操作系统A

Principles of Operating System

北京大学计算机科学技术系 陈向群

Department of computer science and Technology, Peking University

2020 Autumn

本章要求掌握的概念

调度层次	调度时机	进程切换
调度算法设计原则	抢占与非抢占	时间片
调度算法	饥饿	优先级反转
吞吐量	周转时间	响应时间
Linux调度算法	Windows线程调度	……

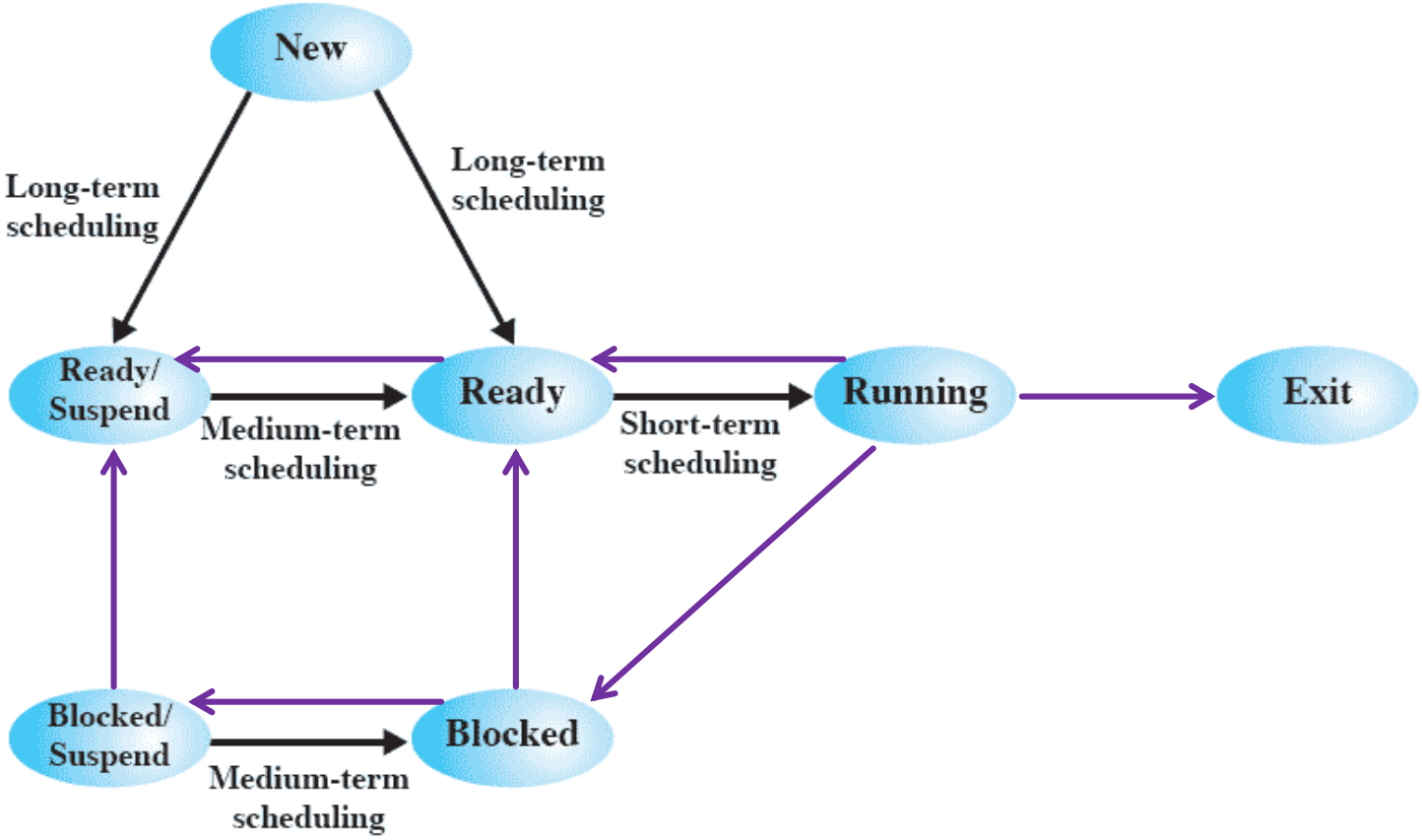
大纲

- 处理机调度的类型
- 设计处理机调度算法的原则
- 典型的处理机调度算法
- 实例：Windows线程调度
- 实例：Linux调度算法

调度层次、调度时机、调度过程……

处理器调度

处理器调度——三个层次



调度和进程状态转换

调度的三个层次小结

- **长程调度**（作业调度或宏观调度）

  创建新进程时 → 决定是否进入当前活跃进程集合

- **中程调度** 进程在内外存之间的交换

  从存储资源管理的角度：把进程的部分或全部换出到外存，可为当前运行进程的执行提供所需内存空间，将当前进程所需部分换入到内存（**交换技术**）

- **短程调度**（微观调度） 从处理机资源分配的角度，需要选择就绪进程或线程进入运行状态
  - **短程调度的时间尺度通常是毫秒级的**
  - 由于短程调度算法的频繁使用，要求实现时做到**高效**

处理器调度

系统场景
- N个进程就绪、等待上CPU运行
- M个CPU，M ≥ 1
- 需要决策：给哪一个进程分配哪一个CPU？

处理器调度

—— 其任务是控制、协调进程对CPU的竞争

即按一定的调度算法从就绪队列中选择一个进程，把CPU的使用权交给被选中的进程

如果没有就绪进程，系统会安排一个系统空闲进程或idle进程

调度程序：挑选就绪进程的内核函数

要解决三个问题

WHAT：依据何原则挑选进程/线程以分配CPU

——调度算法

WHEN：何时分配CPU

——调度时机

HOW：如何分配CPU

——调度过程（进程的上下文切换）

CPU调度的时机(1/2)

事件发生 → 当前运行的进程暂停运行 → 硬件机制响应后 → 进入操作系统，处理相应的事件 → 结束处理后：

某些进程的状态会发生变化，也可能又创建了一些新的进程

→ 就绪队列有调整 → 需要进程调度根据预设的调度算法从就绪队列选一个进程

典型的事件举例：

- 创建、唤醒、退出等进程控制操作
- 进程等待I/O、I/O中断
- 时钟中断，如：时间片用完、计时器到时
- 进程执行过程中出现abort异常

CPU调度的时机(2/2)

- 进程正常终止 或 由于某种错误而终止
- 新进程创建 或 一个等待进程变成就绪
- 当一个进程从运行态进入阻塞态
- 当一个进程从运行态变为就绪态

内核对中断/异常/系统调用处理后返回到用户态前最后时刻

调度过程——进程切换(1/2)

- 进程调度程序从就绪队列选择了要运行的进程：

  这个进程可以是刚刚被暂停执行的进程，也可能是另一个新的进程

→ **进程切换**

- 进程切换：是指一个进程让出处理器，由另一个进程占用处理器的过程

调度过程——进程切换(2/2)

- 主要包括两部分工作：
  - 切换全局页目录以加载一个新的地址空间
  - 切换内核栈和硬件上下文，其中硬件上下文包括了内核执行新进程需要的全部信息，如CPU相关寄存器

切换过程包括了对原来运行进程各种状态的保存和对新的进程各种状态的恢复

上下文切换开销(cost)

什么是上下文
切换的开销?

- 直接开销:

  内核完成切换所用的CPU时间
  - 保存和恢复寄存器……
  - 切换地址空间（相关指令比较昂贵）
- 间接开销
  - 高速缓存(Cache)、缓冲区缓存(Buffer Cache)和TLB(Translation Look-aside Buffer)失效

处理器调度算法的设计

不同操作系统追求的目标

- 不同类型的操作系统
  - 交互式进程（interactive process）
    - 需要经常与用户交互，因此要花很多时间等待用户输入操作
    - 响应时间要快，平均延迟要低于50~150ms
    - 典型的交互式程序：shell、文本编辑程序、图形应用程序等
  - 批处理进程（batch process）
    - 不必与用户交互，通常在后台运行
    - 不必很快响应
    - 典型的批处理程序：营业额计算、利息计算、索赔处理
  - 实时进程（real-time process）
    - 有实时需求，不应被低优先级的进程阻塞
    - 响应时间要短
    - 典型的实时进程：视频/音频、机械控制等

响应时间、均衡性

吞吐量、周转时间、CPU利用率

最后期限、可预测性

CPU调度算法的设计考虑

- 什么情况下需要仔细斟酌调度算法?
  - 批处理系统 → 多道程序设计系统 → 批处理与分时的混合系统 → 个人计算机 → 网络服务器

	用户角度	系统角度
性能	周转时间 响应时间 最后期限	吞吐量 CPU利用率
其他	均衡性 可预测性	公平性 强制优先级 平衡资源

调度算法衡量指标

- 吞吐量 Throughput — 每单位时间完成的进程数目
- 周转时间TT(Turnaround Time)

  每个进程从提出请求到运行完成的时间
- 响应时间RT(Response Time)

  从提出请求到第一次回应的时间

- 其他
  - CPU 利用率(CPU Utilization)

    CPU做有效工作的时间比例
  - 等待时间(Waiting time)

    每个进程在就绪队列(ready queue)中等待的时间

要点讨论

设计调度算法时要考虑以下几个问题：

- 进程控制块PCB中

  需要记录哪些与CPU调度有关的信息？

- 进程优先级及就绪队列的组织
- 抢占式调度与非抢占式调度
- I/O密集型与CPU密集型进程
- 时间片

1. 进程优先级(数)

优先级 与 优先数  静态 与 动态

静态优先级:

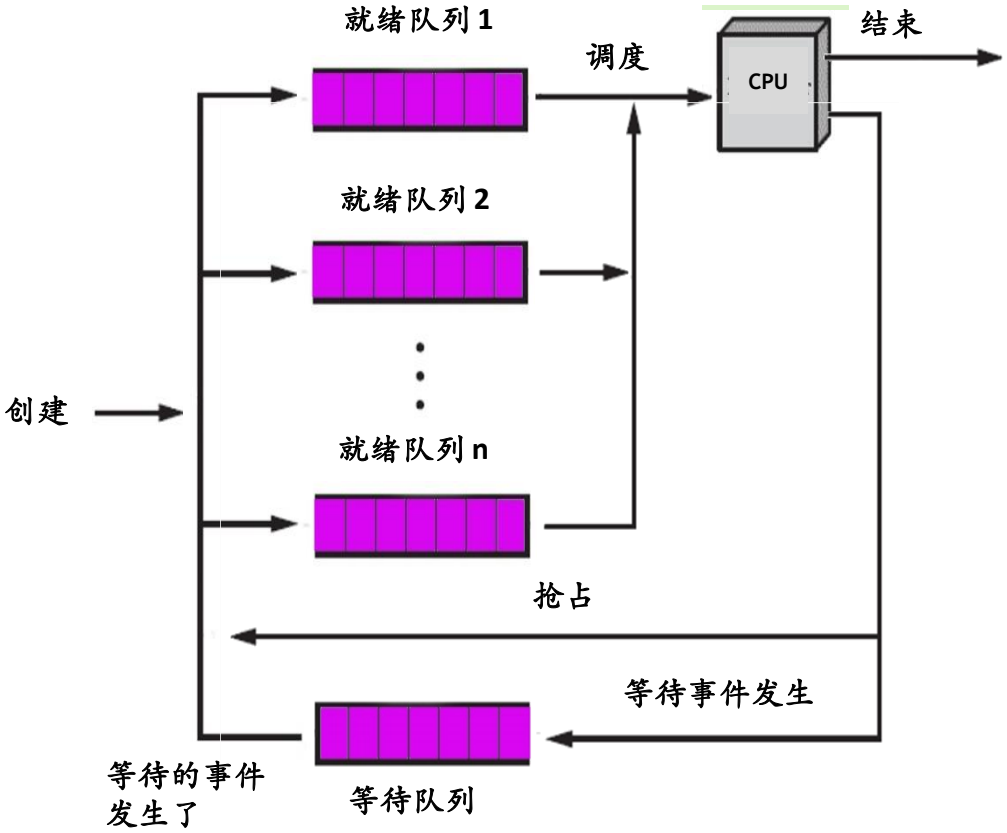
进程创建时指定，运行过程中不再改变

动态优先级:

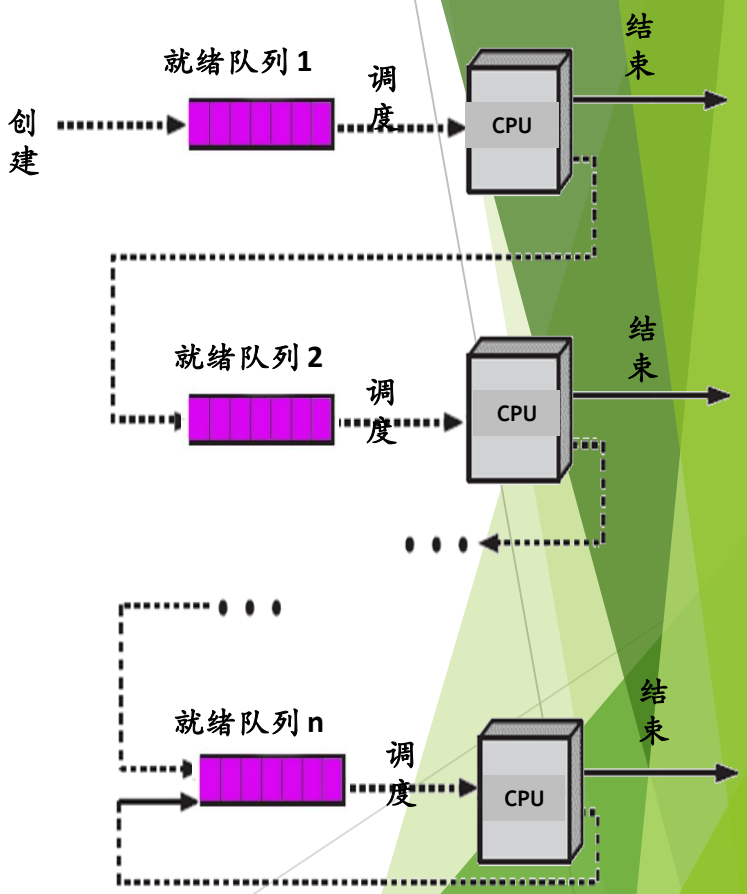
进程创建时指定了一个优先级，运行过程中可以动态变化

如：等待时间较长的进程可提升其优先级

2. 进程就绪队列组织



按优先级排队



另一种排队方式

3. 抢占与非抢占

指占用CPU的方式：

- 可抢占式Preemptive（可剥夺式）

  当有比正在运行的进程优先级更高的进程就绪时，系统可强行剥夺正在运行进程的CPU，提供给具有更高优先级的进程使用

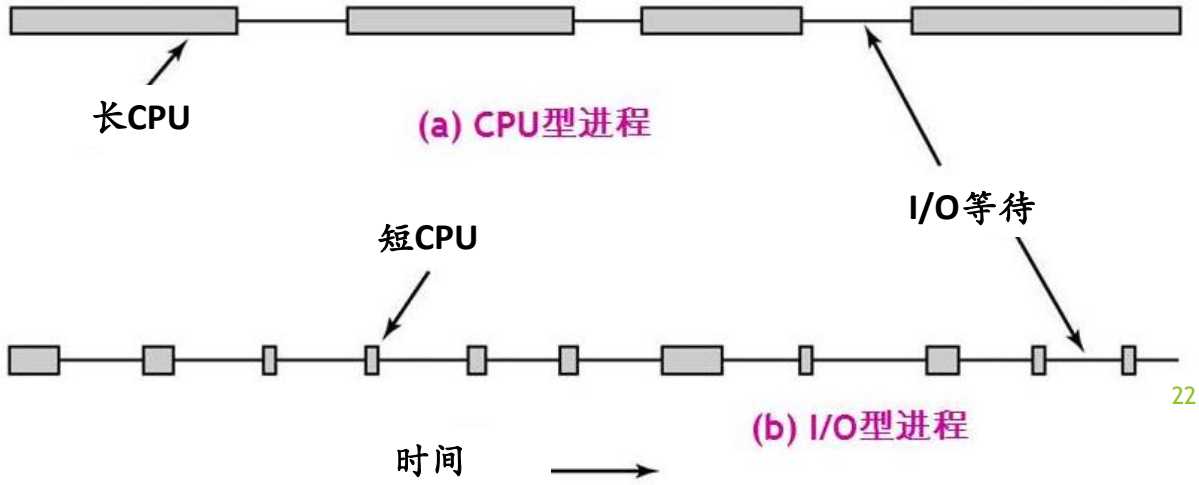
- 不可抢占式Non-preemptive（不可剥夺式）

  某一进程被调度运行后，除非由于它自身的原因不能运行，否则一直运行下去

4. I/O密集型与CPU密集型进程

按进程执行过程中的行为划分：

- **I/O密集型或I/O型(I/O-bound)**
  - 频繁的进行I/O，通常会花费很多时间等待I/O操作的完成
- **CPU密集型或CPU型或计算密集型(CPU-bound)**
  - 需要大量的CPU时间进行计算



5. 时间片

时间片长一点 or 短一点  固定 与 可变

- 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
- 如何选择时间片呢?

考虑因素：
- ✓ 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

FIFO、SSJF、HRRN、RR、……

处理机调度算法

批处理系统中采用的调度算法

- 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（SRT-Shortest Remaining Time Next）
- 最高相应比优先（HRRN-Highest Response Ratio Next）

先来先服务（FCFS）调度算法

- 按照进程就绪的先后顺序使用CPU
- 没有抢占

- 优缺点

  + 公平、简单

  - 长进程之后的短进程需要等很长时间，不利于用户的交互式体验

  - I/O资源和CPU资源的利用率较低

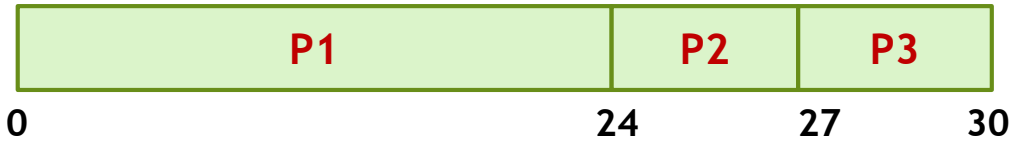
先来先服务调度算法举例

例子：

- 三个进程按顺序就绪：P1、P2、P3

  进程P1 执行需要24s，P2和P3各需要3s

  - 采用FCFS调度算法：



吞吐量：3 jobs/ 30s = 0.1 jobs/s

周转时间TT：P1:24；P2:27；P3:30

平均周转时间：27s

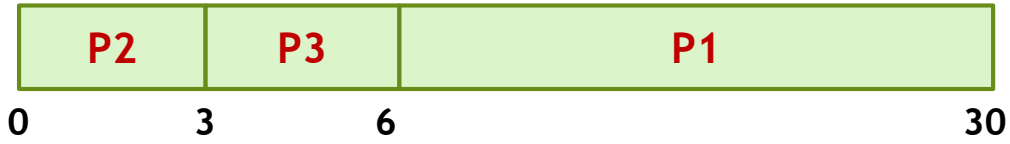
讨论

例子：

- 三个进程按顺序就绪：P1、P2、P3

  进程P1 执行需要24s，P2和P3各需要3s

  - 改变调度顺序：P2 、P3、P1



吞吐量：3 jobs/ 30s = 0.1 jobs/s

周转时间TT：P1:30；P2:3；P3:6；

平均周转时间：13s

短作业优先（SJF）调度算法

- 具有最短完成时间的进程优先执行
- 非抢占式（运行执行时间最短的进程，直到该进程结束或者被I/O阻塞）

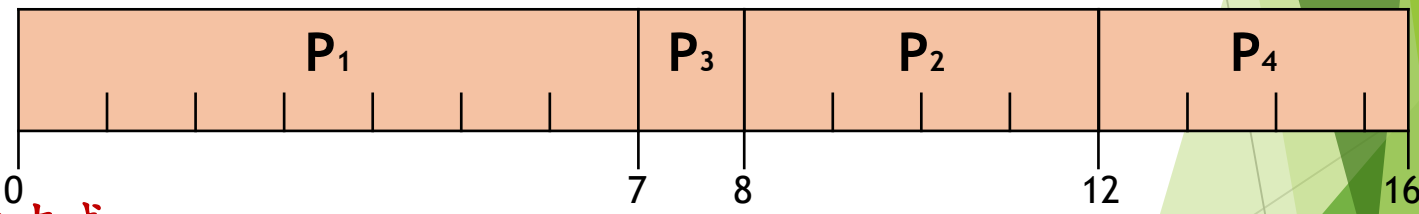
- **最短剩余时间（SRTN）优先**：SJF的抢占版本
- 当一个新就绪的进程比当前运行的进程具有更短的完成时间时，抢占当前进程，而选择新就绪的进程执行

思路：先完成短的作业
改善短作业的周转时间

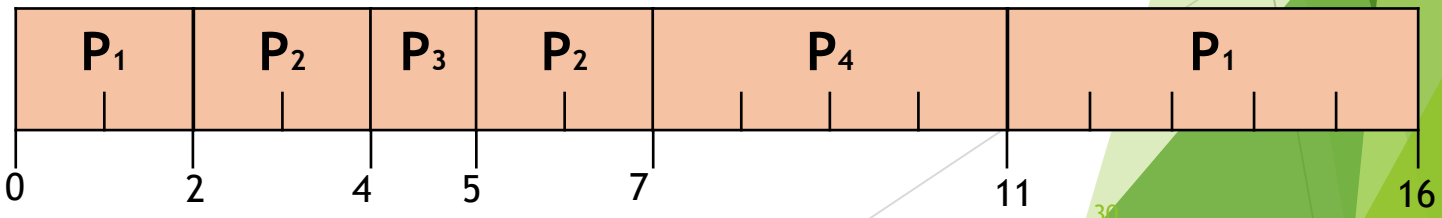
短作业优先SJF调度算法举例

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



抢占式



短作业优先调度算法

- 优点和缺点

  ＋ 最短的平均周转时间

> 在所有进程同时可运行时，采用SJF调度算法可以得到最短的平均周转时间

  - 不公平

  源源不断的短任务可能使长任务得不到任何处理器时间（导致 饥饿）

  - 需要预测未来

最高相应比优先（HRRN）算法

- 是一个综合的算法
- 计算每个进程的响应比R
- 总是选择响应比最高的进程

抢占 or 不可抢占

响应比R = 作业周转时间 / 作业处理时间
= （作业处理时间+作业等待时间）/ 作业处理时间
= 1 + （作业等待时间 / 作业处理时间）

折衷权衡
tradeoff

交互式系统中采用的调度算法

- 轮转调度（RR-Round Robin）
- 优先级调度（HPF—Highest Priority First）
- 多级队列（Multiple queues）

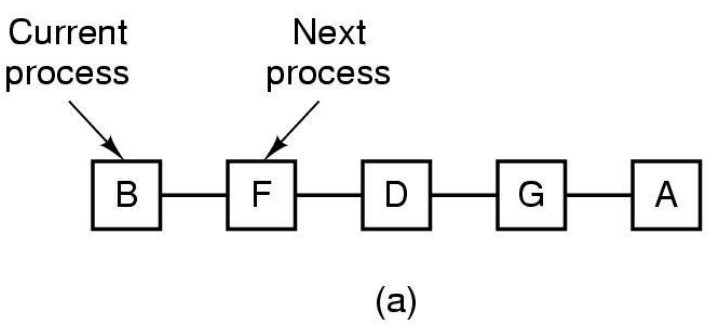
  与 多级反馈队列（Multiple feedback queue）
- 最短进程优先（Shortest Process Next）

抢占与
优先级
反转

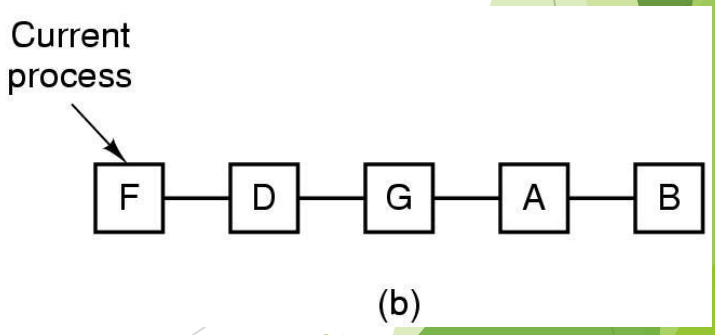
响应时间
均衡性
公平、平衡

时间片轮转调度算法(1/5)

- 解决问题的思路
  - 周期性任务切换
  - 每个进程分配一个时间片
  - 时钟中断 → 轮换
- 目标
  - 为短任务改善平均响应时间



可运行进程列表



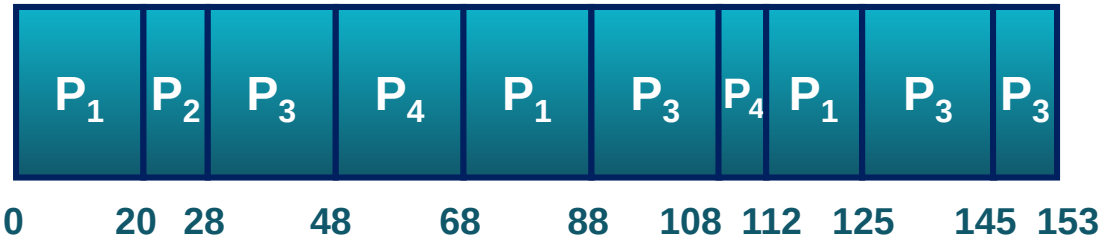
B用完自己的时间片后

时间片轮转调度算法(2/5)

示例:4个进程的执行时间如下

P1	53
P2	8
P3	68
P4	24

时间片为20的RR算法示例



等待时间 $P_1=(68-20)+(112-88)=72$

$P_2=(20-0)=20$

$P_3=(28-0)+(88-48)+(125-108)=85$

$P_4=(48-0)+(108-68)=88$

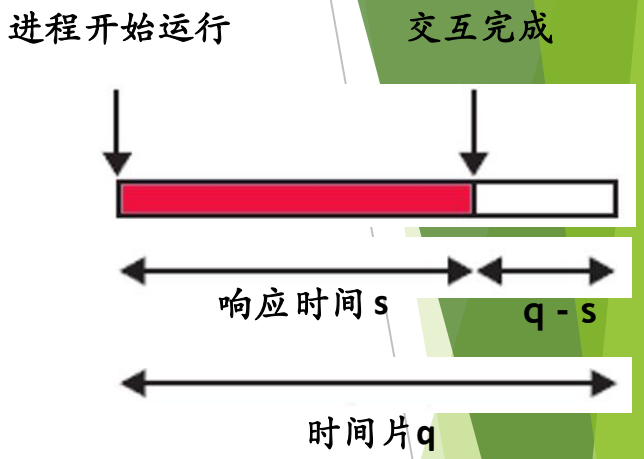
平均等待时间 = (72+20+85+88)/4=66.25

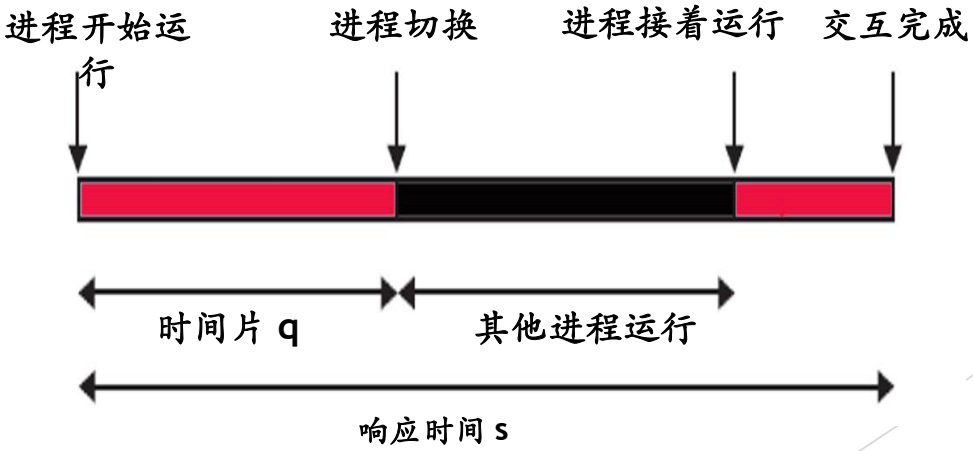
时间片轮转调度算法(3/5)

- 如何选择合适的时间片?

  太长--大于典型的交互时间
  - 降级为先来先服务算法
  - 延长短进程的响应时间

  太短--小于典型的交互时间
  - 进程切换浪费CPU时间





时间片轮转调度算法(4/5)

优缺点

- 公平
- 有利于交互式计算，响应时间快
- 由于进程切换，时间片轮转算法要花费较高的开销

假设时间片 10ms，如果进程切换花费0.1ms，CPU开销约占1%

进程运行时间	时间片	上下文切换次数
10	12	0
10	6	1
10	1	9

时间片轮转调度算法(5/5)

- 优缺点（续）
  - RR对不同大小的进程是有利的

    但是对于相同大小的进程呢?

- 两个进程A、B，运行时间均为100ms
- 时间片大小为1ms
- 上下文切换不耗时

假设

➢ 使用时间片轮转（RR）算法的平均完成时间?

199.5ms

ABABABAB…… …… …… ……A(199)B(200)

➢ 使用先来先服务（FCFS）算法呢?

150ms

FCFS和RR比较

■ 示例：4个进程的执行时间

P1如下	53
P2	8
P3	68
P4	24

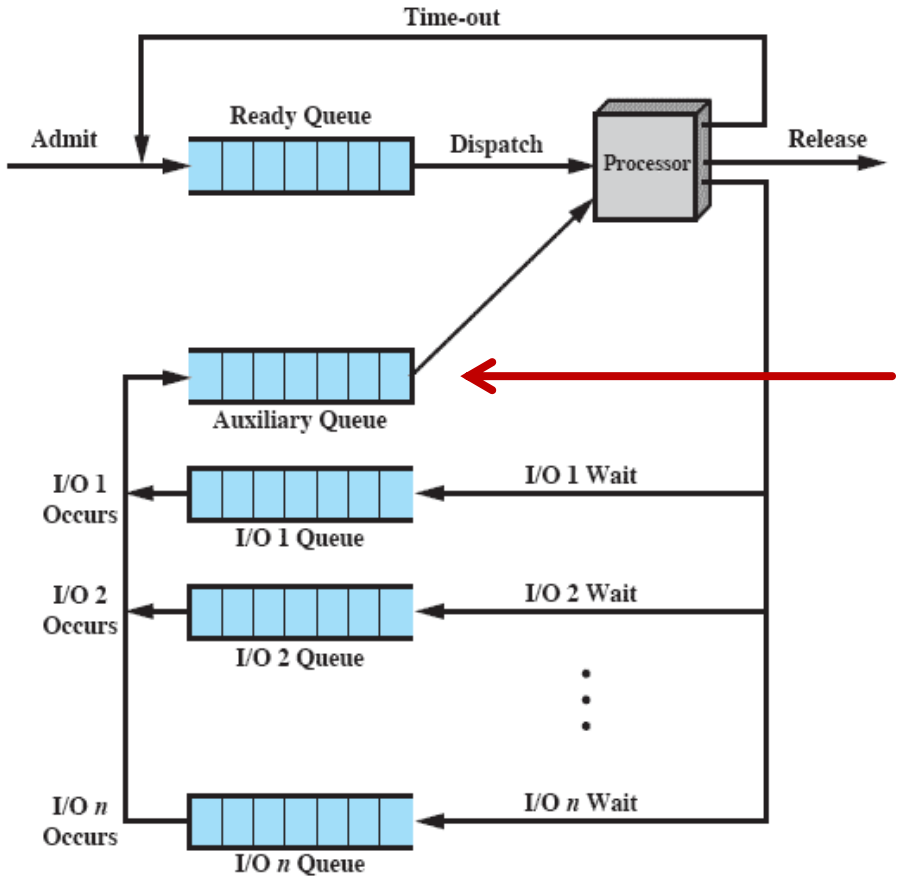
假设上下文切换时间为零

FCFS和RR各自的平均等待时间是多少？

时间片	P_1	P_2	P_3	P_4	平均等待时间
RR(q=1)	84	22	85	57	62
RR(q=5)	82	20	85	58	61.25
RR(q=8)	80	8	85	56	57.25
RR(q=10)	82	10	85	68	61.25
RR(q=20)	72	20	85	88	66.25
BestFCFS	32	0	85	8	31.25
WorstFCFS	68	145	0	121	83.5

虚拟轮转法(Virtual RR)

对计算密集型进程如何分配时间片?



Halder,S. etc.,1991

虚拟轮转调度的队列图

例子(1/4)

Process A

需要1000 ms 进行计算

Process B

需要1000 ms 进行计算

Process C

```
while(1) {
    使用处理器 1 ms
    使用I/O 10 ms
}
```

- C 可以使用磁盘的91%
- A 或者 B 每个都能使用处理器的100%
- 如果将它们一起运行会发生什么?
- 目标: 保持处理器和磁盘忙

例子(2/4)

- 先来先服务
  - 如果A或者B在C之前运行，它们将阻止C发出磁盘I/O请求大约
    - 多达 2000 ms
- 时间片轮转算法（时间片100ms）
  - CA--------B--------CA--------B--------...
  - 当A和B运行时，磁盘大多数时间处于空闲状态
    - 每 200ms 中大约10 ms时间进行磁盘操作

例子(3/4)

问题:
非常多的进程切换(以及进程切换的开销)

- 使用1ms的时间片
  - CABABABABABCABABABABABC...
- C频繁得到调度
  - 因此C可以在上次I/O请求执行完成后立即进行下次I/O操作
- 磁盘利用了接近90%的时间
- 对A或是B的性能几乎没有影响

例子(4/4)

- SRTN

  当C的I/O磁盘操作一结束就运行

  - 因为它具有最短的下次处理器突发
  - CA------CA----------CA-------- …

最高优先级调度算法

- 选择优先级最高的进程投入运行

- 通常：　系统进程优先级　高于　用户进程
  　　前台进程优先级　高于　后台进程
  　　操作系统更偏好　I/O型进程
- 优先级可以是静态不变的，也可以动态调整
  - 优先数可以决定优先级
- 就绪队列可以按照优先级组织
- 实现简单；不公平

饥饿
starvation

优先级反转问题(1/2)

基于优先级的抢占式

- Priority Inversion
- 又称：优先级反置、翻转、倒挂
- 现象

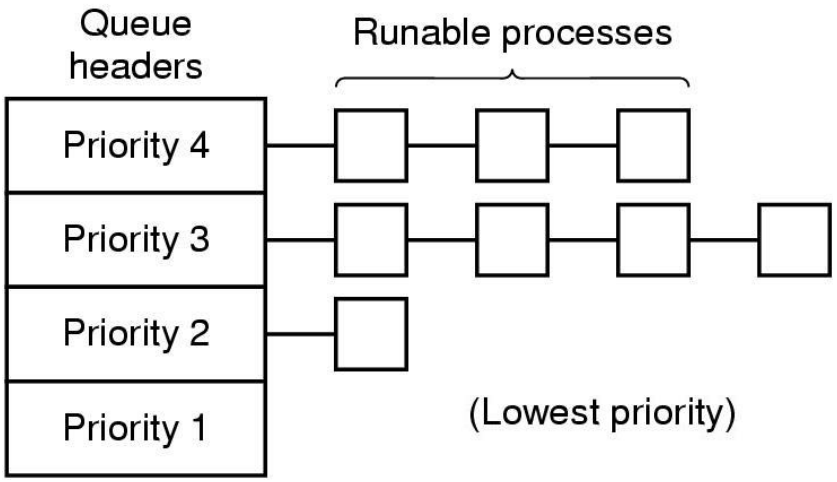
一个低优先级进程持有一个高优先级进程所需要的资源，使得高优先级进程等待低优先级进程运行

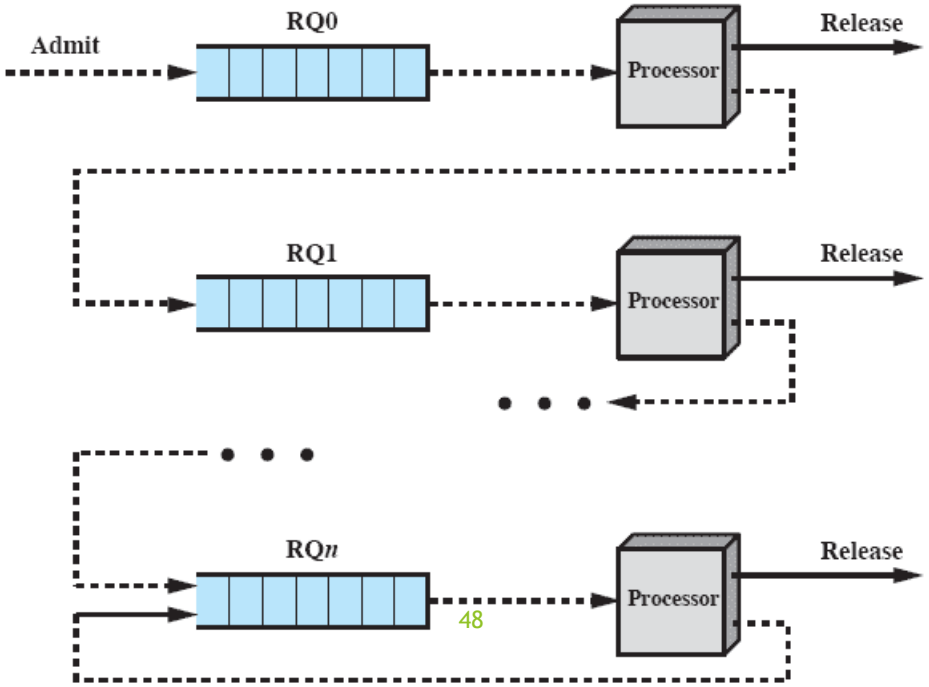
设H是高优先级进程，L是低优先级进程， M是中优先级进程（CPU型）
场景：L进入临界区执行，之后被抢占；
H也要进入临界区，失败，被阻塞；
M上CPU执行，L无法执行所以H也无法执行

优先级反转问题(2/2)

- 影响
  - 系统错误
  - 高优先级进程停滞不前，导致系统性能降低
- 解决方案
  - 设置优先级上限（优先级天花板协议 priority ceiling protocol）
  - 优先级继承
  - 使用中断禁止

多级队列调度算法





根据什么分队列?
排队方式?
哪些进程优先级高?
每一级队列调度策略是否相同?

多级反馈队列调度算法(1/3)

- Multilevel Feedback Queues
- 是UNIX的一个分支BSD（加州大学伯克利分校开发和发布的）5.3版所采用的调度算法
- 是一个综合调度算法

多级反馈队列调度算法(2/3)

- 设置多个就绪队列，第一级队列优先级最高
- 不同就绪队列中的进程分配的时间片长度不同

  第一级队列时间片最小；随着队列优先级别的降低，时间片增大
- 第一级队列为空时，在第二级队列调度，以此类推
- 各级队列按照*时间片轮转方式*进行调度
- 当一个新创建的进程就绪后，进入第一级队列
- 进程用完时间片而放弃CPU时，进入下一级就绪队列
- 由于阻塞而放弃CPU的进程进入相应的等待队列，一旦等待的事件发生，该进程回到原来一级就绪队列（？）

队首 or 队尾?

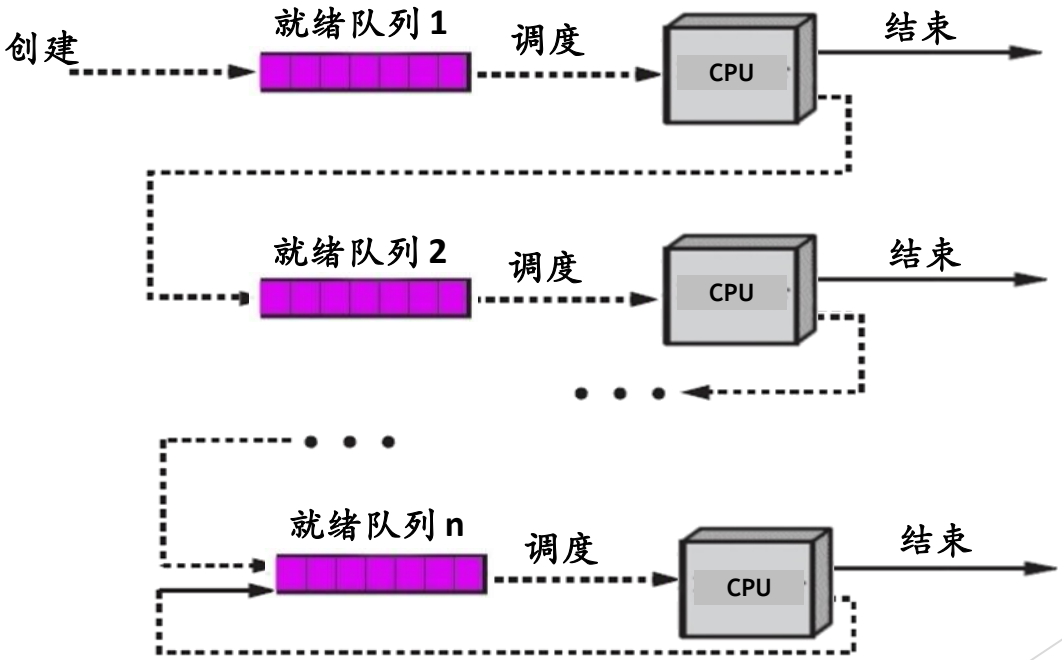
再次被调度上CPU时对时间片的处理?

非抢占式

多级反馈队列调度算法(3/3)

若允许抢占

- 当有一个优先级更高的进程就绪时，可以抢占CPU
  被抢占的进程回到原来一级就绪队列末尾（或者？）



队列模型示意

交互式系统中采用的其他调度算法

- 公平共享调度（Fair-share scheduling）
- 保证调度（Guaranteed scheduling）
- 彩票调度（Lottery scheduling）

各种调度算法的比较

调度算法	选择函数	决策模式	吞吐量	响应时间	开销	对进程的影响	饥饿问题
FCFS	max[w]	非抢占	不强调	可能很高，特别是当进程的执行时间差别很大时	最小	对短进程不利；对I/O密集型的进程不利	无
Round Robin	常数	抢占（时间片用完时）	如果时间片小，吞吐量会很低	为短进程提供好的响应时间	最小	公平对待	无
SJF	min[s]	非抢占	高	为短进程提供好的响应时间	可能较高	对长进程不利	可能
SRTN	min[s-e]	抢占（到达时）	高	提供好的响应时间	可能较高	对长进程不利	可能
HRRN	max((w+s)/s))	非抢占	高	提供好的响应时间	可能较高	很好的平衡	无
Feedback	见算法思想	抢占（时间片用完时）	不强调	不强调	可能较高	可能对I/O密集型的进程有利	可能

w：花费的等待时间；*e*：到现在为止，花费的执行时间；*s*：进程所需要的总服务时间，包括*e*

讨论

A long-established principle

- 机制与策略

  调度机制和调度策略分离的原则

  为什么?

  怎么做?

典型系统所采用的调度算法

- UNIX 动态优先数法
- 5.3BSD 多级反馈队列法
- Windows 基于优先级的抢占式多任务调度
- Linux 抢占式调度
- Solaris 综合调度算法

一个实例

Windows线程调度

Windows 线程调度

- 调度单位是线程
- 采用基于动态优先级的、抢占式调度，结合时间配额调整

- 就绪线程按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让其运行
- 同一优先级的各线程按时间片轮转进行调度
- 多处理机系统中允许多个线程并行运行

Windows 线程调度

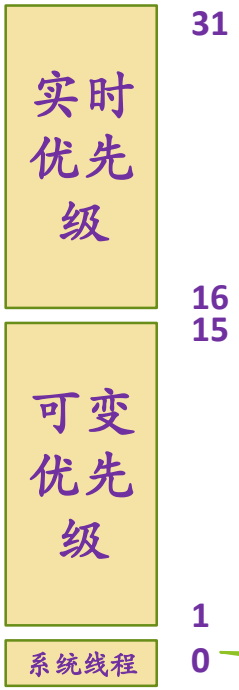
引发线程调度的条件：

- 一个线程的优先级改变了
- 一个线程改变了它的亲和(Affinity)处理机集合

- 线程正常终止 或 由于某种错误而终止
- 新线程创建 或 一个等待线程变成就绪
- 当一个线程从运行态进入阻塞态
- 当一个线程从运行态变为就绪态

线程优先级

- Windows使用32个线程优先级，分成三类



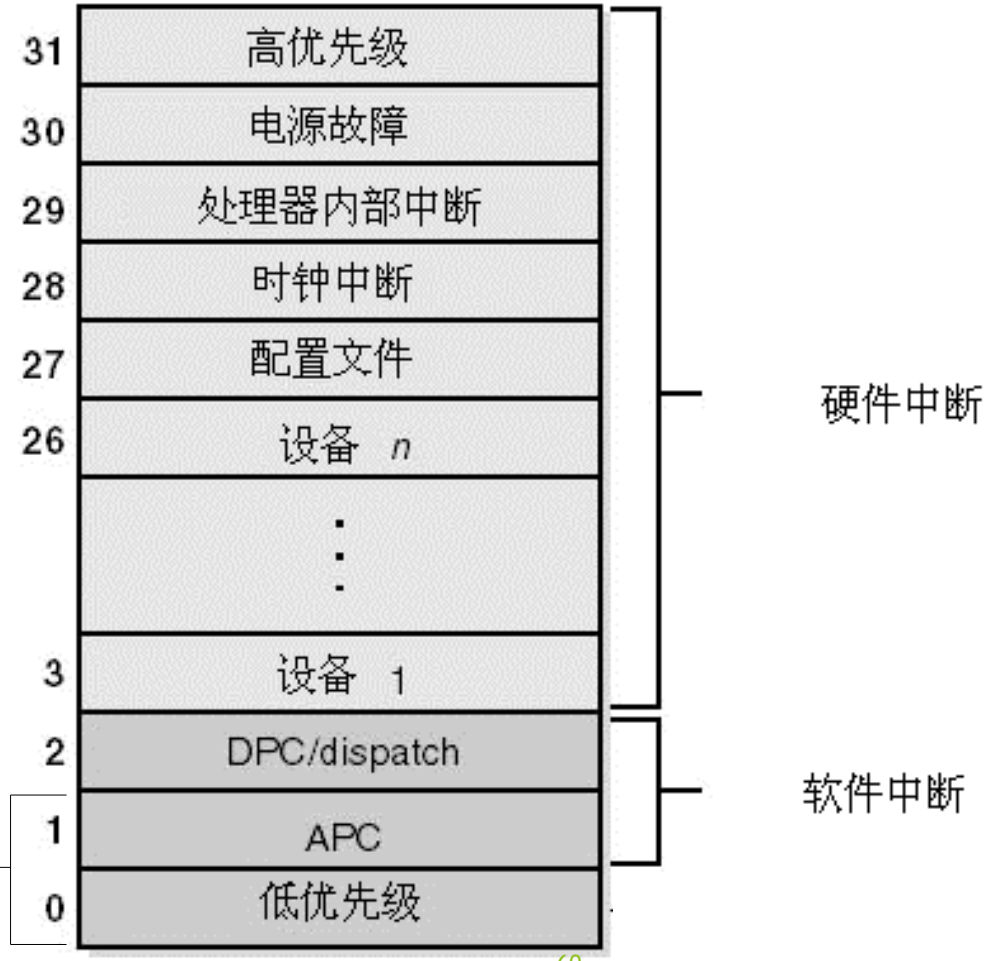
- 实时优先级线程不改变其优先级
- 可变优先级线程：其优先级可以在一定范围内升高或降低
  基本优先级 和 当前优先级

零页线程：用于对系统中空闲物理页面清零

中断优先级与线程优先级的关系

所有线程都运行在中断优先级0和1

用户态线程运行在中断优先级0，内核态的异步过程调用运行在中断优先级1



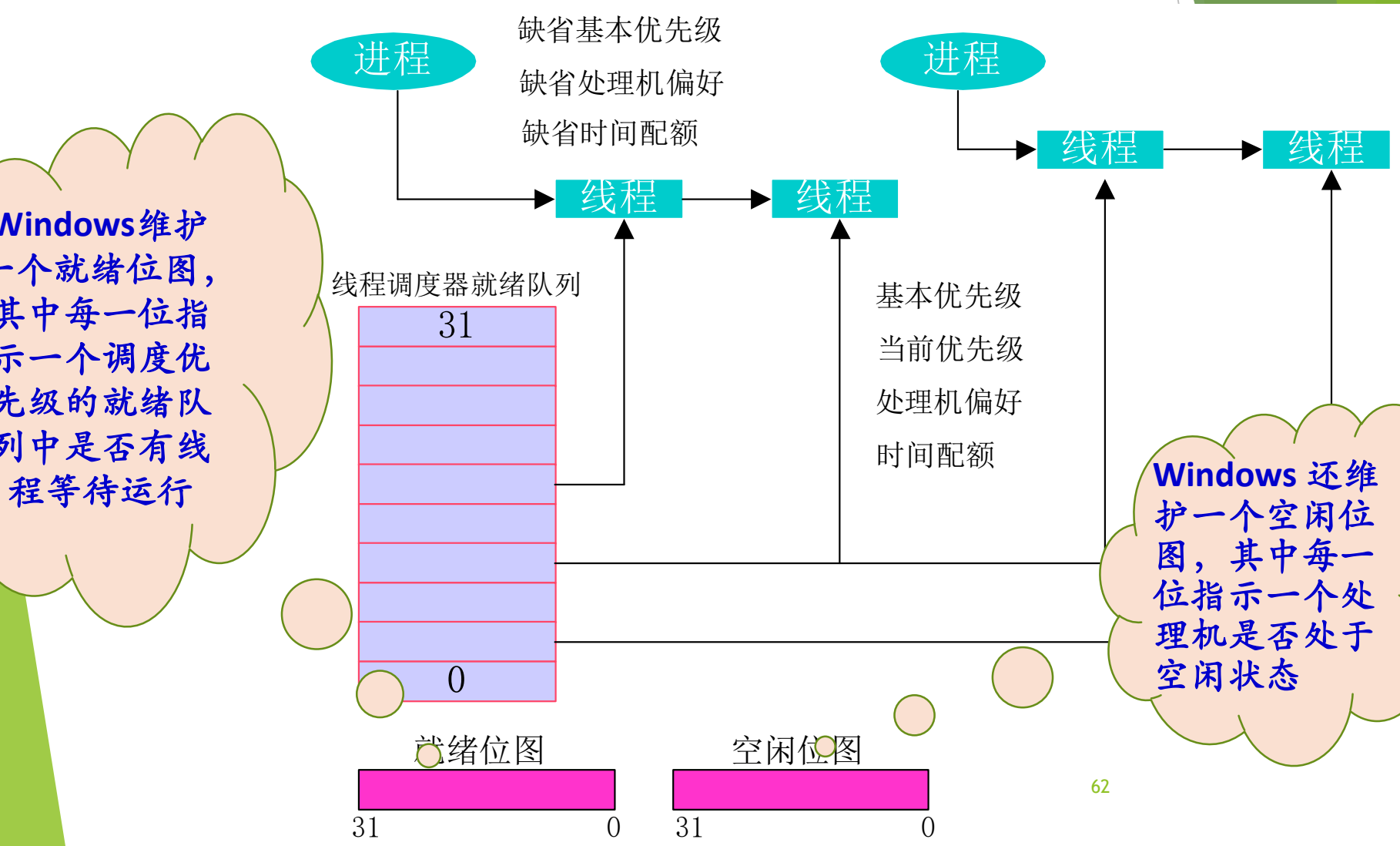
线程的时间配额

- 时间配额不是一个时间长度值，而一个称为**配额单位**(quantum unit)的整数
- KTHREAD：
  Quantum 和 QuantumReset
  #define CLOCK_QUANTUM_DECREMENT 3
  #define WAIT_QUANTUM_DECREMENT 1
- 一个线程用完了自己的时间配额时，如果没有其它相同优先级线程，Windows将重新给该线程**分配一个新的时间配额**，并继续运行

时间配额的一种作用

- 假设用户首先启动了一个运行时间很长的电子表格计算程序，然后切换到一个游戏程序(需要复杂图形计算并显示，CPU型)
- 如果前台的游戏进程提高它的优先级，则后台的电子表格将会几乎得不到CPU时间了
- 但增加游戏进程的时间配额，则不会停止执行电子表格计算，只是给游戏进程的CPU时间多一些而已

调度器数据结构

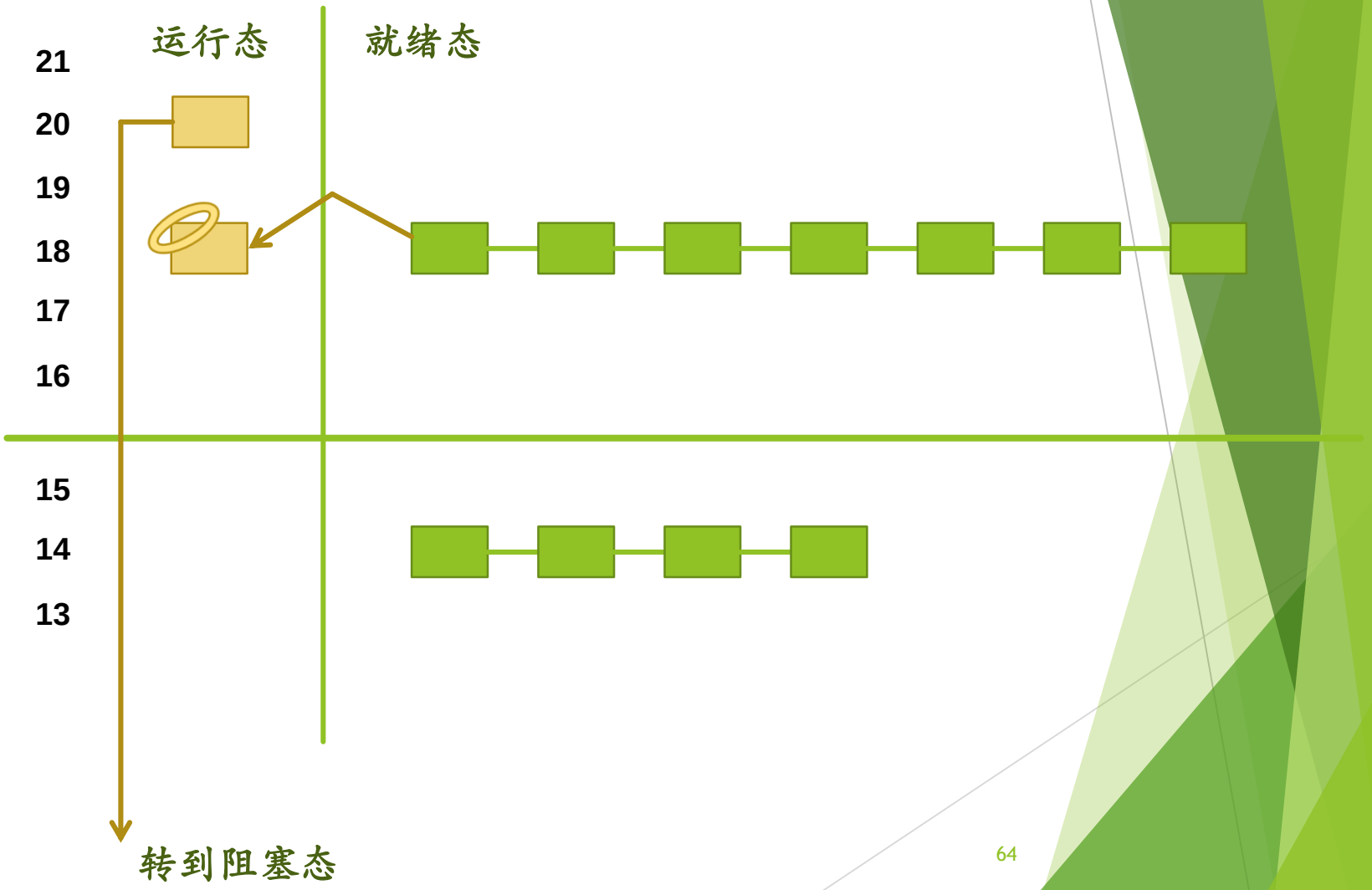

进程
缺省基本优先级
缺省处理机偏好
缺省时间配额
进程
线程
线程
线程
线程
线程调度器就绪队列
31
0
基本优先级
当前优先级
处理机偏好
时间配额
就绪位图
31
0
空闲位图
31
0
Windows维护一个就绪位图，其中每一位指示一个调度优先级的就绪队列中是否有线程等待运行
Windows 还维护一个空闲位图，其中每一位指示一个处理机是否处于空闲状态

调度策略

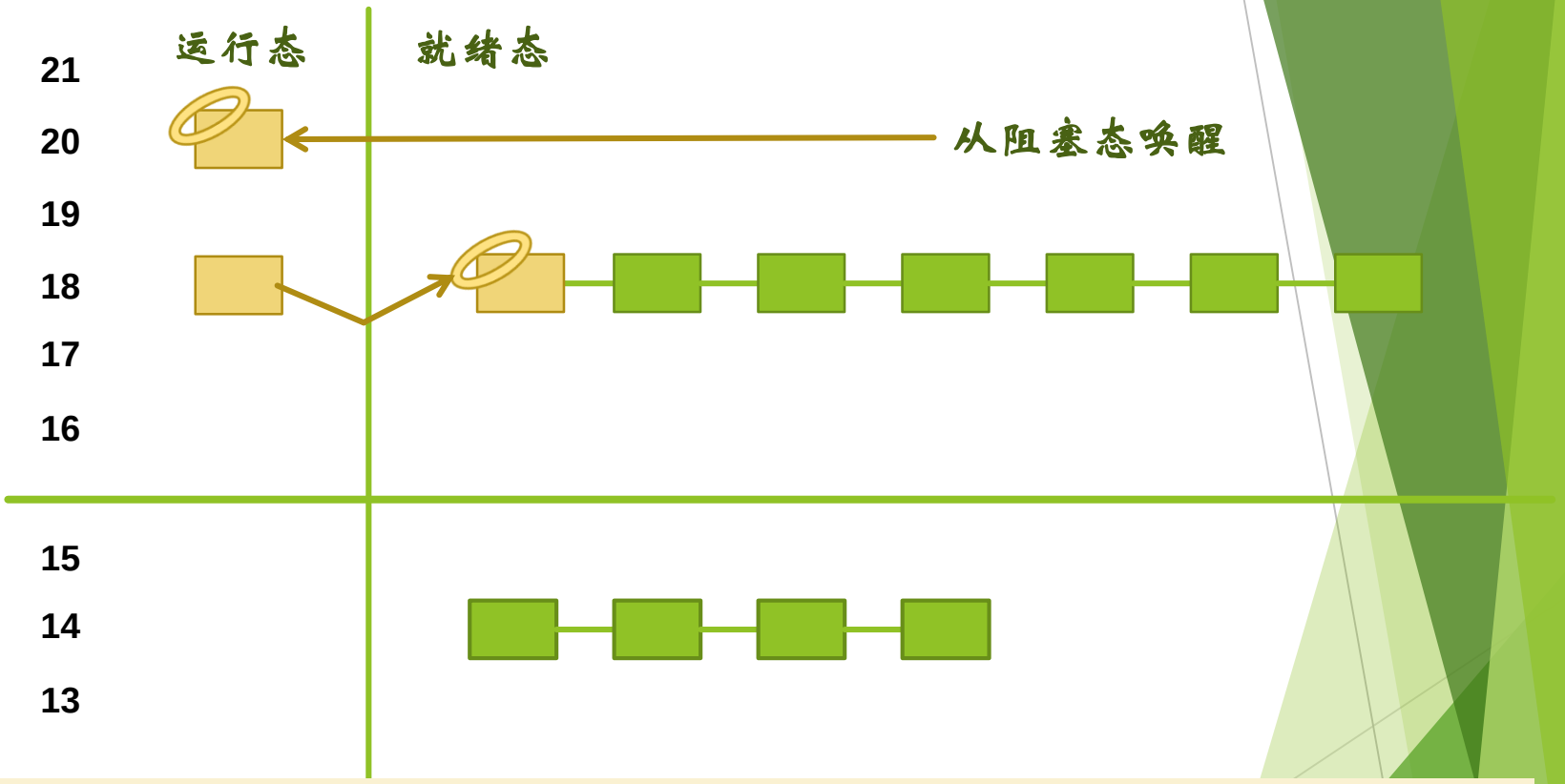
机制已确定

- 主动切换
- 抢占
- 时间配额用完

(1)主动切换



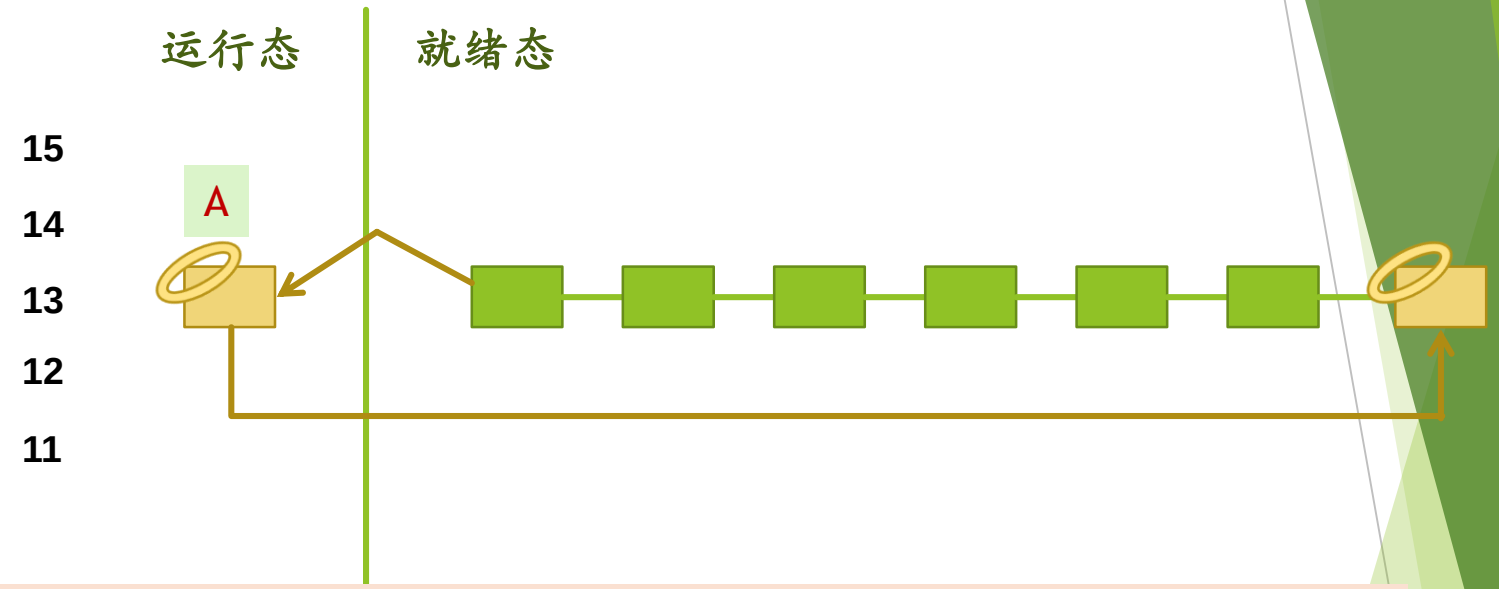
(2)抢占



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，重新得到处理机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

- 下列5种情况，Windows 会提升线程的当前优先级：
  - I/O操作完成
  - 信号量或事件等待结束
  - 前台进程中的线程完成一个等待操作
  - 由于窗口活动而唤醒窗口线程
  - 线程处于就绪态超过了一定的时间还没有运行——“饥饿”现象

针对可变优先级范围内(1至15)的线程优先级

I/O操作完成后的线程优先级提升

- 在完成I/O操作后，Windows 将临时提升等待该操作线程的优先级，以保证该线程能更快上CPU运行进行数据处理
- 线程优先级的实际提升值由设备驱动程序决定，提升建议值在文件“Wdm.h”或“Ntddk.h”中
- 线程优先级的提升幅度与I/O请求的响应时间要求是一致的，响应时间要求越高，优先级提升幅度越大
- 设备驱动程序在完成I/O请求时通过内核函数IoCompleteRequest来指定优先级提升的幅度
- 为了避免不公平，在I/O操作完成唤醒等待线程时将把该线程的时间配额减1

等待事件和信号量后的线程优先级提升

- 当一个等待事件对象或信号量对象的线程完成等待后，它的优先级将提升一个优先级
- 阻塞于事件或信号量的线程得到的处理器时间比CPU型线程要少，这种提升可减少这种不平衡带来的影响
- SetEvent、PulseEvent、ReleaseSemaphore等函数调用可导致事件对象或信号量对象等待的结束
- 提升是以线程的基本优先级为基准，提升后的优先级不会超过15
- 在等待结束时，线程的时间配额减1，并在提升后的优先级上执行完剩余的时间配额；随后降低1个优先级，运行一个新的时间配额，直到优先级降低到基本优先级

前台线程在等待结束后的优先级提升

- 对于前台进程中的线程，一个内核对象上的等待操作完成时，内核函数KiUnwaitThread会提升线程的当前优先级(不是线程的基本优先级)，提升幅度为变量PsPrioritySeparation的值
- 在前台应用完成它的等待操作时小幅提升它的优先级，以使它更有可能马上进入运行状态，有效改进前台应用的响应时间特征
- 用户不能禁止这种优先级提升，甚至是在用户已利用Win32的函数SetThreadPriorityBoost禁止了其他的优先级提升策略时，也是如此

窗口线程被唤醒后的优先级提升

- 这种优先级提升是为了改进交互应用的响应时间
- 窗口线程在被窗口活动唤醒(如收到窗口消息)时将得到一个幅度为2的额外优先级提升
- 窗口系统(Win32k.sys)在调用函数KeSetEvent时实施这种优先级提升，KeSetEvent函数调用设置一个事件，用于唤醒一个窗口线程

“饥饿”线程的优先级提升

- 系统线程“平衡集管理器(balance set manager)”每秒钟扫描一次就绪队列，发现其中存在的排队超过300个时钟中断间隔的线程
- 对这些线程，平衡集管理器将把该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
- 当被提升线程用完它的时间配额后，该线程的优先级立即衰减到它原来的基本优先级

空闲线程

- 如果在一个处理机上没有可运行的线程，Windows会调度相应处理机对应的空闲线程
- 由于在多处理机系统中可能两个处理机同时运行空闲线程，所以系统中的每个处理机都有一个对应的空闲线程
- Windows给空闲线程指定的线程优先级为0，该空闲线程只在没有其他线程要运行时才运行

空闲线程的功能

- 空闲线程的功能就是在一个循环中检测是否有要进行的工作

其基本的控制流程如下：

- 处理所有待处理的中断请求
- 检查是否有待处理的DPC请求。如果有，则清除相应软中断并执行DPC
- 检查是否有就绪线程可进入运行状态。如果有，调度相应线程进入运行状态
- 调用硬件抽象层的处理机空闲例程，执行相应的电源管理功能

对称多处理机系统上的线程调度

- 亲合关系
- 线程的首选处理机和第二处理机
- 就绪线程
- 为特定的处理机调度线程的运行处理机选择
- 最高优先级就绪线程可能不处于运行状态

调度算法的演化、当前最新版本

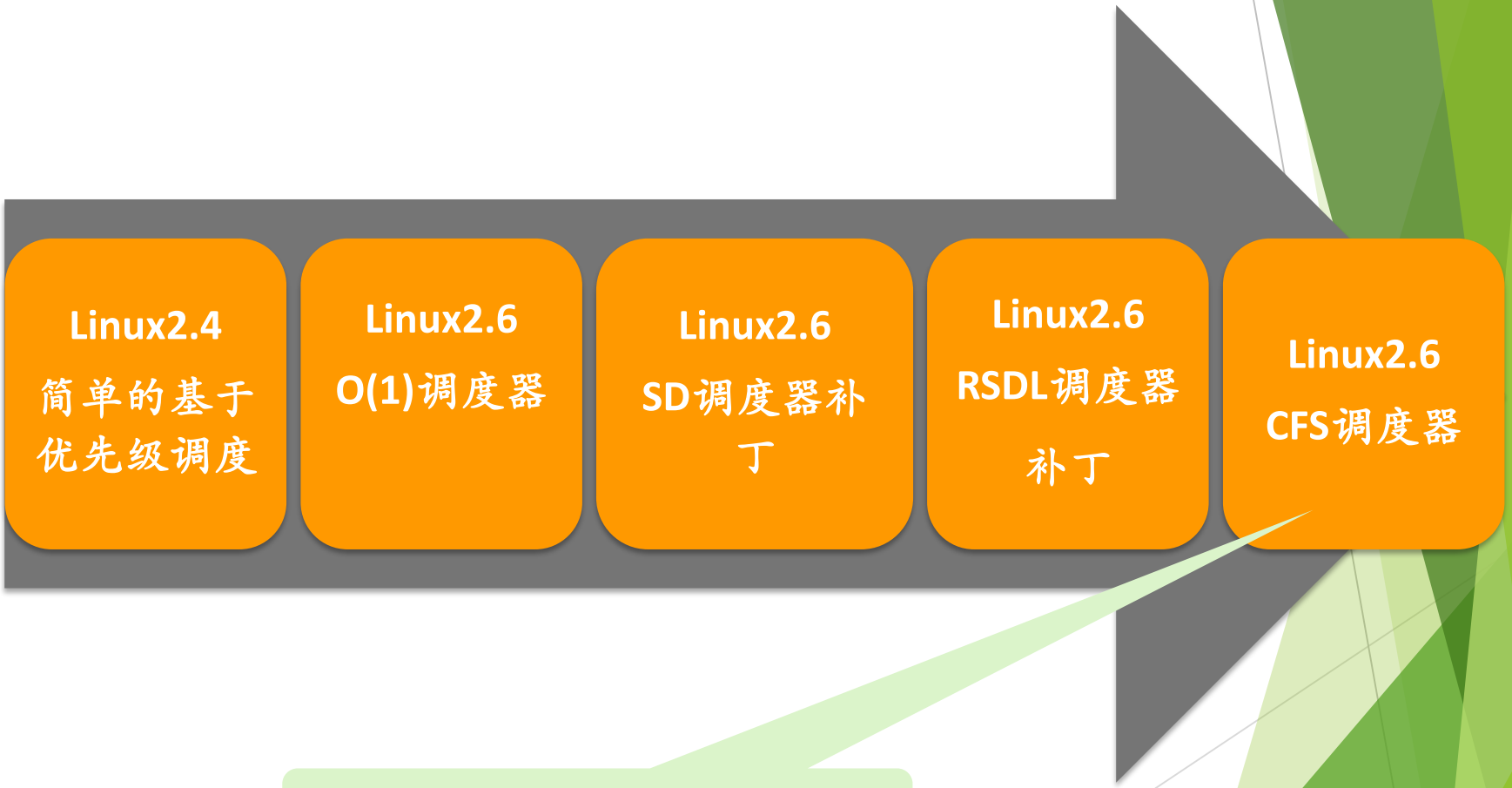
Linux调度算法的设计思想

Linux进程调度

Linux中 进程分类与调度机制

- 实时进程
  - 对调度延迟的要求最高，要求立即响应并执行
  - 调度策略：FIFO、Round Robin
- 普通进程
  - 交互式进程：间或处于睡眠态，对响应速度要求比较高
  - 批处理进程：在后台执行，能够忍受响应延迟
- 普通进程调度策略：CFS

Linux调度算法的演化历史

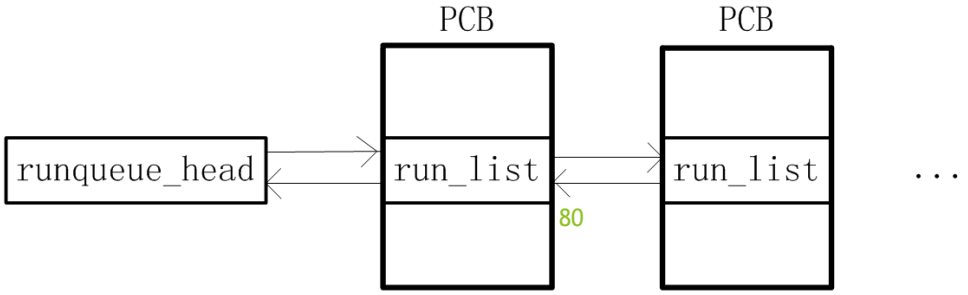


CFS：完全公平调度算法

Linux2.4的调度器（1）

RunQueue + 时间片 + 优先级

- 对 runqueue中所有进程的优先级依次进行比较，选择最高优先级的进程作为下一个被调度的进程
- 创建时进程赋予它一个时间片。时钟中断时递减当前运行进程的时间片，进程的时间片用完时，它必须等待重新赋予时间片才能有机会运行
- 只有当所有 RUNNING 进程的时间片都用完之后，才对所有进程重新分配时间片。这种设计保证了每个进程都有机会得到执行



Linux2.4的调度器（2）

时间片指的就是counter值，每个进程的可能不一样。而且实时进程的counter用完之后会立刻重置counter然后放入就绪队列，而非实时进程要等待runqueue中为空时，统一重新计算counter

counter（时间片）越大，优先级越高

- 普通进程的优先级主要由进程描述符中的 counter 字段决定(再要加上 nice 设定的静态优先级)
- nice从最初的UNIX沿用而来，表示进程的静态负向优先级，其取值范围为19~-20，以-20优先级最高
- 当所有 RUNNING 进程的时间片用完之后，调度器重新计算所有进程的 counter 值（不仅包括 RUNNING 进程，也包括处于睡眠状态的进程）
  - 睡眠状态进程的 counter 本来就没有用完，重新计算时，它们的 counter 值会加上这些原来未用完的部分，从而提高了它们的优先级
  - 系统通过这种方式提高交互式进程的相应速度

Linux2.4的调度器（3）

- 该调度算法的缺点
  - 可扩展性不好
  - 高负载系统上的调度性能比较低
  - 交互式进程的优化并不完善
  - 对实时进程的支持不够

- Ingo Molnar开发了新的O(1)调度器，以解决这些问题

O(1)调度器（1）

在两个方面修改了Linux2.4调度器：

- 进程优先级的计算方法
- pick next算法

O(1)调度器（2）

- 优先级计算
  - 普通进程优先级计算

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

    - 平均睡眠时间被用来判断进程是否是一个交互式进程

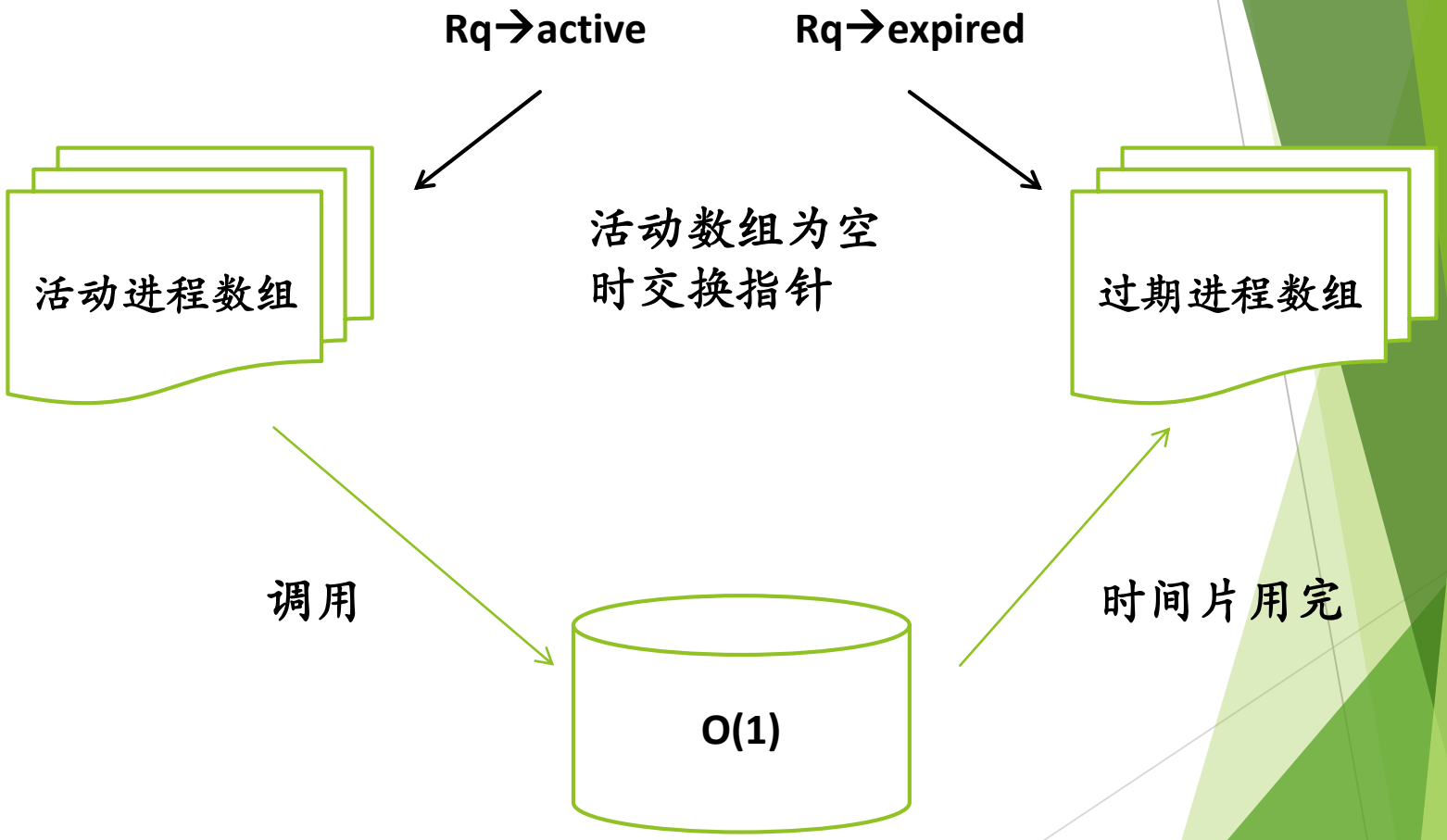
动态优先级 ≤ 3 x 静态优先级 /4 + 28

    - 平均睡眠时间越长，其bonus越大、优先级越高
  - 实时进程优先级计算
    - 实时进程的优先级不会动态修改，而且总是比普通进程的优先级高

O(1)调度器（3）

- pick next算法
- 调度器为每一个CPU维护了两个进程队列数组：active和expired。数组中的元素保存某一优先级的进程队列指针。当需要选择当前最高优先级的进程时，直接从active数组中选择当前最高优先级队列中的第一个进程
- 进程时间片为0时，调度器判断当前进程的类型，若是交互式进程或者实时进程，则重置其时间片并重新插入active数组；否则从active数组中移到expired数组。然而这些进程不能始终留在active数组中，当进程已经占用CPU时间超过一个固定值后，即使它是实时进程或者交互式进程也会被移到expired数组中
- 当active数组中的所有进程都被移到expired数组中后，调度器交换active数组和expired数组。进程被移入expired数组时，调度器会重置其时间片

O(1)调度器工作机制



O(1)调度器（4）

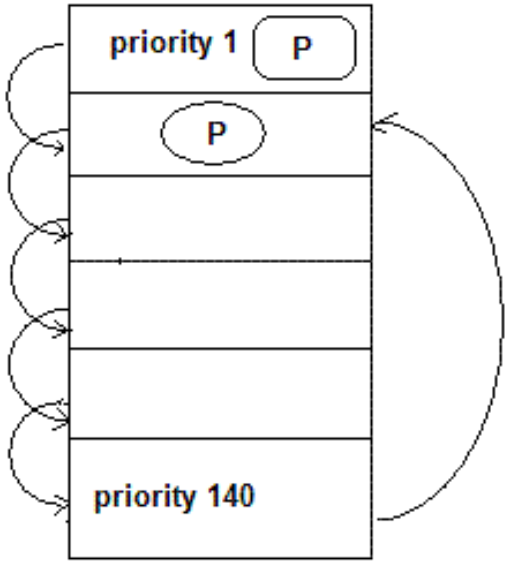
- O(1)调度器区分交互式进程和批处理进程的算法比以前虽有较大改进，但在很多情况下会仍然失效（一些典型的测试程序总会使该调度器的性能下降，导致交互式进程反应缓慢）
- O(1)调度器动态优先级的计算，调度器根据平均睡眠时间和一些经验公式（很难理解）来修正进程的优先级以及区分交互式进程，其代码很难阅读和维护
- Con Kolivas开发了楼梯调度算法SD，以及后来的改进版本RSDL

SD调度器（1）

- 楼梯调度（staircase scheduler）算法与O(1)算法很不同，它抛弃了动态优先级的概念，而采用了一种完全公平的思路
- 与O(1)算法一样，楼梯算法也为每一个优先级维护一个进程列表，并将这些列表组织在active数组中，当选取下一个要调度进程时，SD算法也同样从active数组中直接读取

SD调度器（2）

- 当进程用完了自己的时间片后，并不是被移到expired数组中，而是被加入active数组的低一优先级列表中，即将其降低一个级别（任务本身的优先级并没有改变）。当时间片再次用完，任务被再次放入更低一级优先级任务队列中
- 任务下到最低一级时，如果时间片再次用完，它会回到初始优先级的下一级任务队列中
  - 例如：任务本身优先级为P，当它从第N级台阶开始下楼梯并到达底部后，将回到第N+1级台阶，同时赋予该任务N+1倍的时间片



SD调度器（3）

- 楼梯算法能避免进程饥饿现象，高优先级的进程会最终和低优先级的进程竞争，使得低优先级进程最终获得执行机会
- 对于交互式应用，当进入睡眠状态时，与它同等优先级的其他进程将一步一步地下楼梯，进入低优先级进程队列。当该交互式进程被唤醒时，它还留在高处的楼梯层，从而能更快地被调度器选中，加快了响应时间
- 注意：实时进程还是采用原来的FIFO或者Round Robin调度策略

RSDL调度器（1）

- RSDL（The Rotating Staircase Deadline Schedule
- Con Kolivas开发，对SD算法的改进
- 其核心思想还是“完全公平”，没有复杂的动态优先级调整策略
- RSDL重新引入了expired数组
- 它为每一个优先级都分配了一个“组时间配额”，用Tg表示
- 同一优先级的每个进程都拥有同样的“时间配额”，用Tp表示

RSDL调度器（2）

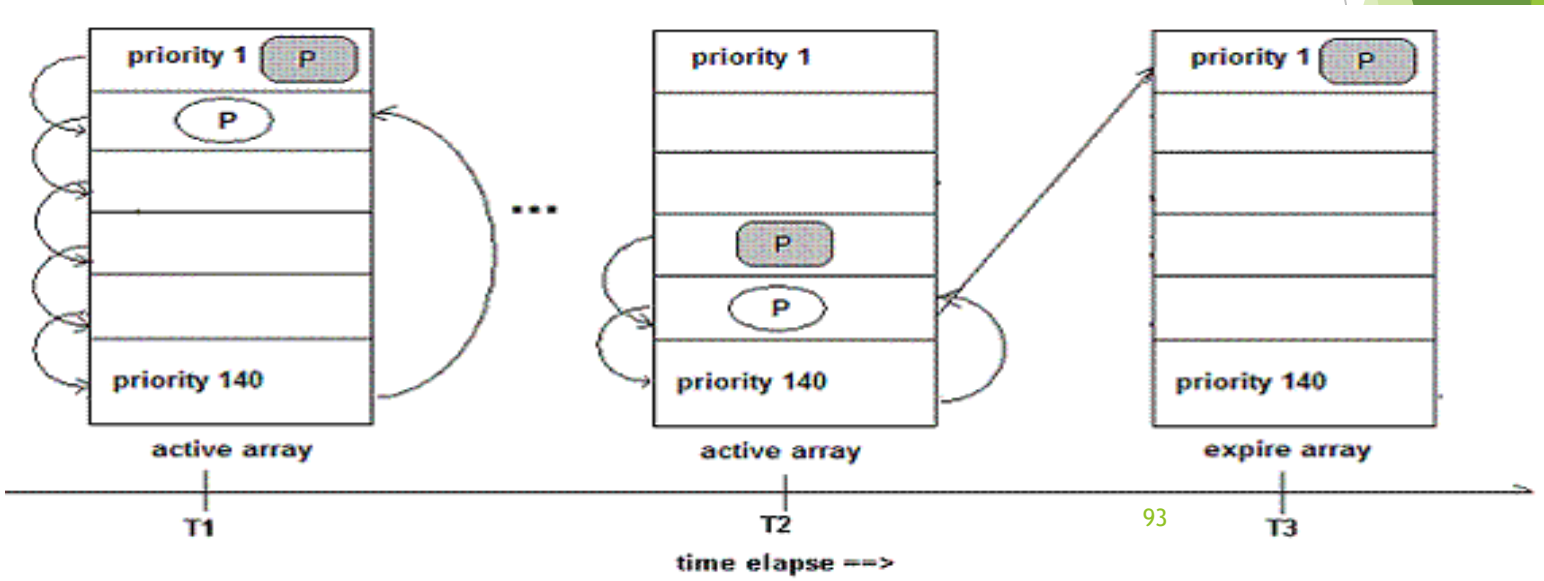
- 当进程用完了自身的Tp时，就下降到下一优先级进程组中（这个过程和SD相同），这一过程称为minor rotation

  注意：Tp与进程的时间片不同，时间片用完后进程直接进入expired数组，一般有Tp<time_slice

- 在SD算法中，低优先级进程必须等待所有的高优先级进程执行完才能获得CPU，因此低优先级进程的等待时间无法确定。而在RSDL中，当高优先级进程组用完了它们的Tg时，无论该组中是否还有进程的Tp尚未用完，所有属于该组的进程都被强制降低到下一优先级进程组中。这样低优先级任务就可以在一个可以预计的未来得到调度

RSDL调度器（3）

- 进程用完了自己的时间片time_slice时，将放入expired数组中它初始的优先级队列中
- 当active数组为空，或所有进程都降低到最低优先级时就会触发major rotation。Major rotation交换active数组和expired数组，所有进程都恢复到初始状态，重新开始minor rotation过程



CFS调度器

- CFS是现在被内核采纳的调度器。它从RSDL/SD中吸取了完全公平的思想，不再跟踪进程的睡眠时间，也不再企图区分交互式进程
- CFS算法中，每个进程都有一个“虚拟运行时间”表示该进程运行了“多长时间”，而调度器会选择虚拟运行时间最小的进程来运行
- 虚拟运行时间的计算与进程实际运行时间成正比，而与进程优先级成反比
- CFS以虚拟运行时间作为键值构造一棵红黑树，从而实现了快速更新和删除

CFS算法介绍

- CFS算法概述
- CFS重要数据结构
- 具体情境分析

CFS算法概述（1）

- 完全公平调度Completely Fair Scheduler
- 核心思想：根据进程的优先级按比例分配运行时间
- 分配给进程的运行时间

  = 调度周期 * 进程权重 / 所有进程权重之和

  (公式1)
- 调度周期：将所有处于TASK_RUNNING态进程都调度一遍的时间

CFS算法概述（2）

- CFS对时钟做抽象，引入了虚拟运行时间vruntime的概念，每个进程有自己的vruntime
- vruntime = 实际运行时间 * NICE_0_LOAD / 进程权重

(公式2)

- 由公式1和2得：

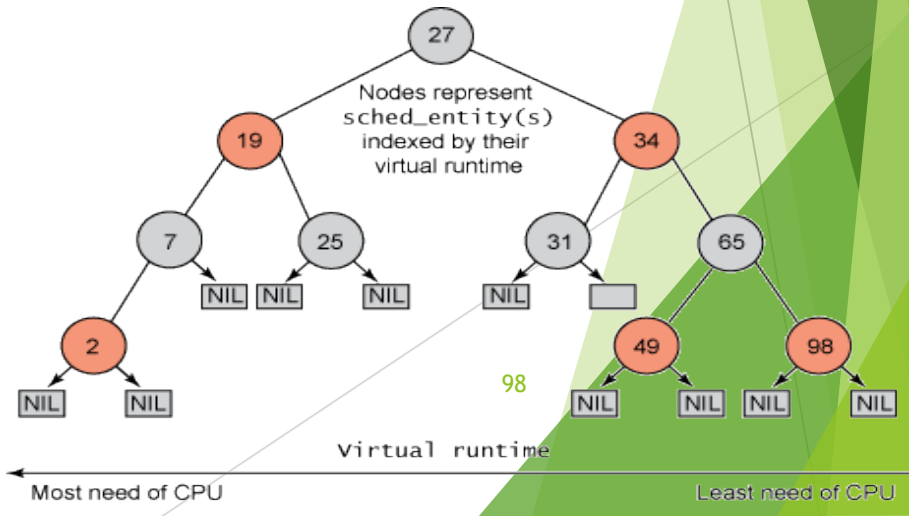
vruntime = (调度周期 * 进程权重 / 所有进程总权重) * NICE_0_LOAD / 进程权重

= 调度周期 * NICE_0_LOAD / 所有进程总权重

- 从vruntime的角度，分配给所有进程的时间是一样的

CFS算法概述（3）

- CFS算法，每次选择vruntime最小的进程运行，所有进程的vruntime增长速度宏观上看是同时推进的
- CFS 维护了一个以vruntime为顺序的红黑树，可以快速高效地插入或删除任务
- 调度器每次选择最左侧结点的进程运行，运行结点从树中删除；进程切换时，切换下来的就绪态进程再重新插入树中，因为换下来的进程一般vruntime比较大所以会靠近树的右侧；总体来说树的内容从右侧迁移到左侧以保持平衡



CFS重要数据结构（1）

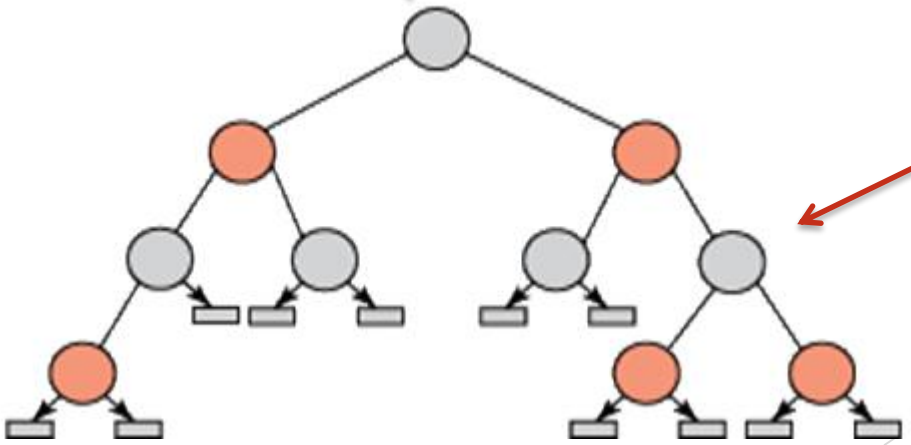
- Linux 内的所有任务都由称为 task_struct 的任务结构表示，该结构（以及其他相关内容）完整地描述了任务并包括了任务的当前状态、其堆栈、进程标识、优先级（静态和动态）等
- task_struct通过sched_entity和sched_class分别来定义调度实体和调度类

```
struct task_struct{
        volative long state;
        void *stack;
        unsigned int flags;
        int prio,static_prio normal_prio;
        const struct sched_class
        *sched_class;
        struct sched_entity se;
        …
};
```

```
struct sched_entity{
        Struct load_weight load;
        struct rb_node run_node;
        struct list_head group_node;
        …
};
```

```
struct cfs_rq{
        …
        struct root tasks_timeline;
        …
};
```

```
struct rb_node{
        unsigned long rb_parent_color;
        struct rb_node *rb_right;
        struct rb_node *rb_left;
        …
};
```



CFS重要数据结构（2）

- struct sched_entity结构包含了完整的信息，用于实现对单个任务或任务组的调度
- struct sched_class该调度类类似于一个模块链，协助内核调度程序工作。每个调度程序模块需要实现struct sched_class 建议的一组函数
- struct cfs_rq对于每个运行队列，都提供了一个结构来保存相关红黑树的信息

讨论几个问题

- 新进程的vruntime的初值是不是0?
- 休眠进程的vruntime一直保持不变吗?
- 休眠进程在唤醒时会立刻抢占CPU吗?
- 进程从一个CPU迁移到另一个CPU上的时候vruntime会不会变?

具体情景分析

- 时钟中断
- 主动调度
- 进程唤醒
- 进程创建

情景分析——时钟中断

- 更新进程运行信息
  - 将进程的这一段的实际运行时间换算为vruntime，然后加在进程原来的vruntime上
- 判断是否满足抢占条件
  - 如果当前进程这一段实际运行的时间超过了一个调度周期内分配给它的时间，则做标记，随后进行主动进行一次进程调度
  - 否则继续运行当前进程

情景分析——主动调度（1）

- 更新当前进程运行信息
- 将当前进程插入红黑树
- 选择要运行的进程
- 将该进程移出红黑树

情景分析——主动调度（2）

- 在“选择要运行的进程”这一步，并不是直接选取红黑树最左侧的结点就算完结了
- cfs_rq有两个指针与进程调度策略有很大关系，last和next。如果这两个指针不为NULL，那么last指向最近被调度出去的进程，next指向被调度上cpu的进程

例如：A正在运行，被B抢占，那么last指向A，next指向B。在选择下一个调度进程时会优先选择next，次优先选择last，选择完后，就清空这两个指针

情景分析——主动调度（3）

以上介绍的优先选择的实现：

- wakeup_preempt_entity(cfs_rq->next, se)
- wakeup_preempt_entity(cfs_rq->last, se)
- 函数返回-1表示新进程vruntime大于当前进程，不能抢占；返回0表示虽然新进程vruntime比当前进程小，但是没有小到调度粒度，一般也不能抢占；返回1表示新进程vruntime比当前进程小的超过了调度粒度，可以抢占

情景分析——进程创建（1）

- 更新新进程的vruntime值
  - 以cfs队列的min_vruntime为基准，再加上进程在一次调度周期中能够增加的vruntime（相当于假设新进程在这一轮调度中已经运行过了）
  - 新进程的vruntime确定之后有一个判断，满足特定条件时，交换父子进程的vruntime
- 将新建进程插入红黑树
- 判断当前进程能否被新建进程抢占
  - 调用wakeup_preempt_entity
  - 如果满足抢占条件则标记，之后进行调度

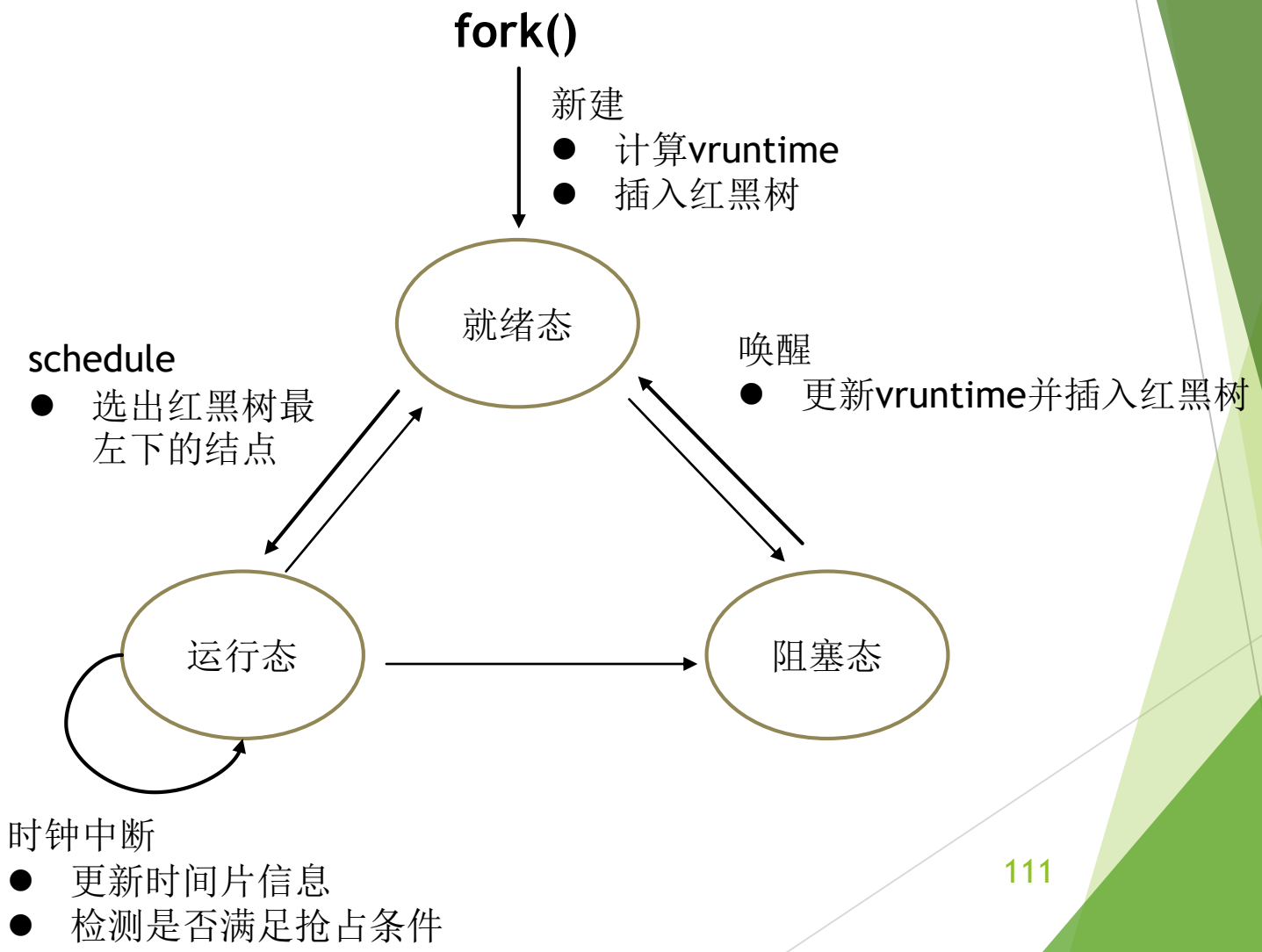
情景分析——进程创建（2）

- 解释下min_vruntime，这是每个cfs队列一个的变量，它一般小于等于所有就绪态进程的最小vruntime，min_vruntime体现了目前所有进程整体虚拟运行时间的下限
- 同时满足以下几个条件时，交换父子进程的vruntime：
  - sysctl设置了子进程优先运行
  - fork出的子进程与父进程在同一个cpu上
  - 父进程不为空
  - 父进程的vruntime小于子进程的vruntime

情景分析——进程唤醒

- 调整唤醒进程的vruntime
  - 用min_vruntime减去一个值，这是对睡眠进程做一个补偿，能让它醒来时可以快速得到CPU
  - 但是有些进程只睡眠很短时间，这样在它醒来后vruntime还是大于min_vruntime，不能让进程通过睡眠获得额外的运行时间，所以最后选择计算出的补偿时间与进程原本vruntime中的较大者
- 将唤醒进程插入红黑树
- 判断是否满足调度条件

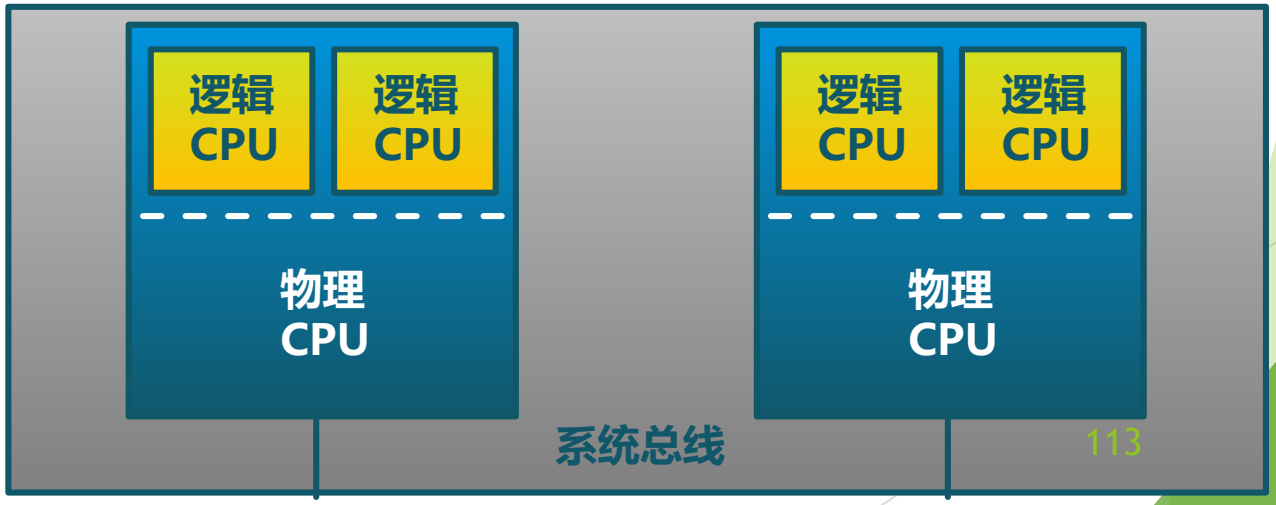
CFS与进程状态转换


fork()
新建
计算vruntime
插入红黑树
就绪态
schedule
选出红黑树最左下的结点
唤醒
更新vruntime并插入红黑树
运行态
阻塞态
时钟中断
更新时间片信息
检测是否满足抢占条件

多处理器调度

多处理器调度

- 多处理器调度的特征
  - ✓ 多个处理器组成一个多处理机系统
  - ✓ 处理器间可负载共享
- 对称多处理器(SMP, Symmetric multiprocessing)调度
  - ✓ 每个处理器运行自己的调度程序
  - ✓ 调度程序对共享资源的访问需要进行同步



Acknowledgement: 清华大学 向勇、陈渝

多处理器调度算法设计

- 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
- 要考虑进程在多个CPU之间迁移时的开销
  - 高速缓存失效、TLB失效
  - 尽可能使进程总是在同一个CPU上执行
    - 如果每个进程可以调度到所有CPU上，假如进程上次在CPU1上执行，本次被调度到CPU2，则会增加高速缓存失效、TLB失效；如果每个进程尽量调度到指定的CPU上，各种失效就会减少
- 考虑负载均衡问题

对称多处理器的进程分配

* 静态进程分配
  * 进程从开始到结束都被分配到一个固定的处理机上执行
  * 每个处理机有自己的就绪队列
  * 调度开销小
  * 各处理机可能忙闲不均

* 动态进程分配
  * 进程在执行中可分配到任意空闲处理机执行
  * 所有处理机共享一个公共的就绪队列
  * 调度开销大
  * 各处理机的负载是均衡的

Acknowledgement: 清华大学 向勇、陈渝

重点小结

- 涉及调度相关的基本概念
  - 调度的层次
  - 进程的行为
  - 调度算法的设计目标
- 典型的调度算法（设计思想及应用）
  - Windows线程调度算法
  - Linux进程调度算法
  - FCFS、RR、SPN、SRT、HRRN、Feedback
- 调度的时机
- 进程切换过程

作业4(1)

1、为什么区分CPU密集型程序和I/O密集型程序对调度程序是重要的?

2、讨论下列几对调度准则在某些情况下如何冲突:

（a）CPU利用率和相应时间

（b）平均周转时间和最大等待时间

（c）I/O设备利用率和CPU利用率

3、对某系统进行监测后发现，在阻塞I/O之前，平均每个进程运行时间为T。一次进程切换需要的时间为S，这里S实际上就是开销。对于采用时间片长度为Q的轮转调度，请给出以下各种情况的CPU利用率的计算公式：（a）Q=∞（b）Q>T（c）S<Q<T（d）Q=S（e）Q趋近于0

作业4(2)

4、轮转调度程序的一个变种是回归轮转（regressive round-robin）调度程序。这个调度程序为每个进程分配时间片和优先级。时间片的初值为50ms。然而，如果一个进程获得CPU并获得它的整个时间片（不会因I/O而阻塞），那么它的时间片会增加10ms并且它的优先级会提升。（进程的时间片可以增加到最多100ms。）如果一个进程在用完它的整个时间片之前阻塞，那么它的时间片会降低5ms而它的优先级不变。回归轮状调度程序会偏爱哪类进程（CPU密集型的或I/O密集型的）？请解释。

作业4(3)

5、阅读《现代操作系统》2.4.4，回答下列问题。

（1）一个实时系统有2个周期为5ms的电话任务，每次任务的CPU时间是1ms；还有一个周期为33ms的视频流任务，每次任务的CPU时间是11ms。试问这个系统是可调度的吗？为什么？如果再加入一个视频流，系统还是可调度的吗?

（2）一个软实时系统有4个周期时间，其周期分别为50ms，100ms，200ms和250ms。假设这4个事件分别需要35ms，20ms，10ms和x ms的CPU时间。保持系统可调度的最大x值是多少?

作业提交时间：
2020年11月6日 晚23:30

XV6源代码阅读要求

阅读XV源代码之进程调度部分，结合代码写出总结报告。

提交时间：2020年11月6日晚23:30

Thanks

The End