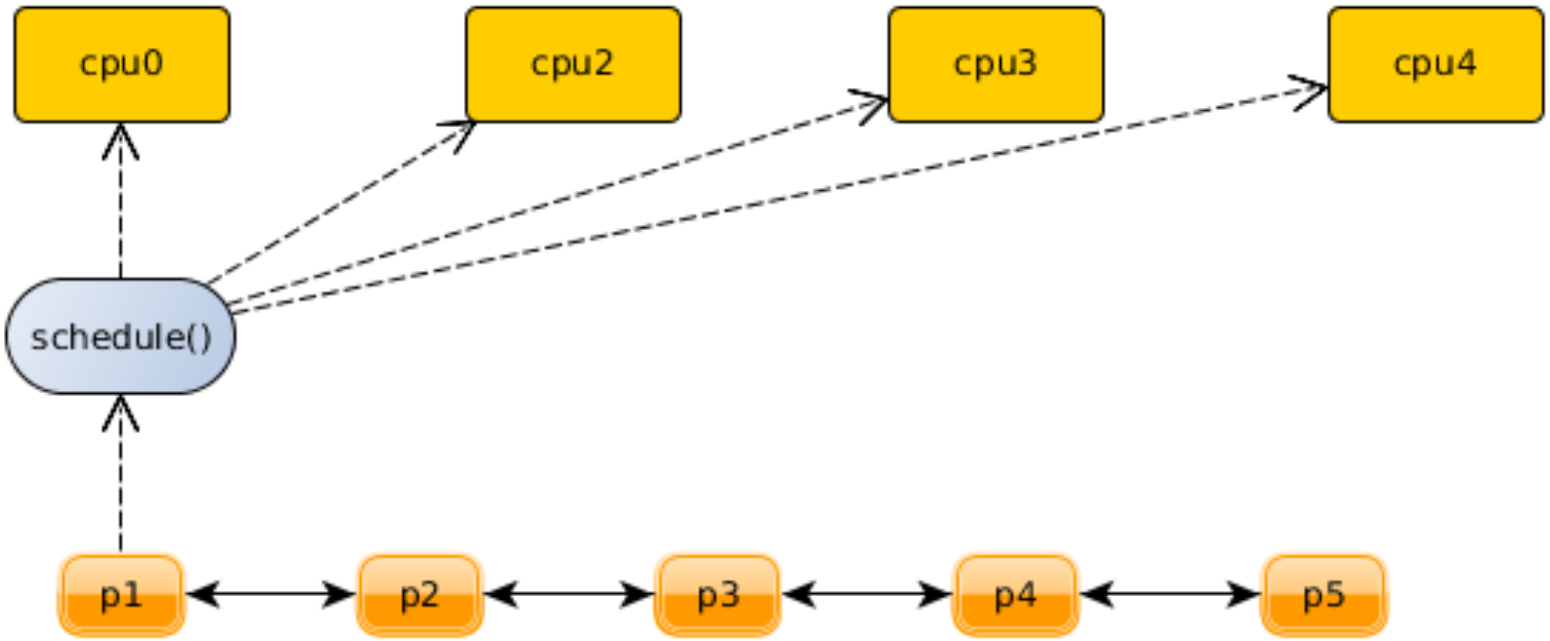
参考网址:
http://abcdxyzk.github.io/blog/2015/01/22/kernel-sched-n1/

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：
  - SCHED_FIFO实时进程使用的FIFO策略，进程会一直占用cpu 除非其自动放弃cpu。
  - SCHED_RR实时进程的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。
  - SHED_OTHER普通进程基于优先级的的时间片轮转调度。

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

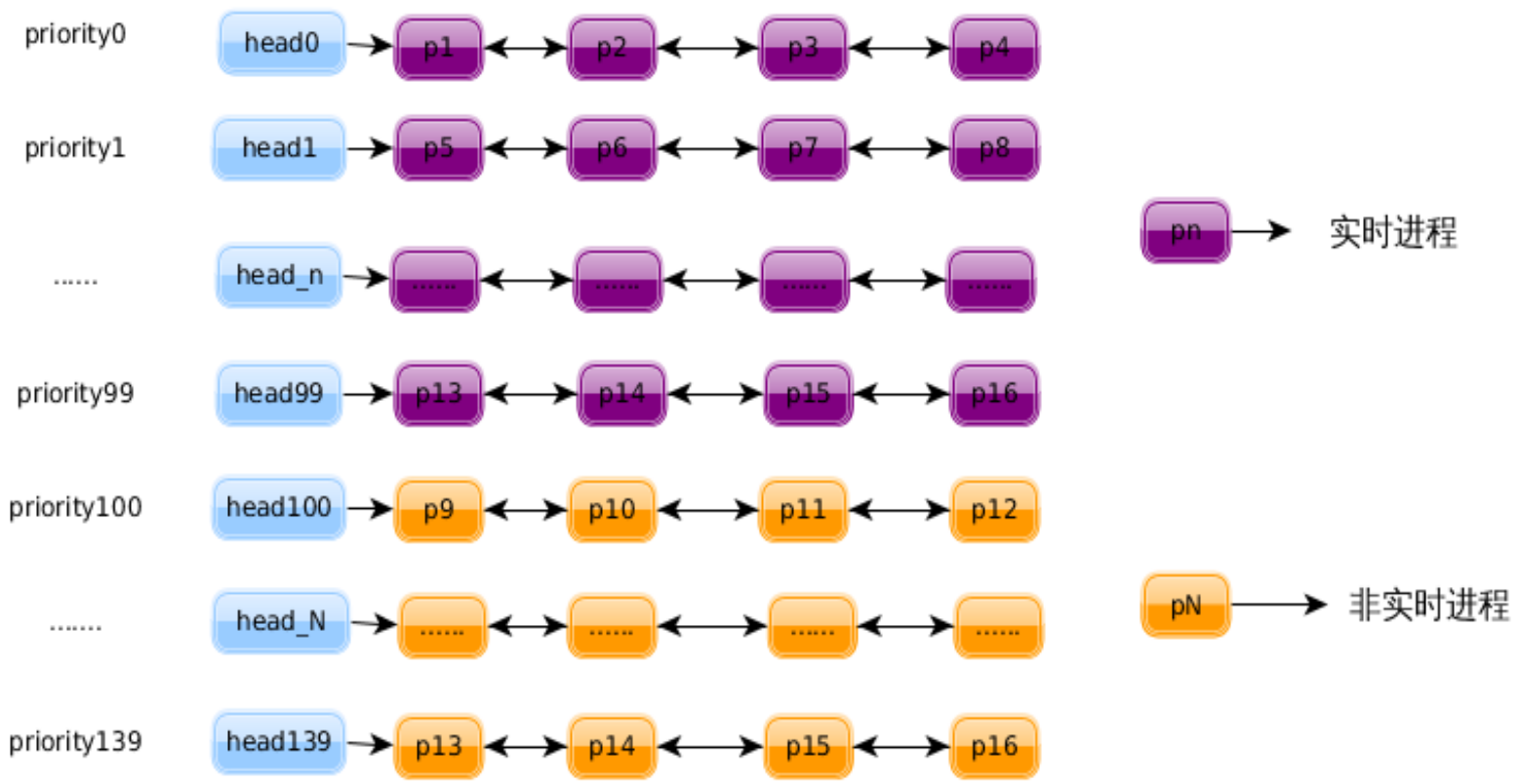
- O(1)调度器每个cpu维护一个自己的运行队列，每个运行队列有自己的active队列与expried队列。
- 当进程的时间片用完后就会被放入expired队列中。当active队列中所有进程的时间片都用完，进程执行完毕后，交换active和expried。这样expried队列就成为了active队列。这样做只需要指针的交换而已。
- 当调度程序要找出下一个要运行的进程时，要根据位图宏来找出优先级最高的且有就绪进程的队列。这样的数据组织下，2.6的调度器的时间复杂度由原来2.4的O(n)提高到O(1)。对smp环境具有较好的伸缩性。

linux 2.6 O(1)调度器

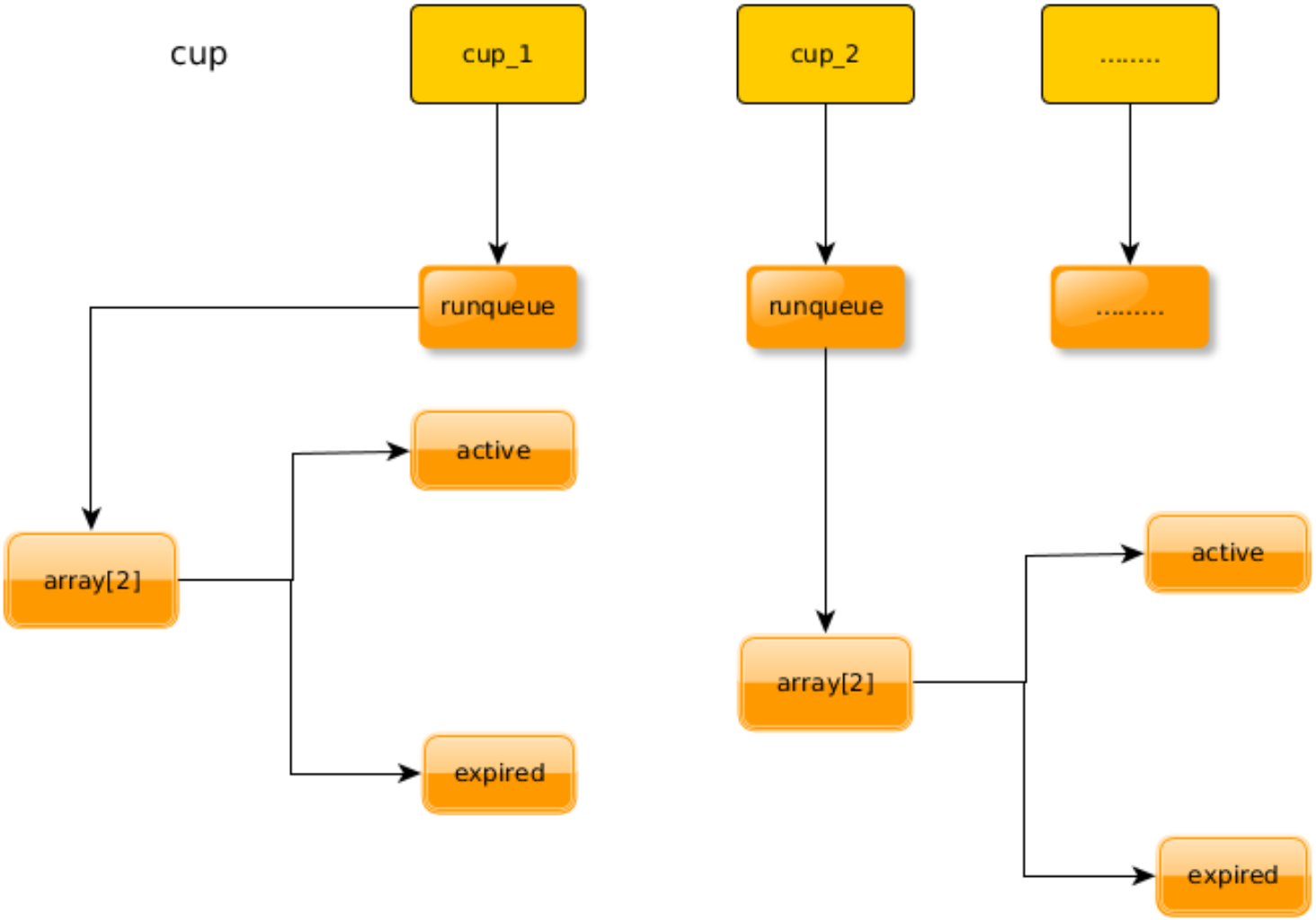
- 调度有140个优先级级别，由0～139, 0～99 为实时优先级，而100～139为非实时优先级。
- 调度策略：与2.4相同。

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。

CFS调度算法

- 自内核版本2.6.23开始，Linux就一直使用CFS算法。
- 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调度算法

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调度算法

- 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