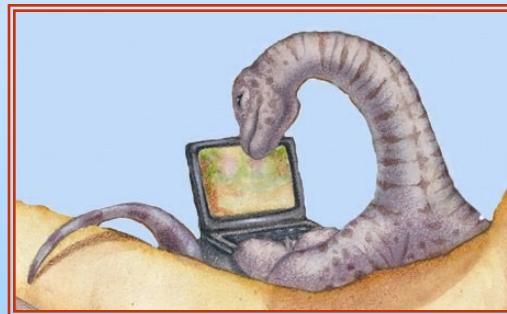


# Chap 5 CPU调度





# 内容

1. 基本概念
2. 调度准则
3. 调度算法
4. 多处理器调度和线程调度
5. 调度实例

# 1、基本概念





# CPU调度概述

1. 长程和短程调度
2. 调度队列
3. 中程调度
4. CPU脉冲周期
5. CPU调度方式
6. 调度过程和时机
7. 调度准则





# CPU调度

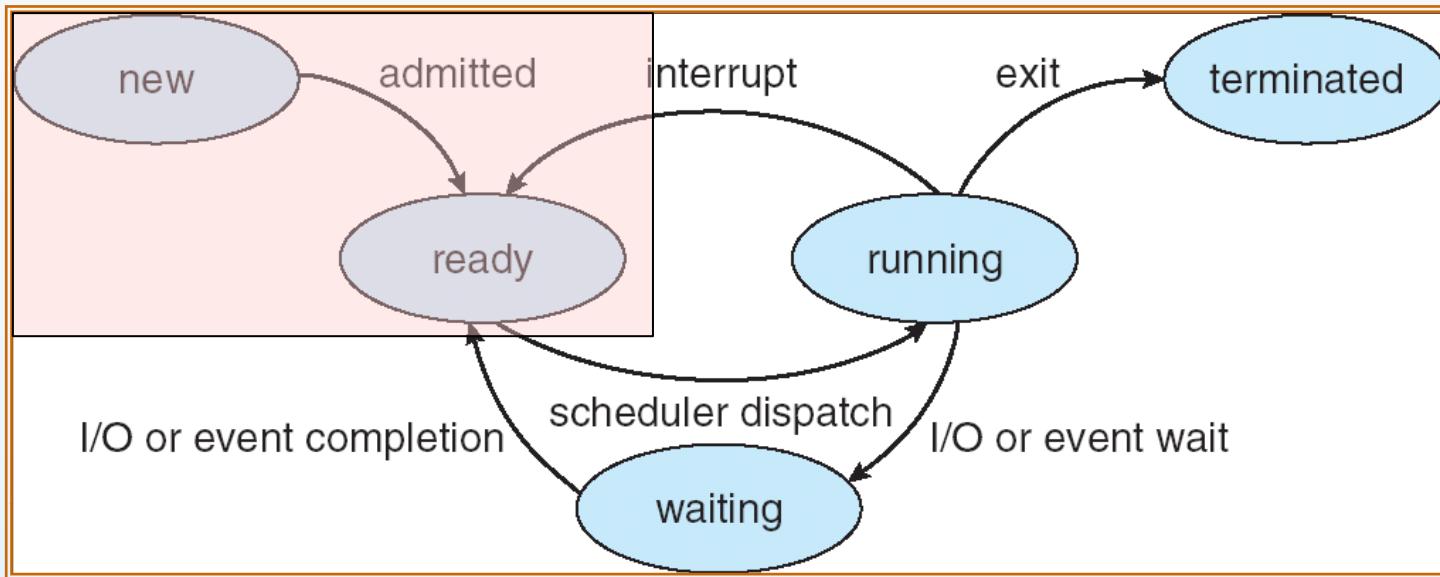
1. 长程调度
2. 中程调度
3. 短程调度





# 长程调度

- “道”：允许在内存中运行的最多进程数



每个用户创建进程的初始状态是“新建”，处于新建状态的进程一般首先被放到外存的进程池中，当内存进程的数量没有达到最多进程数时，操作系统的调度程序才从新建状态选择一个进入内存并转换为就绪状态

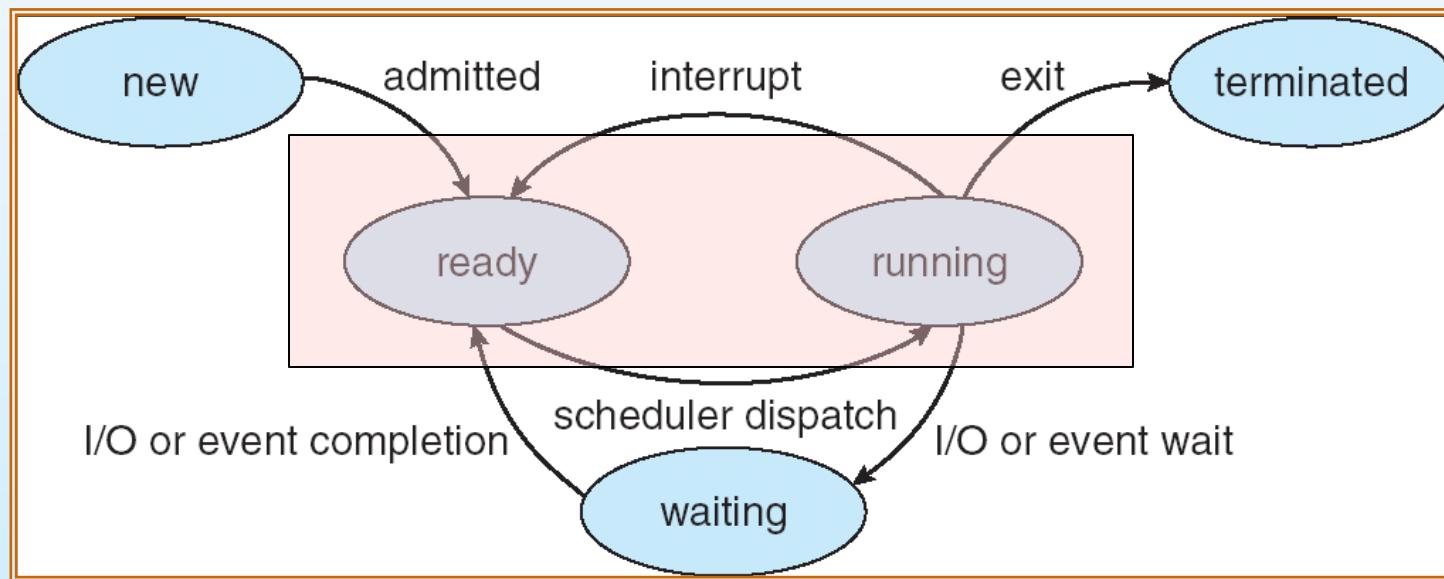
从新建状态转换到就绪状态的操作就是长程调度，又称为作业调度或高级调度





# 短程调度

- 又称为**CPU**调度或低级调度
- 在就绪队列中可能存在不止一个进程，当**CPU**空闲时，操作系统就需要从就绪队列中挑选一个进程让它运行





# 长程调度vs短程调度

	切换频率	切换开销	操作系统中
短程调度	频率高、速度快	开销小	必需
长程调度	频率低、速度慢	开销大	可选



每个进程在其生命周期中只有一次长程调度，而有成千上万次短程调度



长程调度需要把进程的代码和数据调入到内存中，这些I/O操作很耗时





# 中程调度

- 又称为交换
- 从本质上讲，中程调度不属于进程管理的内容，而应该属于内存管理
- 简而言之，一个进程在内存和外存间的换进换出，最大的目的是节省内存
- 当一个进程在内存中长期不运行时，会造成内存浪费
- 为此，操作系统把这些进程从内存换到外存，从而腾出内存空间供运行进程使用
- 当一个在外存的进程接下来需要运行时，操作系统则执行换入操作，把这个进程从外存换入内存





■ 控制内存中运行进程的数量的调度是（）

- A. 长程调度
- B. 短程调度
- C. 中程调度
- D. 进程调度





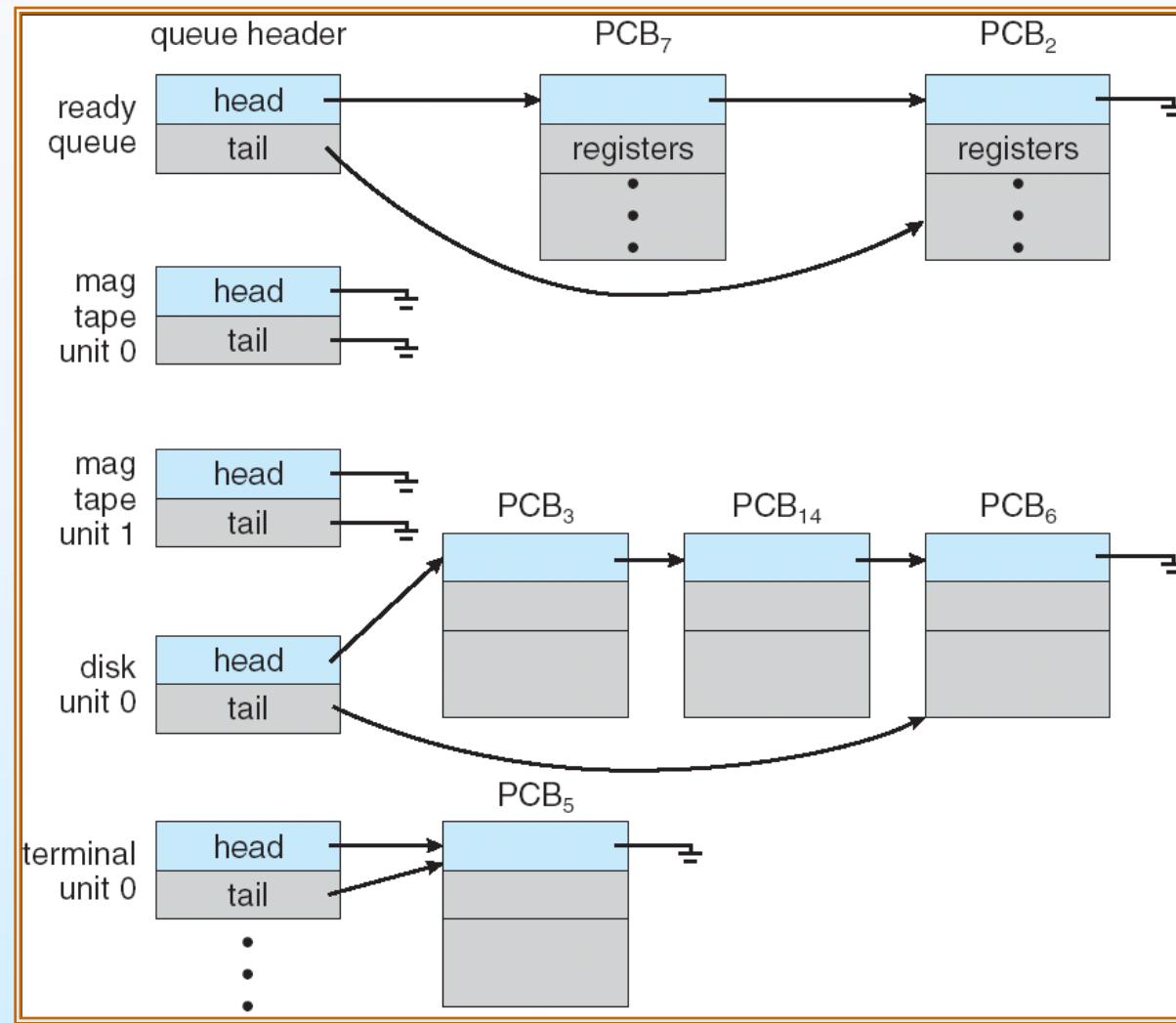
# 进程调度队列

- 为了方便进行CPU调度，操作系统需要对不同状态的进程进行组织和管理。为此操作系统为某些特定的状态设立了一个或多个进程队列，用于管理这些进程。
- 作业队列 (**job queue**) - 在系统中的所有进程的集合
- 就绪队列 (**ready queue**) - 在主内存中的，就绪并等待执行的所有进程的集合
- 设备队列 (**device queue**) - 等待某一I/O设备的进程队列
- 在各种队列之间进程的迁移



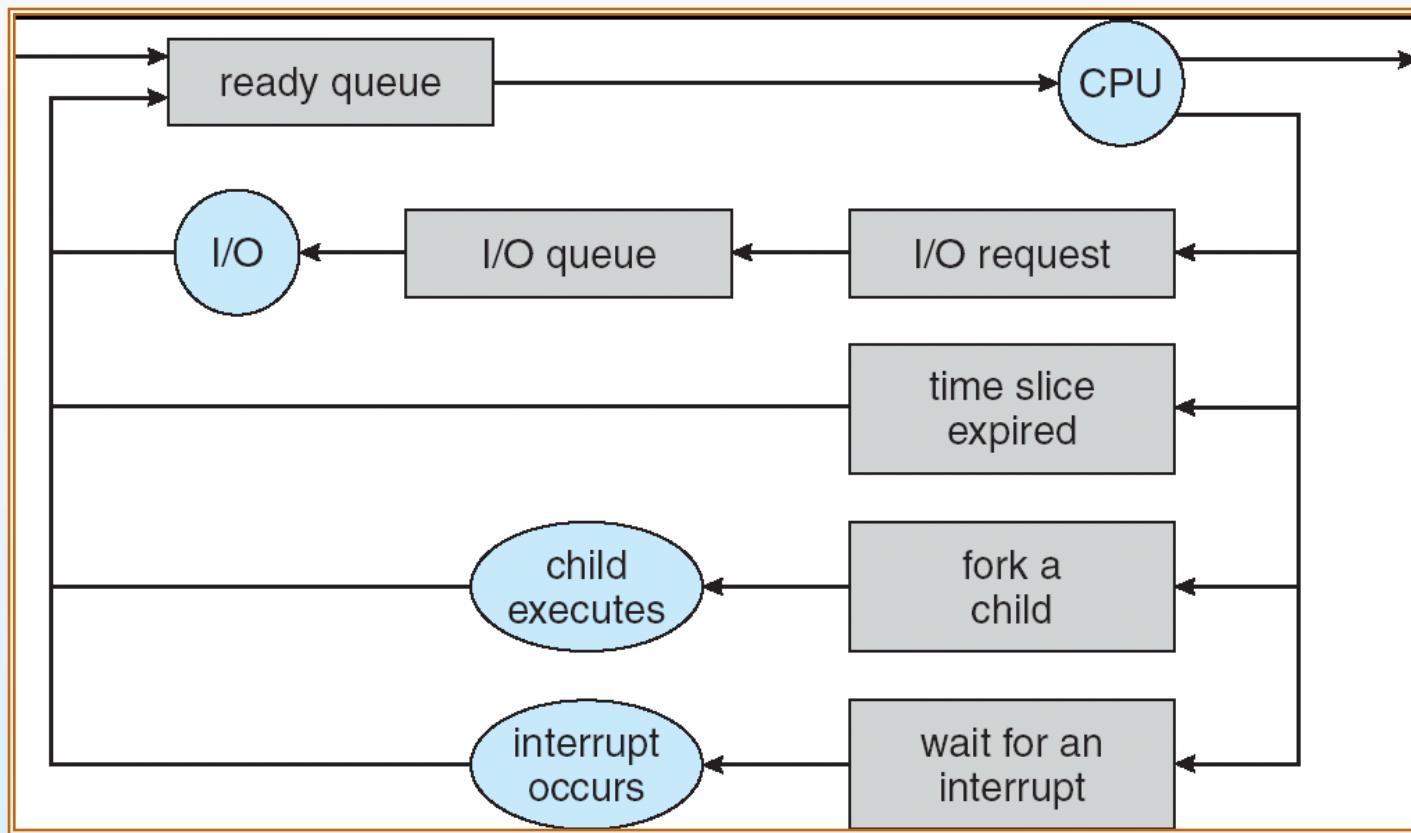


# 就绪队列和各种 I/O 设备队列





# 进程调度的队列表示图





# CPU调度过程

## ■ 调度程序(Scheduler)

- 根据某种策略选择内存中的一个就绪进程
- 一个CPU同时只能运行一个进程
- 做选择

## ■ 分派程序(Dispatcher)

- 负责具体的进程切换工作
- 做切换





# CPU调度的具体操作过程

- ① 利用定时器把CPU的控制权转交CPU调度程序，让调度程序选择一个需要运行的进程；
  - ② 进行进程的上下文切换，把该进程从就绪状态转换到运行状态；
  - ③ 系统切换到用户态，跳转到用户程序的适当位置并重新运行之。
- 
- 分派延迟(*Dispatch latency*) -分派程序终止一个进程的运行并启动进程运行所花的时间。





# 调度方式

## ■ 非抢占式调度

- 一旦把CPU分配给某个进程后，系统不可以抢占已分配的CPU并分配给其他进程。
- 只有进程自愿释放CPU，才可以把CPU分配给其他进程。
- 优点：容易实现，调度开销小，适合批处理系统。
- 缺点：响应时间长，不适合交互式系统。

## ■ 抢占式调度

- 调度程序可以根据某种原则暂停某个正在执行的进程，将已分配给它的CPU重新分配给另一个进程。
- 可防止单一进程长时间独占CPU。
- 系统开销大。

■ 抢占式和非抢占式最大的区别是运行进程是否自愿放弃CPU。

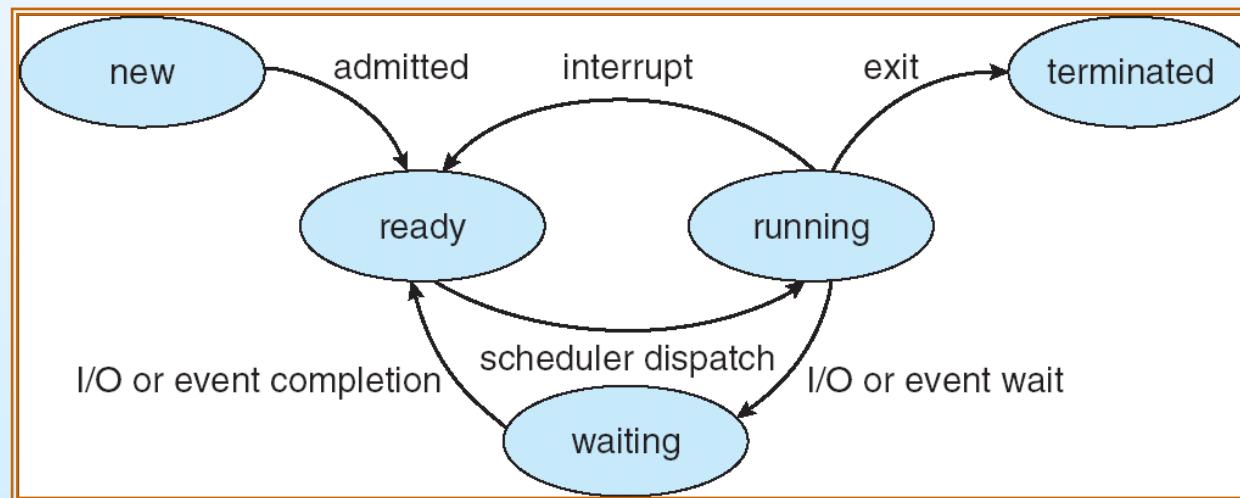




# CPU调度时机

■ CPU调度可能发生在当一个进程:

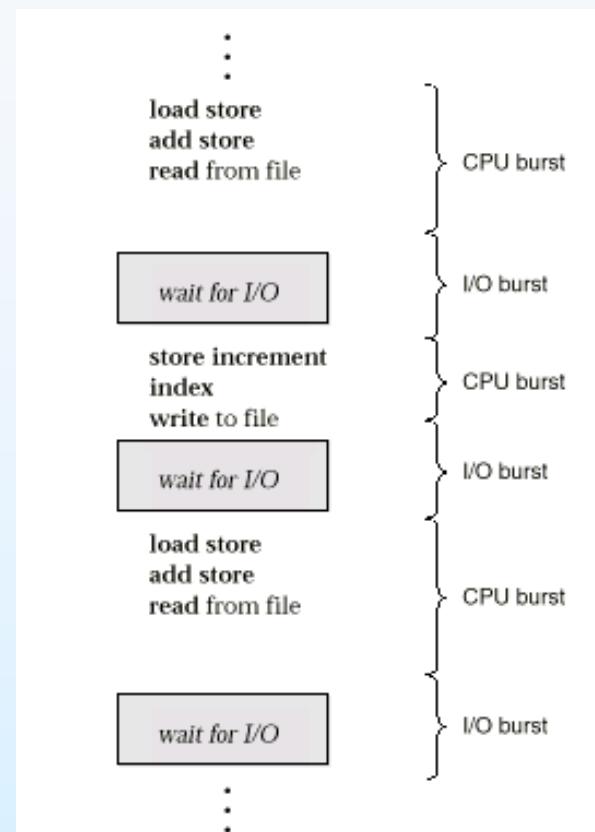
1. 从运行转到等待（非抢占式）
2. 从运行转到就绪（抢占式）
3. 从等待转到就绪（抢占式）
4. 终止运行（非抢占式）



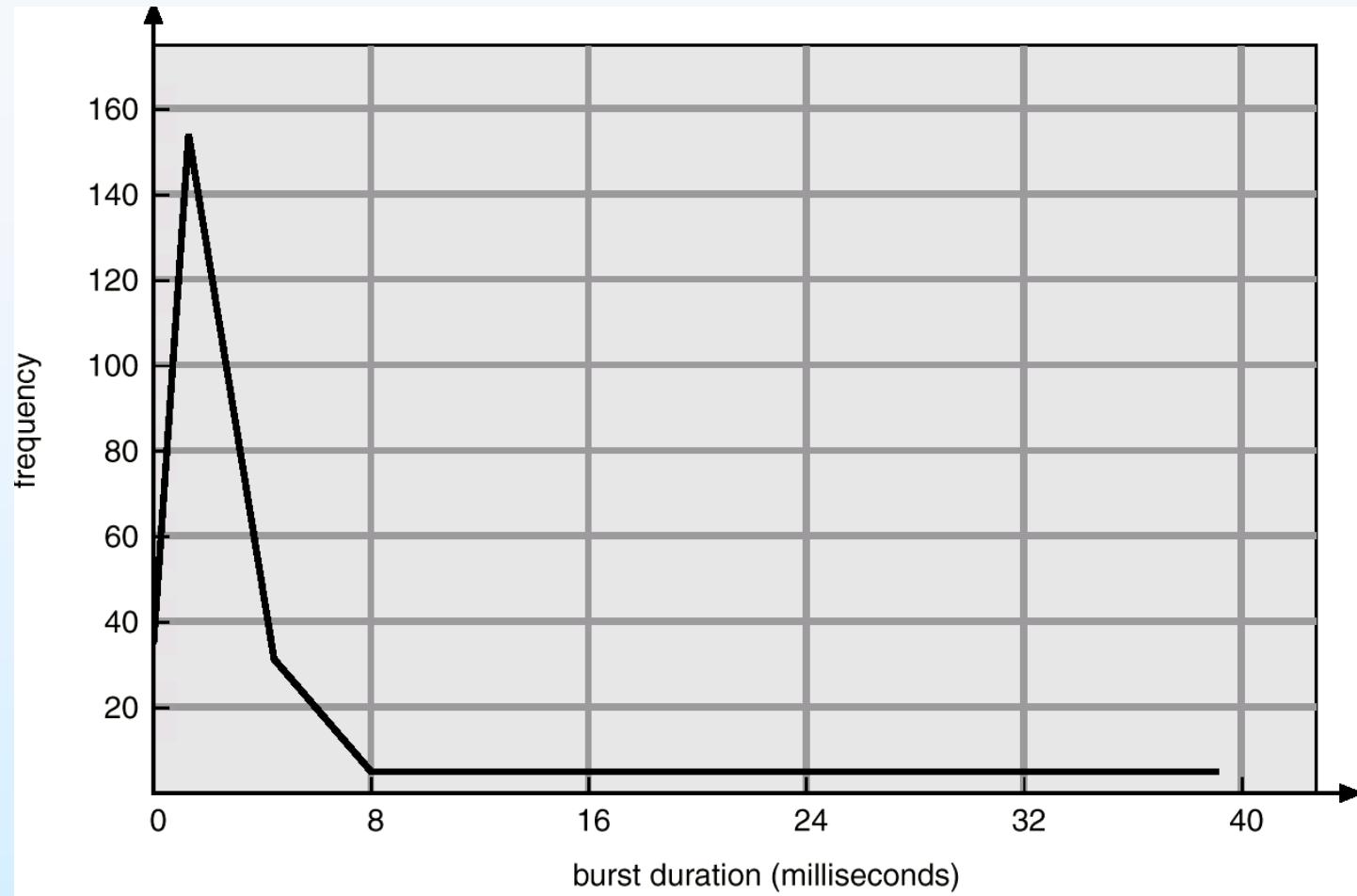


# CPU脉冲周期

- CPU调度（进程调度或线程调度）是多任务操作系统的基础
- 通过多道程序设计得到CPU的最高利用率
- 进程的执行包括进程在CPU上执行和等待I/O
  - CPU脉冲周期
  - I/O脉冲周期



# CPU脉冲周期统计





# CPU调度程序

■ 选择内存中的就绪进程，并分配CPU给其中之一

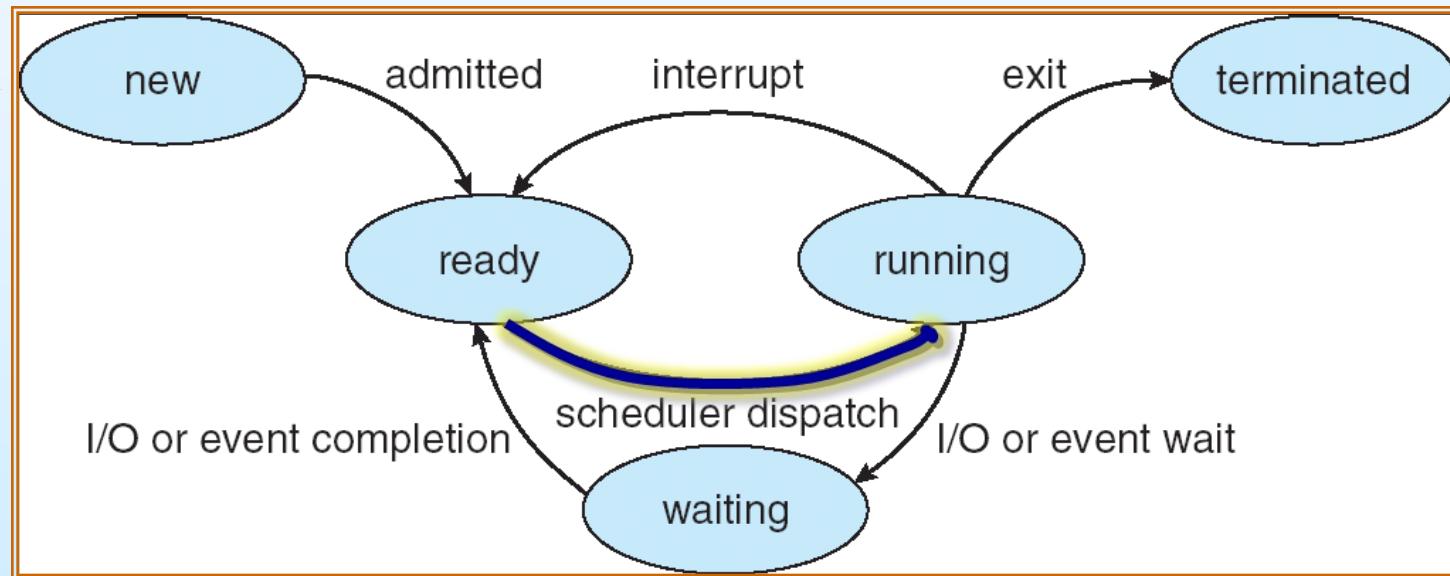
■ 调度方案：

- 非抢占式调度(**nonpreemptive**)

- 一旦把处理机分配给某进程后，系统不允许其他进程抢占已分配给它的CPU。直至该进程完成，自愿释放CPU，或发生某事件而被阻塞，才再把CPU分配给其他进程

- 抢占式调度(**preemptive**)

- ✓ 允许调度程序根据某种原则去暂停某个正在执行的进程，将已分配给她的CPU重新分配给另一进程



## 2、调度准则





- CPU调度的核心任务是提高CPU效率，也就是利用率。
- 那么如何知道CPU的效率是提高了还是降低了？
- 这就需要一系列的指标来衡量。





# 基本指标

- CPU利用率 – 固定时间内CPU运行时间的比例
- 吞吐量 – 单位时间内运行完的进程数
- 周转时间 – 进程从提交到运行结束的全部时间
- 等待时间 – 进程等待调度（不在运行）的时间片总和
- 响应时间 – 从进程提出请求到首次被响应[而不是输出结果]的时间段[在分时系统环境下]，也就是第一段的等待时间
- 周转时间=等待时间+运行时间
- 带权周转时间=周转时间/运行时间



运行时间:  $1+2+3=6$

周转时间:  $10-0=10$

等待时间:  $1+1+2=4$  或  $10-6=4$

响应时间 <= 等待时间

CPU利用率:  $6/10=60\%$





# 优化准则

- 最大的CPU利用率
  - 最大的吞吐量
  - 最短的周转时间
  - 最短的等待时间
  - 最短的响应时间
- 
- 解决方法
    - 调度算法：决定就绪队列中哪个进程被选中运行



### 3、调度算法1



FCFS

SJF





# 内容

## ■ 先来先服务调度算法

- 算法举例
- 优缺点

## ■ 短作业优先调度算法

- 算法介绍
- 算法举例
- 下一个CPU区间长度的预测





# First-Come First-Served (FCFS)

## 先来先服务调度算法

- 调度策略：按照进程请求CPU的先后顺序来使用CPU
- 调度依据：进入就绪队列的时间
- 调度方法：先进入就绪队列的进程被优先选中运行
- 使用FIFO队列实现





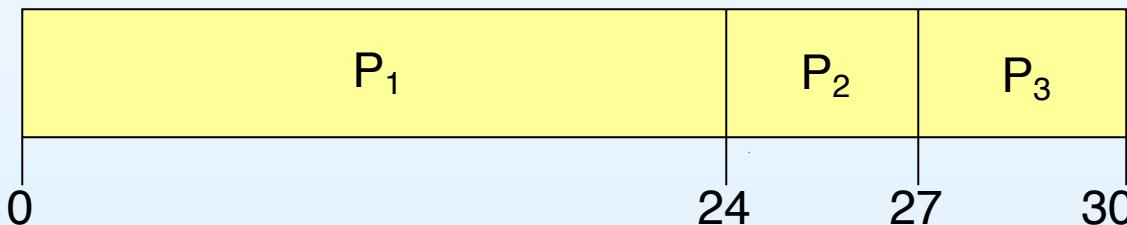
# First-Come First-Served (FCFS)

## 先来先服务调度算法

■ 举例:

<u>进程</u>	<u>区间时间</u>
$P_1$	24
$P_2$	3
$P_3$	3

■ 假定进程到达顺序如下:  $P_1, P_2, P_3$  该调度的Gantt图为:



- 等待时间:  $P_1 = 0; P_2 = 24; P_3 = 27$  平均等待时间:  $(0 + 24 + 27)/3 = 17$
- 周转时间:  $P_1 = 24; P_2 = 27; P_3 = 30$  平均周转时间:  $(24 + 27 + 30)/3 = 27$
- 响应时间:  $P_1 = 0; P_2 = 24; P_3 = 27$  平均响应时间:  $(0 + 24 + 27)/3 = 17$





# FCFS算法

## ■ 算法特点

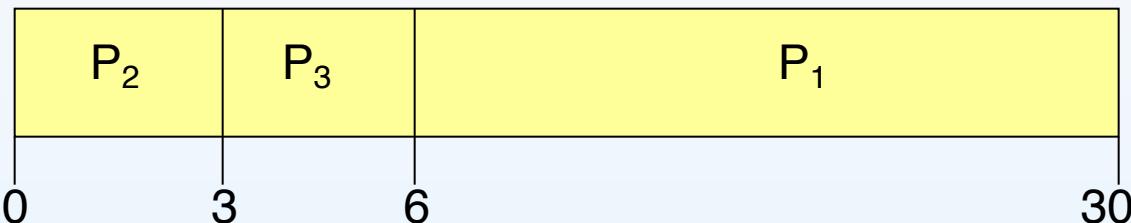
- 实现简单，可使用FIFO队列实现
- 非抢占式调度
- 该调度算法是公平的，每个进程都有被调度的机会，不会被抢占
- 适用于长程调度，后台批处理系统的短程调度
- 对长CPU脉冲的进程有利，对短CPU脉冲的进程不利
  - ◆ 当一个长进程后面的多个短进程，让长进程先执行，会让后面的短进程等待较长时间，从而导致CPU和设备利用率降低，护航效果（convoy effect）





# 先来先服务调度算法

- 假定进程到达顺序如下  $P_2, P_3, P_1$ .
- 该调度的Gantt图为：



- 等待时间:  $P_1 = 6; P_2 = 0; P_3 = 3$  平均等待时间 :  $(6 + 0 + 3)/3 = 3$
- 周转时间:  $P_1 = 30; P_2 = 3; P_3 = 6$  平均周转时间:  $(30 + 3 + 6)/3 = 13$
- 响应时间:  $P_1 = 6; P_2 = 0; P_3 = 3$  平均响应时间 :  $(6 + 0 + 3)/3 = 3$
- 比前例好得多
- 护航效果**convoy effect**: 长进程先于短进程到达
- 有利于长进程, 而不利于短进程





# 讨论

1. FCFS是抢占式调度还是非抢占式调度？
2. FCFS对怎样的进程不利？



# Shortest-Job-First (SJF)

## 短作业优先调度算法

- 从FCFS存在的问题得到启发
- 调度策略：关联到每个进程下次运行的CPU脉冲长度，调度最短的进程
- 调度依据：每个进程下次运行的CPU脉冲长度
- 调度方法：调度最短的进程运行
- 两种模式：
  - 非抢占式调度：一旦进程拥有CPU，它可在该CPU脉冲结束后让出CPU
  - 抢占式调度：有比当前进程剩余时间更短的进程到达时，新来的进程抢占当前运行进程的CPU，也称为最短剩余时间优先调度  
**Shortest-Remaining-Time-First (SRTF)**
- SJF最优 –具有最短的平均等待时间
- 饥饿（Starvation）：长进程可能长时间等待

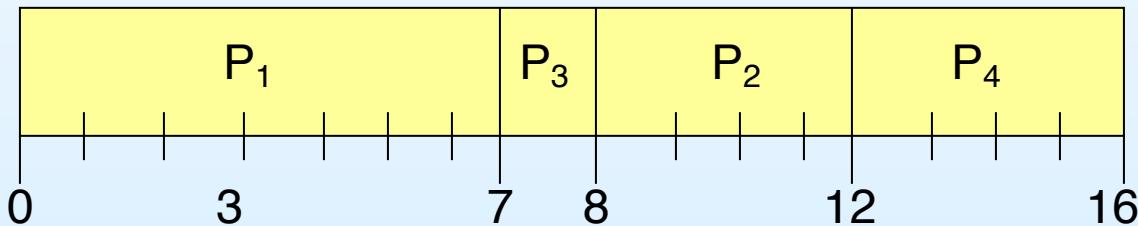




# 非抢占式SJF例子

进程	到达时间	区间时间
P <sub>1</sub>	0.0	7
P <sub>2</sub>	2.0	4
P <sub>3</sub>	4.0	1
P <sub>4</sub>	5.0	4

## ■ SJF (non-preemptive)



- 平均周转时间 =  $(7 + 10 + 4 + 11) / 4 = 8$
- 平均等待时间 =  $(0 + 6 + 3 + 7) / 4 = 4$
- 平均响应时间 =  $(0 + 6 + 3 + 7) / 4 = 4$

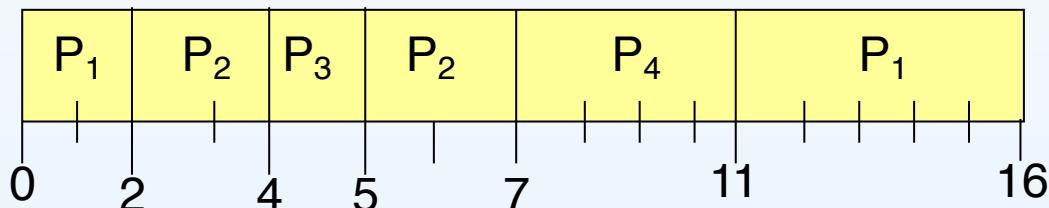




# 抢占式SJF（SRTF）例子

进程	到达时间	区间时间
P <sub>1</sub>	0.0	7
P <sub>2</sub>	2.0	4
P <sub>3</sub>	4.0	1
P <sub>4</sub>	5.0	4

■ SJF (preemptive)



	P1	P2	P3	P4
脉冲时间 0.0	7	未到达	未到达	未到达
脉冲时间 2.0	7-2=5	4	未到达	未到达
脉冲时间 4.0	7-2=5	4-2=2	1	未到达
脉冲时间 5.0	7-2=5	4-2=2	完成	4
脉冲时间 7.0	7-2=5	完成	完成	4
脉冲时间 11.0	7-2=5	完成	完成	完成
脉冲时间 16.0	完成	完成	完成	完成

$$\text{平均等待时间} = (9 + 1 + 0 + 2) / 4 = 3$$

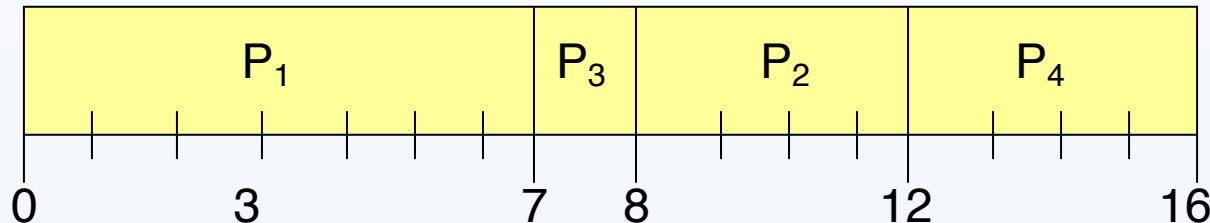
$$\text{平均周转时间} = (16 + 5 + 1 + 6) / 4 = 7$$

$$\text{平均响应时间} = (0 + 0 + 0 + 2) / 4 = 0.5$$





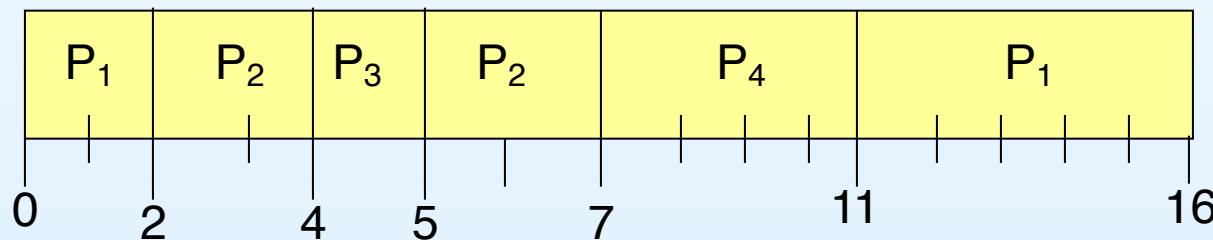
# 非抢占式SJF vs 抢占式SJF



$$\text{平均周转时间} = (7 + 10 + 4 + 11) / 4 = 8$$

$$\text{平均等待时间} = (0 + 6 + 3 + 7) / 4 = 4$$

$$\text{平均响应时间} = (0 + 6 + 3 + 7) / 4 = 4$$



$$\text{平均周转时间} = (16 + 5 + 1 + 6) / 4 = 7$$

$$\text{平均等待时间} = (9 + 1 + 0 + 2) / 4 = 3$$

$$\text{平均响应时间} = (0 + 0 + 0 + 2) / 4 = 0.5$$





# CPU下一次脉冲的探测

- 其长度只能估计
- 可以通过先前的CPU脉冲长度及计算指数均值进行
  1.  $t_n$  = actual lenght of  $n^{th}$  CPU burst
  2.  $\tau_{n+1}$  = predicted value for the next CPU burst
  3.  $\alpha, 0 \leq \alpha \leq 1$
  4. Define :

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n.$$





# 指数平均例子

## ■ $\alpha = 0$

- $\tau_{n+1} = \tau_n$
- 近来历史没有影响

## ■ $\alpha = 1$

- $\tau_{n+1} = t_n$
- 只有最近的CPU区间才重要

## ■ 如果扩展公式，得到：

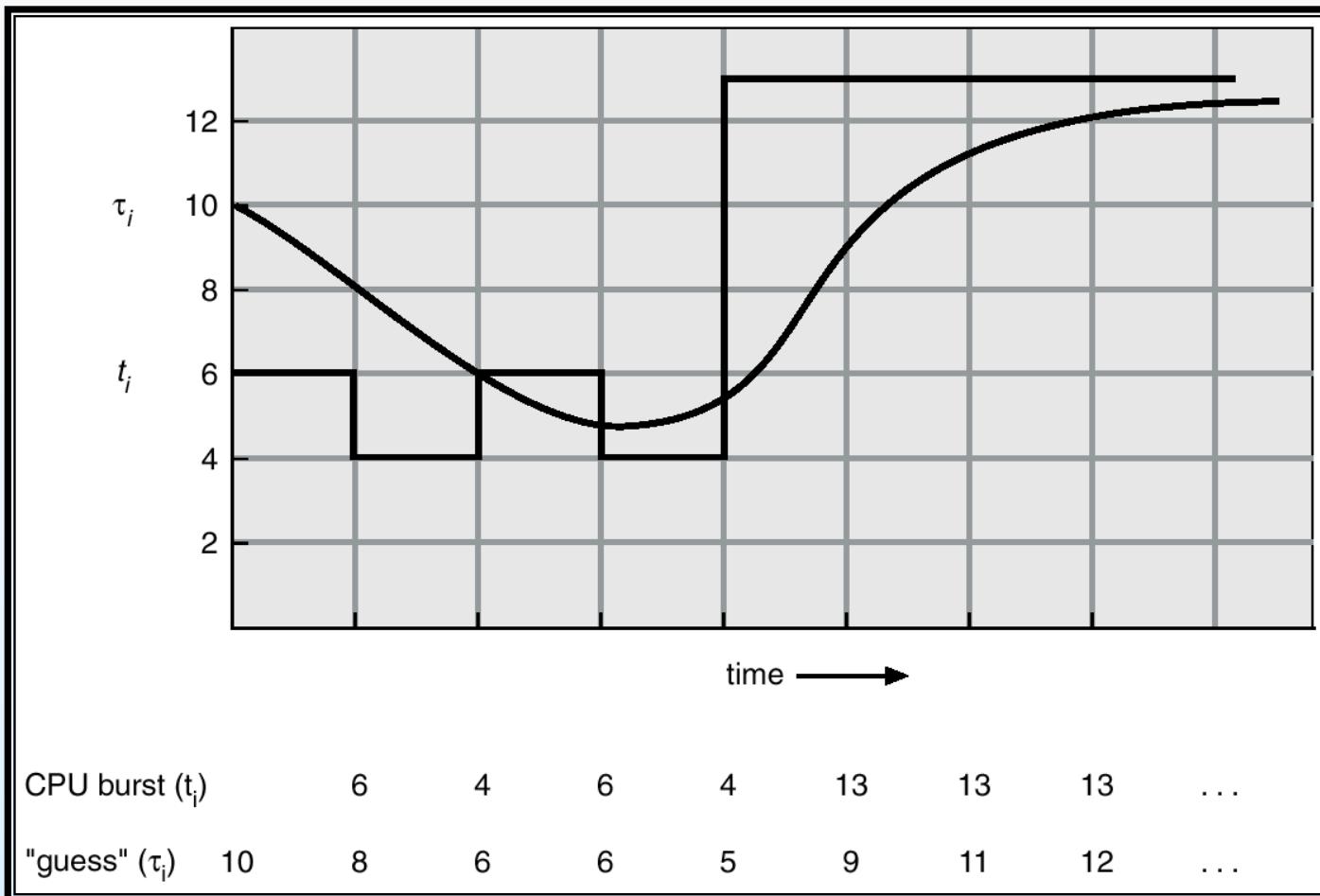
$$\begin{aligned}\tau_{n+1} &= \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \dots \\ &\quad + (1 - \alpha)^j \alpha t_{n-j} + \dots \\ &\quad + (1 - \alpha)^{n+1} \tau_0\end{aligned}$$

## ■ 由于 $\alpha$ 和 $(1 - \alpha)$ 都小于或等于 1，所以后面项的权比前面项的权小





## 例子：CPU下一次脉冲的探测



## 4、调度算法2



优先级调度  
时间片轮转调度



# Priority Scheduling

## 优先级调度

- 调度依据：优先级
- 调度方法：调度优先级最高进程运行
- 优先数：表示优先级的整数[默认：最小整数 → 最高优先级]
- 优先级调度有：
  - Preemptive（抢占式）
  - Nonpreemptive（非抢占式）

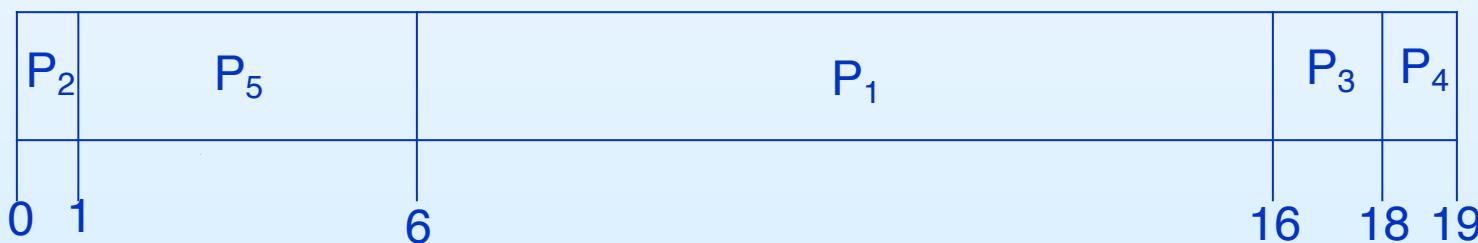




# PR例子（非抢占式）



<u>进程</u>	<u>优先数</u>	<u>区间时间</u>
P1	3	10
P2	1	1
P3	3	2
P4	4	1
P5	2	5



❖ 平均等待时间 =  $(6 + 0 + 16+18+1)/5 = 8.2$





# 优先级调度

- 优先数：
  - 静态优先数：在运行前每个进程获得一个优先数，在运行过程中不可变化
  - 动态优先数：在运行前每个进程获得一个基准优先数，在运行过程中可变化
- 最大特点：灵活（可以模拟其它调度算法）
- 目前大多数现代操作系统常用的调度算法
- 问题：饥饿 – 低优先级的可能永远得不到运行
- 解决方法：老化 – 视进程等待时间的长提高其优先数





# 动态优先级举例

- ❖ 高响应比优先调度算法

- 既考虑进程的等待时间，又考虑进程的运行时间

$$\text{优先数} = \text{响应比} R = \frac{\text{等待时间}}{\text{运行时间}}$$

优先数越大，优先级越高

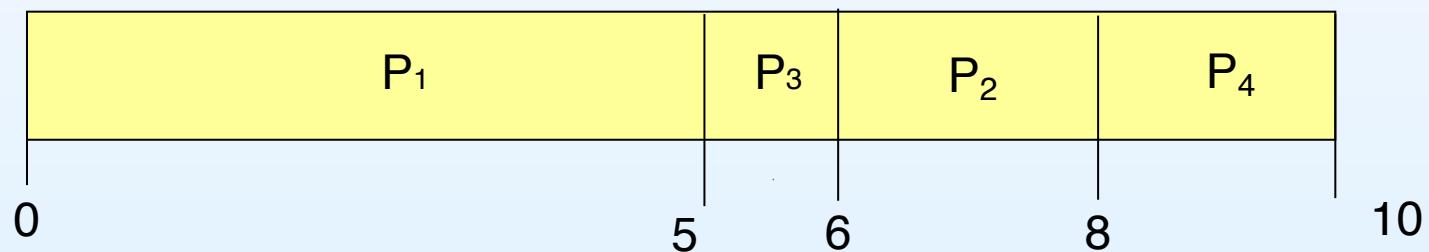
- ❖ 如等待时间相同，运行时间越短，优先级越高，类似于SJF
- ❖ 如运行时间相同，优先级取决于其等待时间，类似于FCFS
- ❖ 长进程的优先级可随等待时间的增加而提高，最终可得到服务
- ❖ 缺点：每次调度之前，都需要计算响应比，增加系统开销





# 非抢占式优先级调度例子

进程	优先级	到达时间	区间时间
P <sub>1</sub>	3	0	5
P <sub>2</sub>	3	1	2
P <sub>3</sub>	1	2	1
P <sub>4</sub>	4	3	2



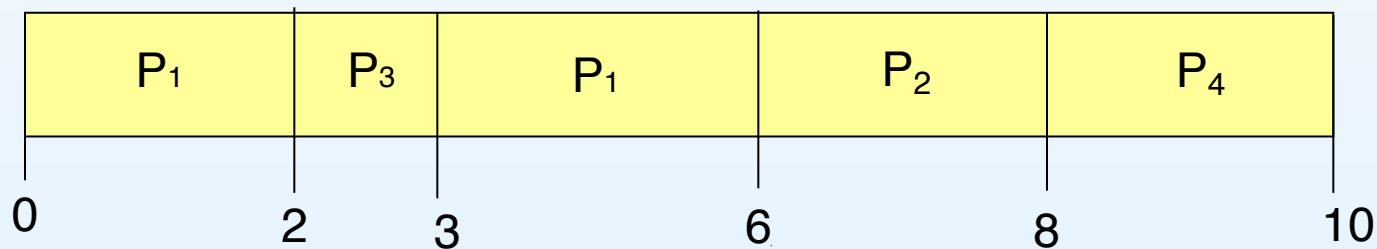
■ 平均等待时间 =  $(0 + 5 + 3 + 5) / 4 = 3.25$





# 抢占式优先级调度例子

进程	优先级	到达时间	区间时间
P <sub>1</sub>	3	0	5
P <sub>2</sub>	3	1	2
P <sub>3</sub>	1	2	1
P <sub>4</sub>	4	3	2



- 平均周转时间 =  $(6 + 7 + 1 + 7) / 4 = 5.25$
- 平均等待时间 =  $(1 + 5 + 0 + 5) / 4 = 2.75$
- 平均响应时间 =  $(0 + 5 + 0 + 5) / 4 = 2.5$





# 响应比高者优先调度算法

$$\text{优先权} = \frac{\text{等待时间}}{\text{要求服务时间}}$$

- 8.0时：选择当时唯一的P1
- 10.0（作业1完成）： P2、P3、P4已到达， 计算响应比：
  - P2:  $(10-8.3)/0.5=3.4$
  - P3:  $(10-8.5)/0.1=\mathbf{15}$
  - P4:  $(10-9.0)/0.4=2.5$
- 10.1(作业3完成)：计算P2、P4响应比：
  - P2:  $(10.1-8.3)/0.5=\mathbf{3.6}$
  - P4:  $(10.1-9.0)/0.4=2.75$
- 选择P2运行， 最后运行P4

进程	到达时间	区间时间
P1	8.0	2.0
P2	8.3	0.5
P3	8.5	0.1
P4	9.0	0.4





# Round Robin (RR)

## 时间片轮转

- 为分时系统设计
- 时间片：较小单位的CPU时间，通常为10-100毫秒
- 调度依据：和FCFS相似
- 调度方法：每个进程运行时间为一个时间片。时间片用完后，该进程将被抢占并插入就绪队列末尾
- 假定就绪队列中有 $n$ 个进程、时间片为 $q$ 
  - 则每个进程每次得到不超过 $q$ 单位的成块CPU时间
  - 没有一个进程的等待时间会超过 $(n-1) q$
  - 在不超过 $nq$ 时间内， $n$ 个进程都运行一次

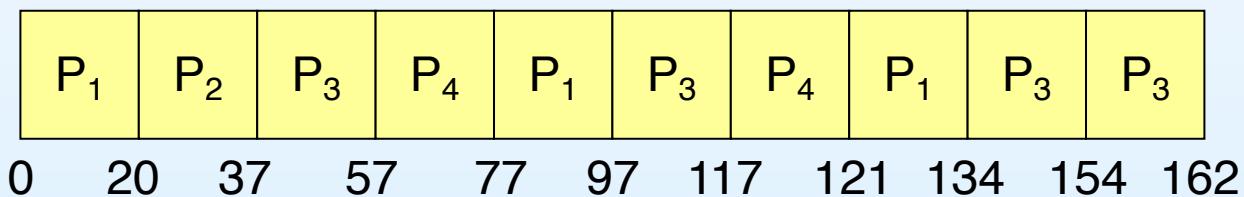




# 时间片为20的RR例子

<u>进程</u>	<u>区间时间</u>
$P_1$	53
$P_2$	17
$P_3$	68
$P_4$	24

- Gantt图如下：



- 平均周转时间:  $(134+37+162+121)/4 = 113.5$
- 平均等待时间:  $(81+20+94+97)/4 = 73$
- 平均响应时间:  $(0+20+37+57)/4 = 28.5$
- RR的平均周转时间比SJF长，但响应时间要短一些





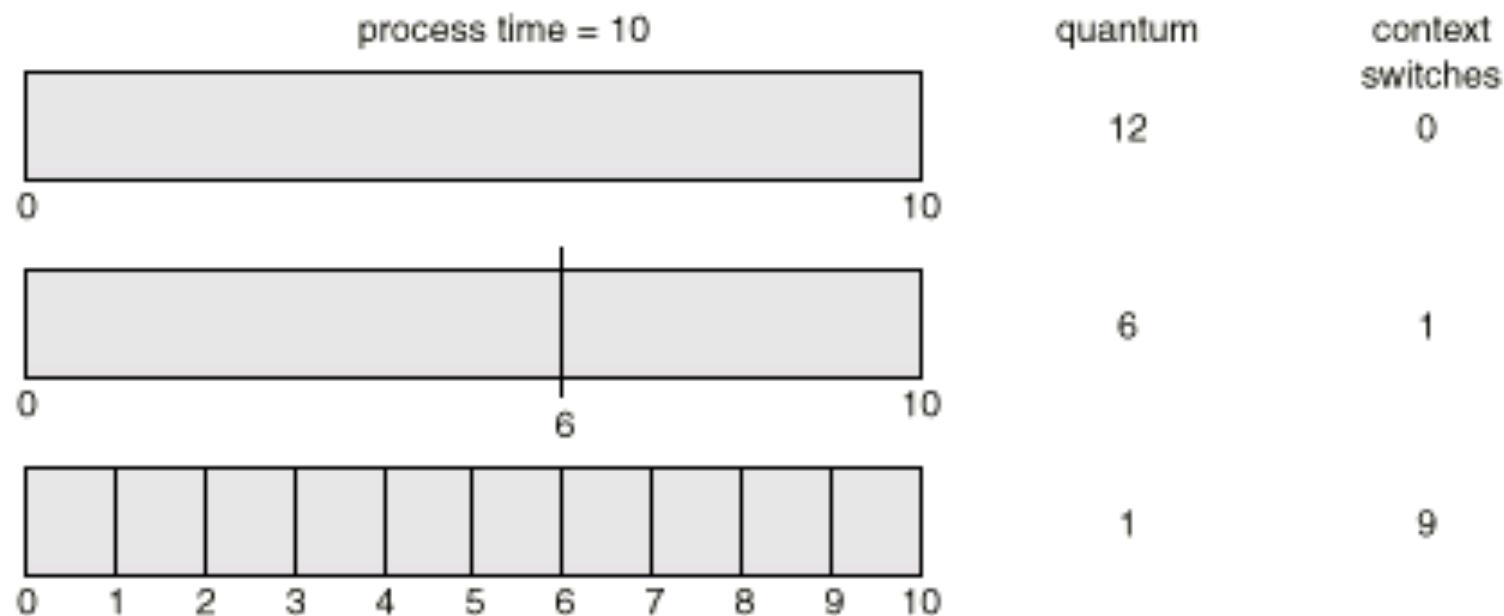
# 时间片和上下文切换次数

时间片大小

$q$  大  $\Rightarrow$  FCFS

$q$  小  $\Rightarrow$  系统开销过大

一般准则：时间片/10>进程上下文切换时间



## 5、调度算法3



混合型算法  
多级队列调度  
多级反馈队列调度

多处理器调度



- 算法的局限性，不能适应各种不同类型的进程
  - SJF算法有利于短进程，而不利于长进程
  - RR算法系统开销大
  - 优先级算法存在饥饿问题等
  - 所有进程采用同一策略，不合理
- 不同类型的进程需要不同策略
  - 交互进程需要较短的响应时间
  - 批处理进程需要较短的等待时间





# Multilevel Queue

## 多级队列

- 多级队列调度(MultiLevel Queue Scheduling)
- 允许系统中存在多个就绪队列，每个就绪队列有自己的调度算法
- 关键问题
  - 需要确定就绪队列的数量
  - 需要确定新进程进入哪个队列
  - 需要确定每一个队列的调度算法

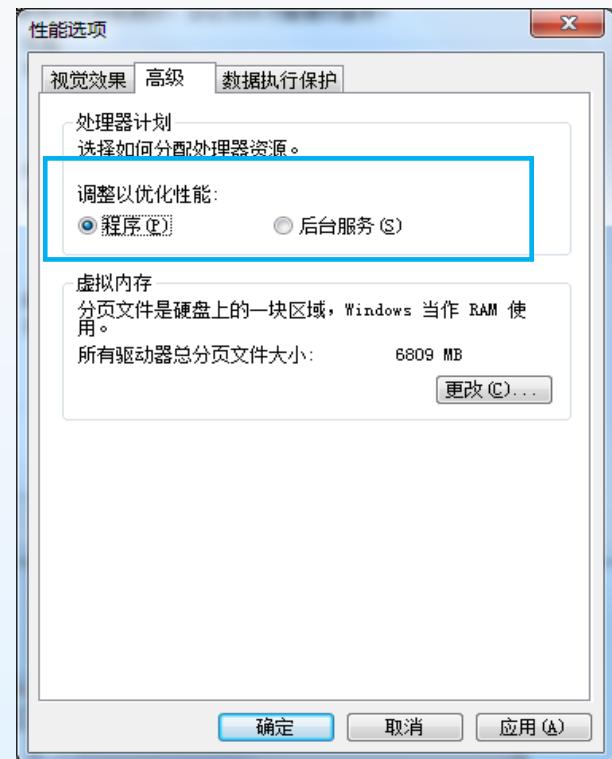




# Multilevel Queue

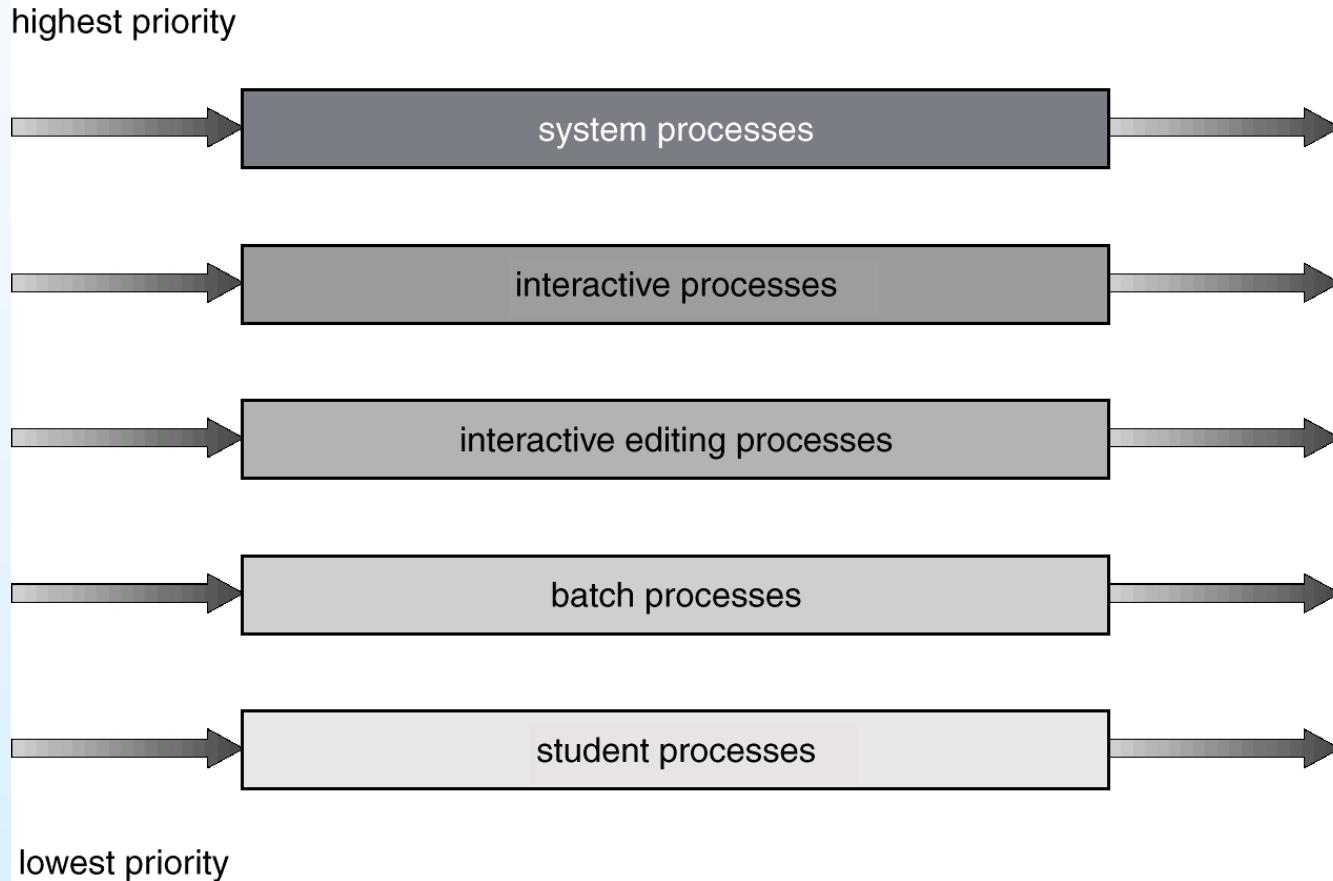
## 多级队列

- 就绪队列分为
  - 前台[交互式]
  - 后台[批处理]
- 每个队列有自己的调度算法
  - 前台 – RR
  - 后台 – FCFS
- 调度须在队列间进行
  - 固定优先级调度，即前台运行完后再运行后台。有可能产生饥饿
  - 给定时间片调度，即每个队列得到一定的CPU时间，进程在给定时间内执行；如，80%的时间执行前台的RR调度，20%的时间执行后台的FCFS调度





# 多级队列调度





# Multilevel Feedback Queue

## 多级反馈队列调度

- 多级反馈队列调度(MultiLevel Feedback Queue Scheduling)
- 多级队列调度算法中，一旦一个进程进入了某个队列，那么在运行过程中不能加入其它队列，也就是不能在不同队列间移动
- 多级反馈队列算法的核心是进程在运行过程中，能在不同队列间移动
- 进程能在不同的队列间移动；可实现老化
- 多级反馈队列调度程序由以下参数定义：
  - 队列数
  - 每一队列的调度算法
  - 决定需要服务的进程将进入哪个队列的方法
  - 决定进程升级的方法（低级队列到高级队列）
  - 决定进程降级的方法（高级队列到低级队列）





# 多级反馈队列调度 (MLFQ)

## ■ 多级反馈队列调度

- 比MLQ算法具有更好的灵活性
- MLFQ很好地解决了CPU调度问题
- Unix, Solaris, Windows的调度算法一定程度上都是MLFQ的变种



1962年，图灵奖得主MIT的Fernando J. Corbató

*An Experimental Time-Sharing System*





# 多级反馈队列调度例子

## ■ 系统有三个就绪队列:

- $Q_0$  – 时间片为8毫秒
- $Q_1$  – 时间片为16毫秒
- $Q_2$  – FCFS

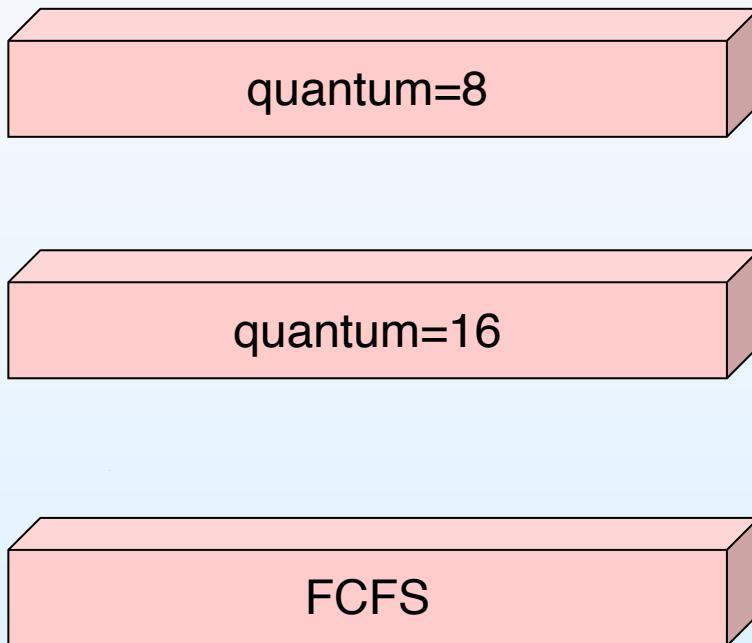
## ■ 调度策略

- 新的作业进入FCFS的 $Q_0$ 队列，它得到CPU时能使用8毫秒，如果它不能在8毫秒内完成，将移动到队列 $Q_1$ ； $Q_0$ 队列的进程采用FCFS算法
- 作业在 $Q_1$ 仍将作为FCFS调度，能使用附加的16毫秒，如果它还不能完成，将被抢占，移至队列 $Q_2$
- 进入队列 $Q_2$ 的进程将采用FCFS算法一次运行完





## MLFQ需要考虑的5个问题：

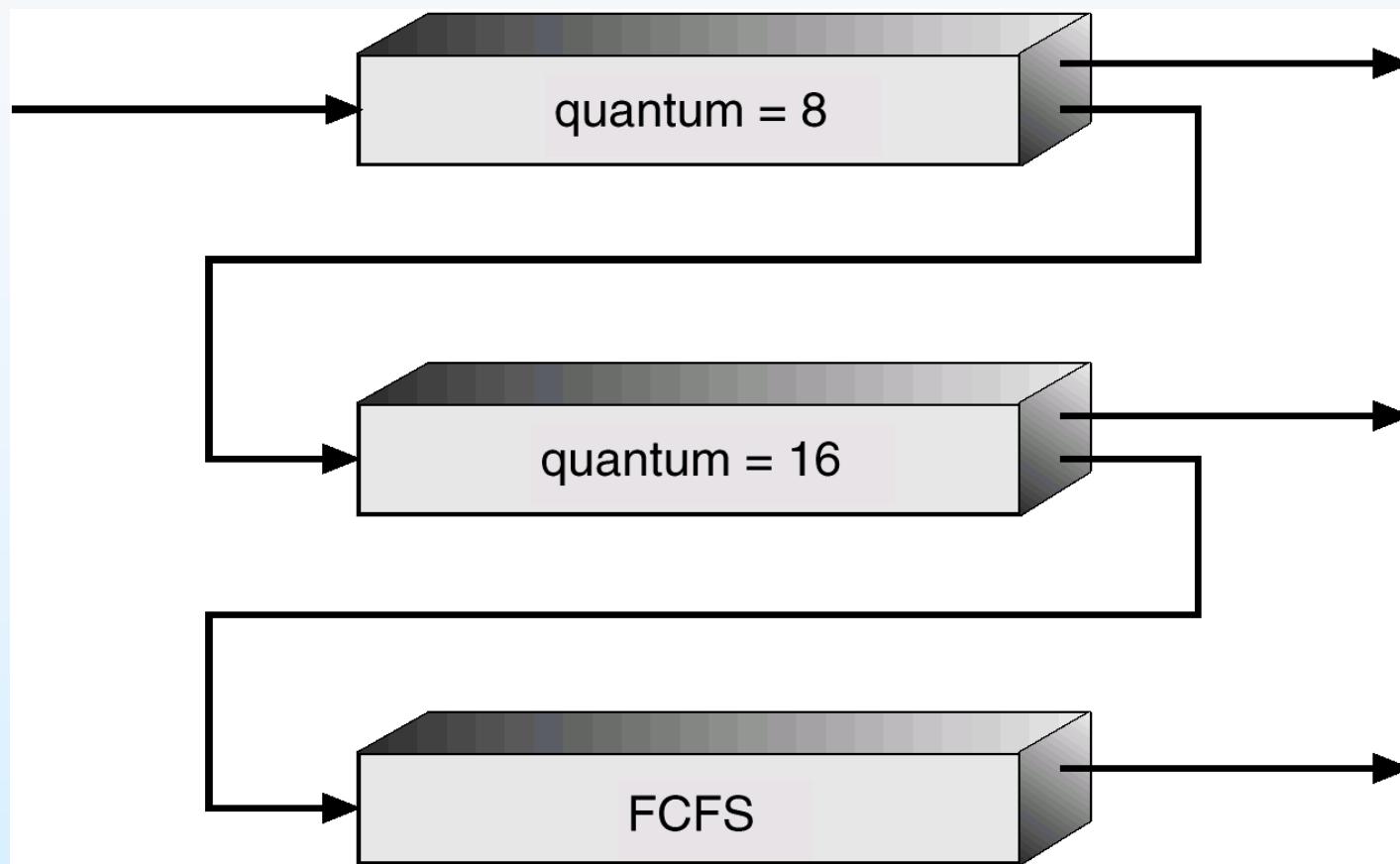


1. 队列数：该算法有3个队列
2. 每一队列的调度算法：每个队列均采用**FCFS**算法
3. 决定需要服务的进程将进入哪个队列的方法：新进程进入 $Q_0$
4. 决定进程降级的方法：每个队列只运行一次；
5. 决定进程升级的方法：没有升级方法





# 多级反馈队列调度



## 4、多处理器调度和线程调度



对称多处理器SMP

非对称多处理器ASMP



# 多处理器调度

- 适用多核处理器的CPU调度
- 多个CPU可用时，CPU调度将更为复杂
- 对称多处理器 (*SMP*) – 每个处理器决定自己的调度方案
- 非对称多处理器(*ASMP*) – 仅一个处理器能处理系统数据结构，减轻了对数据的共享需求
- 所有进程：一个就绪队列
- 处理器：私有就绪队列
- 调度算法：和单处理器相似





# 多处理器调度

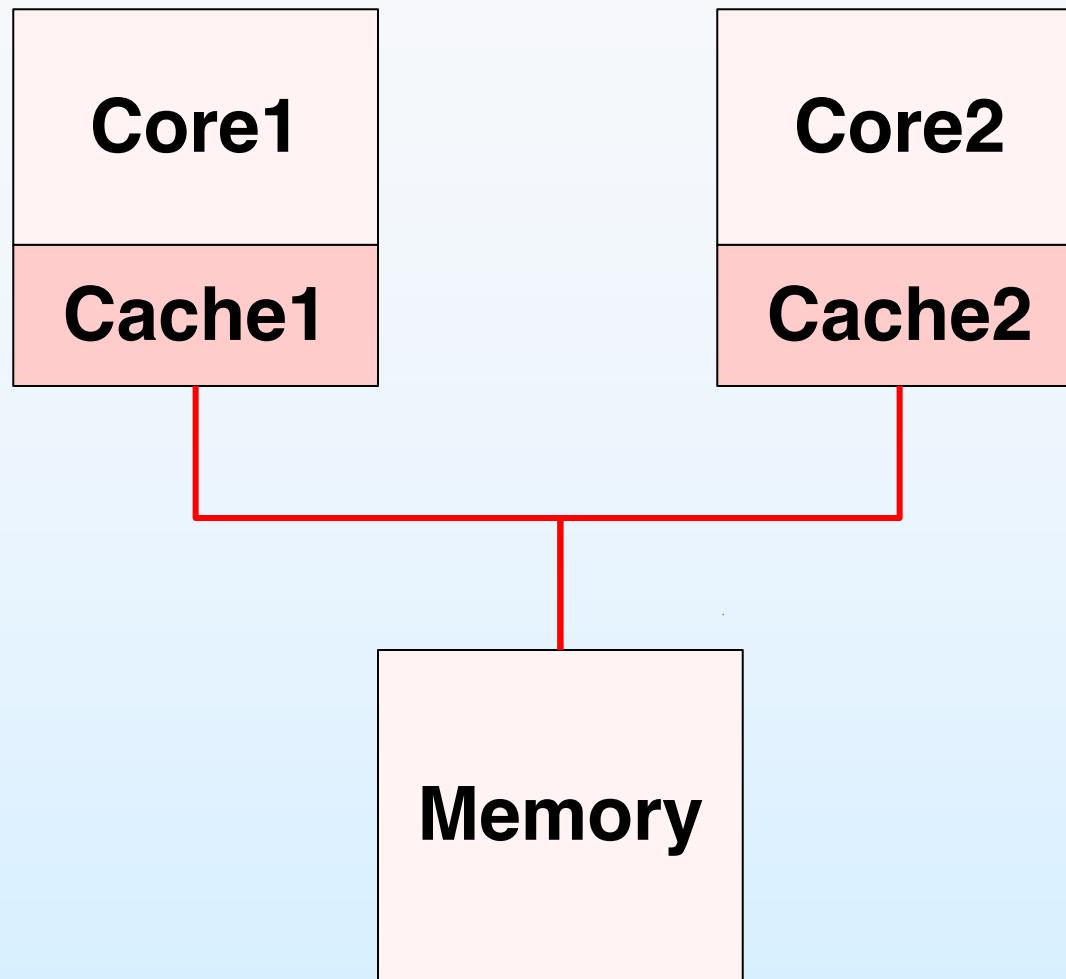
## ■ 负载平衡

- 将任务平均分配给各个处理器
- 针对私有就绪队列
- 和处理器亲和性矛盾

## ■ 处理器亲和性

- 进程要在某个给定的 **CPU** 上尽量长时间地运行而不被迁移到其他处理器的倾向性。
- 高速缓存中的内容
- 软亲和性：进程通常不会在处理器之间频繁迁移
- 硬亲和性：进程不会在处理器之间迁移（Linux）
- 例子：Linux的**task\_struct**。与亲和性（**affinity**）相关的是 **cpus\_allowed** 位掩码。这个位掩码由  $n$  位组成，与系统中的  $n$  个逻辑处理器一一对应，1表示进程可以在对应处理器上运行



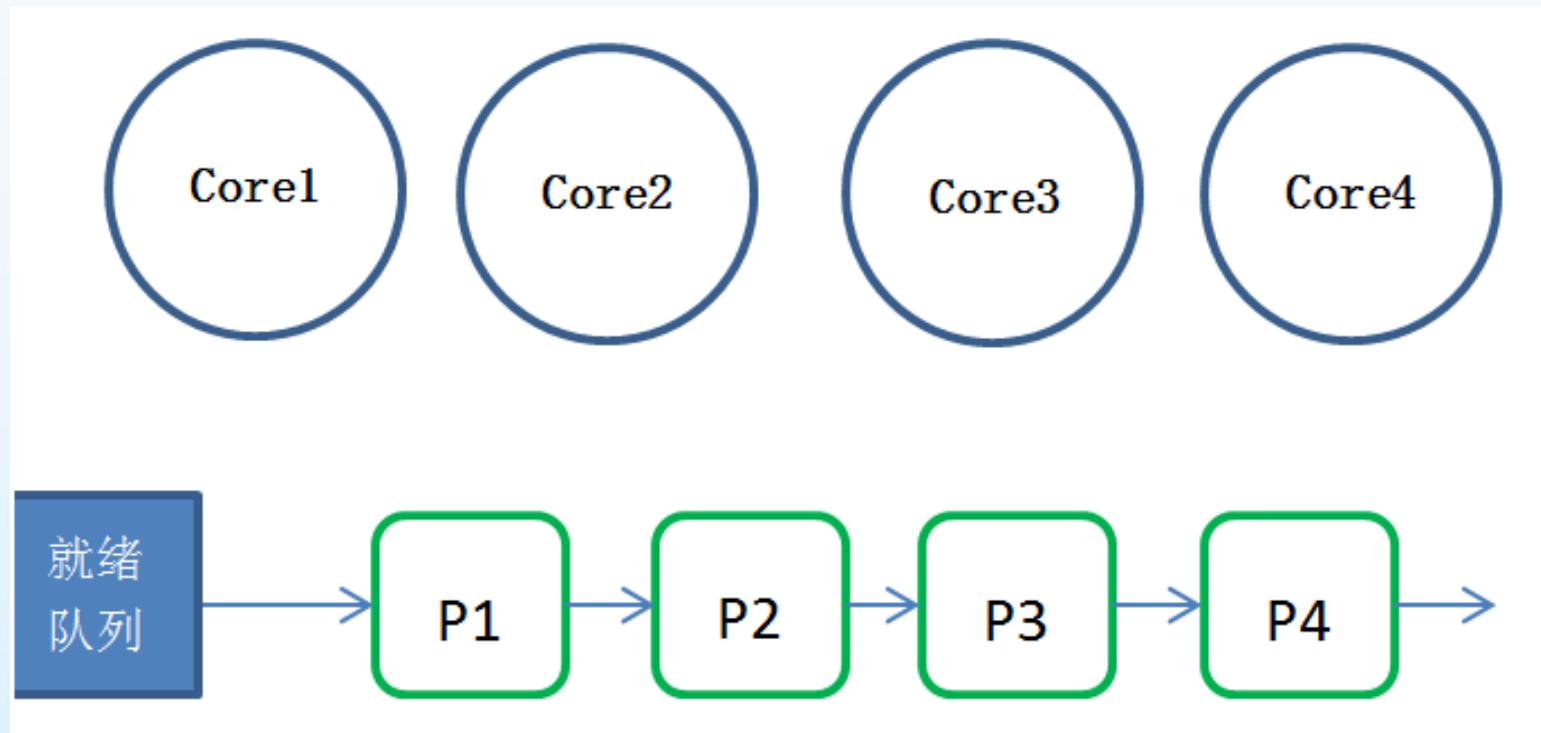




# 单队列调度方法 (SQMP)

- 单队列多核调度方法(Single-Queue MultiProcessor Scheduling)
- 系统有一个就绪队列。当任意一个CPU空闲时，就从就绪队列中选择一个进程到该CPU上运行
- 优点：
  - 容易从单核调度算法推广到多核/多处理器
  - 实现简单，负载均衡
- 缺点：
  - 不具有亲和性，一个进程可能在不同时候被调度到不同的CPU
  - 多核同时访问一个队列，会有加锁问题，从而严重影响调度的性能







# 多队列调度方法 (MQMP)

- 多队列调度方法(Multi-Queue MultiProcessor Scheduling)
- 系统有多个就绪队列，一般每个CPU一个队列。每个就绪队列有自己的调度算法，并且每个就绪队列的调度相对独立
- 优点：
  - 亲和性好
  - 不需要加锁
- 缺点：
  - 负载不均衡
- 策略：“偷”进程

