第5章 进程调度

5.3 多处理器调度

5.3 多处理器调度

- 多处理器系统的类型
  - 非对称多处理（asymmetric multiprocessing）
    - 所有的调度、i/o和其他系统活动处理都在一个处理器上（master），其他处理器负责执行用户代码
    - 优点：只有master处理器访问系统数据（如调度队列），减少数据共享
    - 缺点：master处理器成为瓶颈
  - 对称多处理（symmetric multiprocessing,SMP）
    - 每个处理器运行自己的调度程序
    - 可以共用一个调度队列或者各处理器拥有自己的调度队列

多处理器调度

- 多处理器调度不仅要决定选择哪一个进程执行，还需要决定在哪一个CPU上执行
  - 要考虑处理器亲和性和负载平衡的问题

处理器亲和性

- 处理器亲和性(CPU Affinity)
  - 尽量使进程总是在同一个CPU上执行
- 考虑进程在多个CPU之间迁移时的开销
  - 高速缓存失效、TLB失效
- 两种亲和性：
  - soft affinity：设法保证，不强制
  - hard affinity：强制不允许进程移走

负载平衡

- 负载平衡(Load Balancing)
  - 使工作负载在SMP系统中的所有处理器上均匀分布。
- 实现负载平衡的两种方法
  - Push migration：特殊的任务定期进行检查并将繁忙处理器的任务推送到空闲处理器
  - Pull migration：由空闲的处理器主动拉取繁忙处理器的任务

- 亲和性和负载平衡是有矛盾的
  - 亲和和迁移

多处理器调度的设计要点

- 设计要点之一：如何把处理器分配给进程：
  - 静态分配策略：每个处理器一个就绪队列
    - 调度开销小，但忙闲不均，需要考虑负载平衡
  - 动态分配策略：处理器共享一个就绪队列
    - 容易形成瓶颈；对称处理器还要确保不能选中同一个进程
- 设计要点之二：是否要在单个处理器上支持多道程序设计。
  - 多处理器下，多线程与单处理器的关系
- 设计要点之三：如何指派进程。
  - 复杂的低级调度算法不会提高有效性

多处理器调度算法(1)

1)负载共享调度算法(Load Sharing)

- 基本思想：
  - 进程并不指派到某一个特定处理器，系统维护一个全局性就绪线程队列，当一个处理机空闲时，就选择一个就绪线程占有处理机运行。
- 优点：
  - 保证每个处理机都不是空闲的；无需集中调度（操作系统的调度程序可以运行在不同的处理器上）
- 缺点：
  - 就绪队列必须互斥访问，成为性能瓶颈
  - 被剥夺线程很难在原处理机上运行，高速缓存的恢复影响性能
  - 所有线程在一个公共池中，一个进程的所有线程未必全能获得处理器，若发生同步频繁，则切换进程代价高

多处理器调度算法(2)

2)群调度算法(gang scheduling)

- 基本思想：把**一组相关线程**在同一时间一次性**调度到一组处理器**上运行。
- 优点：
  - 当紧密相关的线程同时执行时，同步造成的等待将减少，进程切换也相应减少，系统性能得到提高。
  - 由于一次性同时调度一组处理器，调度的代价也将减少。
- 问题：对处理器调度分配要求不均衡
  - 如4个cpu，A进程：4个线程，B：1个线程，如果按进程平均分配，则B占优，故需要按比例分配

多处理器调度算法(3)

3)专用处理器调度算法(dedicated processor assignment)

- 基本思想：给**一个应用**指派**一组处理器**，一旦一个应用被调度，它的**每个线程被分配一个处理器**并一直占有处理器运行直到整个应用运行结束。
- 这一算法是群调度的极端情况，处理器**不采用多道程序设计**，即该应用的一个线程阻塞后，线程对应的处理器不会被调度给其他线程，而处于空闲状态。
- 该算法追求大规模处理机的高度并行化，减少低级调度和切换的代价，并不关心单个处理机的效率。

多处理器调度算法(4)

4)动态调度算法

- 基本思想：由操作系统和应用进程共同完成调度。
  - 系统分配处理机给进程。进程负责判断把分配给它的处理器分配给哪些线程。
  - 当发生处理器请求时，如有空闲处理器可用，则满足请求。否则一次性收回多个处理器进行分配；调度时采用先来先服务原则。

第5章 进程调度

5.4 调度实例

5.4 调度实例

1. UNIX
2. Solaris
3. Linux
4. Windows

1. UNIX的进程调度

- UNIX的进程调度采用**多级反馈队列**调度算法，系统设置了多个就绪队列。进程调度程序执行时：
  - 核心首先从处于“内存就绪”或“被剥夺”状态的进程中选择一个**优先级最高**的进程；
  - 若系统中同时有多个进程都具有最高优先级，则将选择其中处于**就绪状态最久**的进程，将它从所在队列移出，恢复其上下文，使之执行；
  - 仅当最高优先级队列中没有进程时，才从次高优先级队列中找出其队首进程，令它执行一个时间片后，又剥夺该进程的执行；
  - 然后，再从优先级最高的队列中取出下一个就绪进程投入运行。
  - 同时UNIX SVR4添加了**静态优先级**的**抢占式**调度。

进程优先级的分类

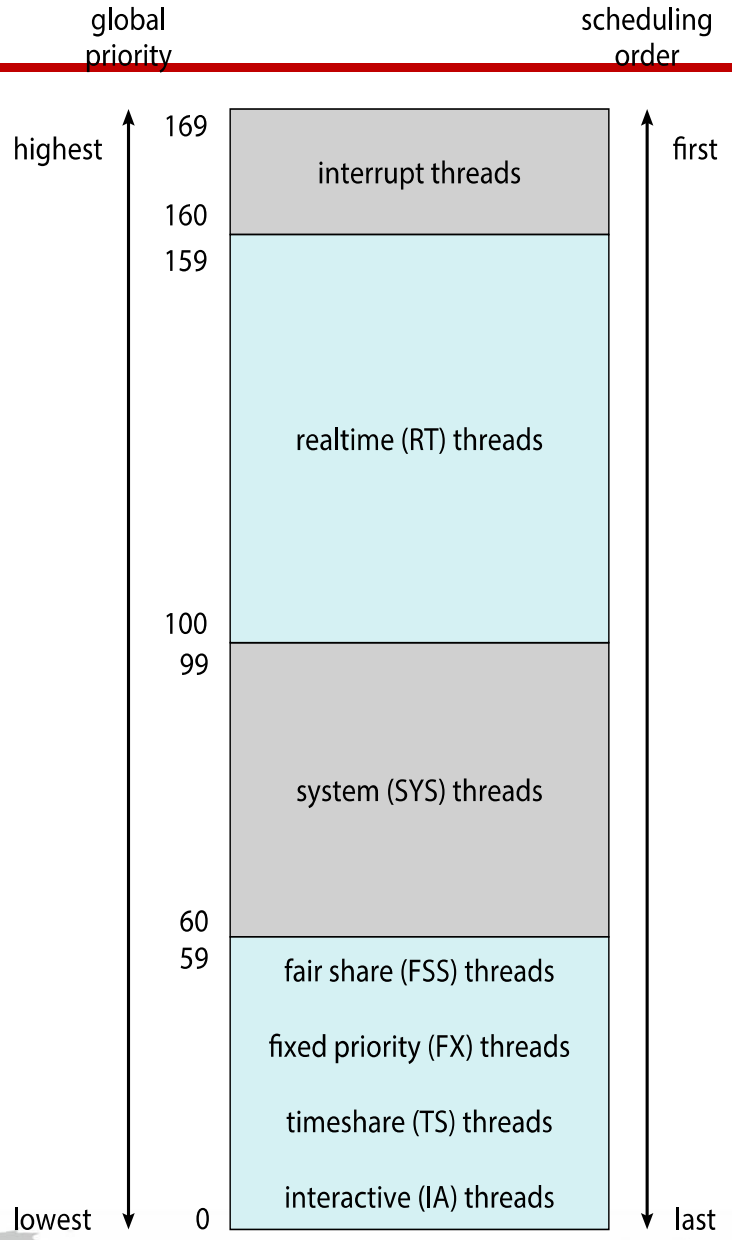
- 在UNIX系统中，进程的优先级分为三类：
  - 实时优先级层次（优先数159-100）：这一层次的进程先于其他层次进程运行，能利用抢占点抢占内核进程和用户进程。
  - 内核优先级层次（优先数99-60）：这一层次的进程先于分时优先级层次进程但迟于实时优先级层次进程运行。
  - 分时优先级层次（优先数59-0）：最低优先级层次，用于非实时的用户应用。

2. Solaris

- 基于优先级的线程调度
- 六个类别优先级
  - 实时(RT) （优先数159-100）
  - 系统 (SYS) （优先数99-60）
  - 分时(默认) (TS) （优先数59-0）
  - 交互 (IA) （优先数59-0）
  - 公平分享 (FSS) （优先数59-0）
  - 固定优先级 (FP) （优先数59-0）
- 每个类别都有自己的调度算法
- 一次只给一个线程分配某一个类别
- 默认分时，采用多级反馈队列
  - 优先级和时间片成反比：优先级越高，时间片越短
  - 交互进程有更好的优先级和响应时间
  - CPU约束程序有更好的吞吐量

Solaris调度原则

- 调度器将面向类别的优先级转换为全局线程的优先级
  - 从中选取全局最高优先级的线程进行执行
  - 线程运行，直到(1)阻塞，(2)时间片用尽，(3)高优先级线程强占
  - 如果有多个线程具有相同优先级，则采用RR算法调度



3. Linux 调度算法

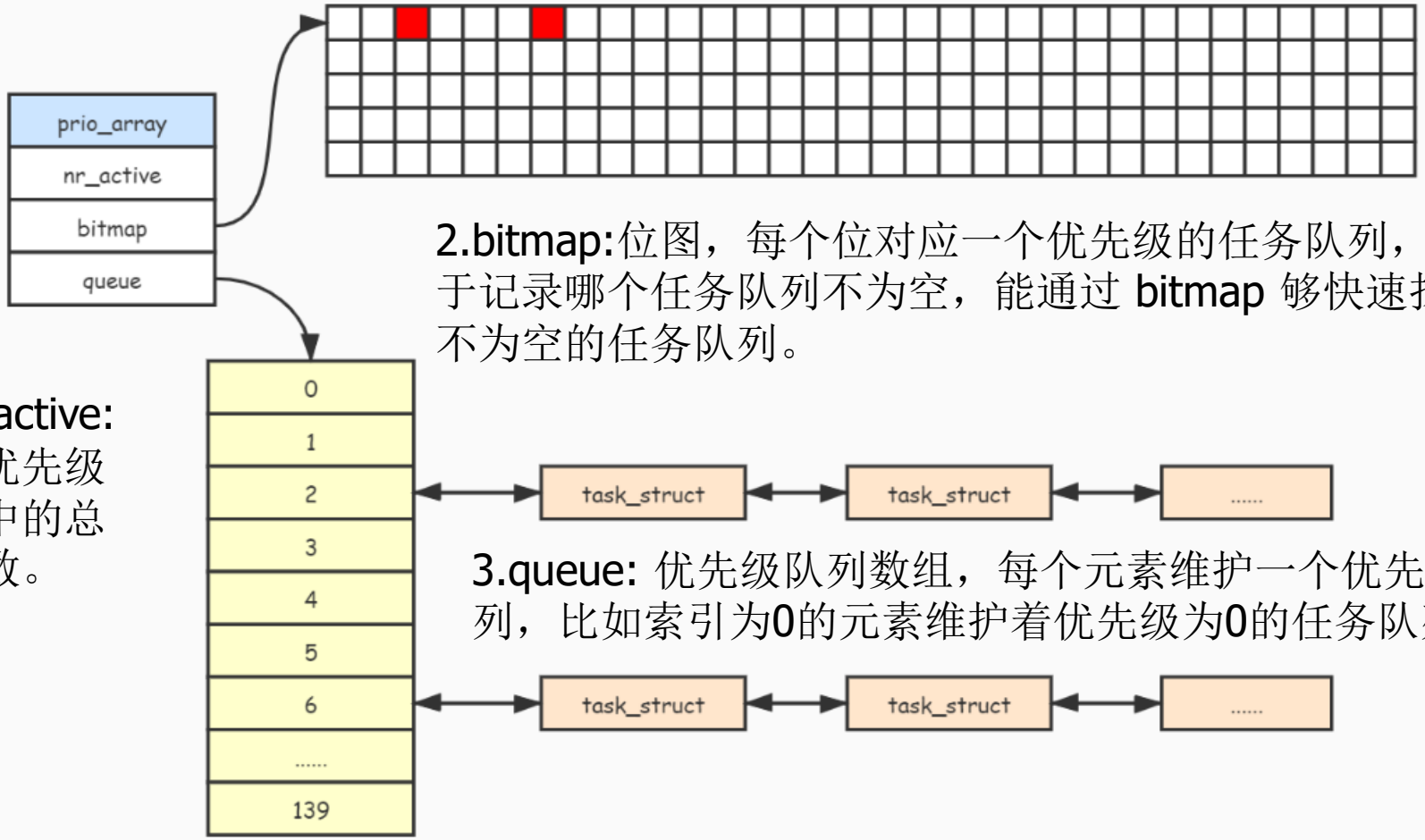
- 2.5版本前，采用标准的Unix调度算法，效率不佳，不支持SMP
- 2.5版本，O(1)调度法：
  - 基于抢占+优先级
  - 具有140个优先级，分成2个范围：实时和普通
    - 实时：0 — 99 （静态优先级）
    - 普通：100 — 139 （动态优先级，时间片用完后重新计算）
  - 时间片分配：200ms — 10ms
    - 进程的时间片用完后会重新计算

O(1)调度法

- O(1)调度法：
  - 当任务在其时间片中具有剩余时间，认为可运行，active
  - 当任务耗尽时间片，则不再执行，expired
  - 为支持SMP，每个CPU维护数据结构
    - 两个优先级队列: active，expired
    - 任务列表：根据优先级进行索引
    - 对于每个cpu，调度程序从active队列选取优先级最高的任务，在该cpu进行执行
    - 当active队列为空，两个队列进行交换
  - 支持SMP、调度开销小，但是需要对交互式任务进行额外的判定和处理，有时候交互任务响应时间不佳

O(1)调度法

- 每个优先级的任务由一个队列来维护。prio_array结构就是用来维护这些任务队列



2.bitmap:位图，每个位对应一个优先级的任务队列，用于记录哪个任务队列不为空，能通过 bitmap 够快速找到不为空的任务队列。

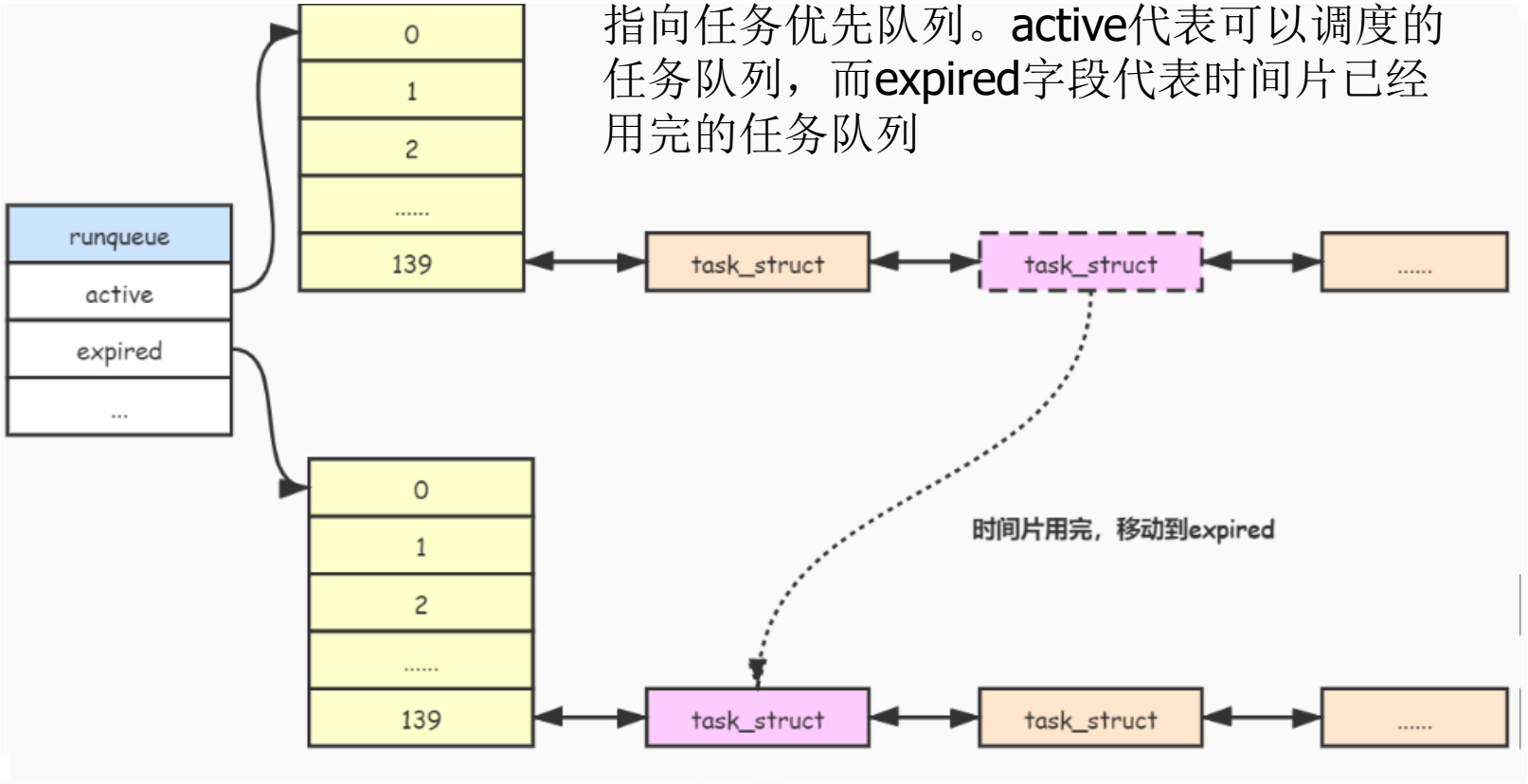
1.nr_active: 所有优先级队列中的总任务数。

3.queue: 优先级队列数组，每个元素维护一个优先级队列，比如索引为0的元素维护着优先级为0的任务队列。

O(1)调度法

- 每个CPU都需要维护一个runqueue结构，runqueue结构主要维护任务调度相关的信息，比如优先队列、调度次数、CPU负载信息等

active和expired 字段的类型为prio_array，指向任务优先队列。active代表可以调度的任务队列，而expired字段代表时间片已经用完的任务队列



O(1)调度法

- 实时进程调度：

实时进程分为FIFO(先进先出)和RR(时间轮询)两种，其调度算法比较简单，如下:

- 1.先进先出的实时进程调度:如果调度器在执行某个先进先出的实时进程，那么调度器会一直运行这个进程，直至其主动放弃运行权(退出进程或者sleep等)。
- 2.时间轮询的实时进程调度:如果调度器在执行某个时间轮询的实时进程，那么调度器会判断当前进程的时间片是否用完，如果用完的话，那么重新分配时间片给它，并且重新放置回active队列中，然后调度到其他同优先级或者优先级更高的实时进程进行运行。

O(1)调度法

- 普通进程调度：

每个进程都要一个动态优先级和静态优先级，静态优先级不会变化(进程创建时被设置)，而动态优先级会随着进程的睡眠时间而发生变化。动态优先级可以通过以下公式进行计算:

```
动态优先级 = max(100, min(静态优先级 - bonus + 5), 139))
```

上面公式的bonus(奖励或惩罚)是通过进程的睡眠时间计算出来，进程的睡眠时间越大，bonus的值就越大，那么动态优先级就越高。

当一个普通进程被添加到运行队列时，会先计算其动态优先级，然后按照动态优先级的值来添加到对应优先级的队列中。而调度器调度进程时，会先选择优先级最高的任务队列中的进程进行调度运行。

O(1)调度法

- 运行时间片计算

当进程的时间用完后，就需要重新进行计算。进程的运行时间片与静态优先级有关，可以通过以下公式进行计算:

静态优先级 < 120，运行时间片 = max((140-静态优先级)*20, MIN_TIMESLICE)

静态优先级 >= 120，运行时间片 = max((140-静态优先级)*5, MIN_TIMESLICE)

Linux 2.6.23的算法

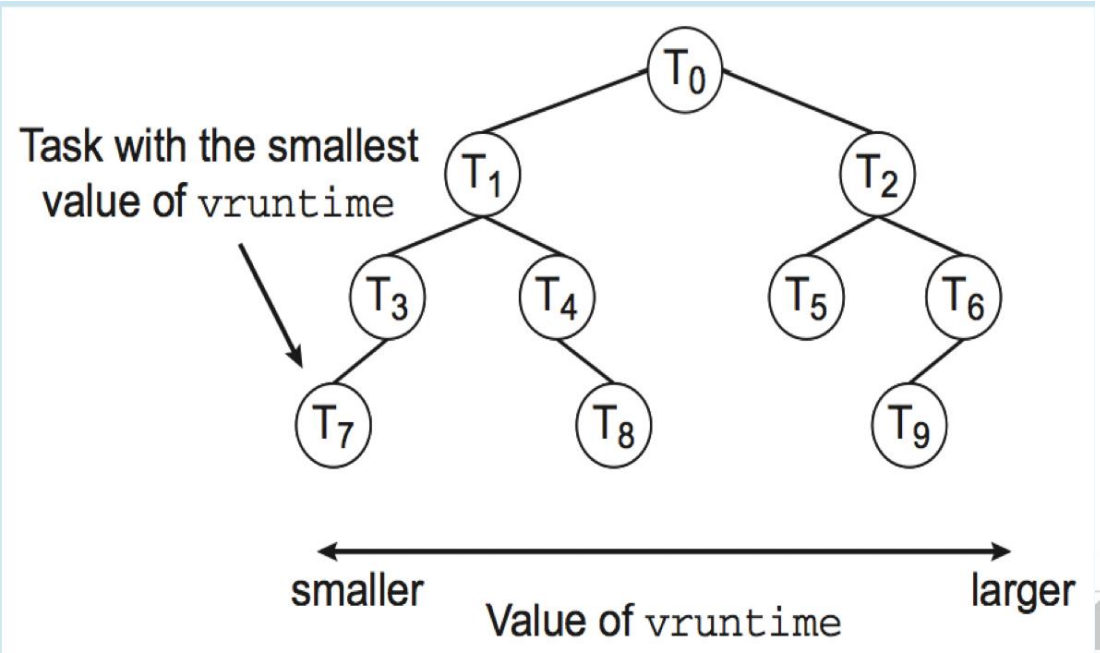
- 采用完全公平调度算法 CFS（Completely Fair Scheduler）
- 调度基于调度类
  - 每个类别，有一个具体的优先级
  - 对于不同类别，采用不同调度算法
  - 选取原则：在最高调度类别中选择最高优先级的任务
  - Linux标准内核实现两个标准类：默认类+实时类
  - 处理时间分配：不用优先级对应某个时间片，而采用为每个任务分配一定比例的CPU处理时间

CFS

- CPU时间配额的问题：
  - 基于nice -20—+19，默认为0
  - 值越低，优先级越高
  - 目标延迟(target latency)：可运行任务运行一次的的时间间隔，用于量化每个类别优先级分配的CPU时间
  - 根据目标延迟，按照比例分配CPU时间
  - 随着系统活动任务的增加，目标延迟可增加
- CFS调度器，为每个任务维护虚拟运行时间变量vruntime，并选取最低vruntime的任务运行
  - 与基于优先级的衰减因子相关：低优先级具有高衰减率
  - 普通默认优先级任务服从：虚拟运行时间=实际运行时间
  - 低优先级任务的>，高优先级则<

CFS性能

- 采用红黑树数据结构(自平衡二叉树)
- 当一个任务可运行，则添加到树上，否则删去
- 将vruntime少的，加到左枝，更多的加到右枝，则最左的叶子为最小vruntime
- 搜索到最左边节点需要O(logN)操作，但为了提高效率，将其值缓存到rb_leftmost中，从而加快检索
- 有利于I/O密集型任务，因其vruntime小于CPU密集型任务，则可以实现抢占



4. Windows调度算法

- Windows NT5之后处理器调度的对象是线程,也称内核级线程调度。
- 采用基于动态优先级、抢先式调度，结合时间配额的调整
  - 就绪进程按照优先级进入相应队列
  - 系统总是选择优先级最高的就绪线程运行
  - 同一优先级的线程按照时间片轮转进行调度
  - 多处理器系统上，允许线程并行执行。线程可在任何可用处理器上运行，但也可允许线程选择亲合处理器集合

线程调度时机

- 线程调度触发事件：
  ① 线程终止
  ② 新线程创建或一个等待线程变就绪
  ③ 当一个线程从运行态转入就绪态
  ④ 当一个线程从运行态变为等待态
  ⑤ 一个线程由于调用系统服务而**改变优先级**或被系统本身改变其优先级。
  ⑥ 一个正在运行的线程**改变了它的亲合处理器集合**。

此时将选择下一个要运行的线程

Windows优先级类别

- 使用32个线程优先级，范围从0到31，分成三大类：
  - 实时优先级（优先数为31-16）：
    - 用于通信任务和实时任务。
    - 实时优先数线程的优先数不可变
    - 一旦就绪线程的实时优先数比运行线程高，它将抢占处理器运行。
  - 可变优先级（优先数为15-1）：
    - 用于交互式任务。
    - 可根据执行过程中的具体情况动态调整优先数，但不能突破15。
  - 1个系统线程优先级（0）
    - 仅用于对系统中空闲物理页面进行清零的零页线程。
- 调度程序会依据优先级从高到底选择线程，如果没有找到就执行idle thread

Windows 优先级类别

- Win32 API 定义的进程优先级
  - REALTIME_PRIORITY_CLASS
  - HIGH_PRIORITY_CLASS
  - ABOVE_NORMAL_PRIORITY_CLASS
  - NORMAL_PRIORITY_CLASS（默认）
  - BELOW_NORMAL_PRIORITY_CLASS
  - IDLE_PRIORITY_CLASS

除了REALTIME_PRIORITY_CLASS，其他都是可变的

Windows 优先级类别

- 具有给定优先级类的一个线程还具有一个相对优先级
  - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE
- 优先级类+相对优先级➔数字化的优先级
- 每个类别中，基本优先级均为NORMAL
- 对与可变优先级范围内（1-15）的线程优先级，有可能会降低或提高
  - 配额用完，要考虑是否降低优先级，但是不会低于本类的基本值
  - 从等待中被释放，调度程序会提升其优先级。

Windows 优先级

可变优先级

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

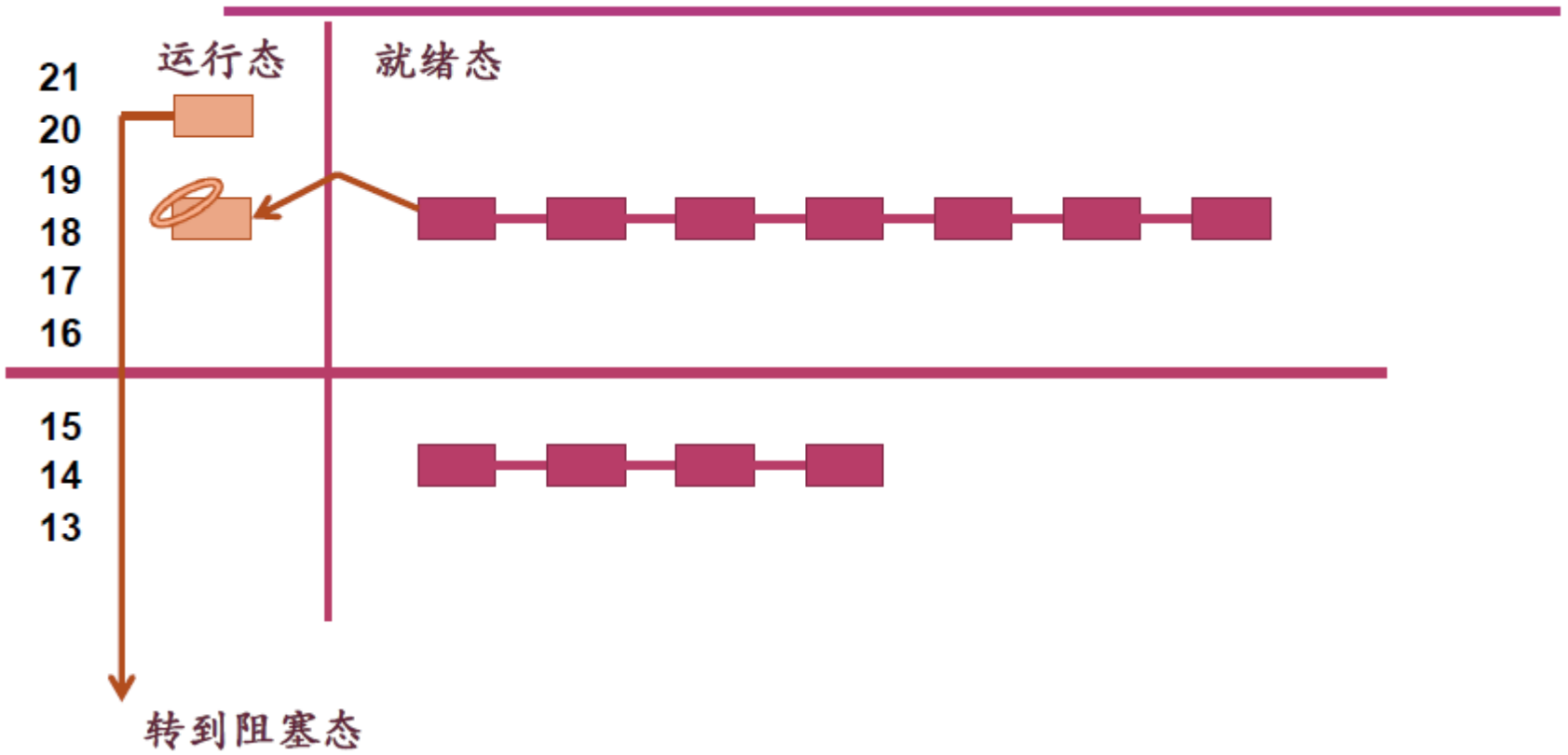
线程的时间配额

- 时间配额不是一个时间长度，而是一个称为配额单位(quantum unit)的整数
- 一个线程用完了自己的时间配额时，如果没有其他相同优先级线程，windows将重新给该线程分配一个新的时间配额，让他继续运行

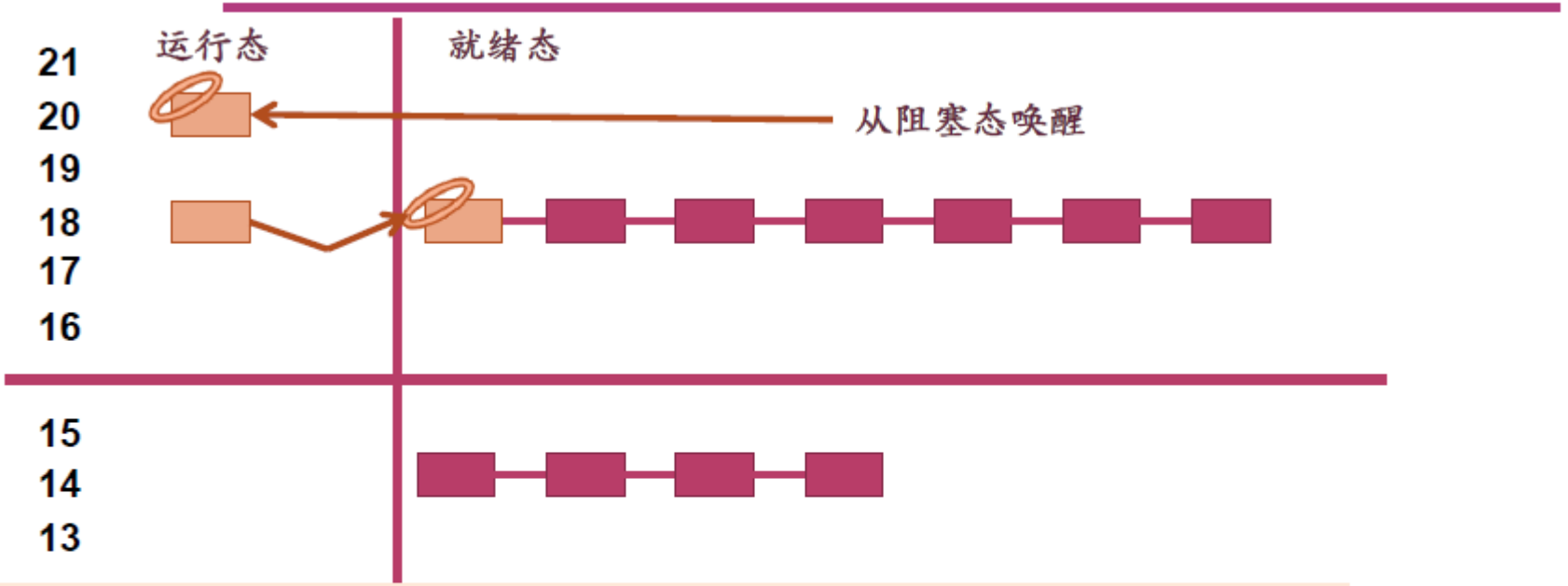
调度策略

- 主动切换
- 抢占
- 时间配额用完

（1）主动切换



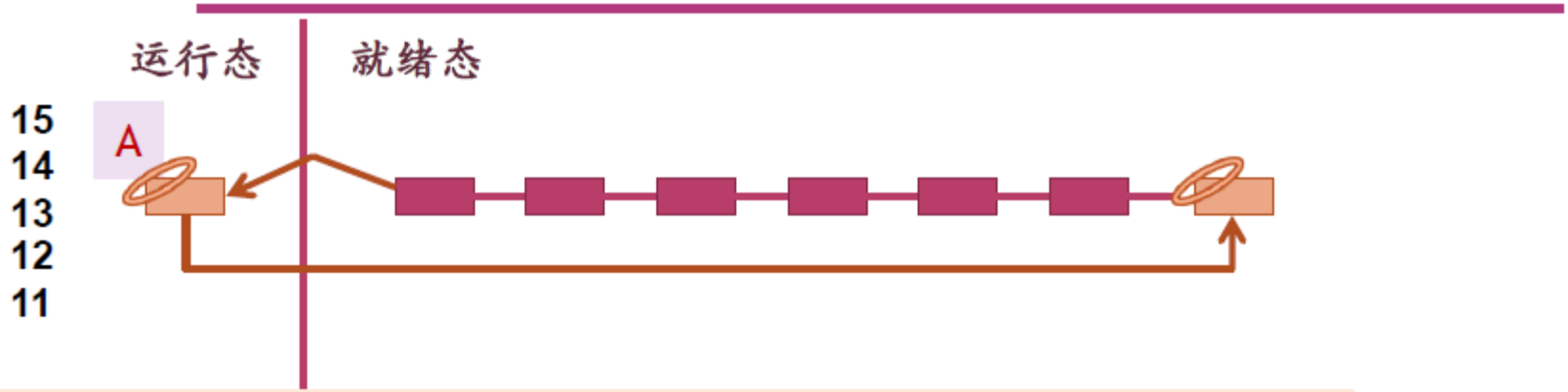
(2) 抢占



当线程被抢占时，它被放回相应优先级的就绪队列的队首

- 处于实时优先级的线程在被抢占时，时间配额被重置为一个完整的时间配额
- 处于可变优先级的线程在被抢占时，时间配额不变，重新得到CPU后将运行剩余的时间配额

（3）时间配额用完



假设线程A的时间配额用完

- A的优先级没有降低
  - ✓ 如果队列中有其他就绪线程，选择下一个线程执行，A回到原来就绪队列末尾
  - ✓ 如果队列中没有其他就绪线程，系统给线程A分配一个新的时间配额，让它继续运行
- A的优先级降低了，Windows 将选择一个更高优先级的线程

线程优先级提升与时间配额调整

- Windows的调度策略
  - 如何体现对某类线程具有倾向性?
  - 如何解决由于调度策略中潜在的不公平性而带来饥饿现象?
  - 如何改善系统吞吐量、响应时间等整体特征?
- 解决方案
  - 提升线程的优先级
  - 给线程分配一个很大的时间配额

线程优先级提升

提升线程当前优先级的情况（针对可变优先级）：

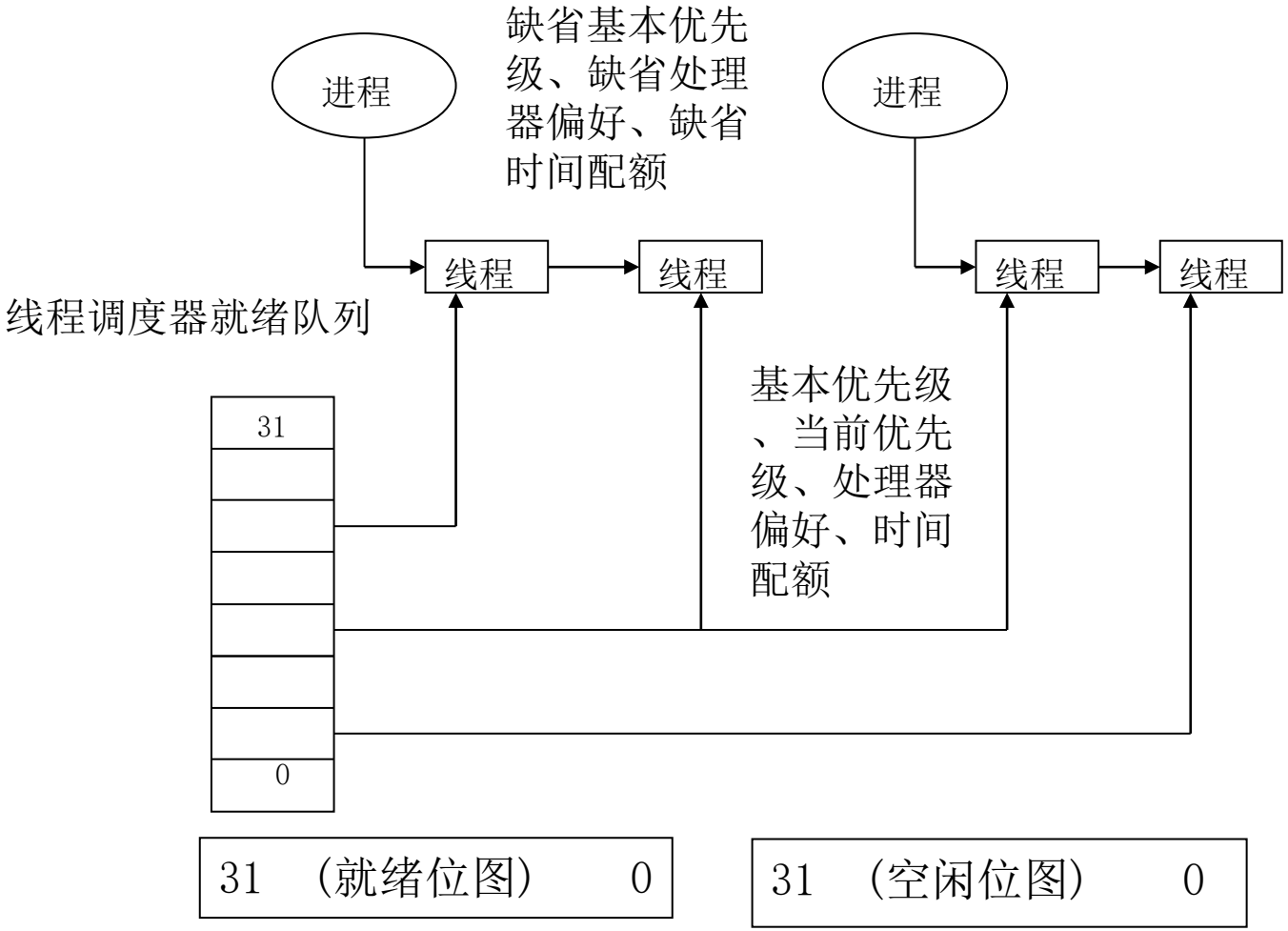
- I/O操作完成
  - 保证等待i/o的线程能够更多机会立即处理所得结果
  - 提升幅度与I/O请求响应时间要求一致
    - 磁盘、光驱、并口和视频为1；
    - 网络、串口和命名管道为2；
    - 键盘和鼠标为6；
    - 音频为8。

      - 为避免不公平，在I/O操作完成唤醒等待线程时会将该线程的时间配额减1

线程优先级提升

- 信号量或事件等待结束
  - 等待结束时候提升，然后随之降低
- 前台进程中的线程完成一个等待操作
  - 提高交互类型应用的响应时间
- 由于窗口活动而唤醒图形用户接口线程
  - 原因同上
- 优先级逆转问题：
  - 当一个高优先级(如11)等待低优先级(如5)的线程
  - 而低优先级线程处于就绪状态超过一定时间，但没能进入运行状态
  - 把这样的低优先级线程提高优先级（如：将线程的优先级提升到15，并分配给它一个长度为正常值4倍的时间配额），当该线程用完时间配额后，立即降低为基本优先级

线程调度数据结构


进程
缺省基本优先级、缺省处理器偏好、缺省时间配额
进程
线程
线程
线程
线程
线程调度器就绪队列
基本优先级、当前优先级、处理器偏好、时间配额
31
0
31 (就绪位图) 0
31 (空闲位图) 0

第5章 进程调度

5.5 算法的评估

5.5 算法的评估

- 如何为OS选择CPU调度算法?
  - 如何定义用于选择算法的标准
    - 最大化CPU使用率、最大响应时间应限定
    - 最大化吞吐量：平均周转时间与总执行时间成正比
  - 如何评估各种算法?

1.确定性模型

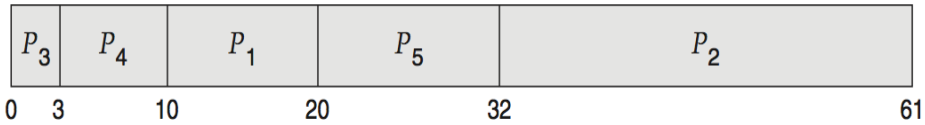
- 分析评估法
  - 给定系统负荷，评估对于该负荷，待比较算法的某性能指标
- 例子：考虑5个进程按所给顺序在0时刻到来

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

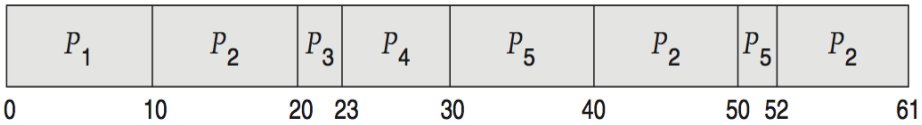
1.确定性模型(Cont.)

- 方法：
  - 计算出哪个算法的平均等待时间最小
    - FCFS: 28ms

    - 非抢占的SFJ: 13ms



    - RR (10ms): 23ms



  - 简单快捷，但需要准确的输入，且有效性依赖于输入
  - 主要用于描述算法和提供实例

2.排队模型

- 实际系统中进程是动态的，但CPU和I/O执行的分布是可确定的，如何求解？
  - 需要描述进程到达系统的分布、CPU和I/O的执行分布
  - 通常为指数的，可通过均值来表示
  - 可计算平均吞吐量、CPU利用率、等待时间等

- 排队网络分析法：
  - 将计算机系统描述为一个网络服务器，每个服务器都有一个等待进程队列
    - CPU视作具有就绪队列的服务器
    - I/O视作具有设备队列的服务器
    - 已知：到达率、服务率
    - 计算：利用率、平均队列长度、平均等待时间等

Little公式

- n = 平均队列长度(不含正在服务的进程)
- W = 队列中的平均等待时间
- λ = 新进程平均到达速度
- Little定律– 当系统处于稳定状态，进程离开队列的数量等于进程到达的数量，即:

$$n = \lambda \times W$$

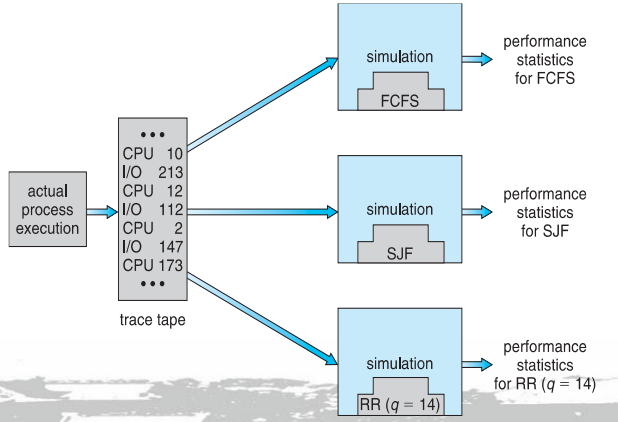
  - 该定律对于任何调度算法和到达分布都有效

- 例： n =14，λ = 7个进程/秒，则W=2秒

- 可用于比较调度算法。但，该算法依赖于数学分布假设，通常不现实。因此有局限性。

3. 仿真法

- 仿真是更为精确的
  - 将计算机系统视为一个可编程的模型
  - 时钟：一个变量
    - 随着变量的增加，模拟程序可修改系统状态，从而反映设备、进程和调度程序的活动
  - 所收集到的统计数据反映了算法的性能
  - 驱动仿真的数据产生方法：
    - 数学方法：随机数生成器，根据概率分布生成进程、CPU执行、到达时间、离开时间等
      - 分布可用数学方法或者经验公式加以定义
    - 跟踪磁带：采用trace tape记录真实系统的事件发生顺序
      - 体现真实系统与真实事件的关联



4. 实现法

- 仿真的缺陷：
  - 仅有有限的准确性
  - 代价随着精确程度而激增
- 所谓实现法，就是在真实系统中实现一个新的调度器，观测其如何工作
  - 高代价、高风险
  - 算法与环境相关
- 灵活的算法应该可被调整
  - 由管理员和用户调整优化
  - 提供API调整优先级
    - 程序优化并不代表更一般性情况的改进

小作业3

- 课本第5章课后习题 5.7、5.8、5.12、5.14
- 补充：现有A、B、C、D、E五个进程，其到达时间分别为0、2、5、7、8，要求运行时间依次为7、9、4、8、2。现请设计一套改进的多级反馈队列调度算法，要求如下：
  - 包括三级队列，高优先级队列会抢占低优先级队列。
  - 第一级队列，要求满足多个用户轮流使用，每隔1个单位时间，要响应一次。
  - 第二级队列，也要求满足多个用户轮流使用，每隔3个单位时间，要响应一次。
  - 第三级队列，要求达到理论上的平均等待时间最少，且该级队列内不考虑抢占。
  - 同时系统要求，当第三级队列调度时，定期每隔5个单位时间，对最近最久没得到服务的进程提升优先级到第一级队列。
  - 假设：对于被高优先级队列抢占的进程，当再度运行到当前队列的该进程时，仅分配上次还未完成的时间片，不再分配该队列对应的完整时间片。
  - 根据所给出的进程序列，给出调度过程。