上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

SJTU

L4-2. CPU调度

宋卓然

上海交通大学计算机系

songzhuoran@sjtu.edu.cn

线程调度

- 用户级线程与内核级线程
  - 用户级线程由线程库管理
  - 内核级线程被操作系统所调度
- 存在线程时，调度的是线程，而非进程
- 在多对一、多对多模型，系统线程库调度用户级线程
  - 进程竞争范围，process-contention scope (PCS) ----竞争CPU发生在同一进程的线程之间，采用优先级调度
- 为了决定将哪个内核级线程调度到物理CPU上，内核采用系统竞争范围，system-contention scope (SCS) ---竞争CPU发生在系统中所有线程之间

线程调度

- PCS 和 SCS 的作用在于定义了在不同的上下文中资源竞争的范围，以确保系统的正确性和可靠性
  - PCS 更侧重于进程内的竞争条件
  - SCS 更广泛地考虑了整个系统的竞争条件
- PCS和SCS有助于操作系统设计者选择适当的同步和竞争范围管理策略，以防止数据不一致和竞争条件的发生

Pthreads调度

- Pthreads API可以指定线程创建时采用PCS或SCS
  - PTHREAD_SCOPE_PROCESS
    - 按PCS来调度线程，将用户级线程调度到轻量级进程（LWP）
    - 轻量级进程的数量由线程库指定
  - PTHREAD_SCOPE_SYSTEM
    - 按SCS来调度线程
    - 创建轻量级进程，并将其与一个用户级线程绑定
    - 使用一对一模型

Pthreads调度

- Pthreads IPC提供两个函数，用于获取和设置竞争范围策略
- pthread_attr_setscope(pthread_ attr_t *attr, int scope)
- pthread_attr_getscope(pthread_ attr_t *attr, int *scope)
- 函数第一个参数包含线程属性集的指针
- pthread_attr_setscope的第二个参数可以设置PCS或SCS

Pthreads调度

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
   int i, scope;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;

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

   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }

   /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

   /* 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);
}
```

多处理器调度

- 多处理器：多个CPU，如何进行负载分配
- 主要关注同构系统
  - 内部所有处理器从功能上来说相同
  - 限制：有一个I/O设备与某个处理器通过私有总线相连
    - 希望使用该设备的进程应调度到该处理器上运行

多处理器调度

- 让一个处理器（主服务器）处理所有调度、I/O处理以及其他系统活动；其他处理器只执行用户代码
  - 非对称多处理
  - 只有一个处理器访问系统数据结构，减少数据共享的需要
- 让每个处理器自我调度
  - **对称多处理**
  - 所有进程处于同一就绪队列；每个处理器都有私有的就绪队列
  - 若多个处理器试图访问、更新共同的数据，每个处理器都需要仔细编程
  - 确保两个处理器不会选择同一进程

处理器亲和性

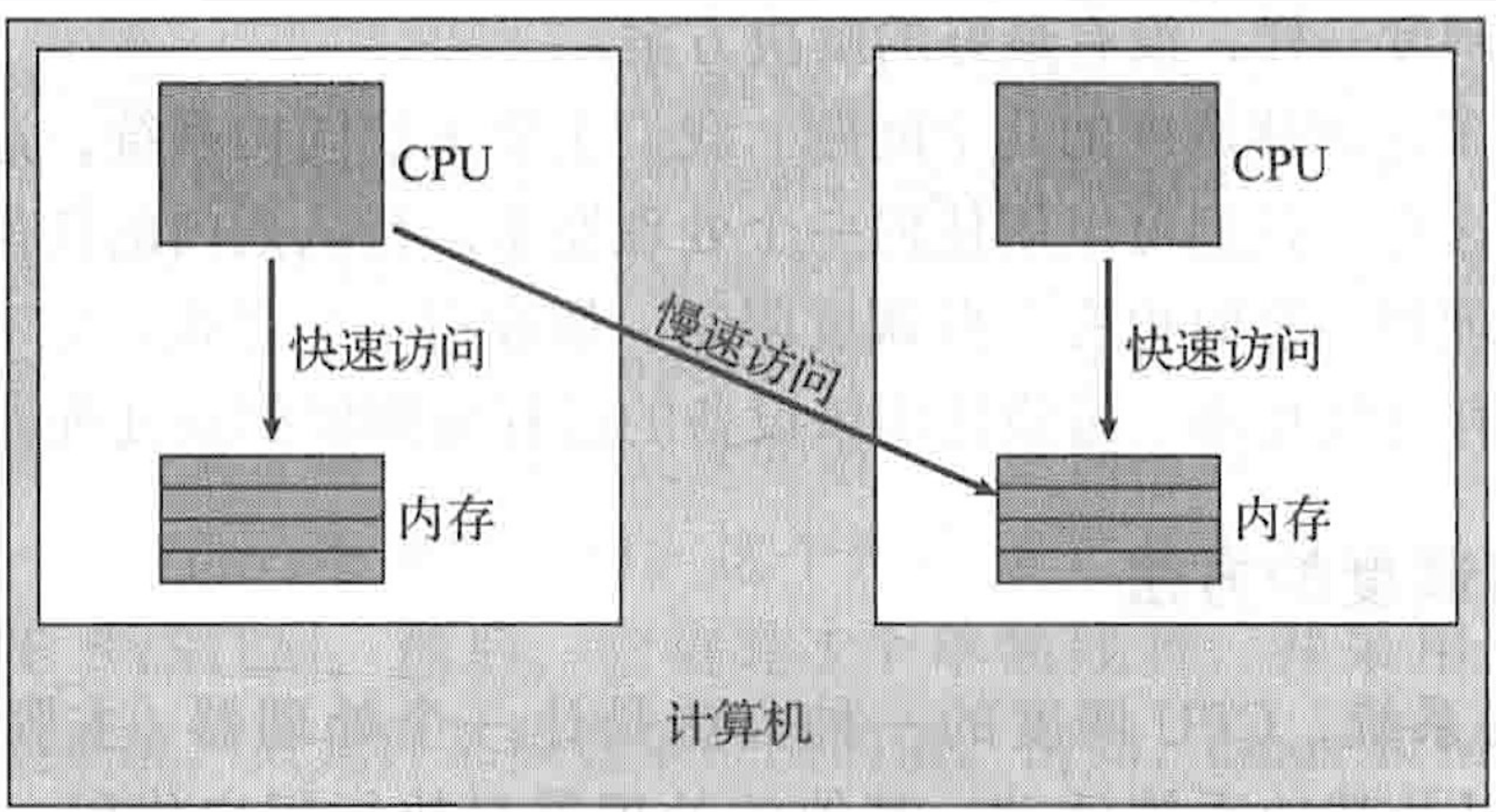
- 当一个进程最近访问的数据更新了处理器的缓存
- 将其从处理器A移到处理器B
- 处理器A的缓存应设为无效
- 处理器B的缓存应重新填充
- 但缓存无效、重新填充代价高
- 避免进程从一个处理器移到另一处理器
- 被称为“处理器亲和性”，一个进程对它运行的处理器具有亲和性

处理器亲和性

- 软亲和性
  - 试图保持进程运行在同一处理器，但不保证
- 硬亲和性
  - 允许某个进程运行在某个处理器子集上
- 系统的内存架构会影响处理器的亲和性
  - 非统一内存访问（non-uniform memory access，NUMA），一个CPU访问内存某些部分比其他部分更快

处理器亲和性

- 非统一内存访问（non-uniform memory access，NUMA）

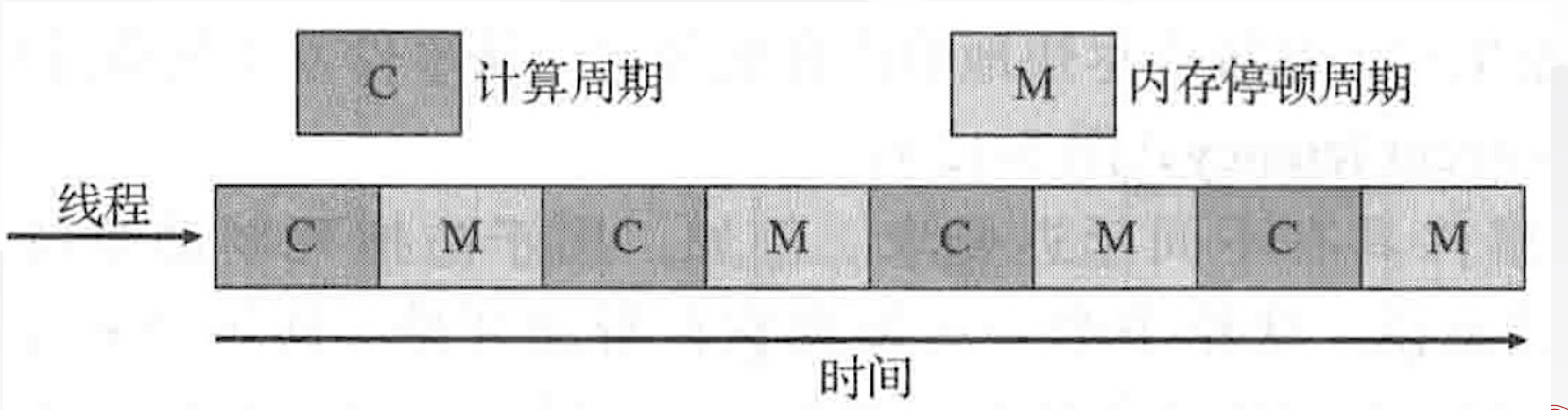


负载平衡

- 目标：保持所有处理器的负载平衡
- 针对每个处理器拥有私有可执行进程队列的系统
- 而对于具有公共队列的系统，负载平衡没有必要
- 方法
  - 推迁移：一个特定的任务周期性检查每个处理器的负载，如果发现不平衡，则通过将进程从超载处理器推到空闲、不太忙的处理器，从而平均分配负载
  - 拉迁移：当空闲处理器从一个忙的处理器上拉一个等待任务时，发生拉迁移
- 可能抵消处理器亲和性的好处（trade-off）

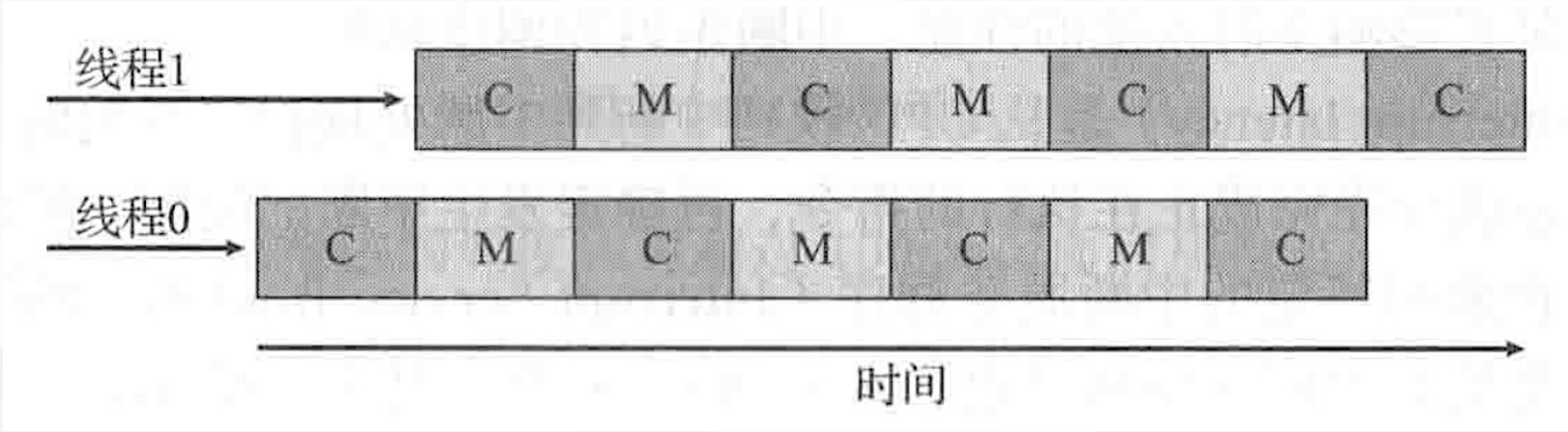
多核处理器

- 多个处理器处于同一个物理芯片，共享MMU，看似多个物理处理器：多核处理器
- 在多核处理器中，可能花费大量时间等待所需数据，发生内存停顿问题
  - 高速缓存未命中
  - 访问的数据不是连续的



多核处理器

- 为每个核分配多个硬件线程
  - 在双线程双核系统，操作系统有4个逻辑处理器
- 当一个线程因等待内存而停顿，可以切换另一线程



多线程多核处理器

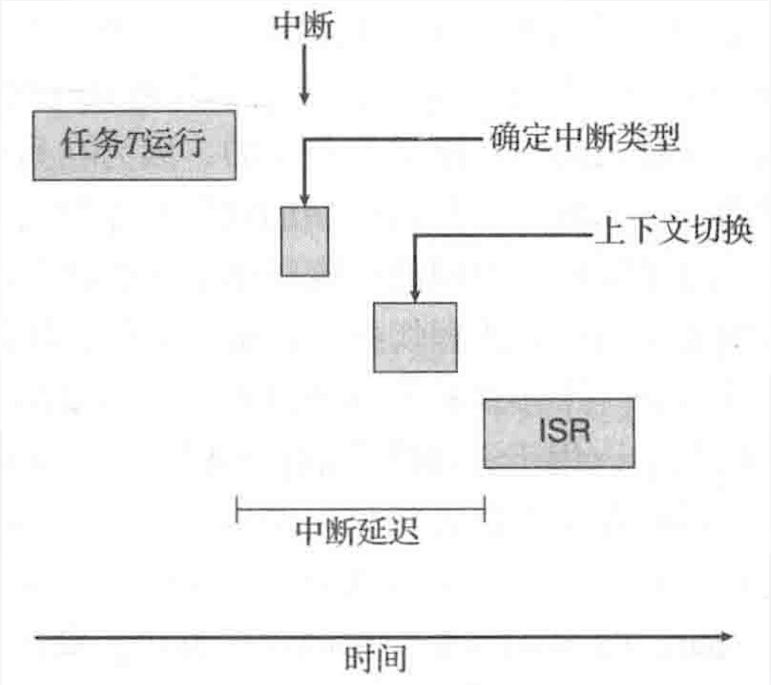
- 粗粒度
  - 线程一直在处理器上执行，直到一个长延迟事件（内存停顿）发生
  - 控制简单
  - 线程切换成本高，新线程需要刷新指令流水线
- 细粒度
  - 多线程在更细的粒度（指令周期的边界上）切换线程
  - 配备了架构设计，具有线程切换的逻辑单元
  - 成本低
- 对于选择调度哪个线程，由操作系统决定（FCFS、SJF……）

实时CPU调度

- 硬实时系统
  - 一个任务应在它的**截止日期**前完成
- 软实时系统
  - 不保证会调度关键实时进程，只保证该类进程的优先级
- 实时系统的事件驱动性质
  - 当一个事件发生时，系统应尽快响应、服务它
  - 从事件发生到得到服务的时间被称为事件延迟
  - 不同事件延迟要求不同：用于防抱死制动系统的要求为3-5ms；飞机的延迟要求高达几秒

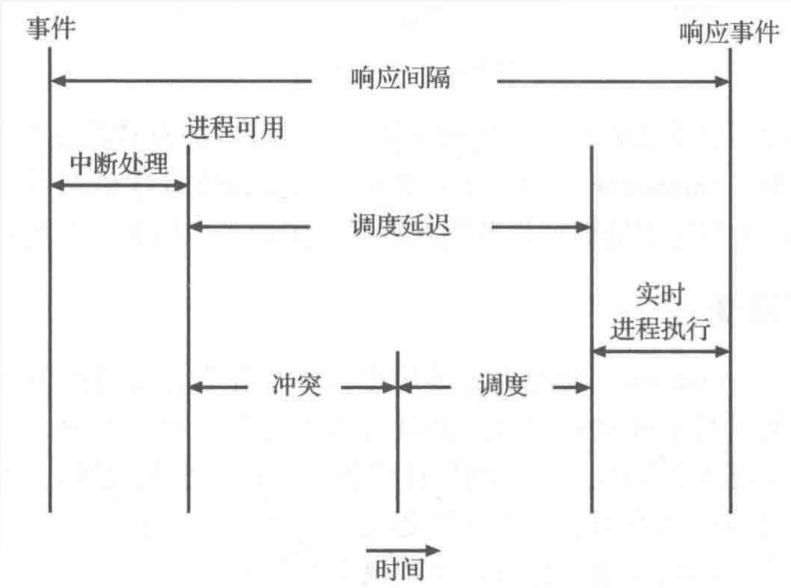
实时CPU调度

- 影响实时系统性能的因素
  - 中断延迟
    - CPU收到中断到中断处理程序（Interrupt Service Routine, ISR）开始的时间



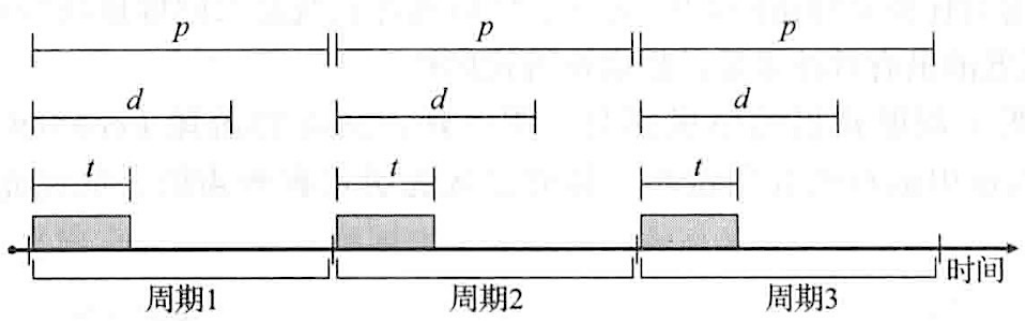
实时CPU调度

- 影响实时系统性能的因素
  - 调度延迟
    - 从停止一个进程到启动另一个进程到时间
    - 提供抢占式内核
  - 调度延迟的冲突阶段
    - 抢占在内核中运行的任何进程
    - 释放高优先级进程所需的、低优先级进程占有的资源（当被抢占的进程正在进行写操作）



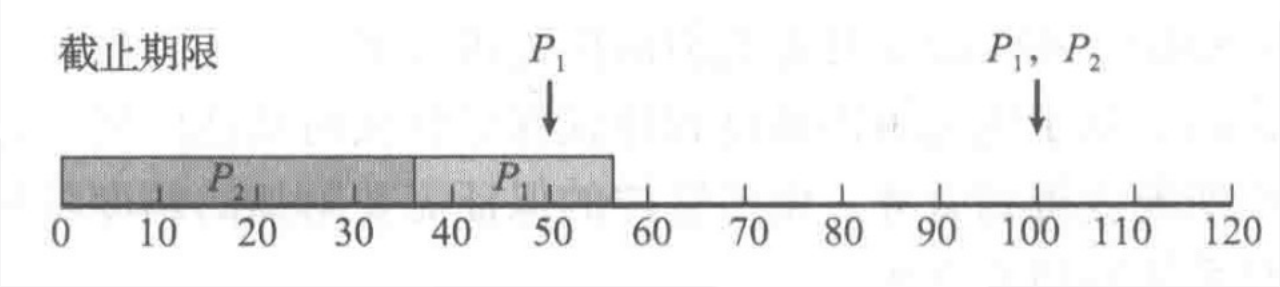
实时CPU调度 优先权调度

- 支持抢占的、基于优先权调度
  - 有一个更高优先级的进程处于就绪， 那么正在运行的、较低优先级的进程会被抢占，尽可能满足实时要求
  - 但仅支持软实时功能
- 硬实时系统应进一步保证实时任务在截止日期前得到服务
  - 周期任务：进程定期需要CPU，进程可能向调度器公布其截止日期要求
    - 固定的处理时间t、CPU应处理的截止期限d，和周期p



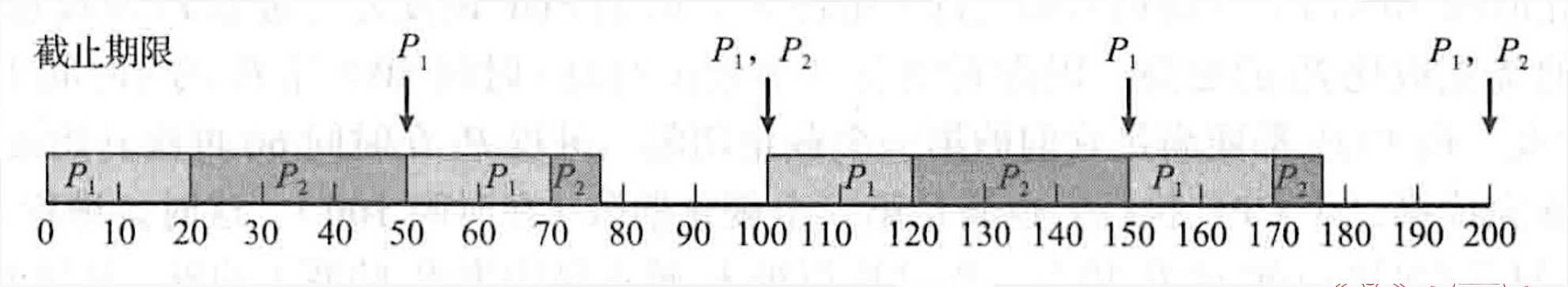
实时CPU调度 单调速率调度

- 单调速率调度算法采用抢占的、**静态优先级**的策略
- 每个周期性任务会分配一个优先级，它与其周期成反比。周期越短，优先级越高，周期越长，优先级越低
  - 更频繁使用CPU的任务应该分配更高的优先级
- 例子：周期p1=50,p2=100;处理时间t1=20;t2=35
- 假设P2优先级高于P1，无法满足截止日期的要求



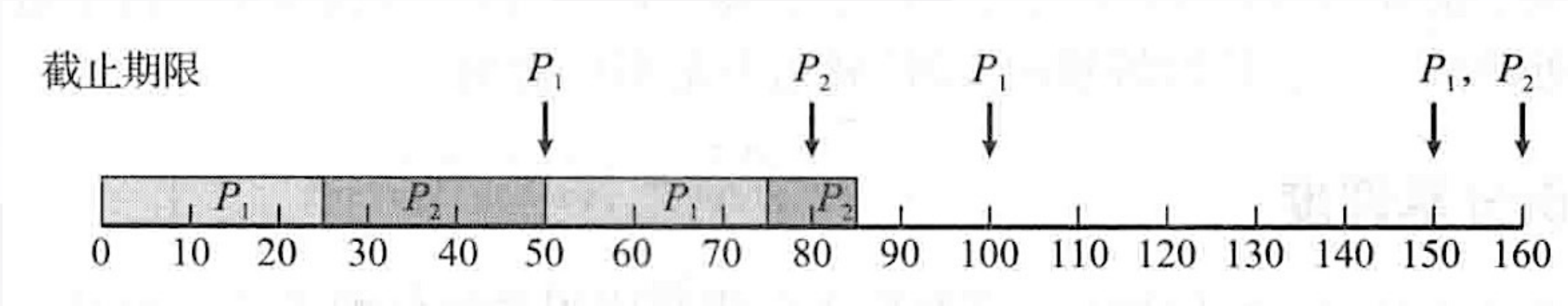
实时CPU调度 单调速率调度

- 例子：p1=50,p2=100;t1=20;t2=35
- 采用单调速率调度
  - P1的优先级高于P2
  - 可以满足截止日期的要求



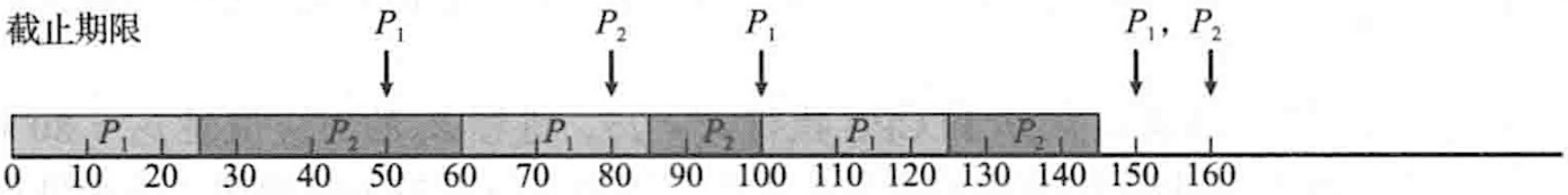
实时CPU调度 单调速率调度

- 例子：p1=50,p2=80;t1=25;t2=35
- 采用单调速率调度
  - P1的优先级高于P2
  - 无法满足截止日期的要求



实时CPU调度 最早截止期限优先调度

- 根据截止期限**动态分配优先级**
  - 截止期限越早，优先级越高；截止期限越晚，优先级越低
  - 当一个进程可运行时，它应向系统公布截止期限要求
- 例子：p1=50,p2=80;t1=25;t2=35
- P2不会被P1抢占，因为P2的截止期限（80）小于P1的截止期限（100）
- 可以满足截止期限
- 但需要进程支持宣布其截止期限



实时CPU调度 POSIX实时调度

- POSIX标准定义两种实时线程调度
  - SCHED_FIFO：先来先服务策略
  - SCHED_RR：轮询
- POSIX API：获取、设置调度策略
  - pthread_attr-getsched_policy (pthread_attr_t *attr, int *policy)
  - pthread_attr_setsched_policy (pthread_attr_t *attr, int policy)

实时CPU调度 POSIX实时调度程序

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

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

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

   /* get the current scheduling policy */
   if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
      fprintf(stderr, "Unable to get policy.\n");
   else {
      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");
   }
```

实时CPU调度 POSIX实时调度程序

```
   /* set the scheduling policy - FIFO, RR, or OTHER */
   if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
      fprintf(stderr, "Unable to set policy.\n");

   /* 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);
}
```

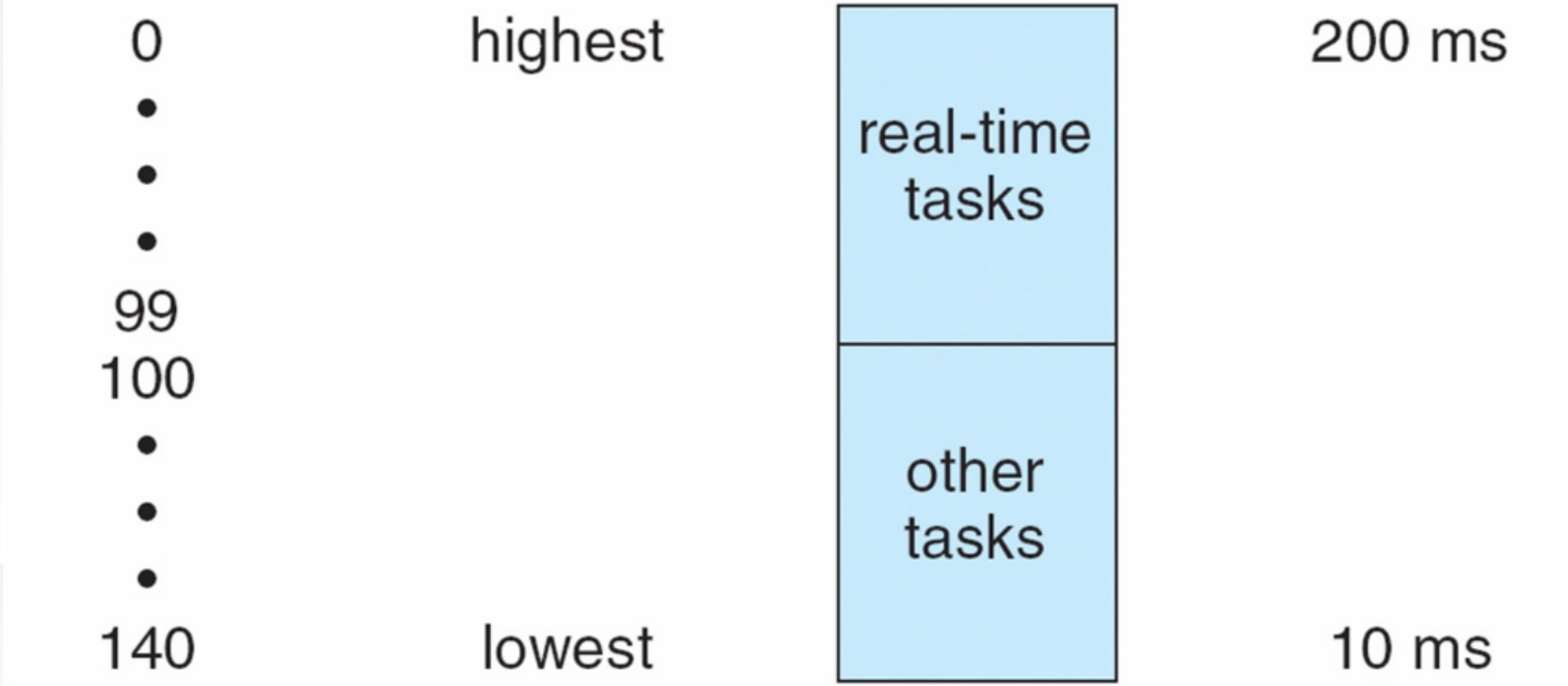
操作系统例子

- Linux调度
- Solaris调度
- Windows XP调度

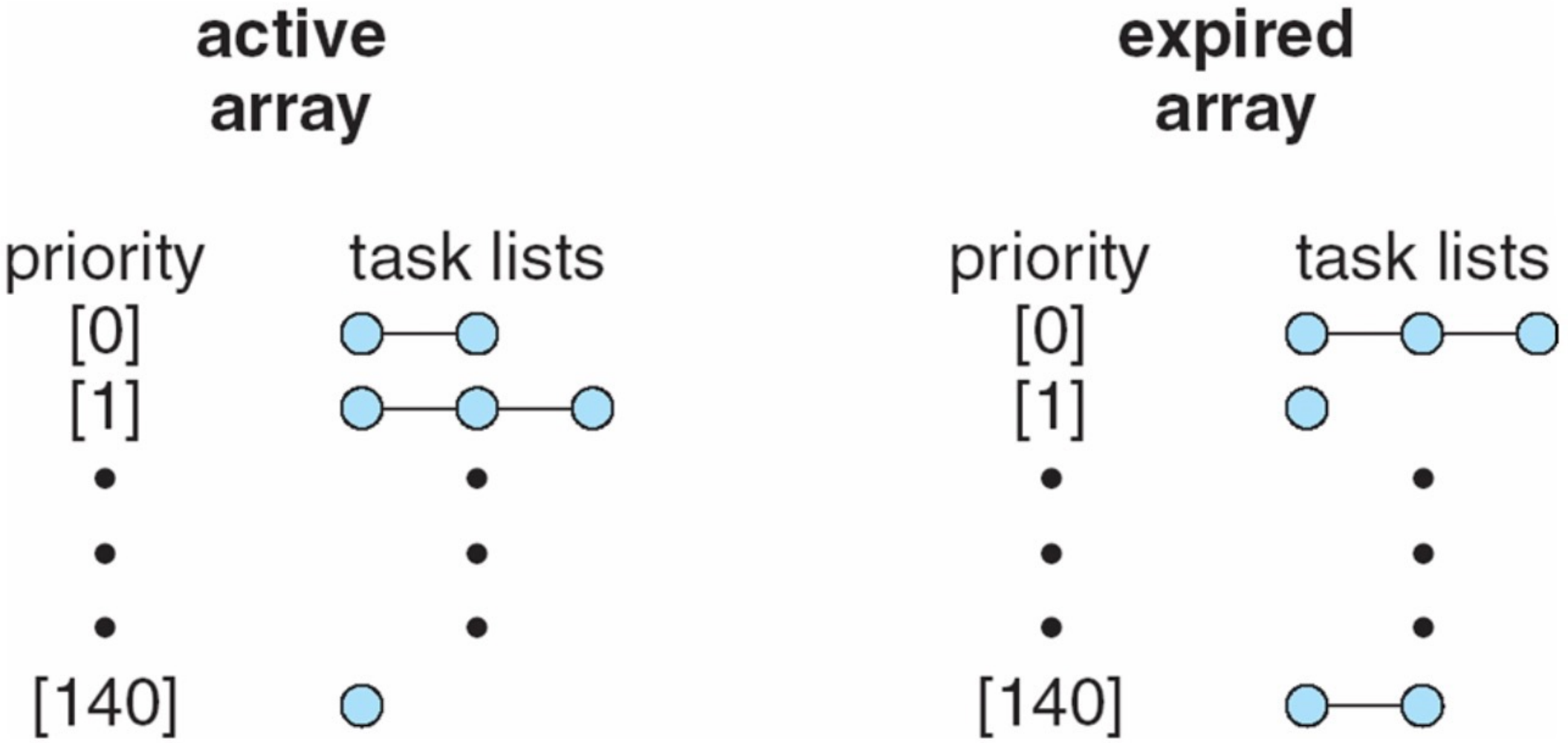
Linux调度

- 2.5版本前，采用传统UNIX调度，没有考虑对称多处理器，性能欠佳
- 2.5版本，O(1)的调度算法，支持对称多处理器、处理器亲和性、负载平衡
- 抢占式、优先级调度
- 高优先级具有较长的时间片
- 如果该进程（过期）到达时间片还没有完成，则等待其他所有进程（激活的）用完它们的时间片
- 采用两个优先级阵列，当没有激活的进程时，阵列交换

Linux调度 抢占式、优先级调度



Linux调度 抢占式、优先级调度

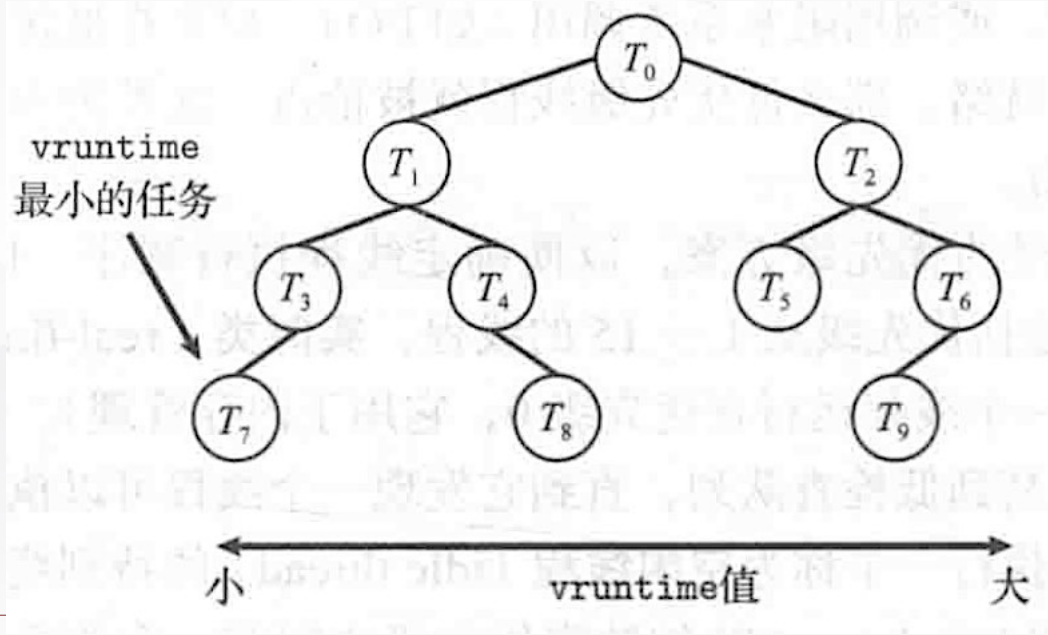


Linux调度

- 2.6.23+版本，采用完全公平调度程序 Completely Fair Scheduler (CFS)
- CFS为每个任务分配一定的CPU处理时间，该时间根据友好值（nice value）来计算，范围从-20到+19，越低表示越高的优先级
- 通过目标延迟---每个可运行任务应当运行一次的时间间隔，根据目标延迟设定友好值
- 设置变量vruntime（虚拟运行时间），该变量与任务优先级的衰减因子θ有关：vruntime+=runtime*θ, θ=weight_nice_0/weight_nice_i
  - 友好值=0，运行200ms，则vruntime=200
  - 友好值较高，运行200ms，则vruntime>200
  - 友好值较低，运行200ms，则vruntime<200

Linux调度

- 每次选择vruntime最小的任务执行
- 抢占式
  - 有一个更高优先级的任务到来，会抢占低优先级的任务
- 根据vruntime构建红黑树（平衡的、二分搜索树），找到最左侧节点仅需O(lg N)操作

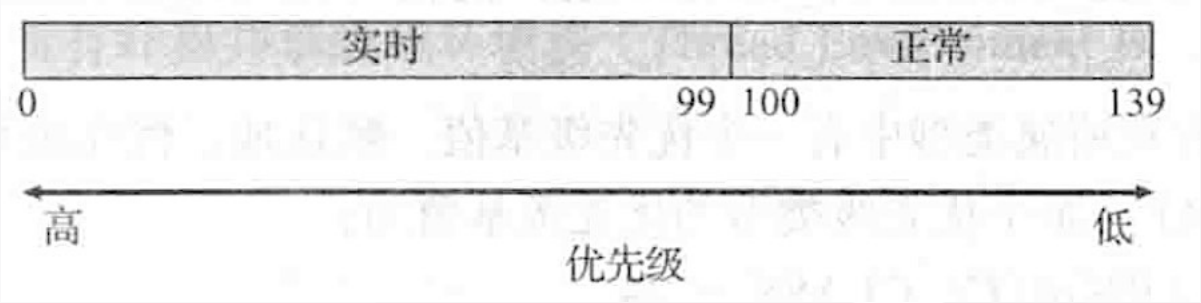


Linux调度

- 假设对于一个I/O密集，一个CPU密集的任务，拥有相同的友好值
- 则I/O密集的vruntime<CPU密集的vruntime
- 导致I/O密集的优先级更高
- 当CPU在运行，而I/O变得有资格运行时，会抢占CPU密集任务

Linux调度

- 实现实时调度
  - 采用SCHED_FIFO或SCHED_RR实时调度
  - 实时任务的优先级更高
- 对于实时+普通任务，采用两个单独的优先级范围
  - 实时任务静态优先级为0～99
  - 普通任务优先级为100～139，友好值-20映射到优先级100，友好值+19映射到优先级139



Windows调度

- 基于优先级的、抢占调度算法
- 确保具有最高优先级的线程在运行
- 线程运行，直到：1）阻塞；2）时间片已到；3）被抢占
- 32级优先级（数值越高优先级越高）
  - 可变类，优先级1～15
  - 实时类，优先级16～31
- 每个优先级分配一个队列

Windows调度

- Windows API定义了进程可能属于的优先级类型
  - REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS,NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
  - 除了REALTIME_PRIORITY_CLASS，其他类的优先级可变
  - 通过API函数SetPriorityClass()修改进程的优先级的类
- 对于给定优先级类的一个线程，由相对优先级
  - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE

Windows调度

- 每个线程的优先级基于它所属的优先级类和它在该类中的相对优先级
- 每个优先级类拥有优先级基值，为优先相对值NORMAL
  - REALTIME_PRIORITY_CLASS- 24
  - HIGH_PRIORITY_CLASS- 13
  - ABOVE_NORMAL_PRIORITY_CLASS – 10
  - NORMAL_PRIORITY_CLASS- 8
  - BELOW_NORMAL_PRIORITY_CLASS – 6
  - IDLE_PRIORITY_CLASS -4

Windows调度

优先级类

相对优先级	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

Windows调度

- 当一个线程的时间片用完，线程被中断，若属于可变优先级类型，则优先级降低，但不能低于优先级基值
- 当一个可变优先级的线程从等待中释放，则提高优先级
  - 等待键盘I/O的线程将得到一个较大提升；而等待磁盘操作的线程将得到一个中等提升。采用这种策略， 正在使用鼠标和窗口的线程往往得到很好的响应时间
- 对于交互程序，具有前台、后台进程
  - 对于移到前台的进程，增加3x时间片

Solaris调度

- 基于优先级调度，每个线程属于6个类型之一
  - 分时
  - 交互
  - 实时
  - 系统
  - 公平分享
  - 固定优先级
- 不同类型具有不同的优先级和调度算法

Solaris调度

- 进程的默认调度类型为分时，策略为动态改变优先级，多级反馈队列，采用不同长度的时间片
  - 优先级与时间片成反比
    - 交互进程具有更高的优先级，时间片小，具有较好的响应时间
    - CPU密集型进程具有更低的优先级，时间片大，具有较好的吞吐量

Solaris调度

- 时间片到期：用完全部时间片而未阻塞的线程所得到的新优先级
  - 表中，优先级降低（数值越高优先级越高）
- 从睡眠中返回:从睡眠(如等待I/O)中返回的线程的优先级
  - 表中，优先级提高

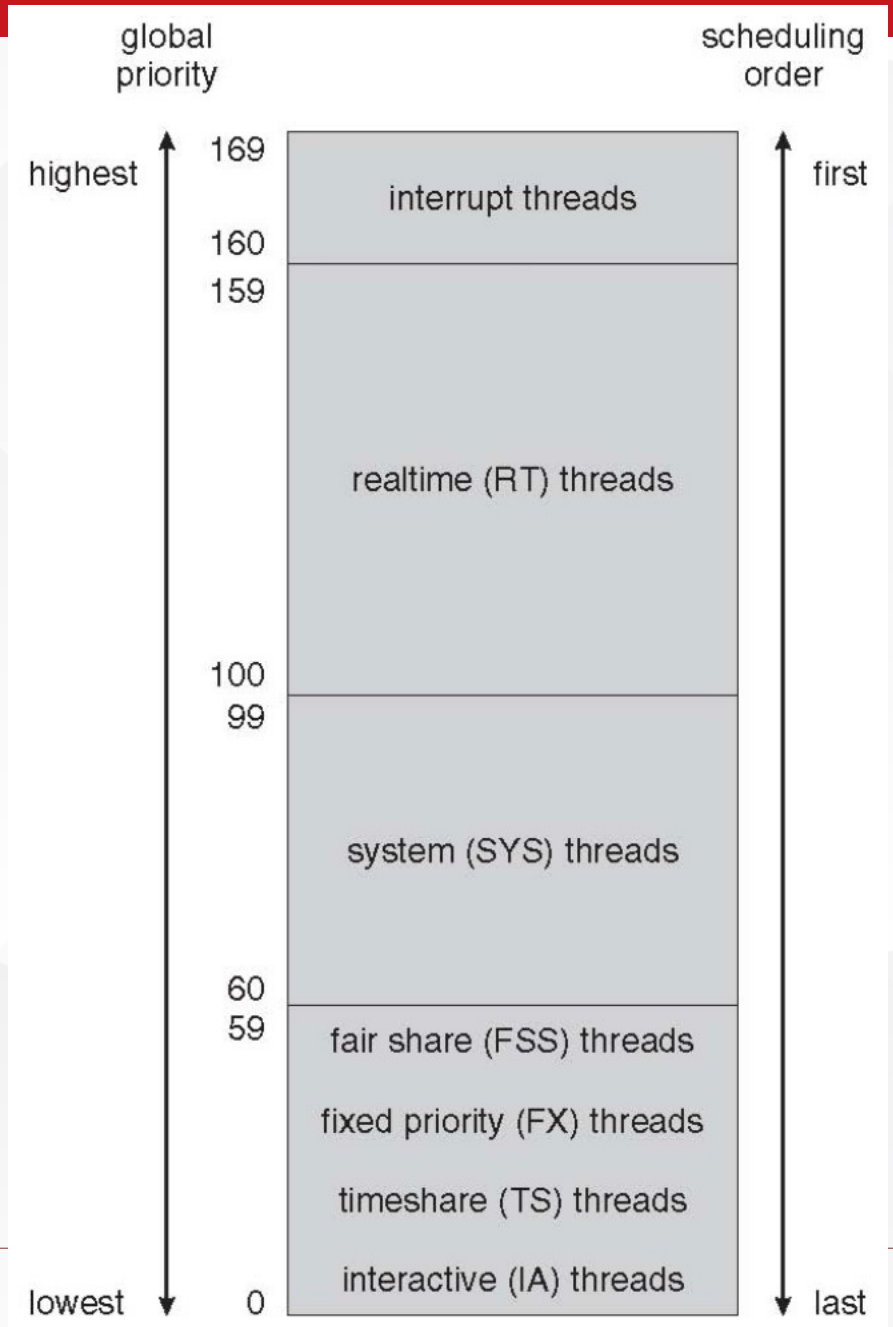
priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Solaris调度

- 对于系统线程，为静态优先级调度
  - 系统类型专门用于操作系统内核
- 固定优先类
  - 优先级范围与分时类相同（60个优先级），但优先级无法动态调整
- 公平分享类
  - 采用CPU共享，而不是优先级
  - CPU 分享表示对可用CPU 资源的授权，且可分配给一组进程

Solaris调度

- 全局优先级
  - 分时
  - 交互
  - 实时
  - 系统
  - 公平分享
  - 固定优先级



算法评估

- 如何选择CPU调度算法?
  - 定义准则
    - 最大化CPU利用率
    - 最大化吞吐量
  - 采用评估方法进行评估
    - 分析评估法：使用给定算法与系统负荷，生成一个公式或数字，以评估在该负荷下的算法性能
    - 仿真：跟踪磁带，监控真实系统，记录事件发生的顺序
    - 实现：对于调度算法，对它进行编程，放入操作系统，观测它如何工作

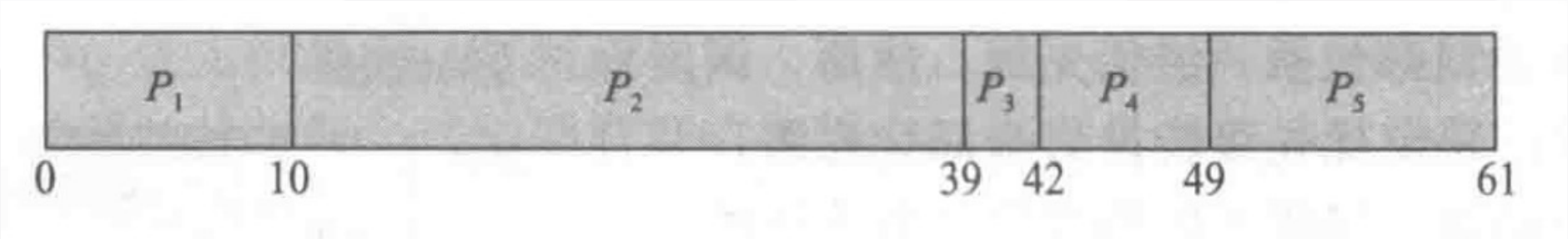
算法评估 分析评估法

进程	执行时间
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

- 评估FCFS、SJF、RR算法，哪个可以给出最小的平均等待时间?

算法评估

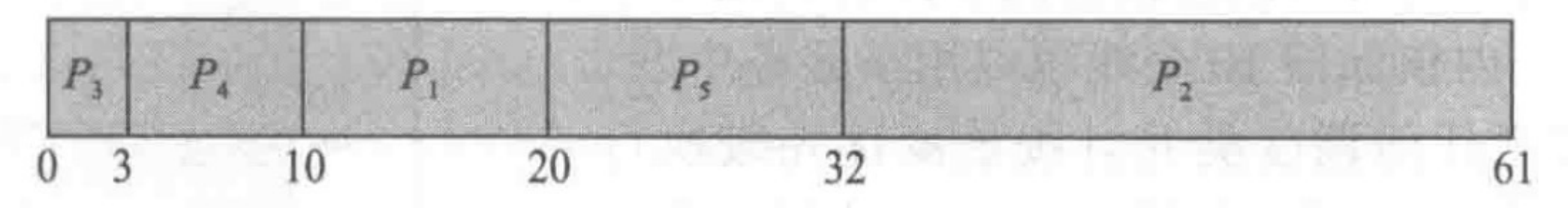
- FCFS：



- 平均等待时间：　(0+10+39+42+49) /5=28ms

算法评估

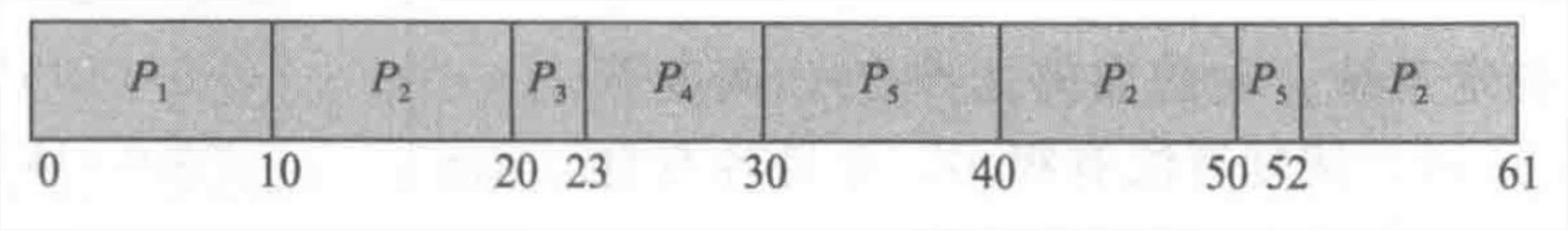
- 非抢占SJF:



- 平均等待时间： (10+32 +0+3+20)/5= 13ms

算法评估

- RR:



- 平均等待时间: (0+10+20+2+20+23+30+10) /5=23ms

算法评估

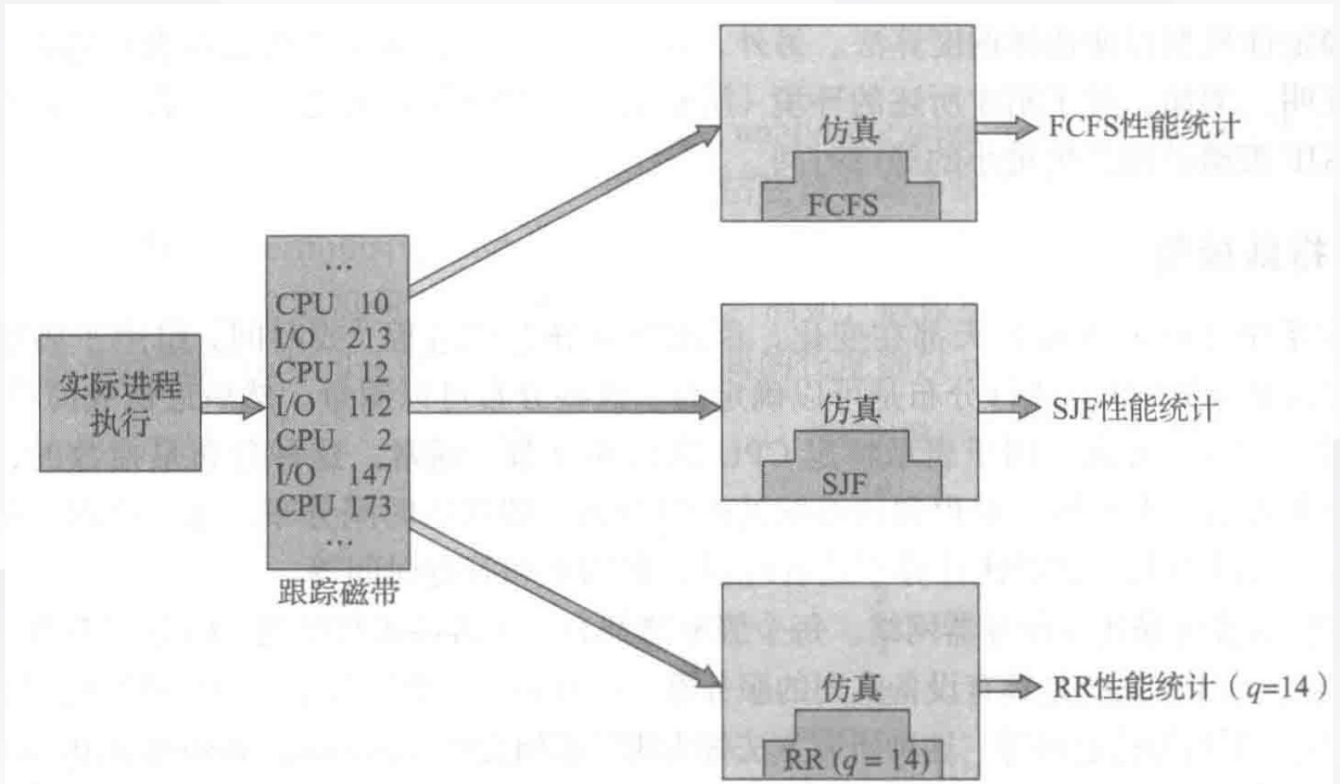
- 确定性模型简单且快速，**但要求输入精确延迟**
- 主要用途
  - 描述调度算法
  - 提供例子，表示趋势
    - 对于刚才所述环境（所有进程都在时间0到达，且它们的处理时间都已知），SJF策略总能产生最小的等待时间

算法评估 仿真

- 仿真涉及对计算机系统进行建模。 软件数据结构代表了系统的主要组成部分。仿真程序有一个代表时钟的变量；随着这个变量值的增加，模拟程序修改系统状态以便反映设备、进程和调度程序的活动。随着仿真的运行，表明算法性能的统计数据被收集并打印
- 驱动仿真的数据可由许多方法产生。最为常见的方法是:通过随机数生成器，根据概率分布生成进程、CPU执行、到达时间、离开时间等。分布可以数学地(均匀的、指数的、泊松的)或经验地加以定义

算法评估 实现

- 由于实际系统的连续事件之间的关联，分布驱动仿真可能不精确，可以使用“跟踪磁带”，监视真实系统并记录事件发生顺序，以驱动仿真



课堂习题

1.采用Windows 调度算法，求出如下线程的优先级数值
a. 类REALTIME_PRIORITY_CLASS 的线程具有相对优先级NORMAL
b. 类ABOVE_NORMAL_PRIORITY_CLASS的 线程具有相对优先级HIGHEST
c. 类BELOW_NORMAL_PRIORITY_CLASS 的线程具有相对优先级ABOVE_NORMAL
2.假设两个任务A和B运行在一个Linux 系统上。A和B的友好值分别为-5和+5。采用CFS调度程序作为指南，针对如下情景，请描述这两个进程的vruntime 如何变化
a. A和B都是CPU密集型的
b. A是I/O密集型的，B 是CPU密集型的
c. A是CPU密集型的，B是I/O密集型的