

# 3、处理机调度

---

薛瑞尼

计算机科学与工程学院

16/3/19

# 处理机调度

- 分配处理机的任务是由进程调度程序完成的。
- 处理机是最重要的计算机资源，提高处理器的利用率及改善系统性能（吞吐量、响应时间），在很大程度上取决于进程调度性能的好坏。

# 调度的目标、原则和方式

目标

提高处理机的利用率

提高系统吞吐量

尽量减少进程的响应时间

防止进程长期得不到运行

原则

用户需要

系统需要

# 调度方式



# 高级调度

- 作业调度、长程调度、接纳调度
- 将外存作业调入内存，创建PCB等，插入就绪队列。
- 一般用于批处理系统，分/实时系统一般直接入内存，无此环节。
- 调度特性
  - 接纳作业数（内存驻留数）
    - 太多→周转时间T长
    - 太少→系统效率低
  - 接纳策略：即调度算法

# 中级调度

- 中程调度
- 为提高系统吞吐量和内存利用率而引入的内外存对换功能

# 低级调度

- 进程调度，短程调度
- 由分派程序（Dispatcher/Scheduler）分派处理机
- 运行频率：低>中>高。

# 进程调度算法基本类型

- 非抢占(Non-Preemptive): 就绪进程不可以从运行进程手中抢占CPU。
  - 一旦进程处于运行状态，直到终止或者阻塞，才释放CPU
- 抢占(Preemptive): 就绪进程可以从运行进程手中抢占CPU。
  - 允许调度程序根据某种策略暂停当前运行的进程，将其转移到就绪状态，并选择另一个进程运行。
  - 时机：
    - 新进程到达时，中断的发生，阻塞进程置为就绪态

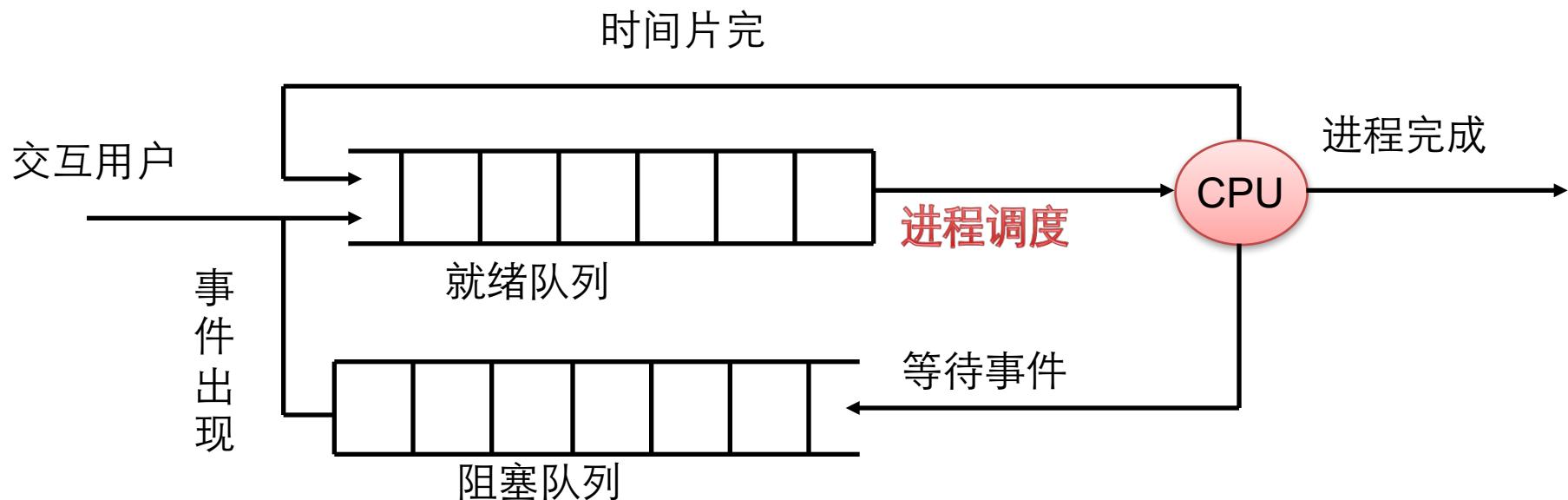
# 思考

- 三种调度的触发事件
  - 长程：任务创建
  - 中程：交换
  - 短程：时间片/事件发生/抢占

# 单处理机调度

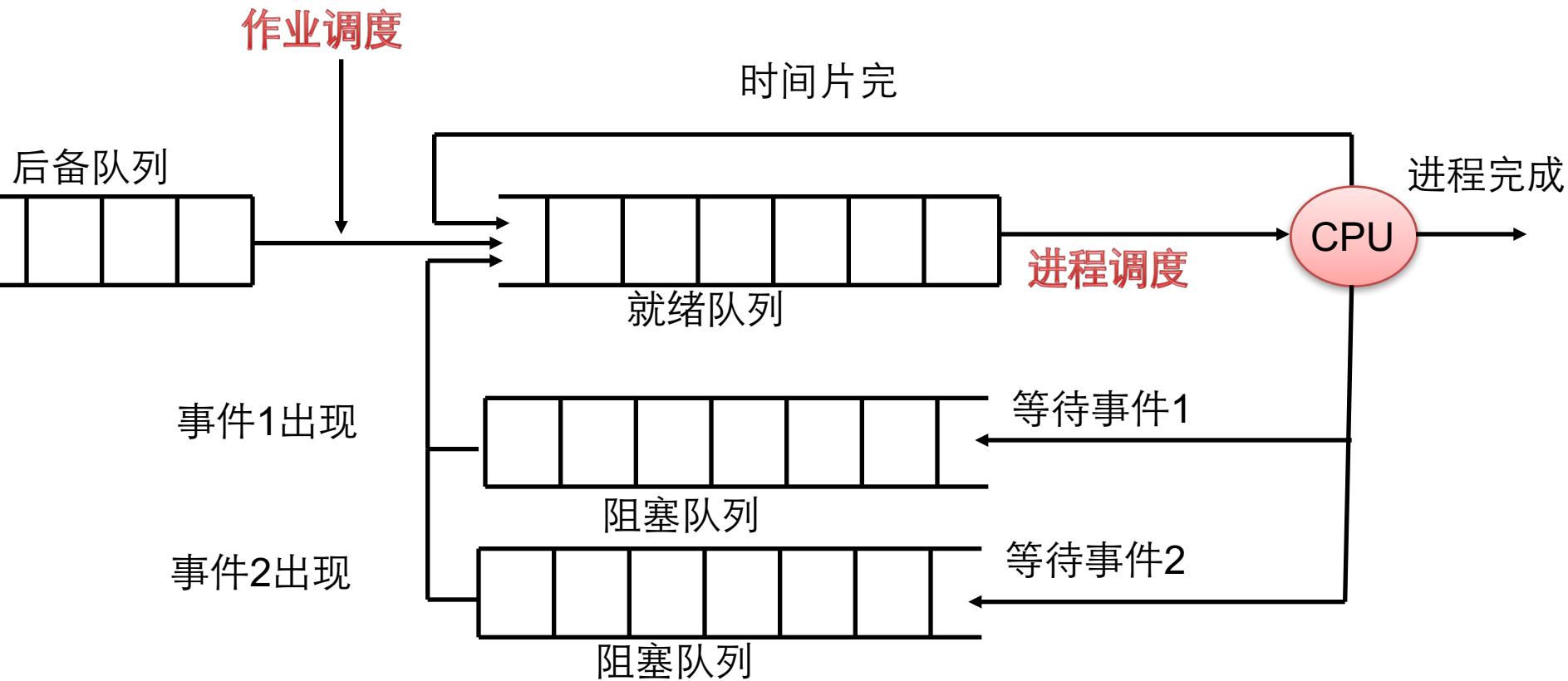
# 调度的队列模型

- 仅有进程调度的队列模型

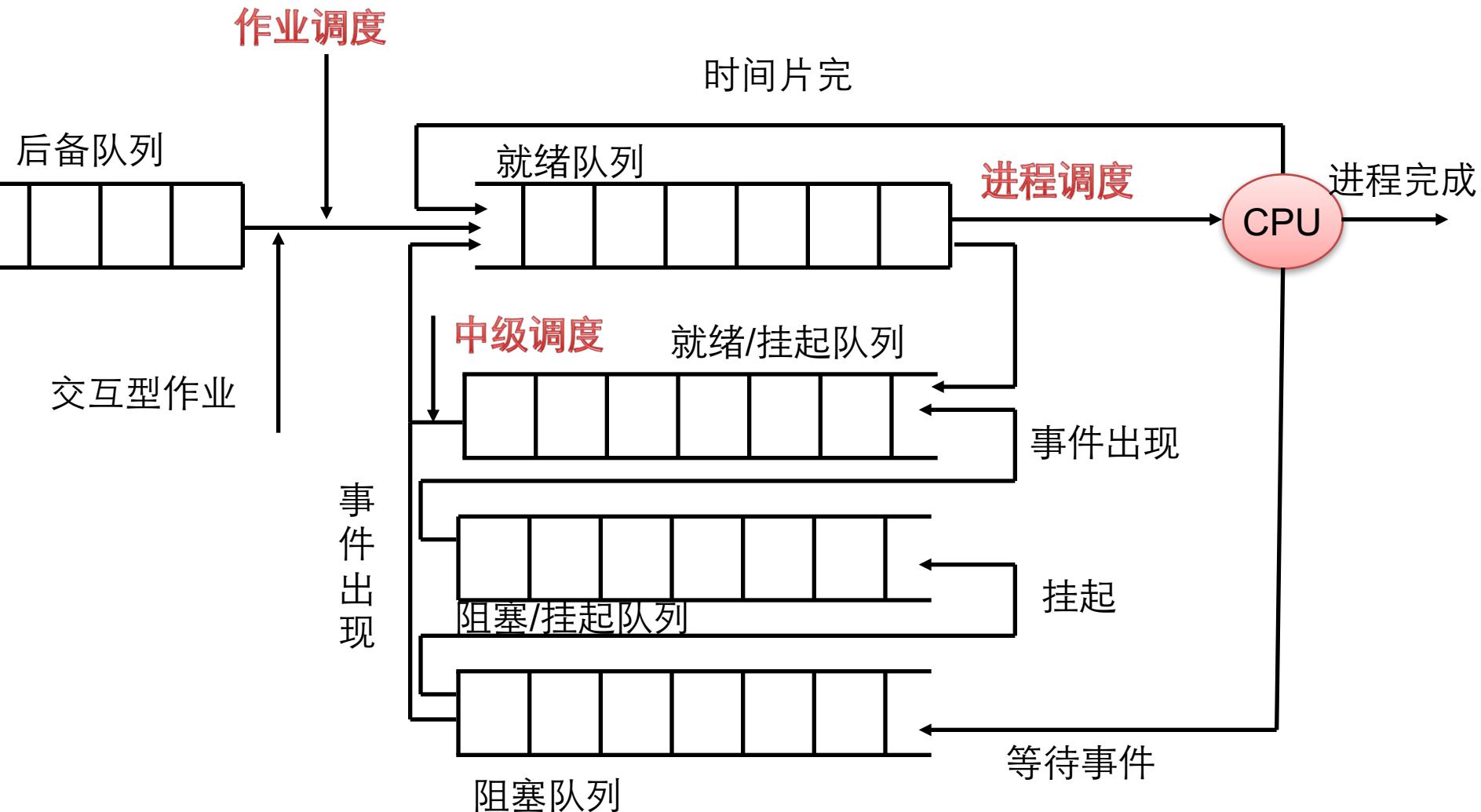


# 调度的队列模型

- 具有高/低级模型



# 三级调度



# 调度的原则

面向用户的原则

周转时间

响应时间

截止时间

优先级

面向系统的原则

吞吐量

利用率

公平性

优先级

# 面向用户的准则

- 周转时间，  
**Turnaround time**
  - 作业提交→完成的时间
  - 批处理系统



# 周转时间

- 平均周转时间

$$T = \frac{1}{n} \sum_{i=1}^n T_i$$

- 平均带权

$$W = \frac{1}{n} \sum_{i=1}^n \frac{T_i}{T_{i,s}}$$

- $T_i$ : 周转时间
- $T_{i,s}$ : 实际运行时间 (占用CPU时间)

# 响应时间 Response Time

- 提交请求到首次响应时间
- 交互式作业



# 其它

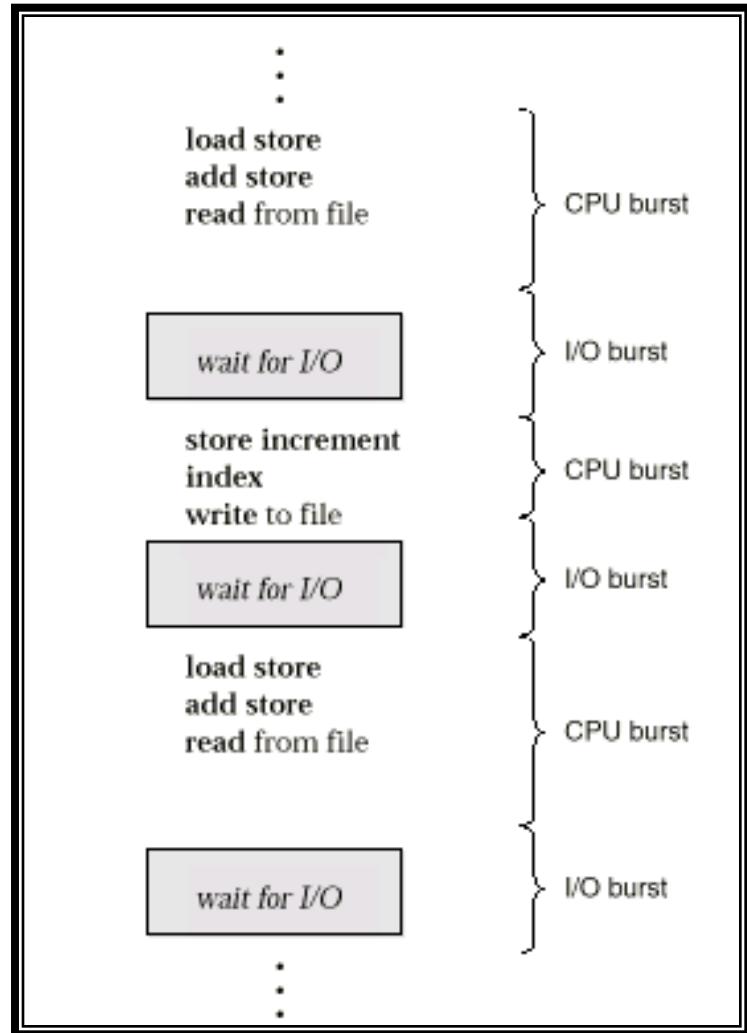
- 截止时间 (deadline)
  - 开始截止时间
  - 完成截止时间
- 优先级准则
  - 需要抢占/剥夺调度

# 面向系统的准则

- 吞吐量 (Throughput) 高
  - 单位时间完成作业数
  - 批处理系统
- 处理机利用率高 (Utilization)
  - keep running!
- 各类资源的平衡利用 (Balancing Resources)
- 公平性 (Fairness)
- 优先级 (Enforcing Priority)

# 程序特征

- 阵发期
  - CPU burst cycle: 进程(线程)使用CPU计算；
  - I/O burst cycle: 进程(线程)使用设备I/O。
- 进程运行行为
  - CPU、I/O交替



# 调度算法——资源分配问题

- 根据系统的资源分配策略所规定的资源分配算法。对于不同的系统目标，通常采用不同的调度算法。



# 先来先服务——FCFS

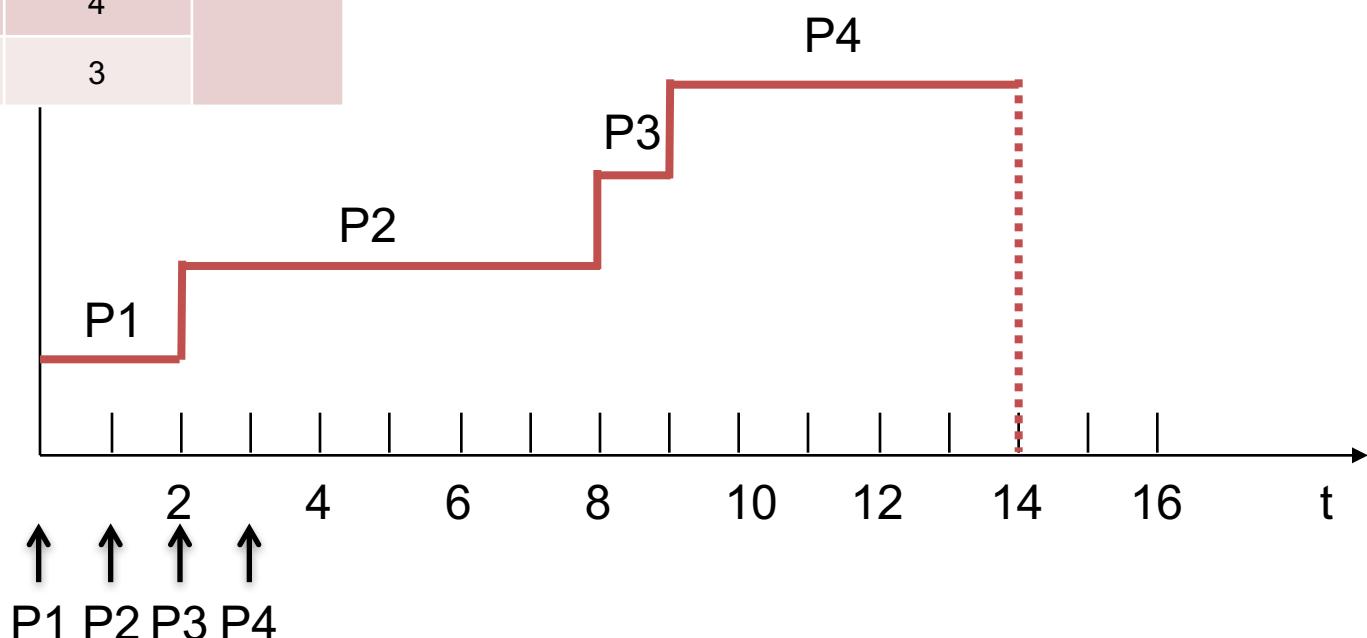
- 先来先服务 (First-Come-First-Served)
  - 当每个进程就绪后，它加入就绪队列（队尾）。
  - 当正在运行的进程停止执行时，选择在就绪队列中存在时间最长的进程运行（队首）

# 举例

进程名	产生时间	服务时间	优先级	时间片
P1	0	2	2	
P2	1	6	1	
P3	2	1	4	1
P4	3	5	3	

# FCFS

进程名	产生时间	服务时间	优先级	时间片
P1	0	2	2	
P2	1	6	1	
P3	2	1	4	
P4	3	5	3	



$$\text{平均周转时间: } ((2-0)+(8-1)+(9-2)+(14-3))/4 = 6.75$$

# 先来先服务——FCFS

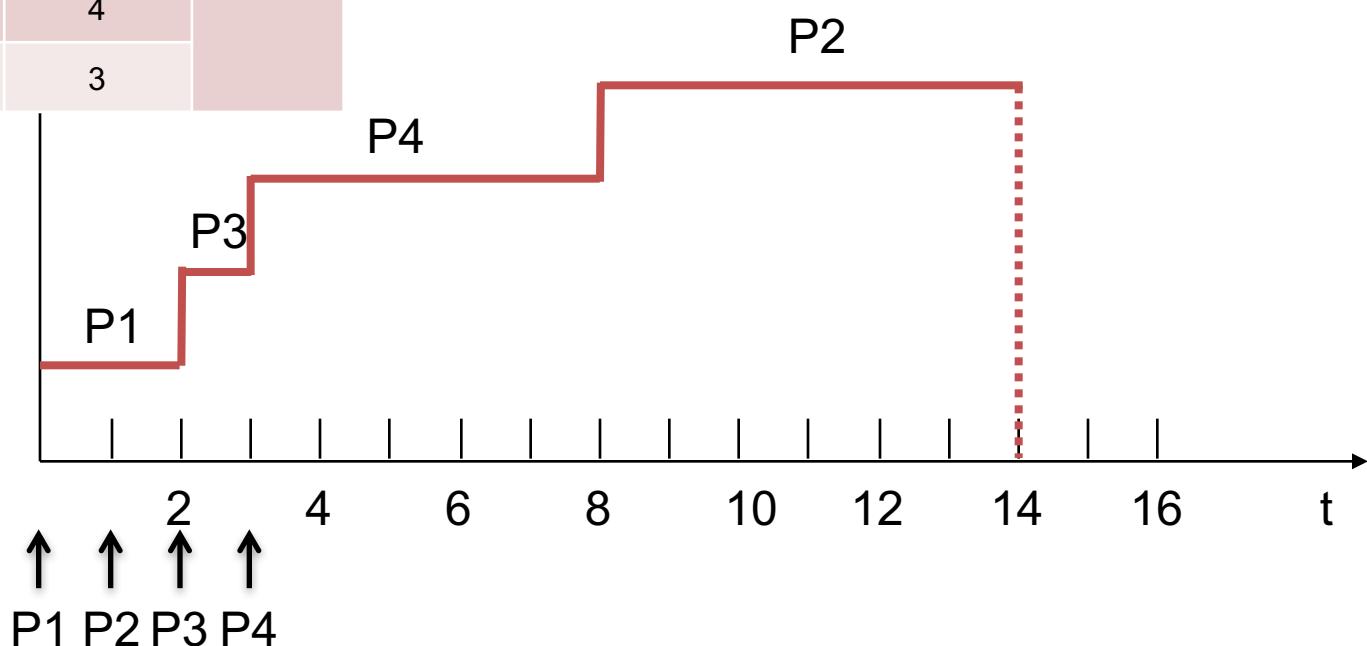
- 评价
  - 非抢占调度
  - 对于长进程有利，而不利于短进程。
  - 有利于CPU繁忙型的进程，不利于I/O繁忙型的进程
  - 不能直接用于分时系统
  - 往往与其它调度算法综合使用

# 最短作业优先

- Shortest Job/Process First/Next,  
SJF/SPF/SJN/SPN
- 非剥夺，下一次选择所需处理时间最短的进程。
- 如果两个进程剩余时间相同，则使用FCFS来调度。

# SJF

进程名	产生时间	服务时间	优先级	时间片
P1	0	2	2	
P2	1	6	1	
P3	2	1	4	
P4	3	5	3	



平均周转时间:  $(2+13+1+5)/4 = 5.25$

# 最短作业优先

- 评价
  - 有利于短进程，提高了平均周转时间
  - 长进程可能被饿死（starvation）
  - 需要知道或估计每个进程的处理时间。

# 基于优先权/级的调度算法

- 优先级
  - 每个进程设有一个优先级，调度程序选择具有较高优先级的进程。
- 静态优先级(static)
  - 优先数在进程创建时分配，生存期内不变。
  - 响应速度慢，开销小。
  - 适合批处理进程
- 动态优先级(dynamic)
  - 进程创建时继承优先级，生存期内可以修改。
  - 响应速度快，开销大。

# 静态优先级

- 进程优先级在整个运行期不变
- 确定优先级依据
  - 进程类型（重要性、紧迫性）
  - 进程对资源的需求
  - 均衡系统资源使用
  - 用户需求

# 优先级调度

- 问题
  - 低优先级的进程可能会饿死（无穷阻塞）
- 改进
  - 一个进程的优先级随着它的时间或执行历史而变化——老化策略(aging)。

# 动态优先级

- 执行过程中不断调整其优先级
  - 如：优先级随执行时间增加而下降，随等待时间增加而升高。
- 优点：长短兼顾

# 轮转调度RR(Round Robin)

- 时间片调度(time slicing): 以一定的时间间隔周期性产生时钟中断，当前正在运行的进程被置于就绪队列尾，然后基于FCFS选择下一个就绪进程运行。
- 时间片的长度从几ms~几百ms
- 专门为分时系统设计

# 轮转调度

- 时间片长度变化的影响
  - 过长：退化为FCFS，进程在一个时间片内都执行完，
  - 过短：用户的一次请求需要多个时间片才能处理完，上下文切换次数增加。
- 评价：
  - 相对公平
  - 倾向于CPU型的进程
  - 中断开销。

# 最短剩余时间调度SRT

- Shortest Remaining Time (First), SRT(F)
- 原理：
  - 对SJF增加了剥夺机制
  - 选择预期剩余时间最短的进程，当一个新进程加入就绪队列时（时机），它可能比当前运行的进程具有更短的剩余时间。
- 优点：
  - 既不偏爱长进程，也不像RR算法那样会产生额外的中断，从而减少了开销。
  - 周转时间方面，SRT比SJF性能要好，短作业可以立即被选择执行。

# 最短剩余时间调度SRT

- SRT问题：
  - 需要知道或估计每个进程所需处理时间；
  - 若持续有短进程存在，长进程可能被饿死；
  - 记录过去的服务时间（以便计算剩余时间）→ 增加了开销。

# 高响应比优先算法HRRN

- Highest Response Ratio Next

- 非抢占式

- 响应比R:

$$R = \text{周转时间}/\text{服务时间} = (w + s)/s$$

$w$  = 等待时间,  $s$  = 服务时间

- 评价:

- FCFS和SJF的结合，克服了两种算法的缺点

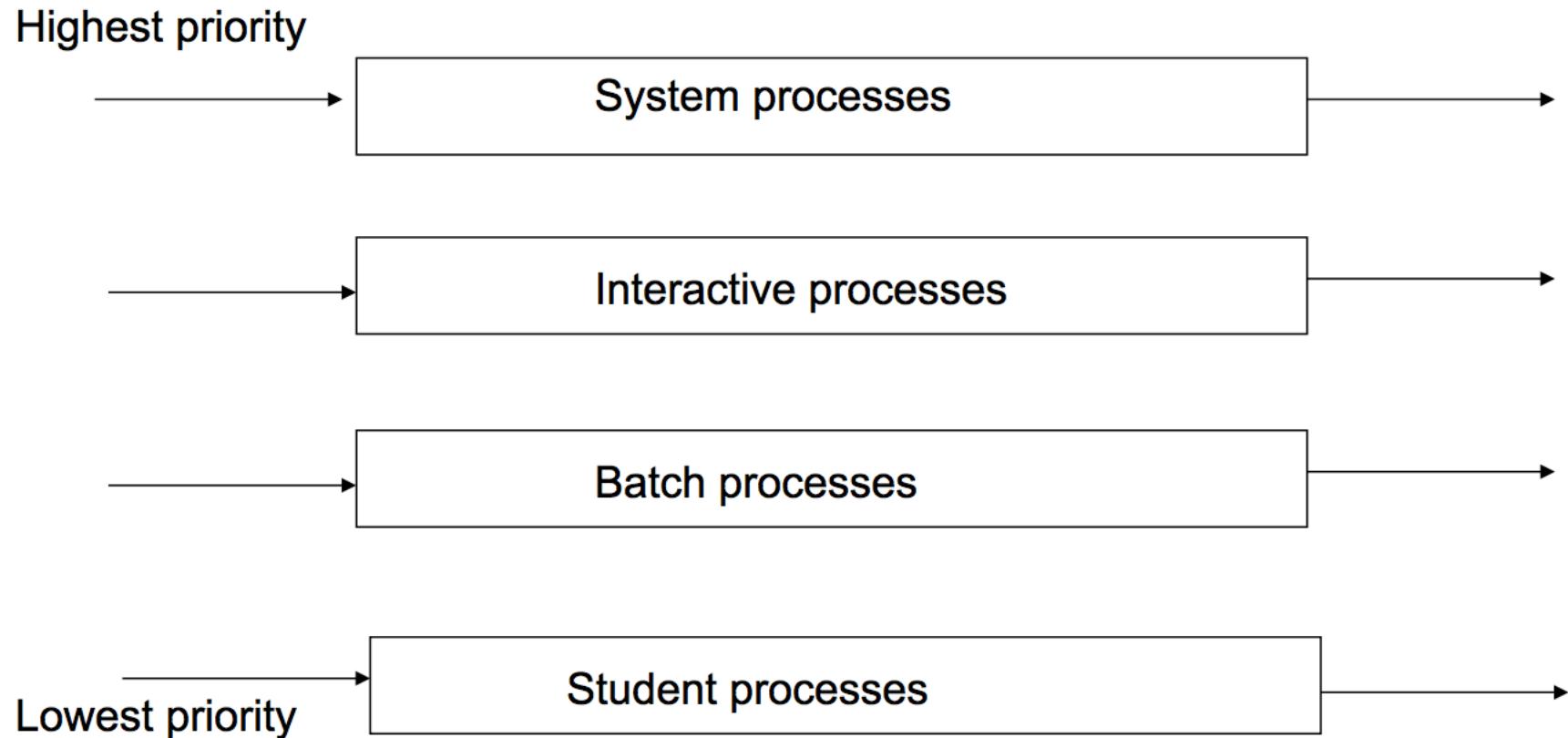
- 公平，吞吐率大

- 需要估计服务时间，增加了计算，增加了开销

# 多级队列调度算法

- Multilevel Queues：将就绪队列分成多个独立队列，进程所属的队列固定。通过对各队列的区别对待，达到一个综合的调度目标。
- 例如：通用系统中：
  - 队列1：实时进程就绪队列（优先级）
  - 队列2：分时进程就绪队列（RR）
  - 队列3：批处理进程就绪队列（优先级）

# 多级队列调度算法



# 多级队列调度算法

- 各队列不同处理：不同队列可有不同的调度策略等。
  - 如前台对队列用RR，后台队列用FCFS。
- 队列之间的区别：采用固定优先级、可抢占调度来实现。
  - 如前台队列优先级高于后台队列优先级。
- 只有优先级高的队列中没有进程时，才可以调度优先级低的队列中的进程

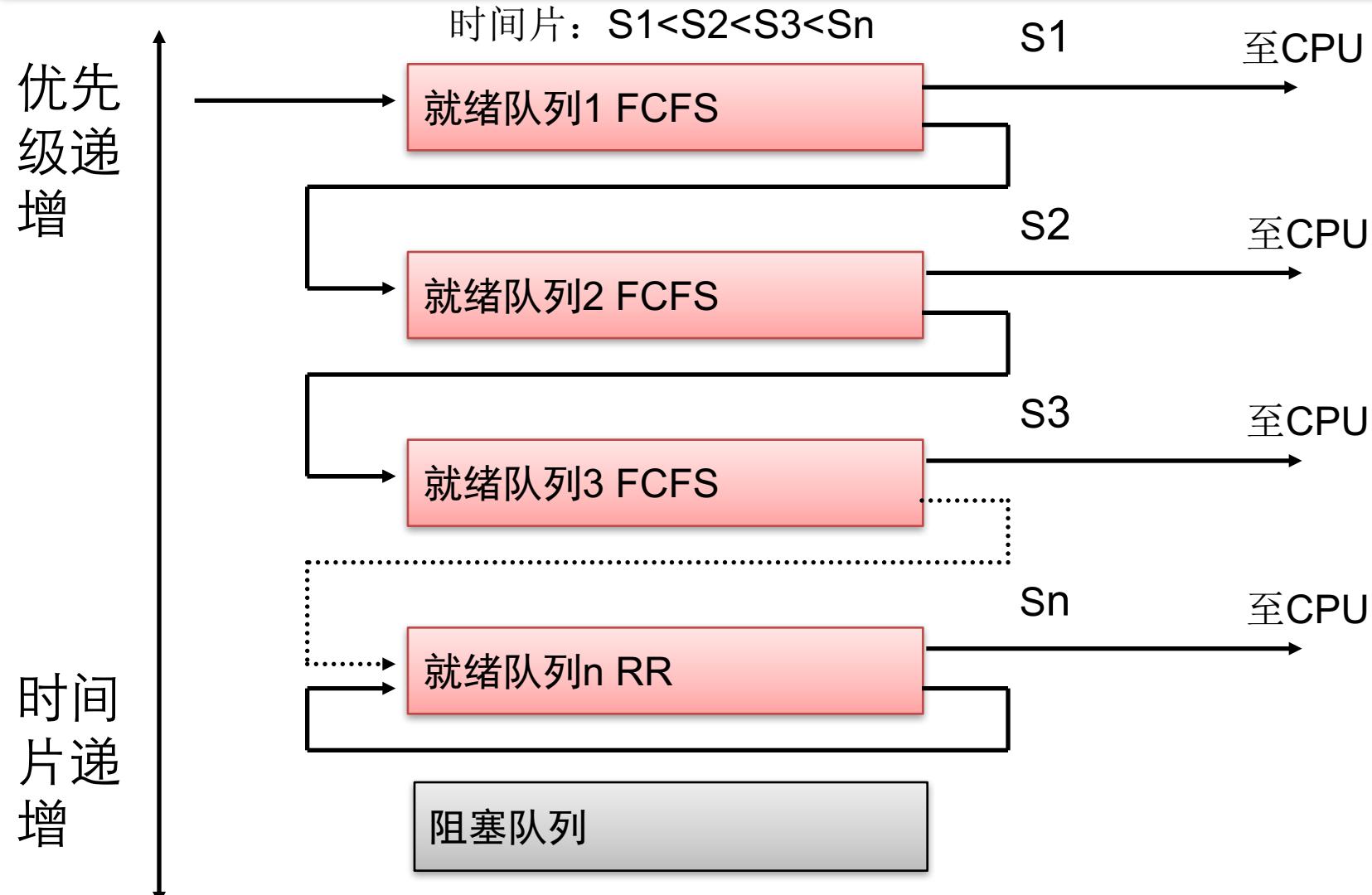
# 多级反馈队列调度

- 出发点
  - SJF, SRT都需要知道进程的运行时间，有局限性。
  - 如果没有关于各个进程相对长度的信息，可以以已经执行了的时间进行衡量
  - 按时间片、等待时间，使用动态优先级机制
  - 调度基于剥夺原则
- 多级反馈队列算法是时间片轮转算法和优先级算法的综合和发展

# 多级反馈队列调度

- Multilevel Feedback Queues
  - 多个就绪队列，进程所属队列可变，即进程可以在不同的就绪队列之间移动
  - 多个就绪队列分别赋予不同的优先级
  - 队列优先级逐级降低，而时间片长度逐级递增

# 多级队列反馈调度算法

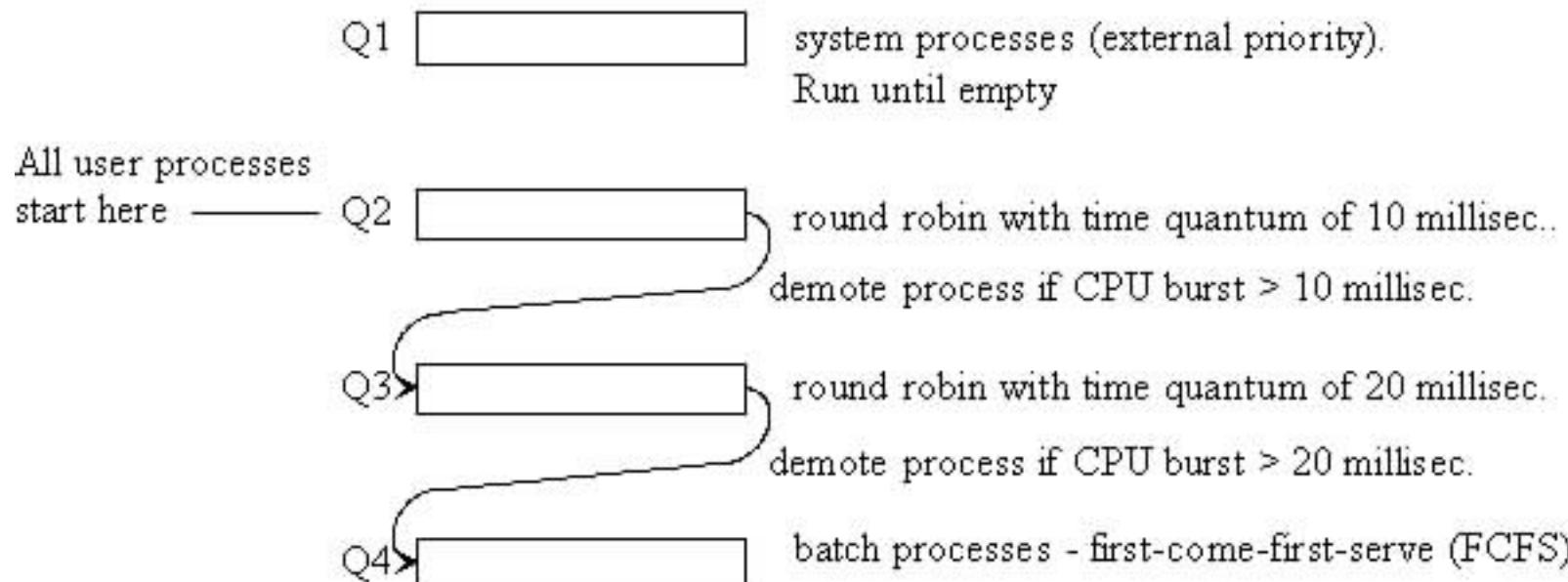


# 多级反馈队列调度过程

- 新进程进入内存后，先放入队列1的末尾，按FCFS顺序调度；
- 若执行过程中阻塞，则离开队列，被唤醒后，被放入同一优先级队列尾部；
- 若在一个时间片未能执行完，则降低优先级投入队列2的末尾，同样按FCFS算法调度；如此下去，直到最后的队列；
  - 对I/O繁忙进程有利
- 最后队列按RR算法调度直到完成。
- 若在最后队列阻塞或等待时间过长，提升优先级；
- 仅当较高优先级的队列为空，才调度较低优先级的进程执行。
- 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。

# MFQ

## Example of Multilevel Feedback Queues



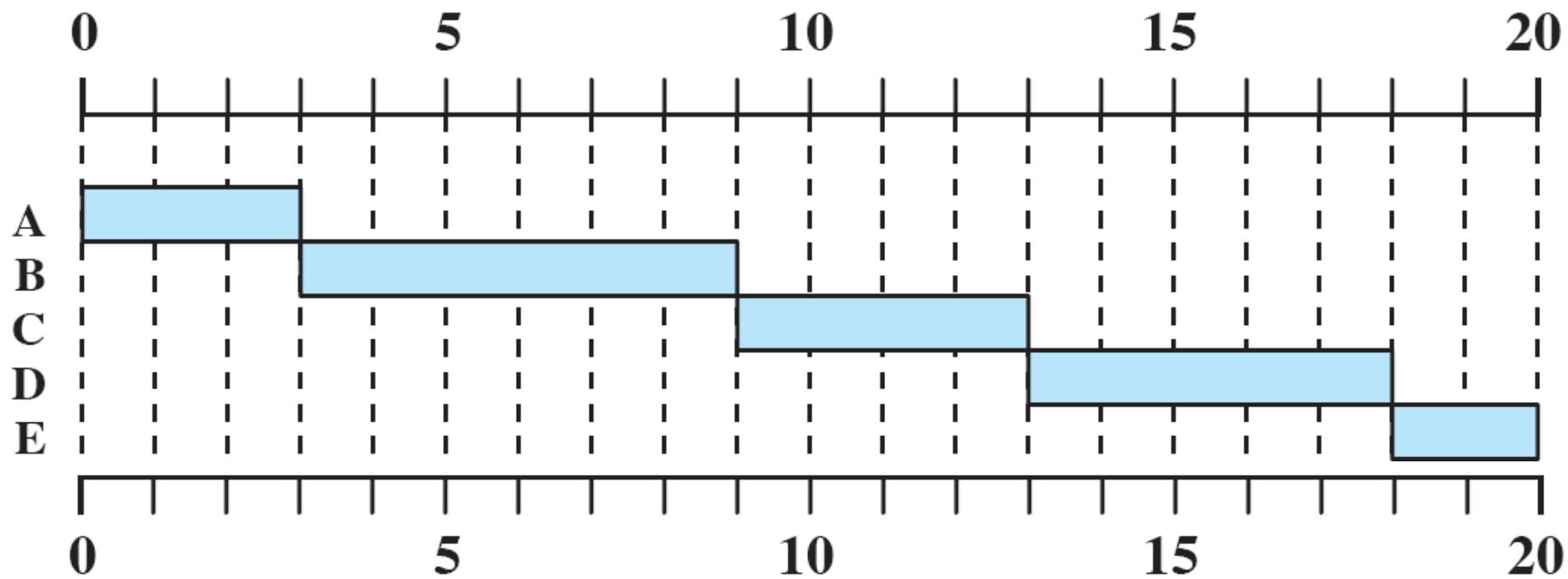
Processes in Q3 and Q4 might be promoted to the next higher queue if they have not run in 1 second.

# 调度算法举例——甘特图

进程	到达时间	服务时间
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

# 先来先服务(FCFS)

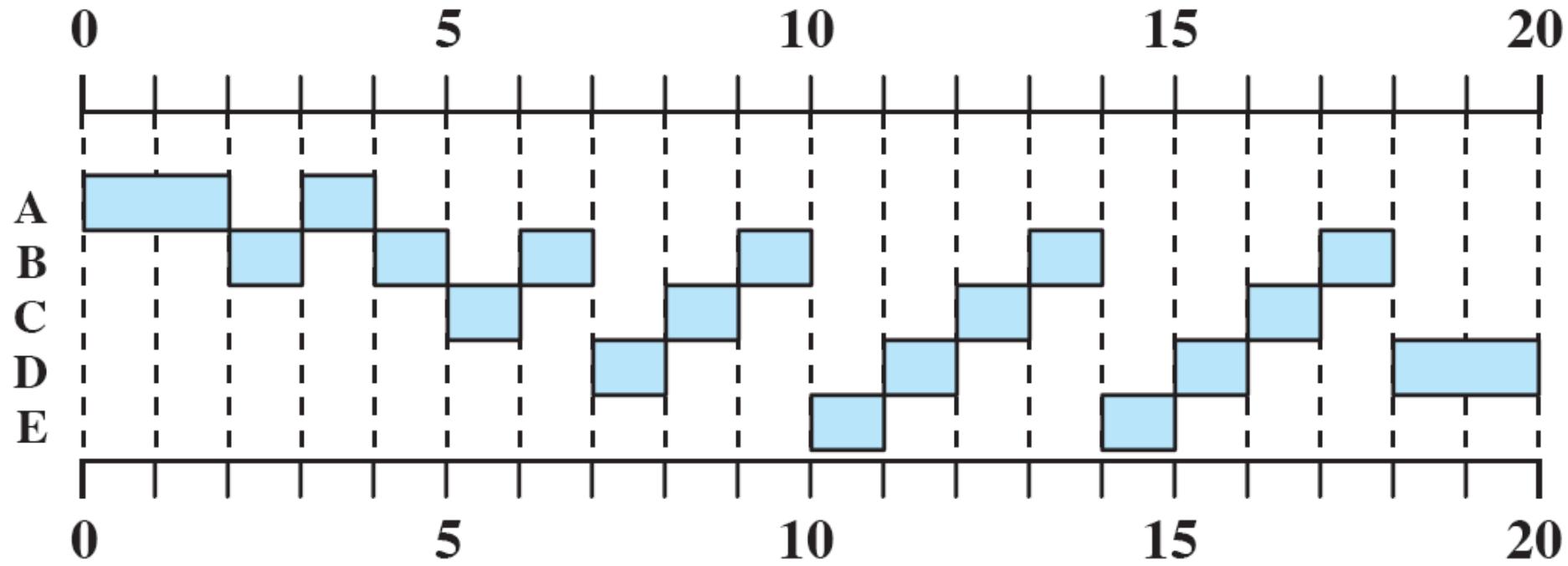
进程	到达时间	服务时间
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



# 时间片轮转调度( $q=1$ )

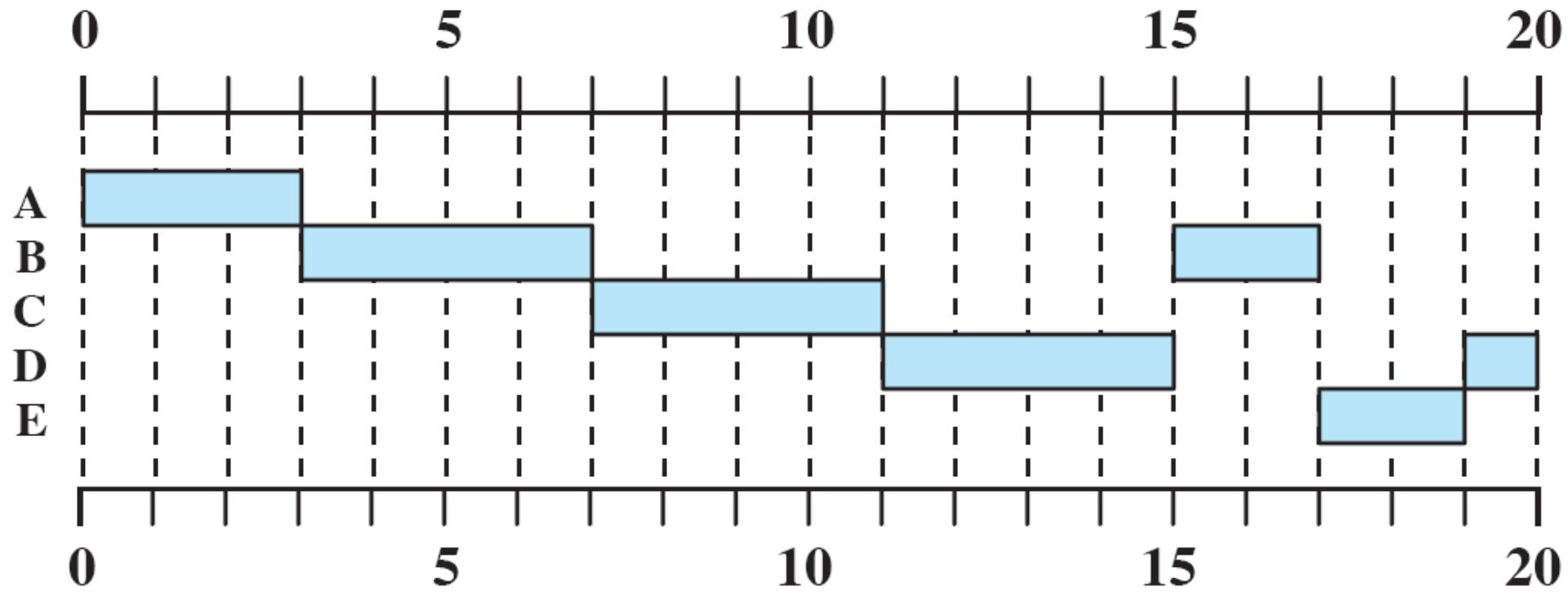
假定：当前进程调度结束置于队列尾部时，同一时刻进入的进程先放入队列。

进程	到达时间	服务时间
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



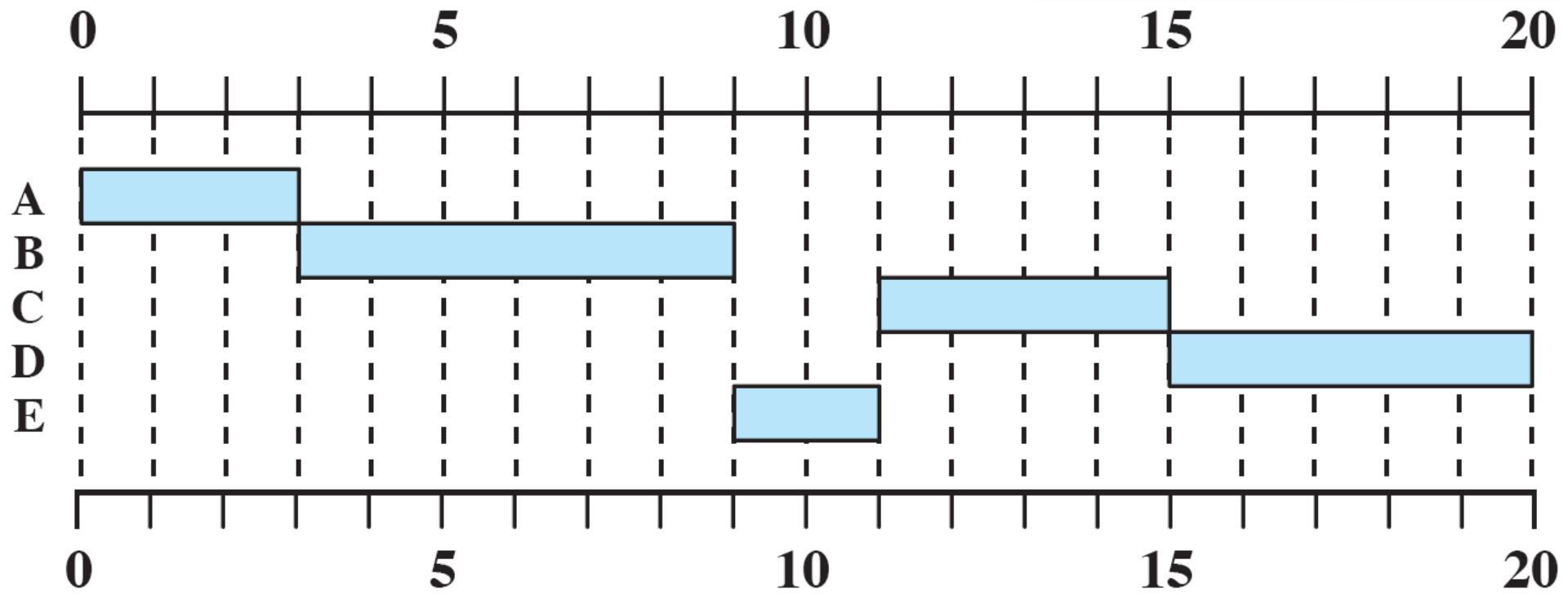
# 时间片轮转调度(q=4)

进程	到达时间	服务时间
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



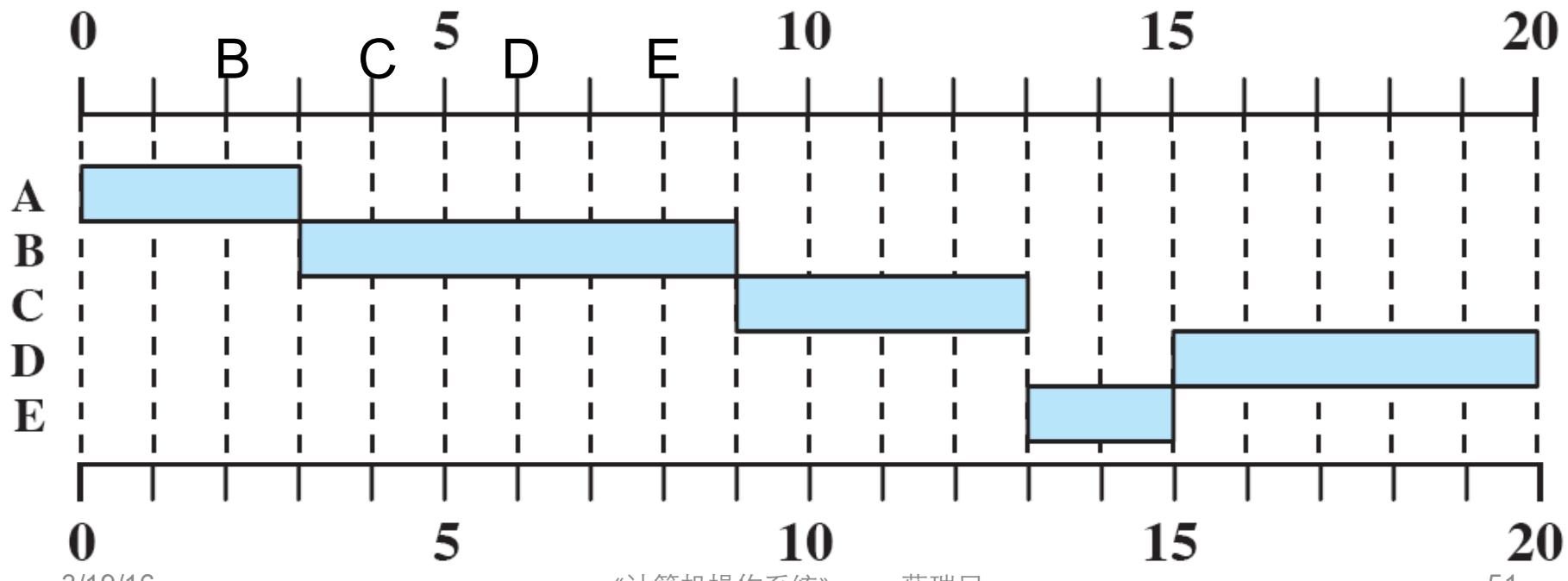
# 短作业优先调度(SJF)

进程	到达时间	服务时间
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



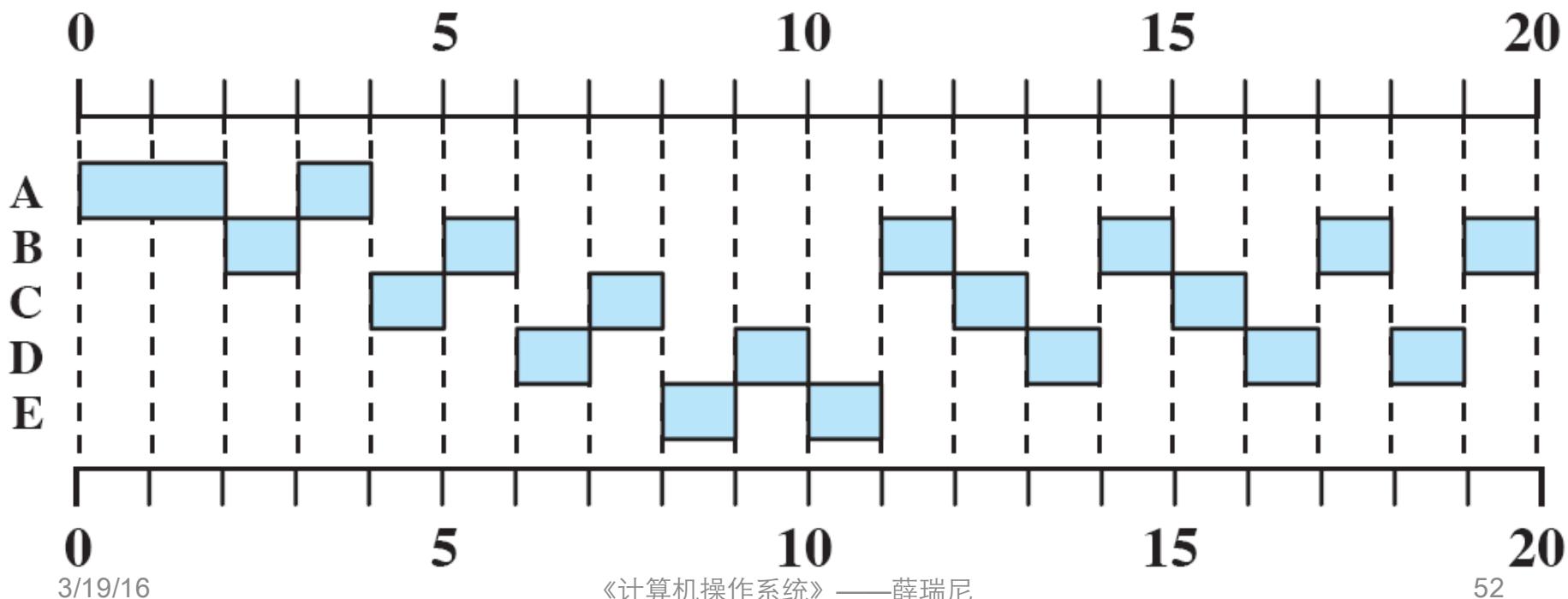
# 响应比优先调度(HRRN)

进程	到达时间	服务时间
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



# 多级反馈队列调度( $q=2^i$ )

进程	到达时间	服务时间
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



# 调度算法例子

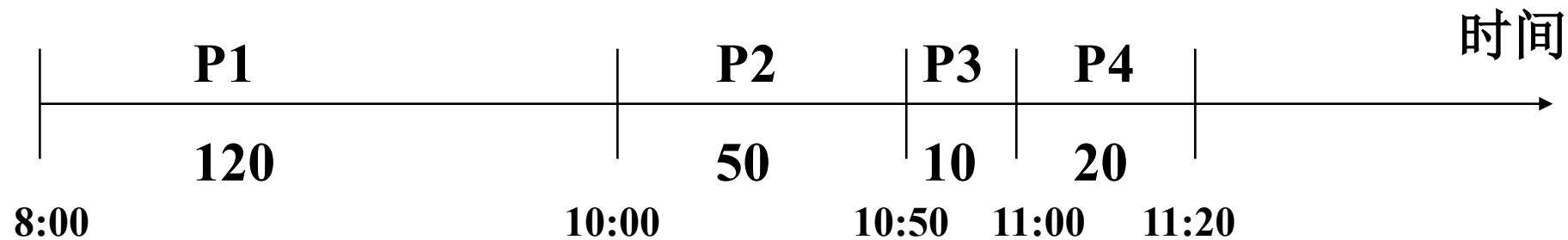
# 例子

- 假设在单道批处理环境下有四个进程，已知它们进入系统的时间、估计运行时间：

进程	进入时间	估计运行时间（分钟）
P1	8:00	120
P2	8:50	50
P3	9:00	10
P4	9:50	20

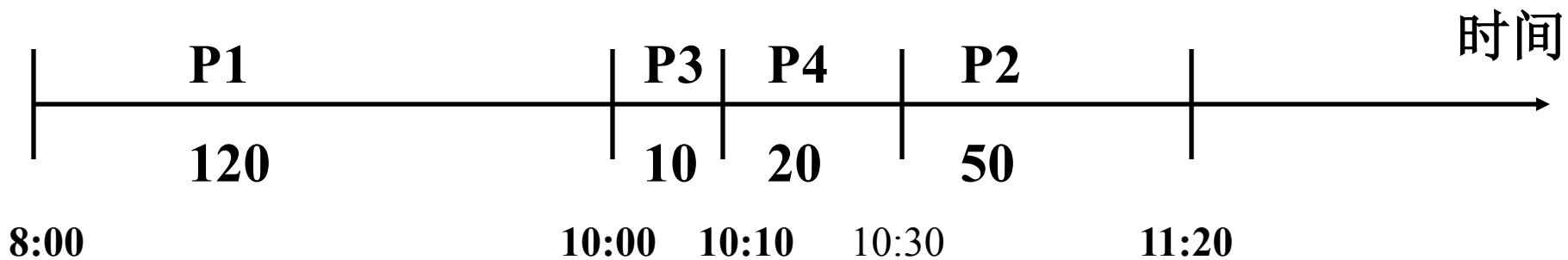
- 要求：应用FCFS、SJF、HRRN，分别给出进程调度的顺序，并计算出平均周转时间

# 先来先服务 (FCFS)



	进入时间	估计运行时间 (分钟)	开始时间	结束时间	周转时间 (分钟)
P1	8:00	120	8:00	10:00	120
P2	8:50	50	10:00	10:50	120
P3	9:00	10	10:50	11:00	120
P4	9:50	20	11:00	11:20	90
平均周转时间 T = 112.5					450

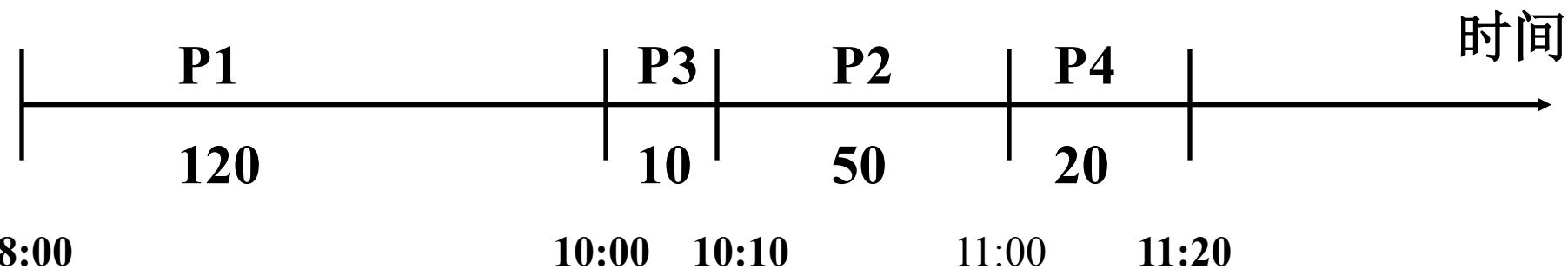
# 最短作业优先算法(SJF)



	进入时间	估计运行时间 (分钟)	开始时间	结束时间	周转时间 (分钟)
P1	8:00	120	8:00	10:00	120
P2	8:50	50	10:30	11:20	150
P3	9:00	10	10:00	10:10	70
P4	9:50	20	10:10	10:30	40
平均周转时间 T = 95					380

# 最高响应比优先算法(HRRN)

$$\frac{70+50}{50}, \frac{60+10}{10}, \frac{10+20}{20} \quad \frac{80+50}{50}, \frac{20+20}{20}$$



	进入时间	估计运行时间 (分钟)	开始时间	结束时间	周转时间 (分钟)
P1	8: 00	120	8: 00	10: 00	120
P2	8: 50	50	10: 10	11: 00	130
P3	9: 00	10	10: 00	10: 10	70
P4	9: 50	20	11: 00	11: 20	90
平均周转时间 T = 102.5					410

算法  
比较项目

FCFS

RR

SJF

SRT

优先级

HRRN

MFQ

选择依据	$\max[w]$	常量	$\min[s]$	$\min(s-e)$	$\max(p)$	$\frac{\max((w+s)/s)}{s}$	$\max(p)$
调度方式	非抢占式	抢占式 (时间片)	非抢占式	抢占式 (进程到达)	抢占式 (进程到达)	非抢占式	抢占式 (优先级, 时间片)
吞吐量	不突出	如果时间 片太小， 可能很低	高	高	高	高	不突出
响应时间	可能很高	好	短作业好	好	好	好	不突出
开销	小	中	中	可能高	可能高	中	可能高
对进程的 影响	不利于短 作业和 I/O繁忙 型作业	公平对待	不利于长 作业	不利于长 作业	不利于低 优先级	良好的均衡	I/O繁忙 型作业优
饥饿问题	无	无	可能	可能	可能	无	可能

# 问题

- 对单道程序而言，各种算法的平均周转时间可能不同，总周转时间是否也不同？

Real-time Scheduling

# 实时调度

# 实时系统



# 实时调度的基本条件

- 就绪时间：成为就绪状态的时间
  - 周期性任务
- 开始/完成截止时间
  - 软、硬
- 处理时间
- 资源要求
- 优先级

# 系统处理能力强

- 假定系统中有m个周期性的硬实时任务，它们的处理时间为Ci，周期为Pi，则在单/多处理机情况下，可调度的必要条件：

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq N$$

# 调度方式

- 剥夺方式
  - 一般都采用此方法
- 非剥夺方式
  - 一般应使实时任务较小，以及时放弃CPU。

# 具有快速切换机制

- 对外部中断的快速响应能力
  - 为使在紧迫的外部事件请求中断时系统能及时响应，要求系统具有快速硬件中断机构，还应使禁止中断的时间间隔尽量短，以免耽误时机。
- 快速的任务分派能力
  - 应使系统中的每个运行功能单位适当的小，以减少任务切换的时间开销。

# 实时调度算法的分类

## 基于时间片轮转调度

- 数秒

## 基于优先级的非抢占调度

- 数百毫秒~秒

## 基于抢占点的抢占调度

- 数毫秒~数十毫秒

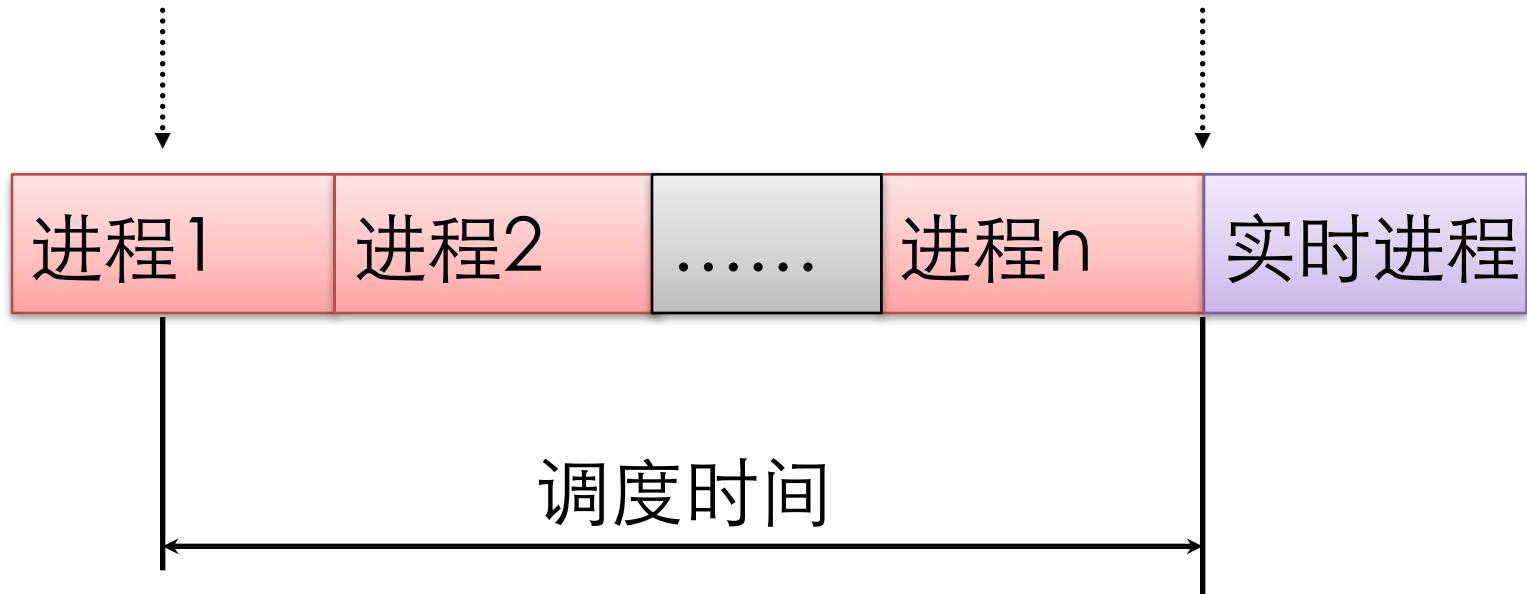
## 立即抢占调度

- 微秒~毫秒

# 时间片轮转调度

实时进程要求调度

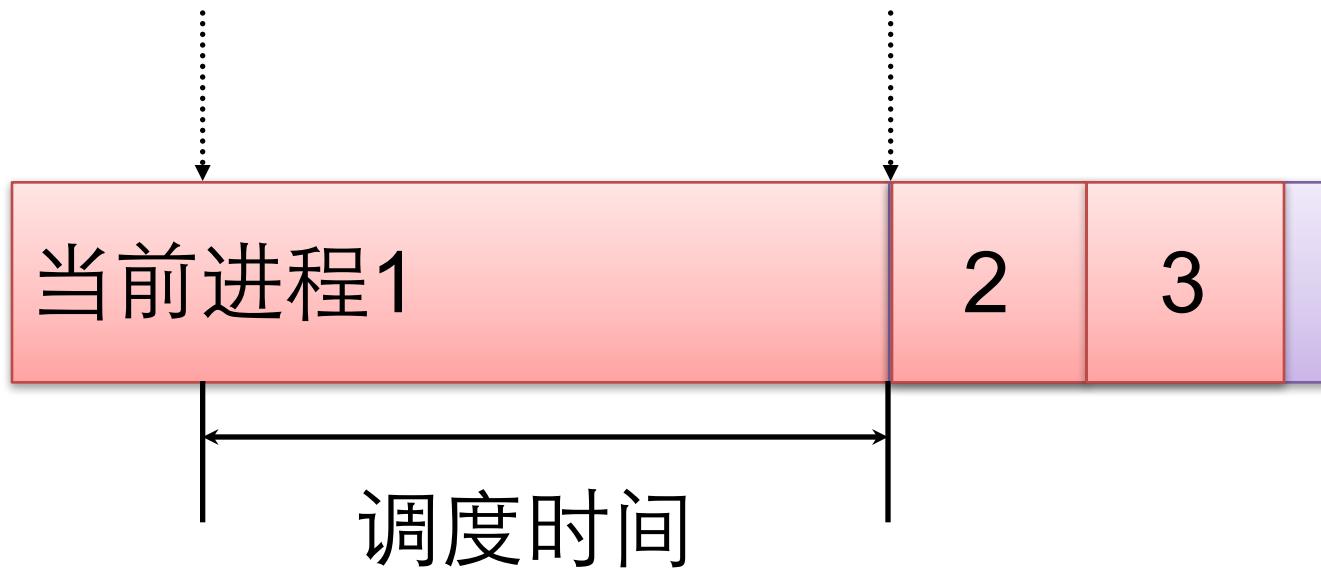
调度实时进程运行



# 基于优先级的非抢占调度

实时进程要求调度

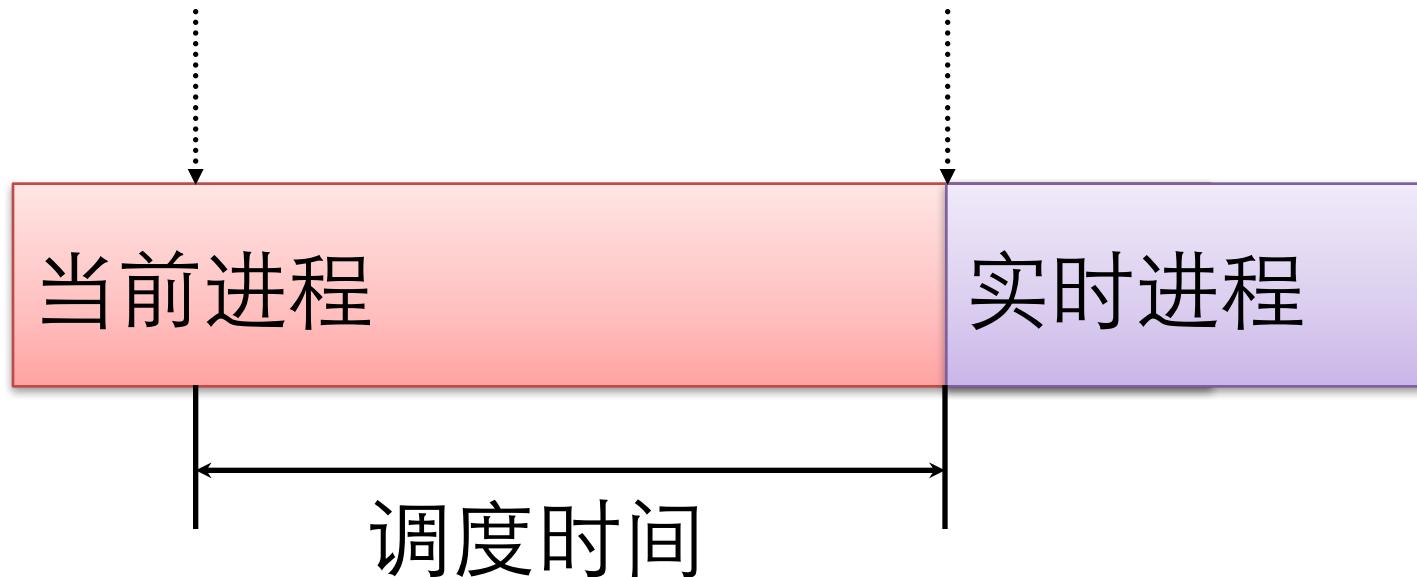
当前进程运行完成



# 基于抢占点的抢占调度

实时进程要求调度

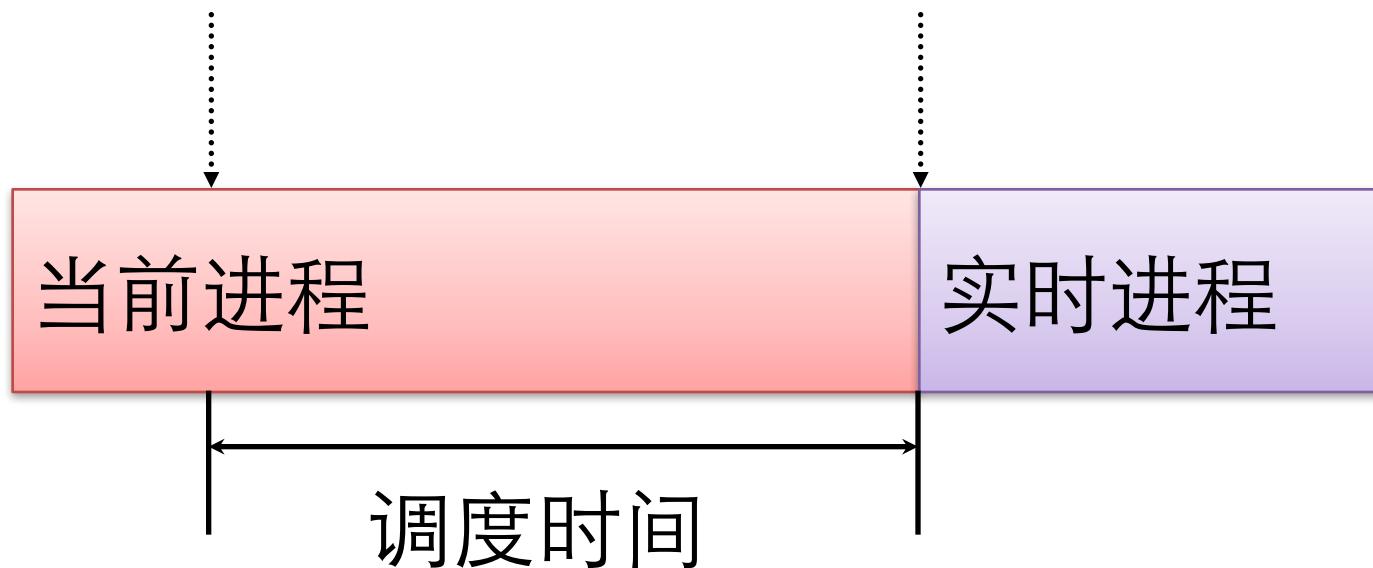
抢占点到达时



# 立即抢占调度

实时进程要求调度

立即抢占

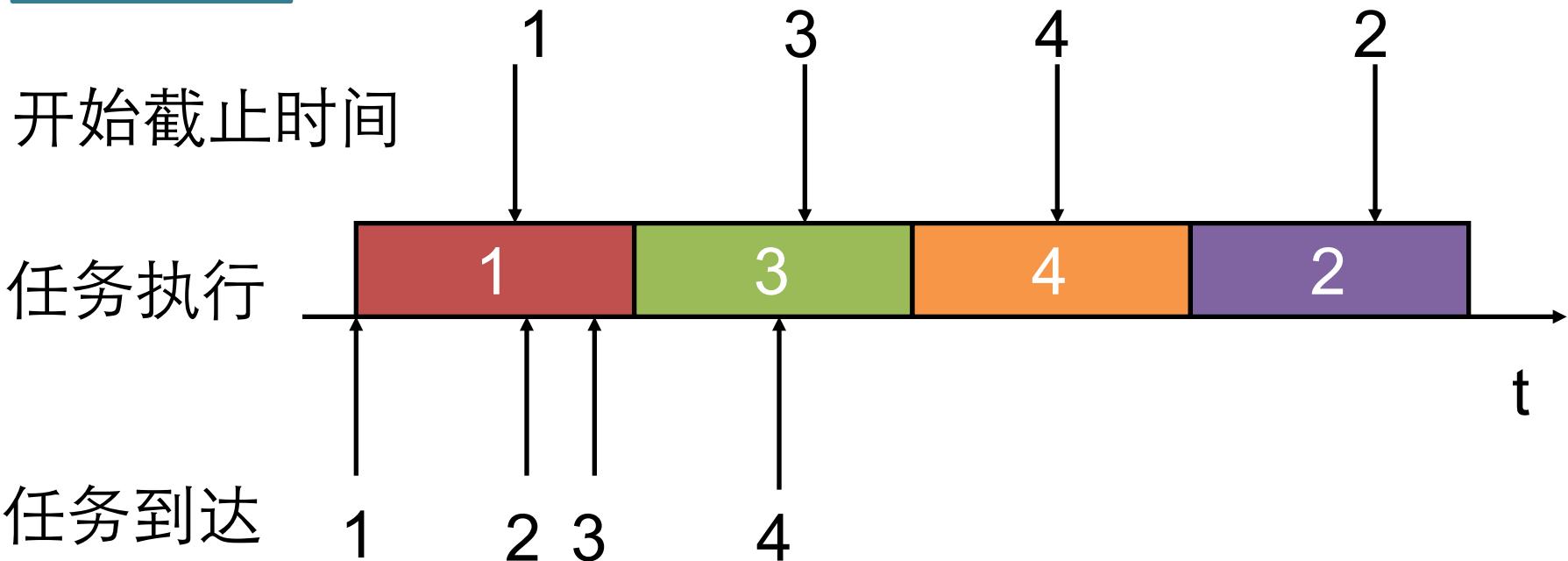


# 实时调度算法

- 最早截止时间优先EDF
  - Earliest Deadline First
  - 根据任务的截止时间来确定任务的优先级
  - 截止时间越早，优先级越高
  - 可以是抢占式或非抢占式

# EDF实例

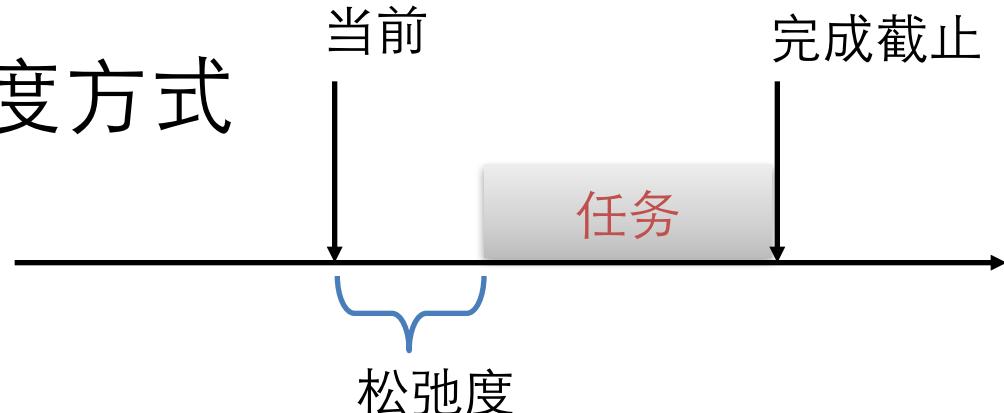
任务长度



EDF算法用于非抢占调度方式

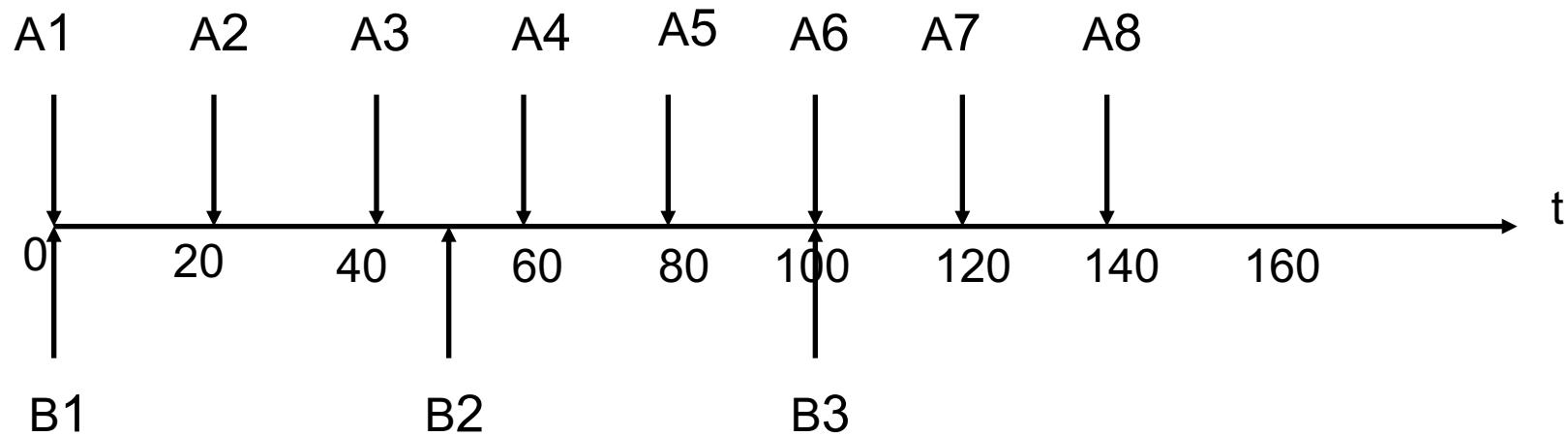
# 最低松弛度优先算法

- LLF: Least Laxity First
  - 松弛度=完成截止时间-当前时间-剩余执行时间
  - 若A进程需在200ms时完成，运行需要100ms，当前时刻是10ms，则A的松弛度： $200 - 10 - 100 = 90$
- 主要用于可抢占调度方式

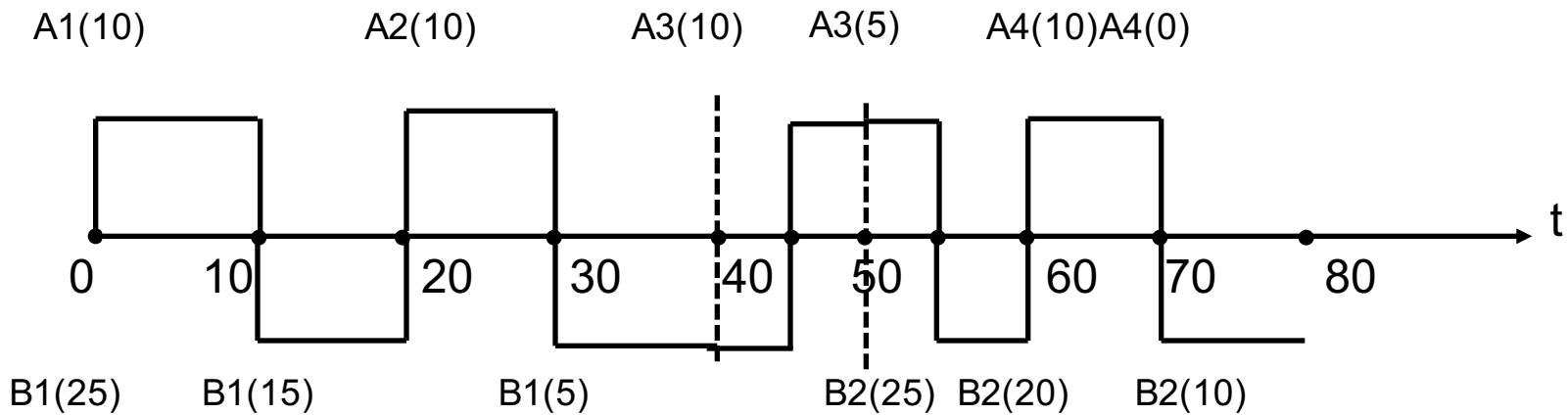


# LLF例子

- 在一个实时系统中，有两个周期性实时任务A和B；抢占点：10n
  - A: 周期: 20ms, 执行时间: 10ms;
  - B: 周期: 50ms, 执行时间: 25ms。



# LLF例子



松弛度=完成截止时间-剩余执行时间-当前时间

A: 周期: 20ms, 执行时间: 10ms;

B: 周期: 50ms, 执行时间: 25ms。

抢占点: 10n

# 速度单调调度

- Rate Monotonic Scheduling
- 任务周期：周期性任务
- 任务速度： $1/\text{任务周期}$
- 速度越快，优先级越高

Multi-processor Scheduling

# 多处理机调度

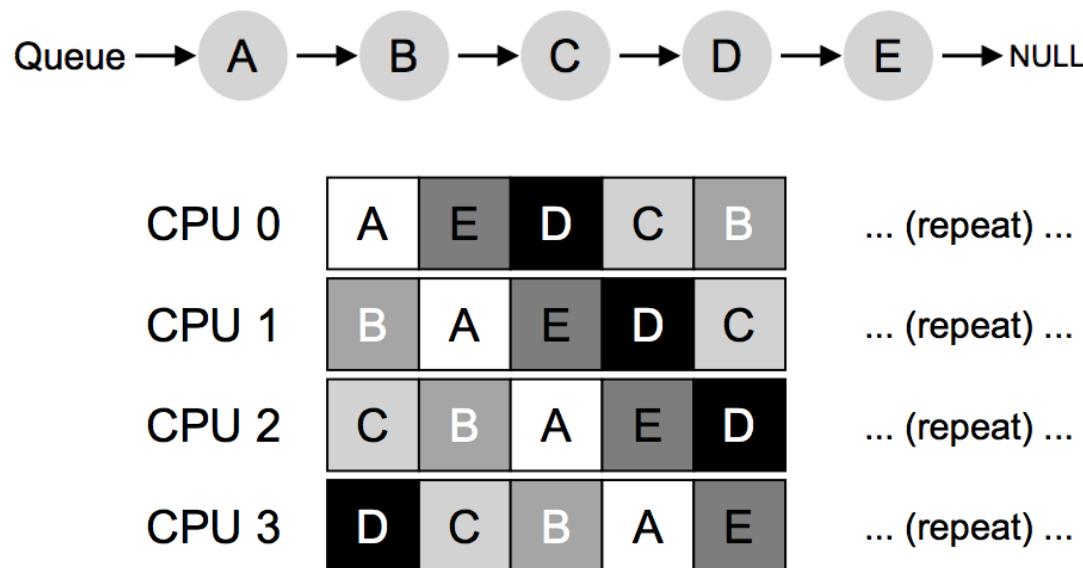
# 多处理器系统中的调度

- 紧密耦合
  - 共享RAM和I/O
  - 高速总线和交叉开关连接
- 松弛耦合
  - 独立RAM和I/O
  - 通道和通信线路连接
- 对称多处理器系统
  - SMP: Symmetric Multiprocessing
- 非对称多处理器系统
  - AMP: Asymmetric Multiprocessing

# 进程分配方式

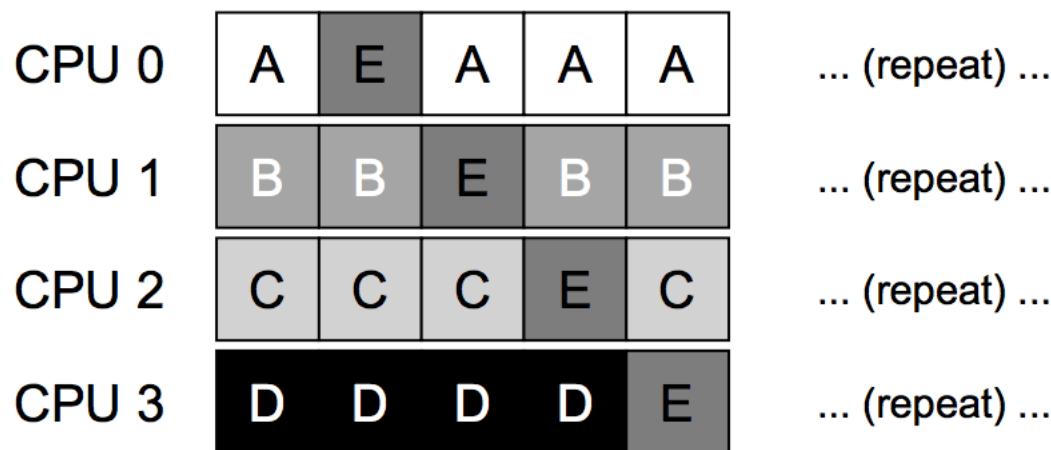
- SMP中进程分配方式
  - 静态分配
  - 动态分配
    - 可防止系统中多个处理器忙闲不均
- 非SMP中进程分配方式
  - 进程调度在主处理器上执行
  - 有潜在的不可靠性

- Single Queue Multiprocessor Scheduling
- 各个处理机自行在就绪队列中取任务（先纵向看，再横向看）



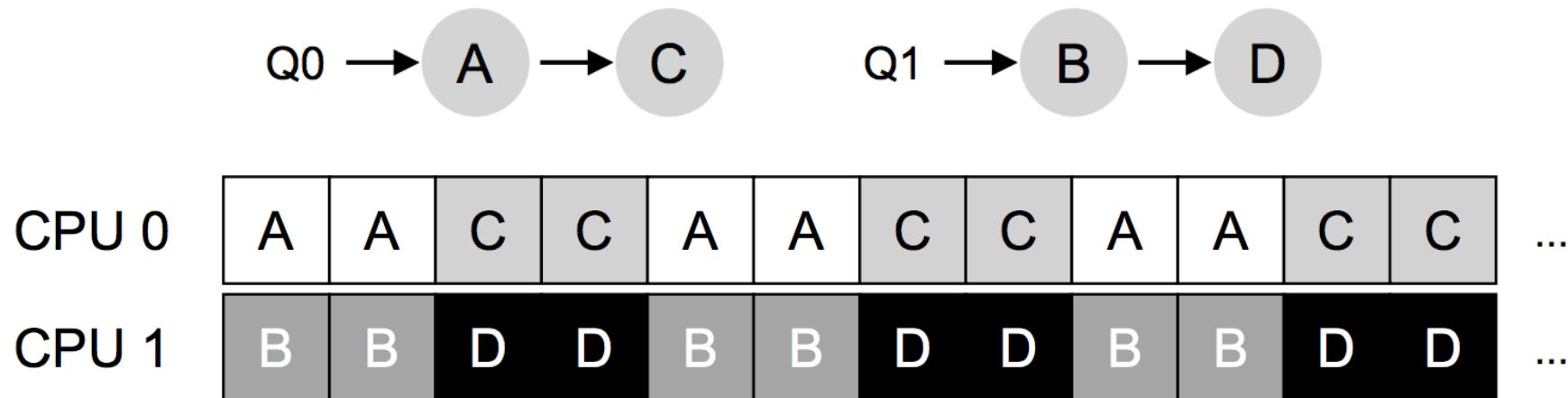
# SQMS

- 简单，分布式调度，多个CPU利用率都好
- 缺点：
  - 瓶颈问题（单队列→共享资源→锁）
  - 低效性：cache affinity （下图为改进）



# MQMS

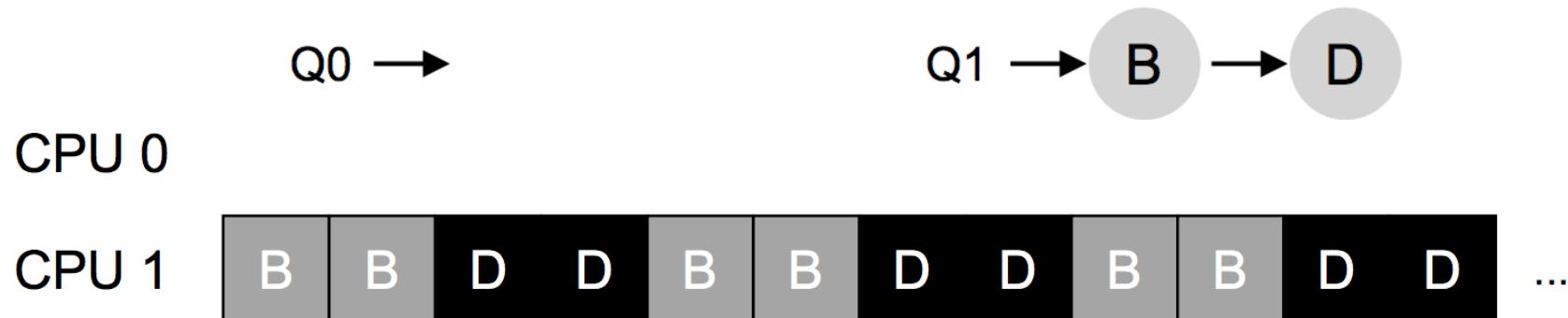
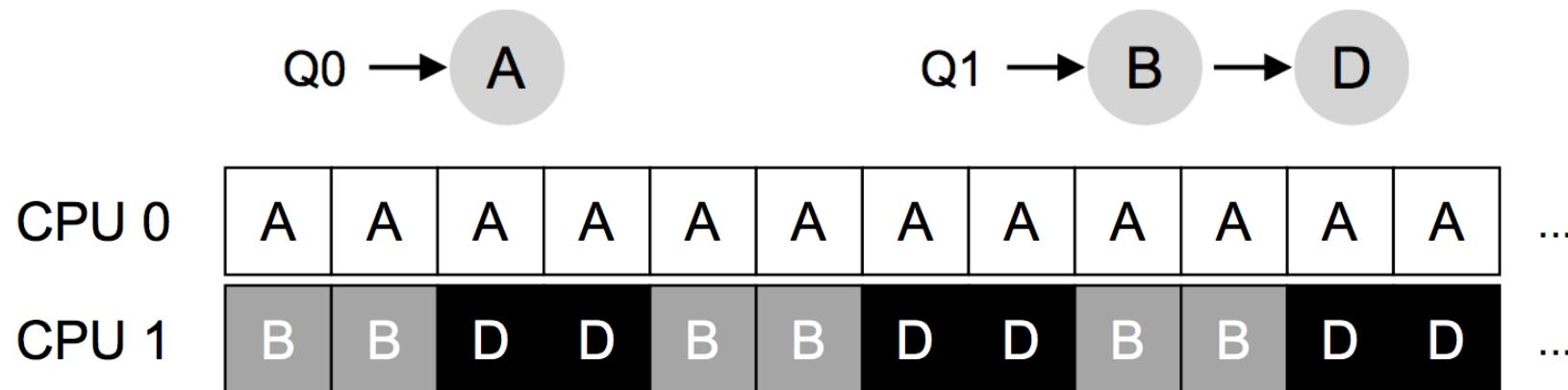
- Multiple Queues
- 每个CPU一个队列：没有SQMS的问题



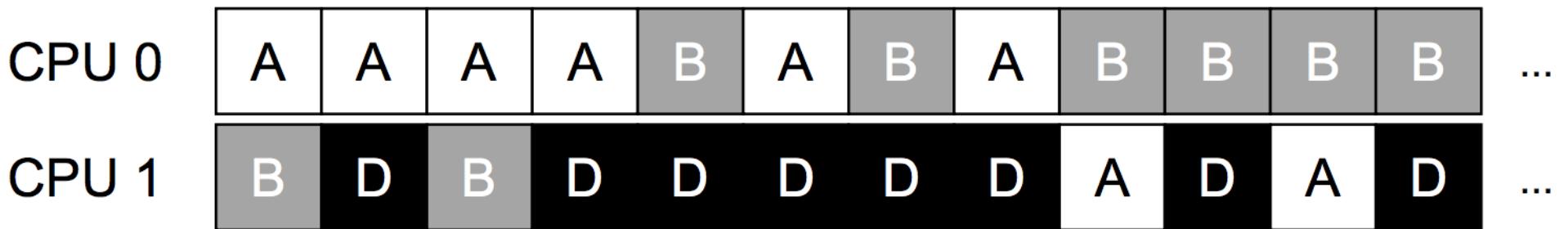
- 新问题？

# 负载不平衡

- Load Imbalance



# Migration & Work Stealing



# 成组调度——Gang scheduling

- 优点：
  - 对相互合作的进（线）程组调度，可以减小切换，减小系统开销。
  - 每次分配一组CPU，减少了调度频率。
- 分配时间
  - 面向程序
  - 面向线程：使处理机利用率更高。

# 专用处理器分配

- Dedicated Processor Assignment
- 多处理机系统，每个处理机不再属宝贵资源。
- 特点：每个进程（线程）专用处理器，使其切换小，提高效率。
- 主要用于大型计算，实时系统

# 操作系统调度实例

# Windows XP

- 优先级：0-31（数值越大，优先级越高）
- 新进程默认优先级
  - IDLE (4)
  - BELOW\_NORMAL (6)
  - NORMAL (8)
  - ABOVE\_NORMAL (10)
  - HIGH (13)
  - REALTIME (24)
- NORMAL进程：当前进程（当前窗口）**时间配额(time quanta)**适当延长
- 每个进程初始一个主线程，再创建子线程
- 内核负责调度线程

# Windows XP Threads

- 线程优先级区分为两组：
  - 可变优先级 (Variable class) (0-15)
  - 实时优先级 (Real-time class) (16-31)
- 处理器绑定 processor affinity
  - CPU可以是物理的或者虚拟的(hyper-threading)
- 每个优先级一个队列
- MQ调度策略：从高到底扫描队列
  - 就绪状态
  - Processor affinity就绪

# Windows XP

- 高优先级实时线程可抢占其它线程
- NORMAL线程时间片结束后，优先级降低
- NORMAL线程等待事件发生后，优先级增大
  - 越慢的I/O（kbd）时间，优先级提升越大
- 当前窗口的线程优先级增大

# Linux

- 基于优先级的可抢占式调度
  - 实时任务被赋予较高优先级，以便与其它进程区分
- 时间片结束后，在所有其它进程时间配额使用完毕前，本进程不被调度。
- 优先级随时间片结束变化。

# Java

- 基于优先级调度 (*loosely-defined scheduling policy based on priorities*)
  - 与父线程相同， 默认1-10。
  - 保持不变， 直到调用： `setPriority`
  - 在支持抢占调度的系统里，在高优先级线程就绪时，低优先级线程仍可能继续执行。
- 基于时间片， 线程一直执行至：
  - 时间配额用完；
  - I/O等待；
  - 退出`run()`函数

# 参考资料

- [http://en.wikipedia.org/wiki/Scheduling\\_algorithm](http://en.wikipedia.org/wiki/Scheduling_algorithm)
- [http://en.wikipedia.org/wiki/O\(1\)\\_scheduler](http://en.wikipedia.org/wiki/O(1)_scheduler)
- [http://en.wikipedia.org/wiki/Completely\\_Fair\\_Scheduler](http://en.wikipedia.org/wiki/Completely_Fair_Scheduler)
- [http://en.wikipedia.org/wiki/Multiprocessor\\_scheduling](http://en.wikipedia.org/wiki/Multiprocessor_scheduling)

# 谢谢！