



# 第4章 进程线程调度

## ■ 本章内容

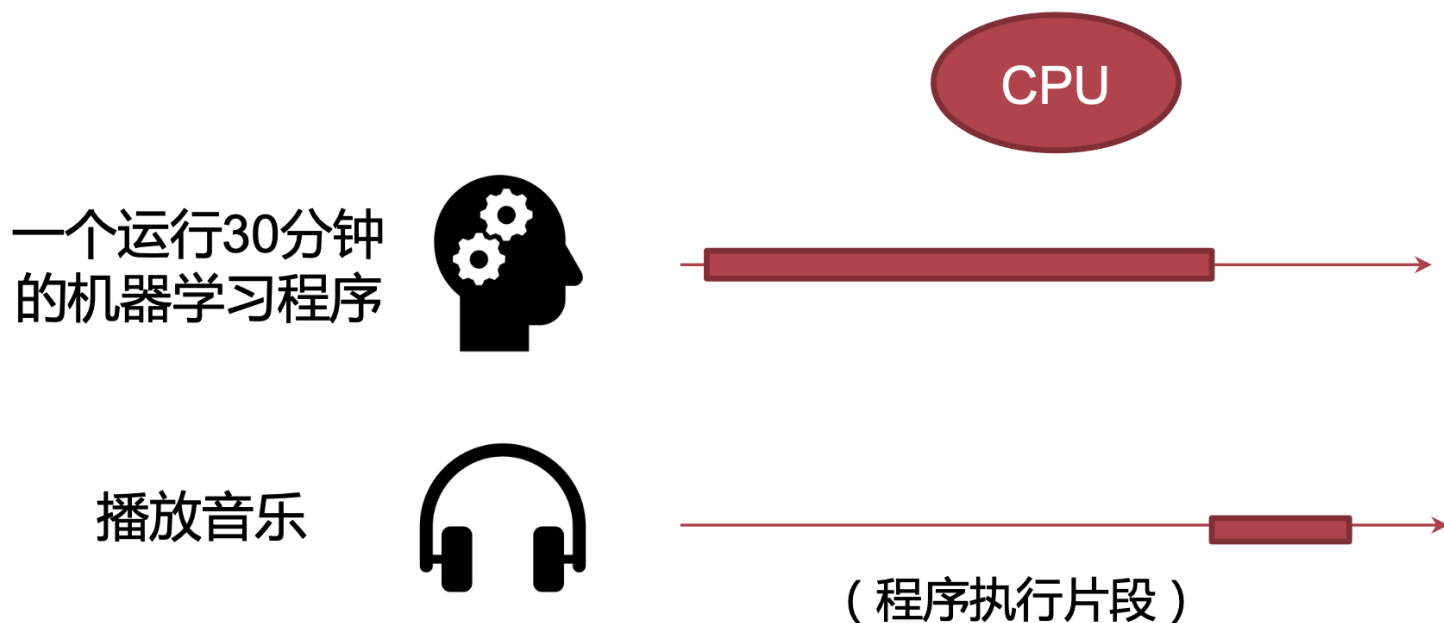
- 1. 多级调度、引发调度的原因、调度过程的实现
- 2. 调度算法设计准则
- 3. 经典的进程调度算法
- 4. 操作系统调度算法实例
- 5. 多处理器调度算法





# 一个应用场景

- 单核CPU上执行30分钟机器学习程序和播放音乐

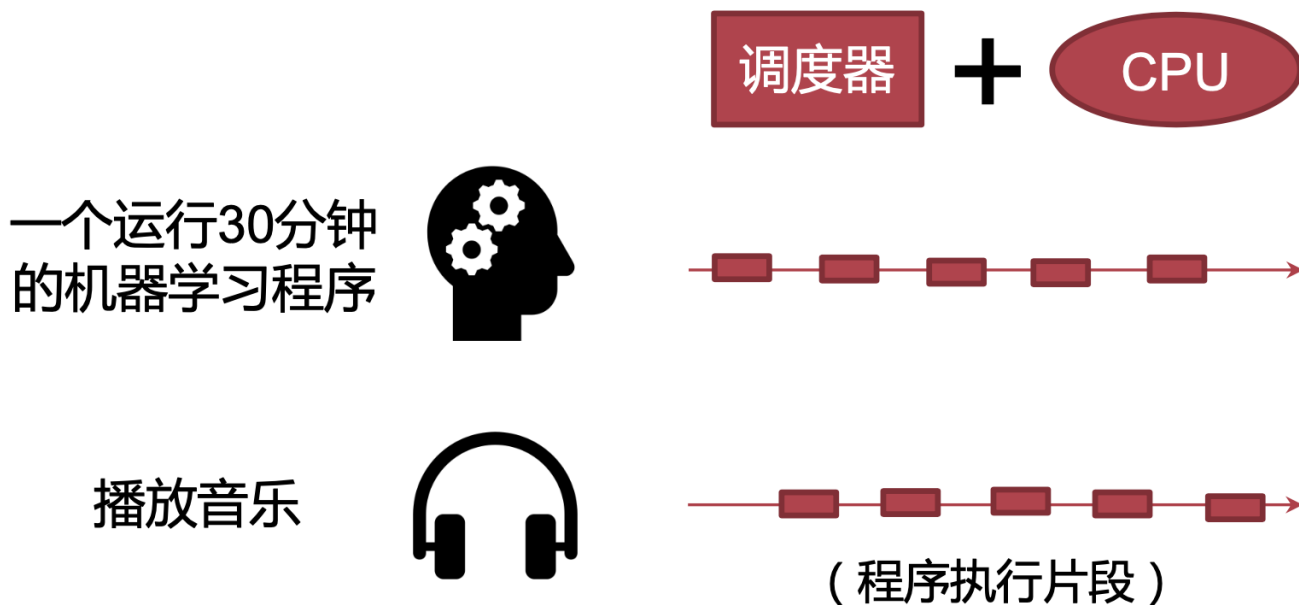


如果没有调度器。。。



# 一个应用场景

- 单核CPU上执行30分钟机器学习程序和播放音乐



调度器将程序切片执行  
程序员可以边听音乐边等待程序运行



# 为什么要研究计算机的调度？

- 系统面临同时处理多个资源请求，但系统资源有限



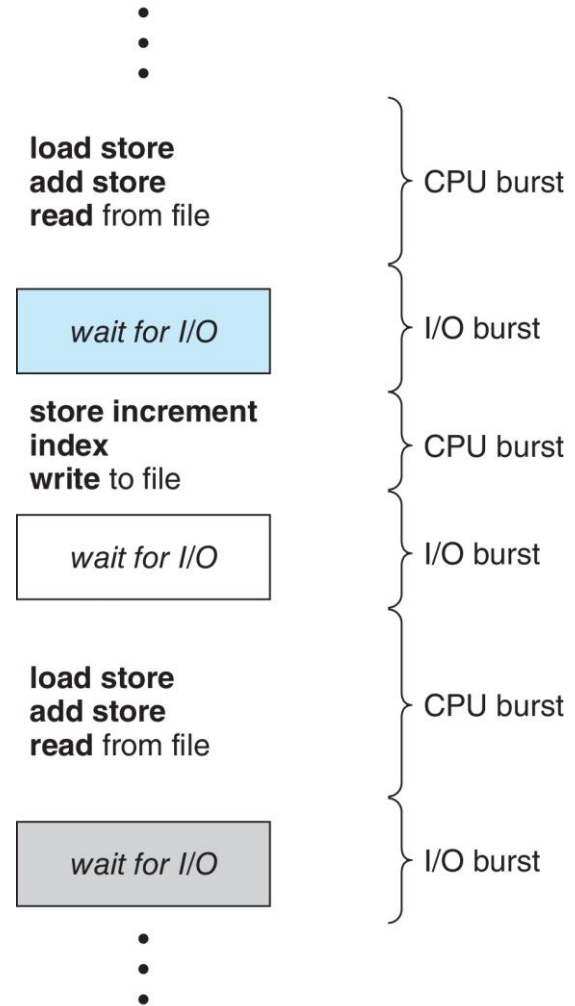
- 调度就是来协调每个请求对资源的使用的方法
- 调度策略有：排队、优先级、时间片轮转等
- CPU、内存、磁盘、网络等资源都面临调度管理





# CPU-I/O执行周期

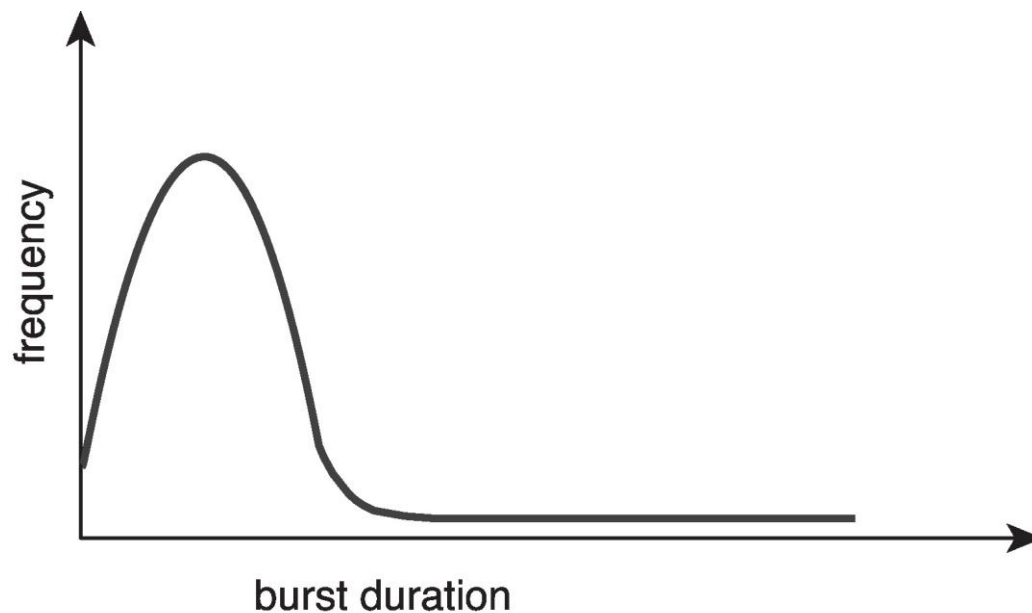
- 通过多道程序设计使得CPU的利用率最大化
- 进程的执行过程是由CPU的执行和等待I/O交替构成的
- CPU执行后跟着I/O执行
- CPU执行时间的分布需要重点关注





# CPU执行时间的直方图

- 大量的短CPU执行时间
- 少量的长CPU执行时间





# 处理机调度

- 根据一定的算法和原则将处理机资源进行重新分配的过程。
- 前提：作业/进程数远远大于处理机数
- 目的：提高资源利用率，减少处理机空闲时间。
- 调度程序：一方面要满足特定系统用户的需求(快速响应)，另一方面要考虑系统整体效率(系统平均周转时间)和调度算法本身的开销。





# 系统中的任务数远多于处理器数

```
1 [|||] 1.3% 5 [|||] 1.3%
2 [|||] 1.3% 6 [|||] 1.3%
3 [|||] 1.3% 7 [|||] 2.0%
4 [|||] 2.0% 8 [|||] 2.0%

Mem[|||||] 7.13G/15.5G Tasks: 169; 1 running
Swp[|||||] 806M/2.00G Load average: 0.01 0.06 0.08
Uptime: 7 days, 05:14:22

PID USER PRI NI VIRT RES SHR S CPU% MEM% TIME+ Command
1 root 20 0 220M 6036 3772 S 0.0 0.0 0:30.34 /sbin/init splash
31077 dongzy 20 0 5080M 288M 21232 S 0.0 1.8 1:00.45 | java -server -Xms512m -Xmx512m -XX:NewRatio=3 -XX:SurvivorRatio=4 -XX:TargetSurvivorRatio=90
30484 root 20 0 296M 10808 9368 S 0.0 0.1 0:00.03 | /usr/sbin/cups-browsed
30483 root 20 0 109M 12520 7036 S 0.0 0.1 0:00.34 | /usr/sbin/cupsd -l
30152 dongzy -31 10 973M 162M 98684 S 0.0 1.0 0:05.66 | /usr/bin/python3 /usr/bin/update-manager --no-update --no-focus-on-map
3450 root 20 0 548M 17832 6060 S 0.0 0.1 0:24.26 | /usr/lib/fwupd/fwupd
3259 dongzy 20 0 496M 2204 1680 S 0.0 0.0 0:00.34 | /usr/lib/gnome-settings-daemon/gsd-printer
3113 dongzy 20 0 335M 5032 3036 S 0.0 0.0 1:46.61 | /usr/lib/ibus/ibus-x11 --kill-daemon
3083 dongzy 9 -11 1309M 7304 5368 S 0.0 0.0 0:01.11 | /usr/bin/pulseaudio --start --log-target=syslog
2902 dongzy 20 0 425M 4936 3756 S 0.0 0.0 0:00.83 | /usr/bin/gnome-keyring-daemon --daemonize --login
14814 dongzy 20 0 11304 632 304 S 0.0 0.0 0:00.08 | | /usr/bin/ssh-agent -D -a /run/user/1000/keyring/.ssh
1580 root 20 0 438M 34448 8204 S 0.0 0.2 0:14.03 | /usr/lib/packagekit/packagekitd
1577 root 20 0 289M 2992 2620 S 0.0 0.0 0:00.06 | /usr/lib/x86_64-linux-gnu/boltd
```

任务（Task）：线程、单线程进程

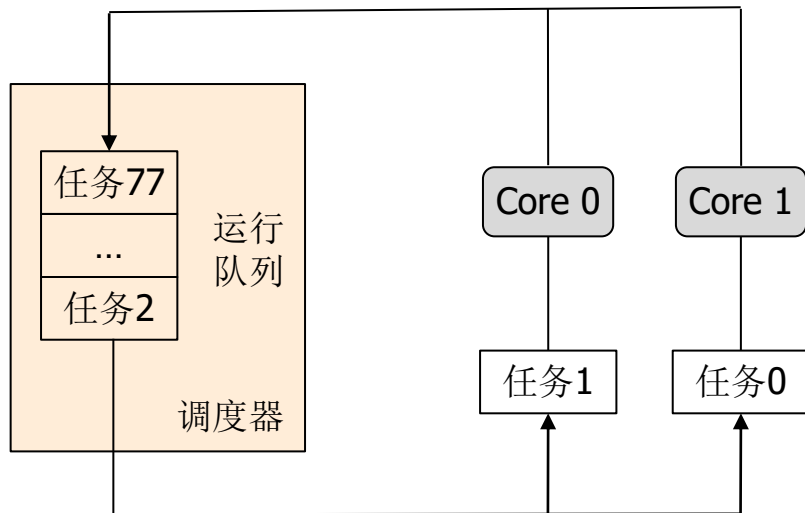
仅有8个处理器，如何运行169个任务？







# 如何进行CPU调度？



任务调度器示意图

- 任务数量远远超过CPU核心
- 调度器通过维护运行队列(run queue)的方式来管理任务
- 任务触发以下条件会停止执行
  - 执行了指定的时间片
  - 任务发起了I/O请求
  - 任务主动停止
  - 任务被系统中断打断
- 调度器的作用
  - 从队列中选择下一个执行的任务
  - 决定执行该任务的CPU核心
  - 决定该任务允许执行的时间片大小





# 进程的多级调度

## ■ 长程调度

- 当一个程序尝试运行时，OS是否立刻为其创建相应的进程？
- 长程调度决定当前真正可被调度的进程的数量

## ■ 短程调度

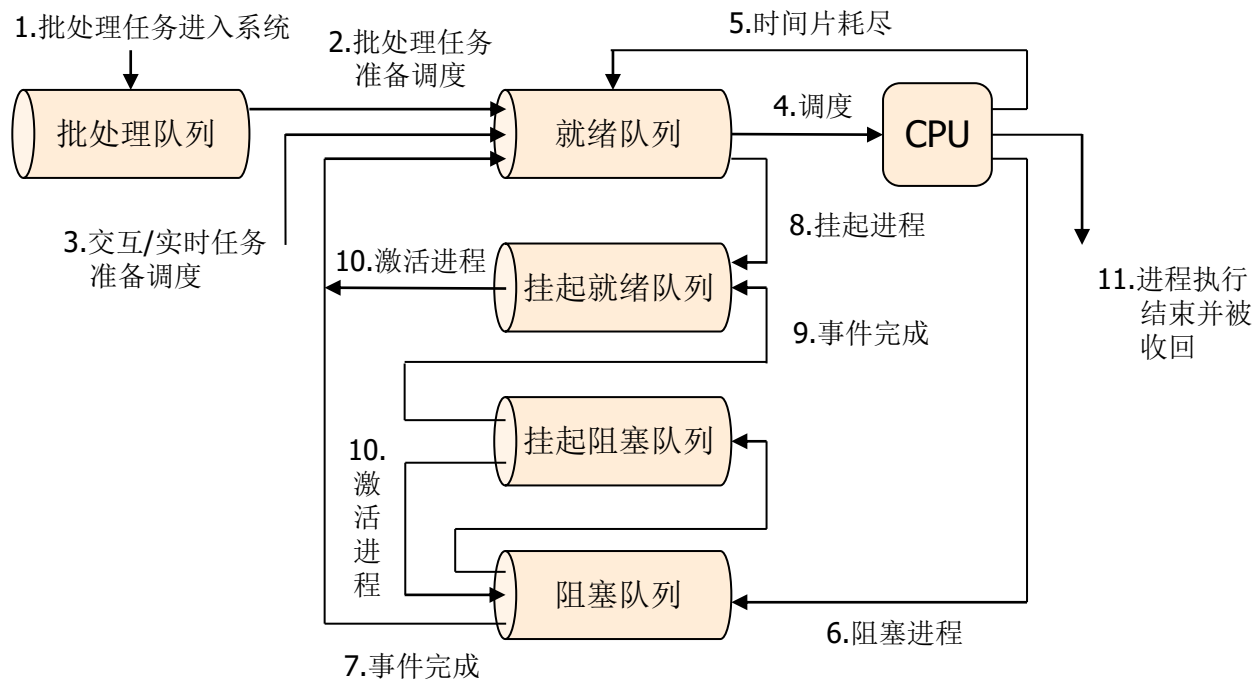
- 负责进程在就绪、运行和阻塞状态间的转换
- 触发短程调度的事件：进程创建、执行结束、硬件中断（特别是时钟中断）、系统调用等

## ■ 中程调度

- 负责避免内存使用过多
- 中程调度是换页机制的一部分



# 进程调度



进程调度示意图

- 长程调度：负责调控系统整体，为进程带来额外时延
- 中程调度：负责监控内存资源的使用
- 短程调度：决定进程是否可执行




# 调度方式

- 剥夺式/抢占式调度
  - 立即暂停当前进程
  - 分配处理机给另一个进程
  - 原则：优先权/短进程优先/时间片原则
- 非剥夺/非抢占式调度
  - 若有进程请求执行
  - 等待直到当前进程完成或阻塞
  - 缺点：适用于批处理系统，不适用分时/实时系统





# 调度时机

- 进程运行完毕
- 进程时间片用完
- 进程要求I/O操作
- 执行某种原语操作 
- 高优先级进程申请运行(剥夺式调度)






# 调度过程

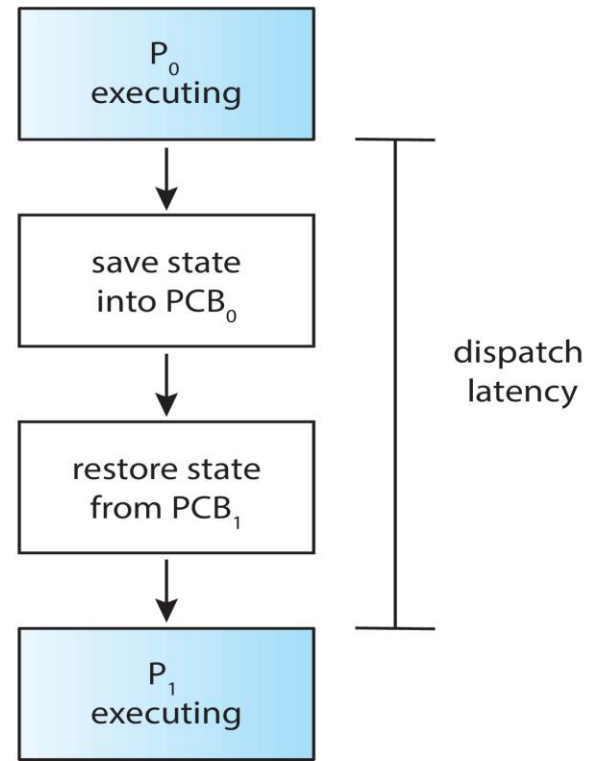
- 1.保存镜像：记录进程现场信息
- 2.调度算法：确定分配处理机的原则
- 3.进程切换：分配处理机给其它进程
- 4.处理机回收：从进程收回处理机





# 进程切换

- 调度程序将CPU的控制权交给CPU调度器选出来的进程
  - 切换上下文 
  - 切换用户模式
  - 跳转到用户程序的适当位置来启动程序
- 调度延迟：调度程序停止一个进程并启动另一个进程运行所需的时间





# 衡量调度算法的指标

- CPU利用率：衡量CPU被有效利用的程度的指标。理想情况下，希望CPU能够一直被占用
- 吞吐量：衡量单位时间内系统完成任务数量的指标
- 周转时间：从任务提交到任务完成所经过的时间，包括等待时间和执行时间
- 等待时间：衡量任务在就绪队列中等待的总时间
- 响应时间：从任务提交到首次开始执行所经过的时间







# 经典的单核进程调度算法

- 先来先服务
- 最短作业优先
- 最短剩余时间优先
- 最高相应比优先
- 轮转调度
- 优先级调度
- 多级队列
- 多级反馈队列



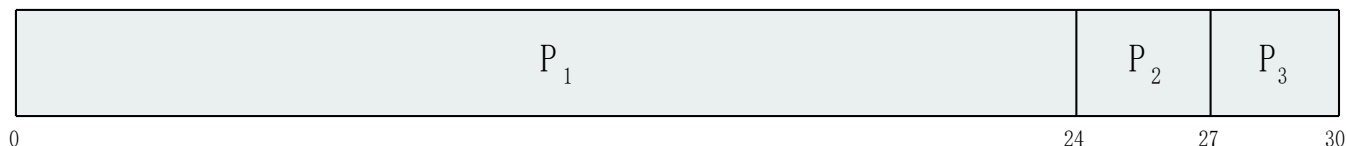


# 先来先服务FCFS调度

- 有3个进程

<u>Process</u>	<u>Burst Time</u>
P1	24
P2	3
P3	3

- 假设进程的到达顺序为: P1 , P2 , P3



进程调度甘特图

- 进程等待时间:  $P1 = 0; P2 = 24; P3 = 27$
- 平均等待时间:  $(0 + 24 + 27)/3 = 17$
- 周转时间  $(24+27+30) / 3 = 27$





# FCFS调度

- 假设进程到达的顺序为:

P2 , P3 , P1



进程调度甘特图

- 进程等待时间:  $P1 = 6$ ;  $P2 = 0$ ;  $P3 = 3$
- 平均等待时间:  $(6 + 0 + 3)/3 = 3$
- 周转时间  $(3 + 6 + 30) / 3 = 13$
- 性能优于前一种情况
- 车队效应- 短进程排在长进程后面
  - 考虑一个CPU繁忙型进程和多个I/O繁忙型进程





# FCFS的特征

- 优点:简单
- 缺点:
  - 平均等待时间波动较大
  - 短作业/任务/进程可能排在长进程后面
  - I/O资源和CPU资源的利用率较低

CPU密集型进程会导致I/O设备闲置时，I/O密集型进程也等待

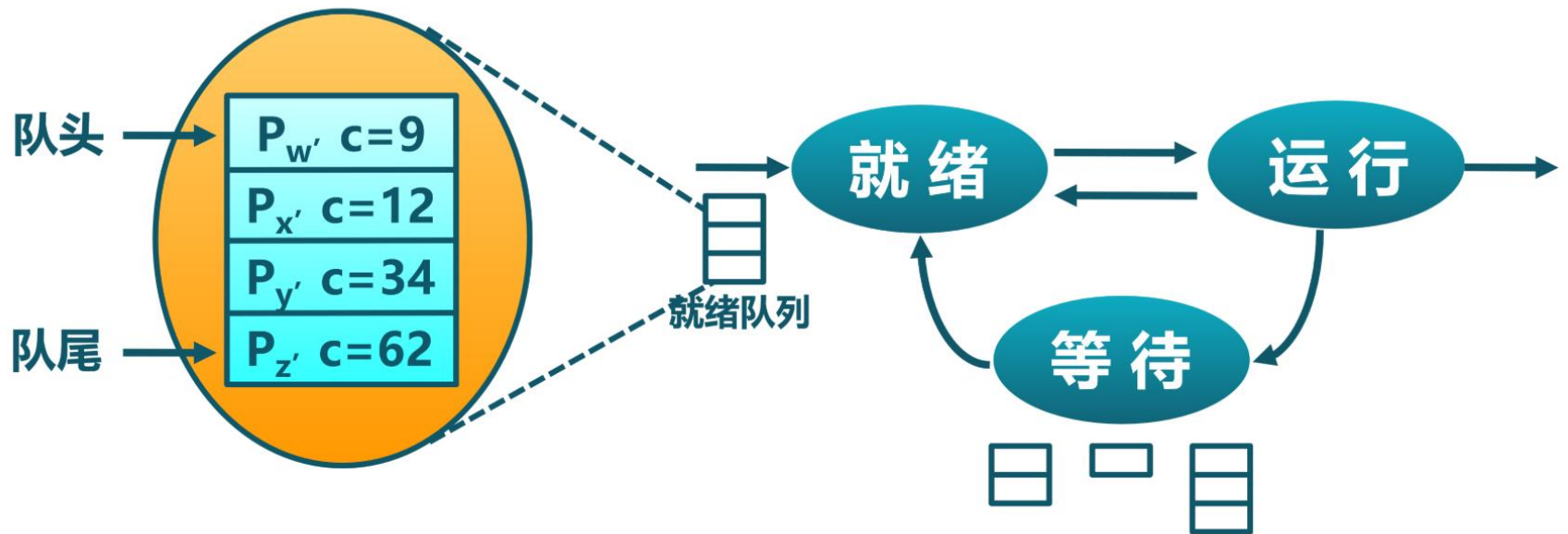




# 最短作业优先SJF调度

- 将进程和后续进程的CPU执行时间进行比较，选择调度CPU执行时间最短的进程
- 对于相同的进程集，SJF调度可以使得平均等待时间最短
- 如何确定后续进程的CPU执行时间呢？
  - 询问用户
  - 估算

# SJF





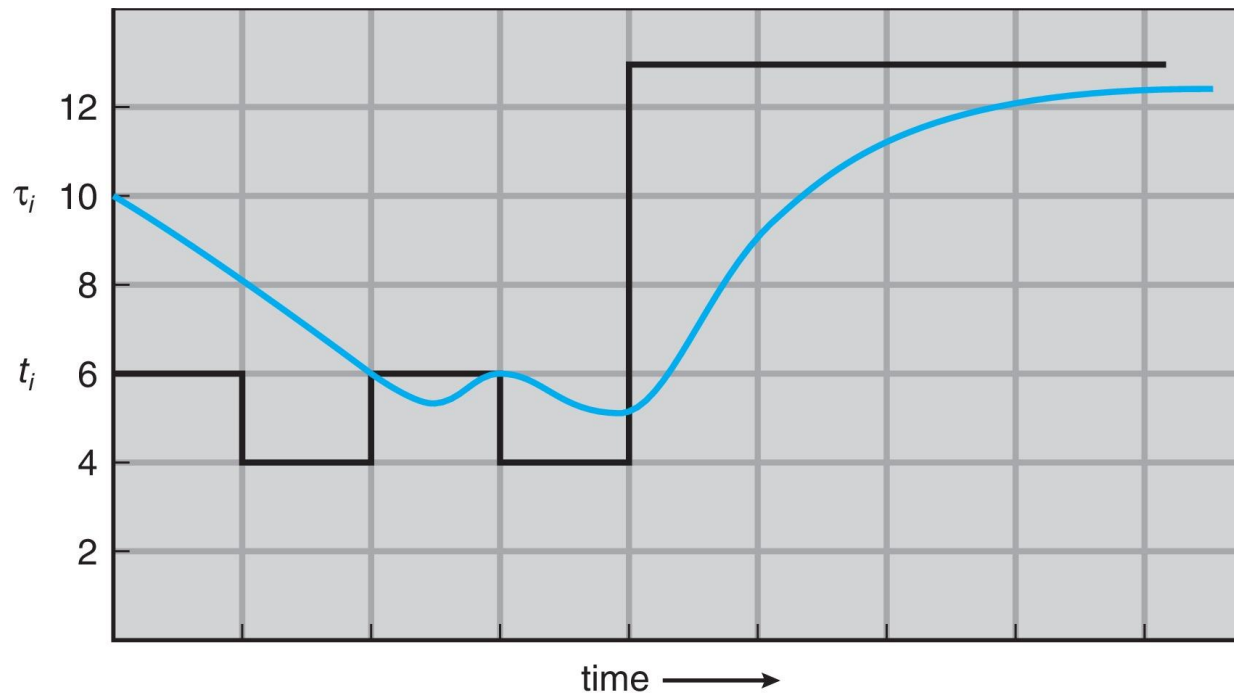
# 后续CPU执行时长

- CPU执行时长：估算为与先前的时长类似
  - 选择后续CPU执行时间最短的进程
- 利用前面的CPU执行时长，通过指数平均法来进行估算：
  - 1.  $t_n$  = 第 $n$ 个CPU执行时间的实际长度
  - 2.  $\tau_{n+1}$  = 下一个CPU执行时长的预测值
  - 3.  $\alpha$ ,  $0 \leq \alpha \leq 1$
  - 4.  $\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$   
通常， $\alpha$ 设置为 $\frac{1}{2}$

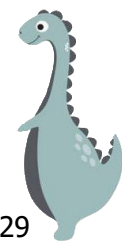




# 预测后续CPU执行时长



CPU burst ( $t_i$ )	6	4	6	4	13	13	13	...	
"guess" ( $\tau_i$ )	10	8	6	6	5	9	11	12	...







# 指数平均法示例

- $\alpha = 0$

- $\tau_{n+1} = \tau_n$
- 不考虑最近的历史情况

- $\alpha = 1$

- $\tau_{n+1} = \alpha t_n$
- 只考虑实际最近的CPU执行时长

- 将公式继续扩展:

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

- 由于 $\alpha$  和  $(1 - \alpha)$  都小于或等于 1, 因此每个后续项的权重都比其前任小

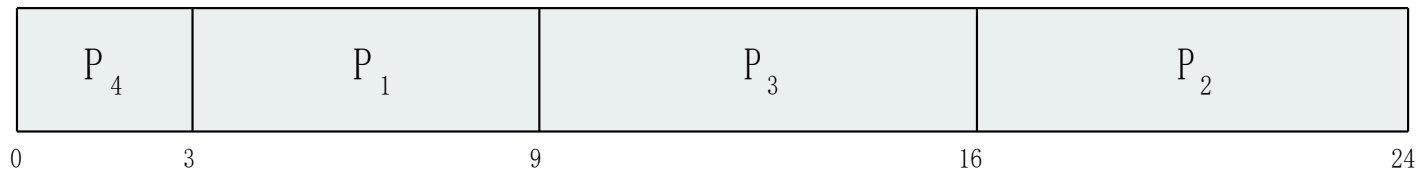




# SJF示例

<u>Process</u>	<u>Burst Time</u>
P <sub>1</sub>	6
P <sub>2</sub>	8
P <sub>3</sub>	7
P <sub>4</sub>	3

## ■ SJF 调度甘特图



周转时间 =  $(3 + 16 + 9 + 24) / 4 = 13$

■ 平均等待时间 =  $(3 + 16 + 9 + 0) / 4 = 7$





# 短作业优先调度算法的特征

- 优点：平均等待/周转时间最少
- 可能导致饥饿
  - 连续的短作业/进程流会使长作业/进程无法获得CPU资源
- 需要预知未来
  - 如何预估下一个CPU计算的持续时间？
  - 简单的解决办法：询问用户
    - 用户欺骗就杀死相应进程
    - 用户不知道怎么办？





# 最短剩余时间优先SRT调度

- 是SJF抢占式的版本
- 每当新进程到达就绪队列时，利用SJF算法重新决定下一个要调度的进程
- 对于给定的一组进程，SRT在平均最小等待时间方面是否比SJF更“优”？



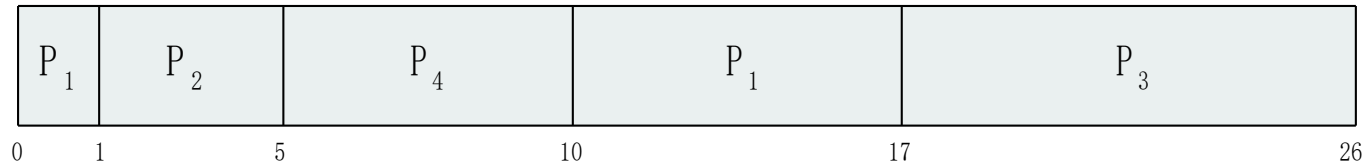


# 抢占式SJF示例

- 考虑进程的到达时间，且允许抢占

<u>Process</u>	<u>ArrivalTime</u>	<u>Burst Time</u>
P <sub>1</sub>	0	8
P <sub>2</sub>	1	4
P <sub>3</sub>	2	9
P <sub>4</sub>	3	5

- 抢占式 SJF 甘特图



- 平均等待时间=  $[(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5$

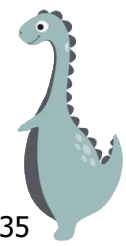




# 最高响应比优先HRRN调度



- 响应比 = (等待时间 + 服务时间) / 服务时间
- 选择响应比高的进程执行
- 非抢占式的调度算，旨在公平地处理各种长度的作业，同时尽量减少作业的平均等待时间和平均周转时间
  - 等待时间相同，短作业优先
  - 运行时间相同，等待时间长的作业优先

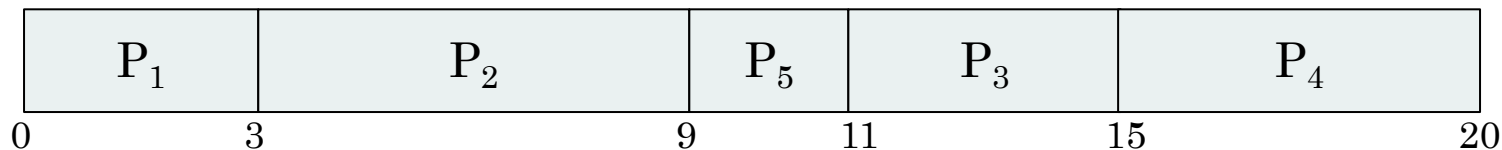




# HRRN示例

<u>Process</u>	<u>ArrivalTime</u>	<u>Burst Time</u>
$P_1$	0	3
$P_2$	1	6
$P_3$	2	4
$P_4$	3	5
$P_5$	4	2

## ■ 甘特图



- 0:  $P_1$  执行,  $P_2 P_3 P_4$  依次到达;
- 3:  $r_2=1+2/6$ ,  $r_3=1+1/4$ ,  $r_4=1$ ;  $P_2$  先运行;
- 9:  $r_3=1+7/4$ ,  $r_4=1+6/5$ ,  $r_5=1+5/2$ ;  $P_5$  先运行;
- 11:  $r_3=1+9/4$ ,  $r_4=1+8/5$ ;  $P_3$  先运行;





# HRRN的特征

- 在短作业优先算法的基础上改进
- 关注进程的等待时间
- 防止无限期推迟
- 长作业等待越久响应比越高，更容易获得处理机。







# 时间片轮转RR调度

- 每个进程获得一小段CPU时间（时间片 $q$ ），通常为10-100毫秒。此时间用完后，进程会被抢占并添加到就绪队列的末尾
- 若就绪队列中有 $n$ 个进程且时间片为 $q$ ，则每个进程一次最多获得CPU时间的 $1 / n$ ，进程等待不会超过 $(n-1)q$ 个时间片
- 时间片到，由计时器发出中断以调度下一个进程
- 性能
  - $q$ 设置过大，则退化为FIFO
  - $q$ 设置过小，则开销太高，因此 $q$ 必须大于上下文切换时间

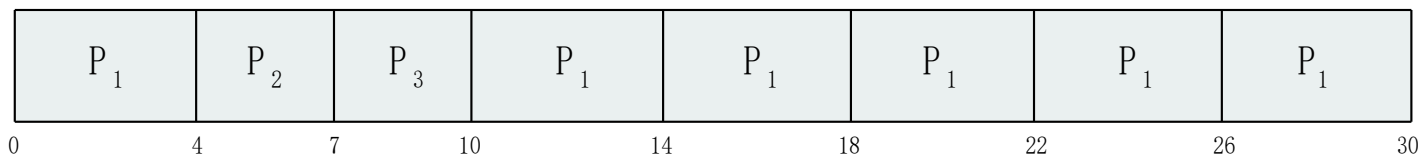




# RR调度示例（时间片=4）

<u>Process</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

## ■ 甘特图:

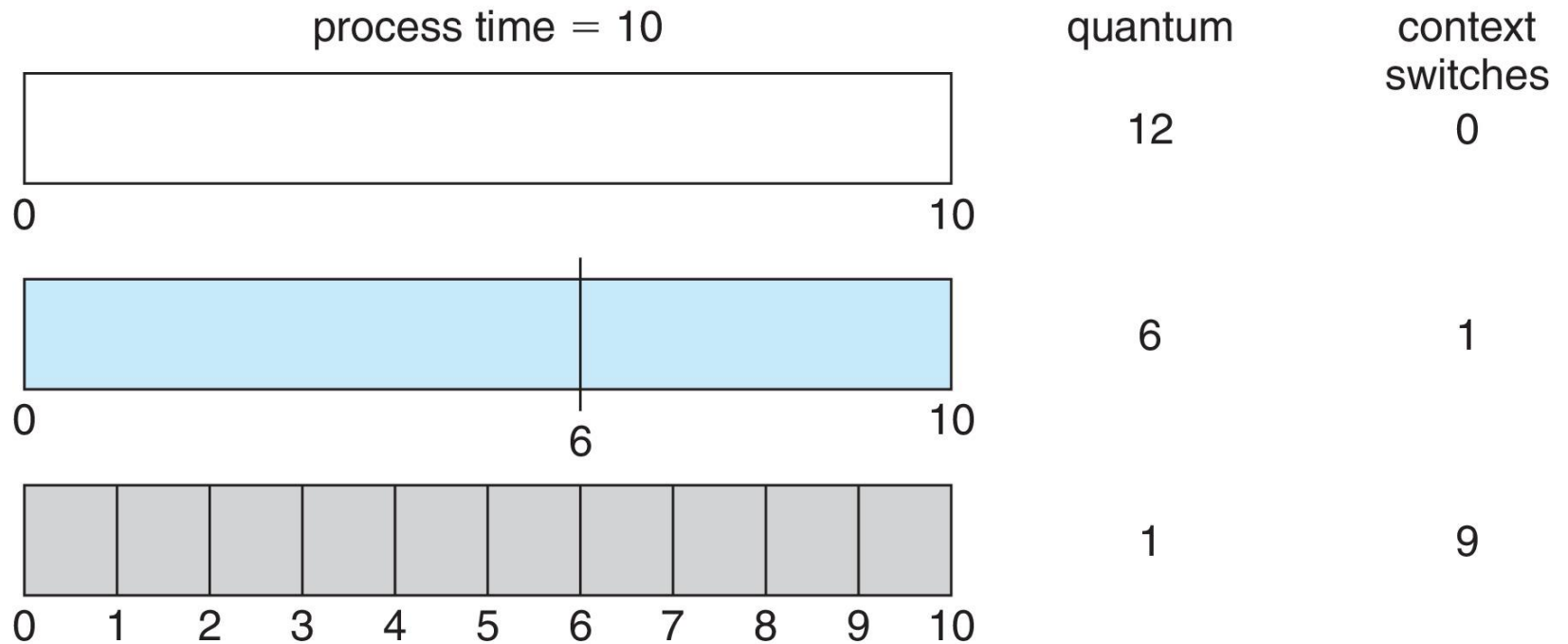


- RR调度通常比SJF的周转时间长，但响应快
- 时间片 $q$  应该长于上下文切换的时间
  - $q$  的值通常在 10~100 毫秒
  - 上下文切换时间 < 10 毫秒



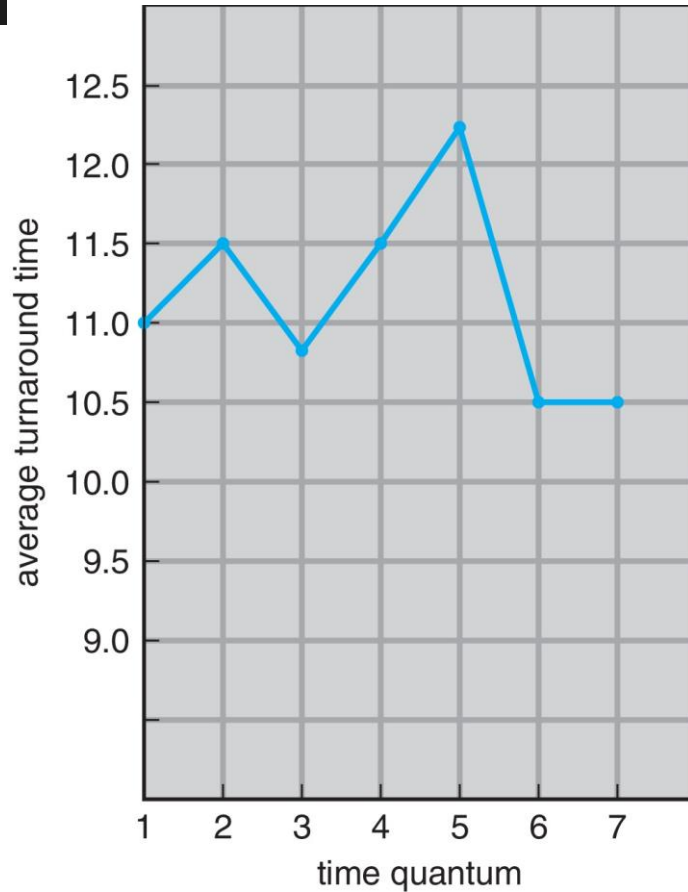


# 时间片和上下文切换时间





# 时间片影响周转时间




process	time
$P_1$	6
$P_2$	3
$P_3$	1
$P_4$	7

80% CPU执行时间应  
小于时间片长度






# RR的特征

- 公平， 响应快，适用于分时系统。
- 时间片决定因素：系统响应时间、就绪队列进程数量、系统处理能力。
- 时间片太大，相当于FCFS；太小，处理机切换频繁，开销增大。





# 优先级调度

- 每个进程都与一个优先级数字（整数）相关联
- CPU分配给具有最高优先级的进程（通常，最小整数 = 最高优先级）
- 两种方案：
  - 可抢占式 
  - 不可抢占式
- 问题：饥饿—低优先级进程可能永远不会执行
- 解决方案：老化—随着时间的推移，增加进程的优先级
- 注意：SJF可以看做是优先级调度，其中优先级是预测的下一个CPU执行时间的倒数

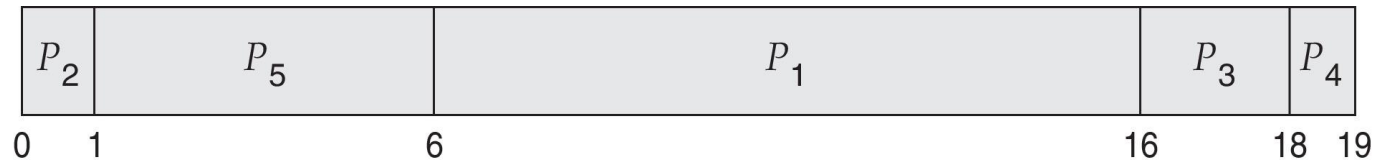




# 优先级调度示例

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	10	3
$P_2$	1	1
$P_3$	2	4
$P_4$	1	5
$P_5$	5	2

## ■ 优先级调度甘特图



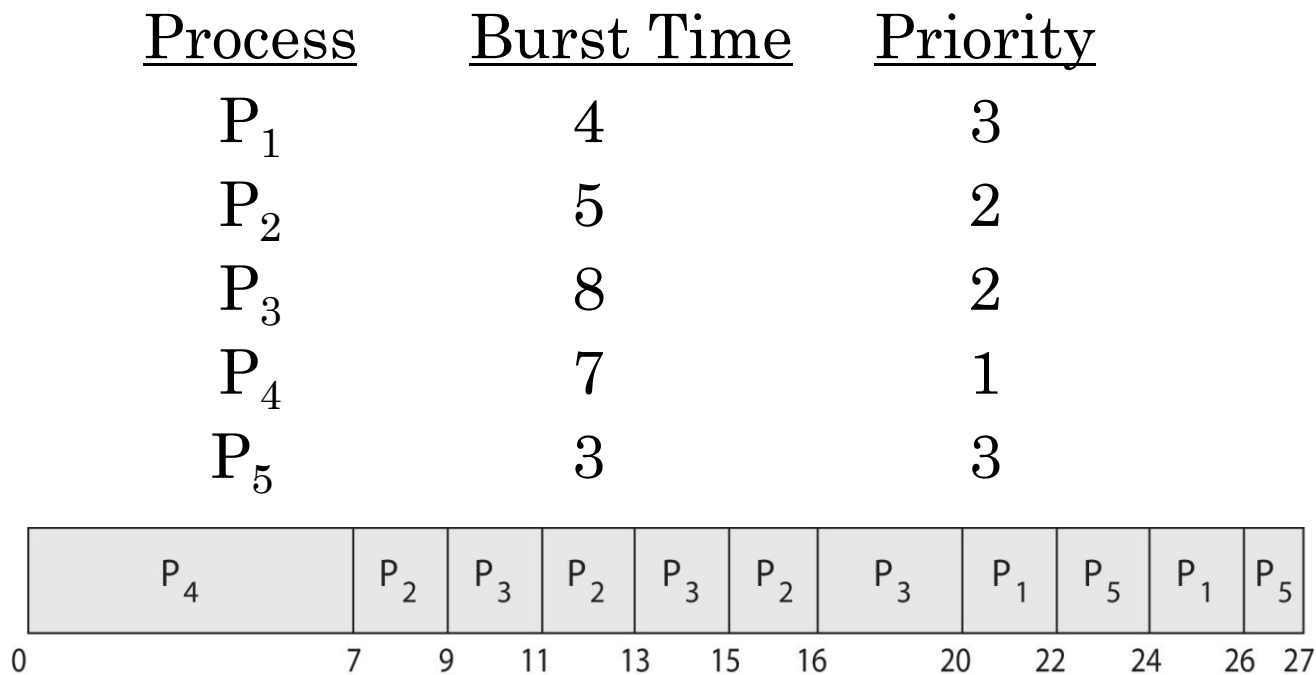
## ■ 平均等待时间 = 8.2





# 结合RR的优先级调度

- 调度最高优先级的进程，若进程优先级相同，则按RR方式进行调度。
- 示例：



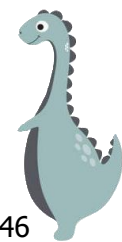
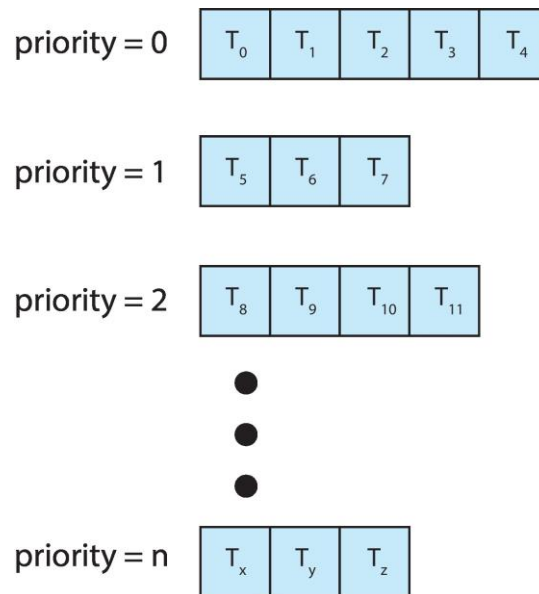
时间片为2的甘特图





# 多级队列

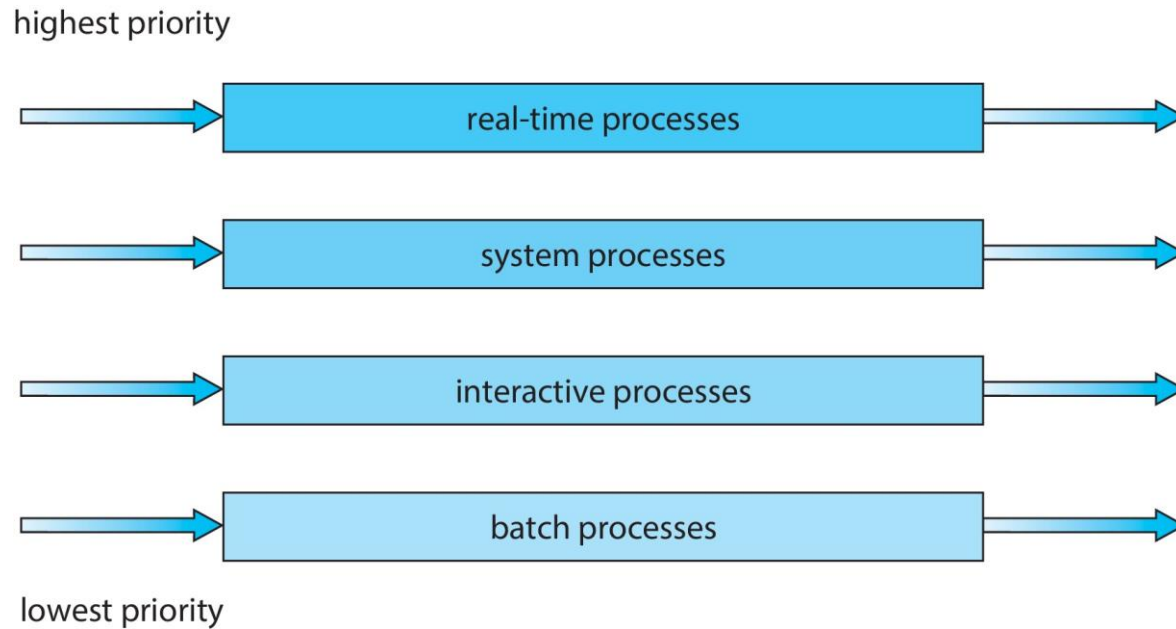
- 就绪队列由多个队列构成
  - 不同的优先级都有各自独立的队列
  - 调度从最高优先级的队列开始





# 多级队列

- 根据进程的类别来确定优先级





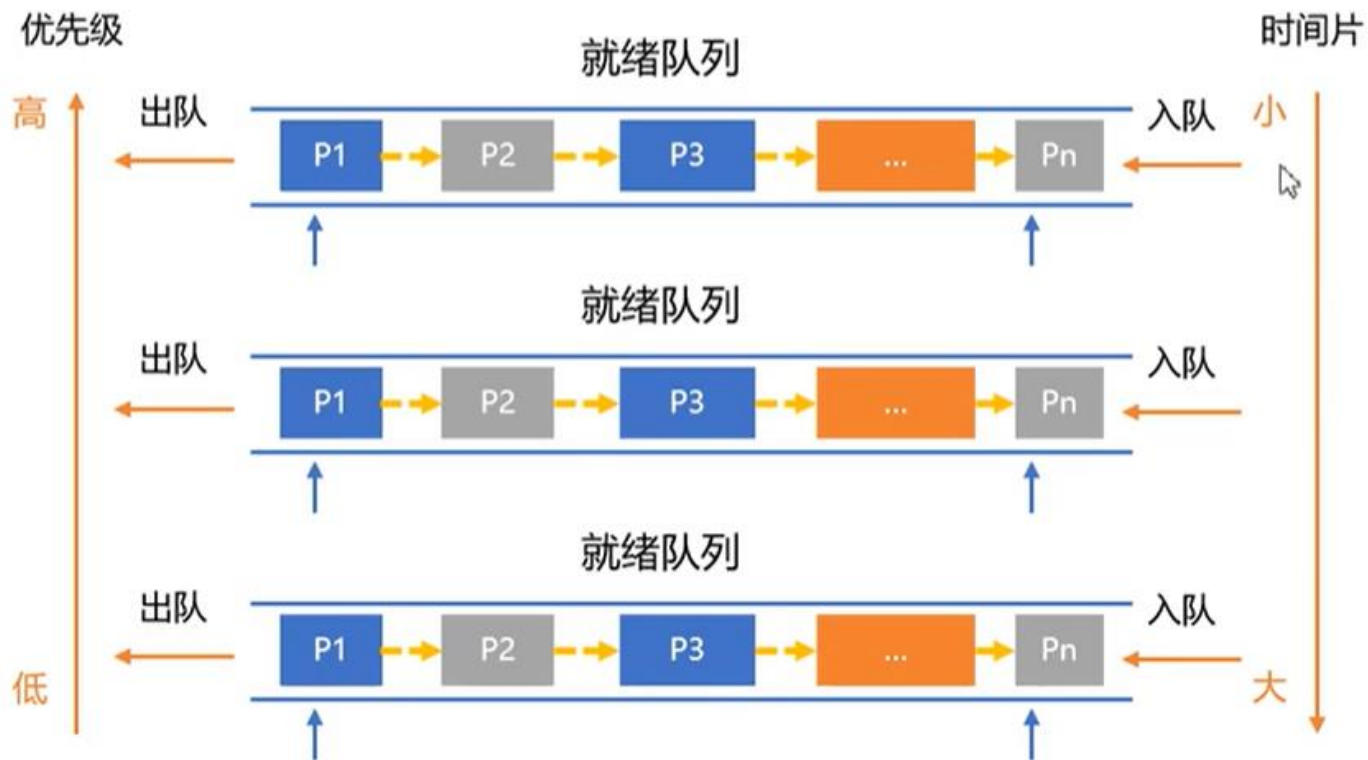
# 多级反馈队列MFQ

- 一个进程可以在各个队列之间移动
- 多级反馈队列的调度程序由以下参数定义：
  - 队列数量
  - 每个队列的调度算法
  - 何时升级进程的方法
  - 何时降级进程的方法
  - 进程将进入哪个队列的方法
- 可以通过多级反馈队列来实现老化机制(Aging), 防止低优先级进程长时间得不到调度





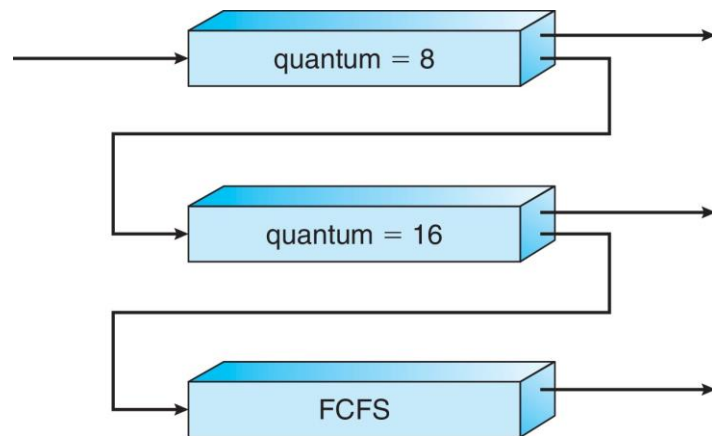
# 多级反馈队列





# 多级反馈队列示例

- 三个队列：
  - $Q_0$  – 时间片8 毫秒
  - $Q_1$  – 时间片16 毫秒
  - $Q_2$  – FCFS
- 调度策略
  - 新进程进入队列 $Q_0$ 等待RR调度
    - 若进程获得CPU，则可执行8毫秒
    - 8毫秒内未完成，则进入队列 $Q_1$
  - 队列 $Q_1$ 中的进程以16毫秒的时间片进行RR调度
    - 若进程依旧未完成，则进入队列 $Q_2$





# MFQ的特征

- 优缺点:
- 对各类型相对公平； 快速响应；
- 终端型作业用户： 短作业优先
- 批处理作业用户： 周转时间短
- 长批处理作业用户： 在前几个队列部分执行





# 操作系统调度算法实例

- 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





# BSD多级反馈队列调度

- 根据进程的行为和特征动态地调整进程的优先级和时间片
  - 系统维护多个就绪队列，每个队列有不同的优先级和时间片长度，一般来说，优先级越高的队列，时间片越短。
  - 新创建的进程被放入最高优先级的队列中，如果该队列为空，则调度该进程运行。
  - 如果一个进程在分配给它的时间片内完成或主动放弃CPU，则保持其优先级不变，重新放入原来的队列中等待下一次调度。
  - 如果一个进程在分配给它的时间片内没有完成或被抢占，则降低其优先级，放入下一级队列中等待下一次调度。
  - 如果一个进程在最低优先级的队列中运行，且该队列中有其他进程，则采用轮转法进行调度。







# BSD多级反馈队列调度

## ■ 调度效果

- 对于短作业或交互式作业，给予较高的优先级和较快的响应时间，从而提高用户体验和系统吞吐量。
- 对于长作业或计算密集型作业，给予较低的优先级和较长的时间片，从而避免过多的上下文切换开销和饥饿现象。
- 对于不同类型或特征的作业，根据其历史行为进行动态调整，从而实现自适应和公平的调度。





# Windows 基于优先级的抢占式多任务调度

- Windows 为每个线程分配一个初始优先级和一个当前优先级，用 0 到 31 的数字表示，数字越大，优先级越高。
- Windows 维护一个线程就绪表，用来记录哪些优先级下有就绪的线程。如果优先级数量小于等于 32，就绪表可以用一个 32 位的变量表示，每一位对应一个优先级；如果优先级数量大于 32，就绪表可以用一个 8 位数组表示，每个元素对应 8 个优先级。
- Windows 在每次调度时，首先查找就绪表中最高的非零位，得到最高优先级的编号。然后在该优先级下的线程链表中选择一个线程执行。如果该链表中有多个线程，可以采用轮转法或者其他方法选择。
- Windows 在执行一个线程时，会给它分配一个时间片，当时间片用完或者该线程主动放弃 CPU 时，Windows 会重新调度下一个线程。
- Windows 在切换线程时，会保存当前线程的上下文，并恢复下一个线程的上下文。
- Windows 在创建、删除、阻塞、唤醒或者改变线程的优先级时，会更新就绪表和线程链表，保证它们反映了当前的线程状态。





# Linux抢占式调度算法

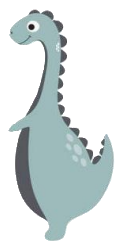
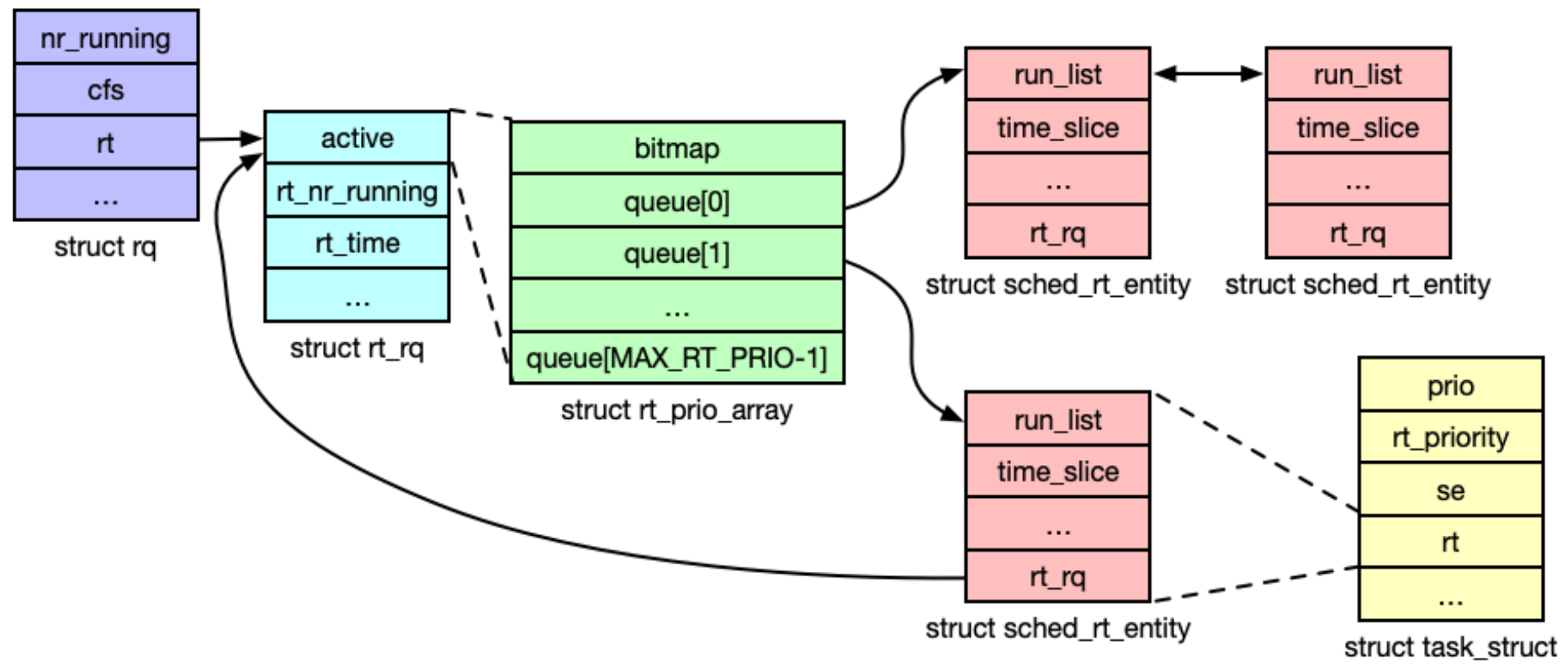
- Linux的调度器维护一个就绪队列(runqueue)，按照完全公平调度算法CFS从就绪队列中选出一个进程，将其分配到CPU上执行一个时间片。
- 当时间片用完时,则触发一个时钟中断,调度器会强制暂停当前进程的执行,将其重新放回就绪队列,然后选取另一个进程执行。
- CFS算法根据进程的优先级和等待时间来计算每个进程的vruntime值,vruntime越小,进程获得CPU时间片的机会就越大。
- CFS维护一个红黑树,根据vruntime值将就绪进程排序,树中序遍历最小的那个进程将被先调度。
- 当一个进程用完时间片被抢占时,调度器会重新计算它的vruntime值,并在红黑树中调整它的位置。
- 为避免线程长期占用CPU,CFS设置了抢占时间片的上限,强制线程放弃CPU执行权。
- Linux也支持优先级抢占,高优先级的进程可以抢占低优先级进程的执行。







# Linux调度机制：RT Run Queue





# Solaris综合调度算法

- Solaris采用多种调度算法相结合的策略
  - 优先权调度
  - 时间片调度
  - 交互式进程优先
  - 多级反馈队列
  - 抢先式调度
- 调度效果
  - 有效地平衡了实时性、交互性和吞吐量等不同的需求，使得系统能够在不同的工作负载下都能保持良好的性能





# Solaris综合调度算法

- 所有的进程都被分配到不同的调度级别。每个调度级别都有自己的调度策略和优先级范围。
  - 实时进程：拥有最高的调度级别，拥有严格时间限制的任务，如科学计算和音频视频处理
  - 系统进程：如内核线程和中断处理程序
  - 交互式用户进程：如文本编辑器和浏览器
  - 批处理进程：拥有最低的调度级别，通常在系统空闲时运行
- 优先级根据多种因素动态计算
  - 考虑进程的类型、进程的老化情况（即等待时间）以及进程最后一次执行后的CPU使用情况等
- 任何时候只要有一个更高优先级的进程变为就绪状态，都会立即抢占当前进程





# 多处理器调度

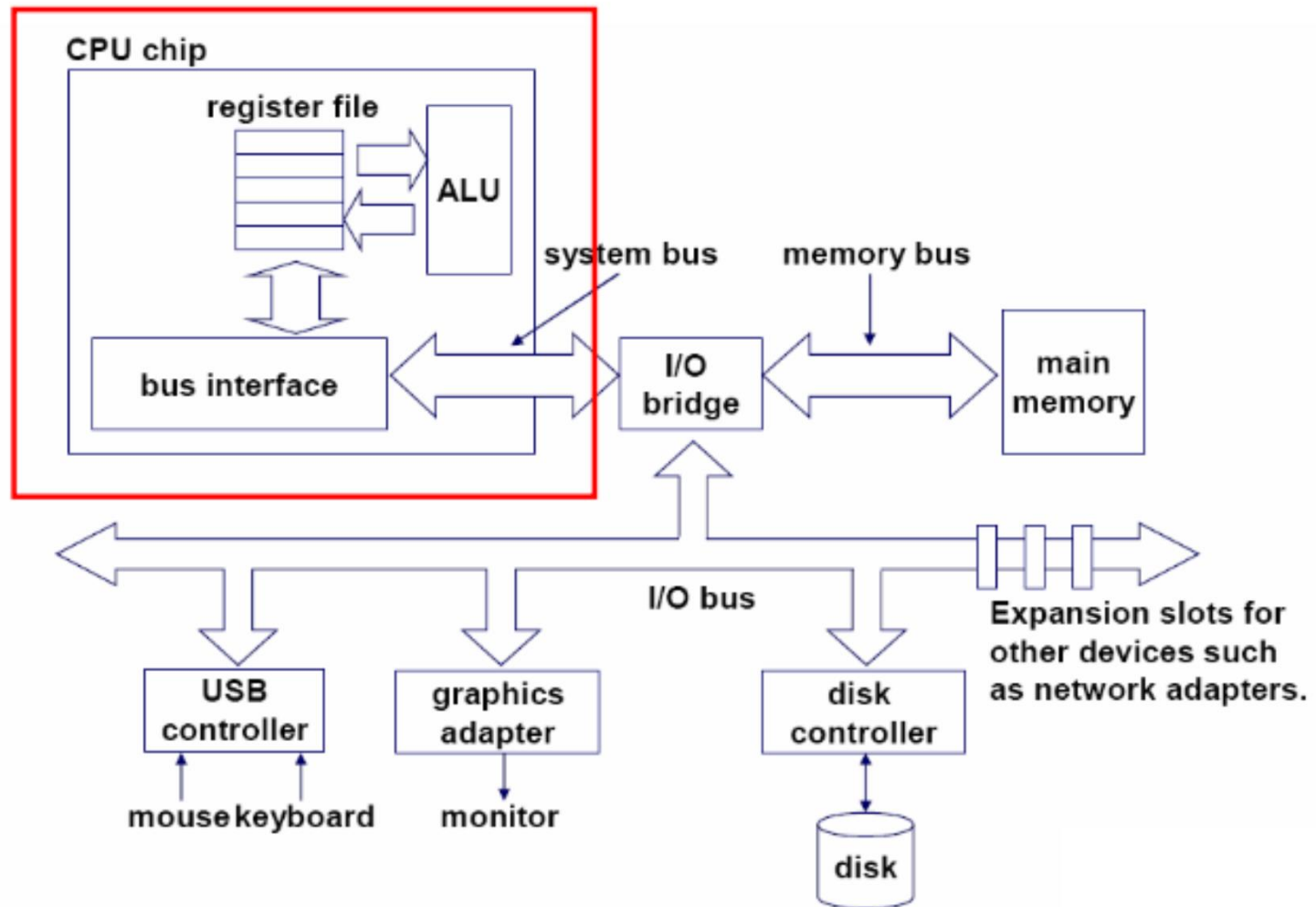
- 当前需要调度哪个/哪些任务？
- 每个调度的任务应该在哪个CPU核心上执行？
- 每个调度任务应该执行多久？
- 如何在多个处理器（或者多核）之间分配和调度进程，做到负载均衡？
- 在多核场景下有哪些调度策略？





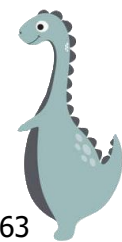
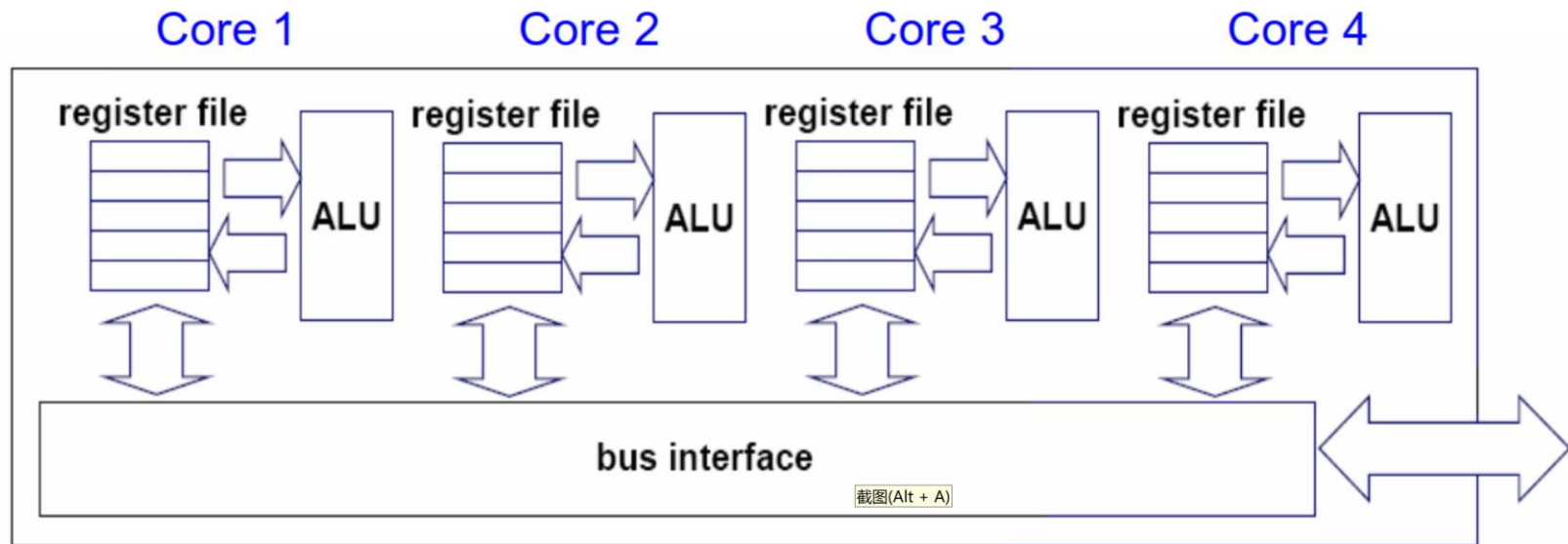


# 单核处理器





# 多核处理器





# 多处理器架构

## ■ 定义：

- 多处理器架构：指一个计算系统中包含了两个或更多的处理器，这些处理器可以并行地执行多个任务，以提高系统的性能
- 主要优点：可以实现任务的并行处理，从而提高系统的吞吐量和性能





# 多处理器架构的分类

- 按连接方式和资源共享方式分类
  - **紧耦合多处理器**：在一个系统中有多处理器**共享同一存储器和I/O设备**，所有处理器可以访问所有的存储和设备。优点是资源共享，缺点是需要复杂的硬件和软件支持。
  - **松耦合多处理器**：在一个系统中有多处理器，每个处理器都**有自己的存储器和I/O设备**，处理器之间通过通信网络连接。优点是简单，易于扩展，缺点是资源利用率低。





# 多处理器架构的分类

- 根据多处理器的工作方式分类
  - 对称多处理器（SMP）：在SMP系统中，所有的处理器都是对等的，任何一个处理器都可以执行任何一个任务，所有的处理器共享同一存储器和I/O设备。
  - 非对称多处理器：在非对称多处理器系统中，不同的处理器有不同的任务，例如，一些处理器专门用于I/O操作，一些处理器专门用于计算。





# 调度对多处理器架构的影响

## ■ 调度目标

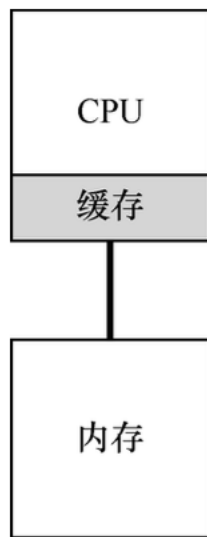
- 调度算法如何将任务分配给不同的处理器，以达到性能最优
- 调度算法如何满足任务之间的依赖关系，确保任务按正确的顺序执行



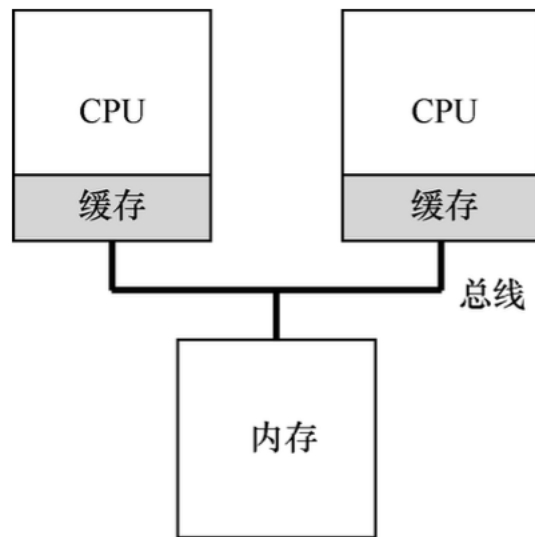


# 多处理器架构带来的问题

- 多CPU的情况下的缓存一致性问题
- 缓存亲和度问题
  - 指进程或线程**倾向于在特定的**处理器或处理器集合上运行的特性



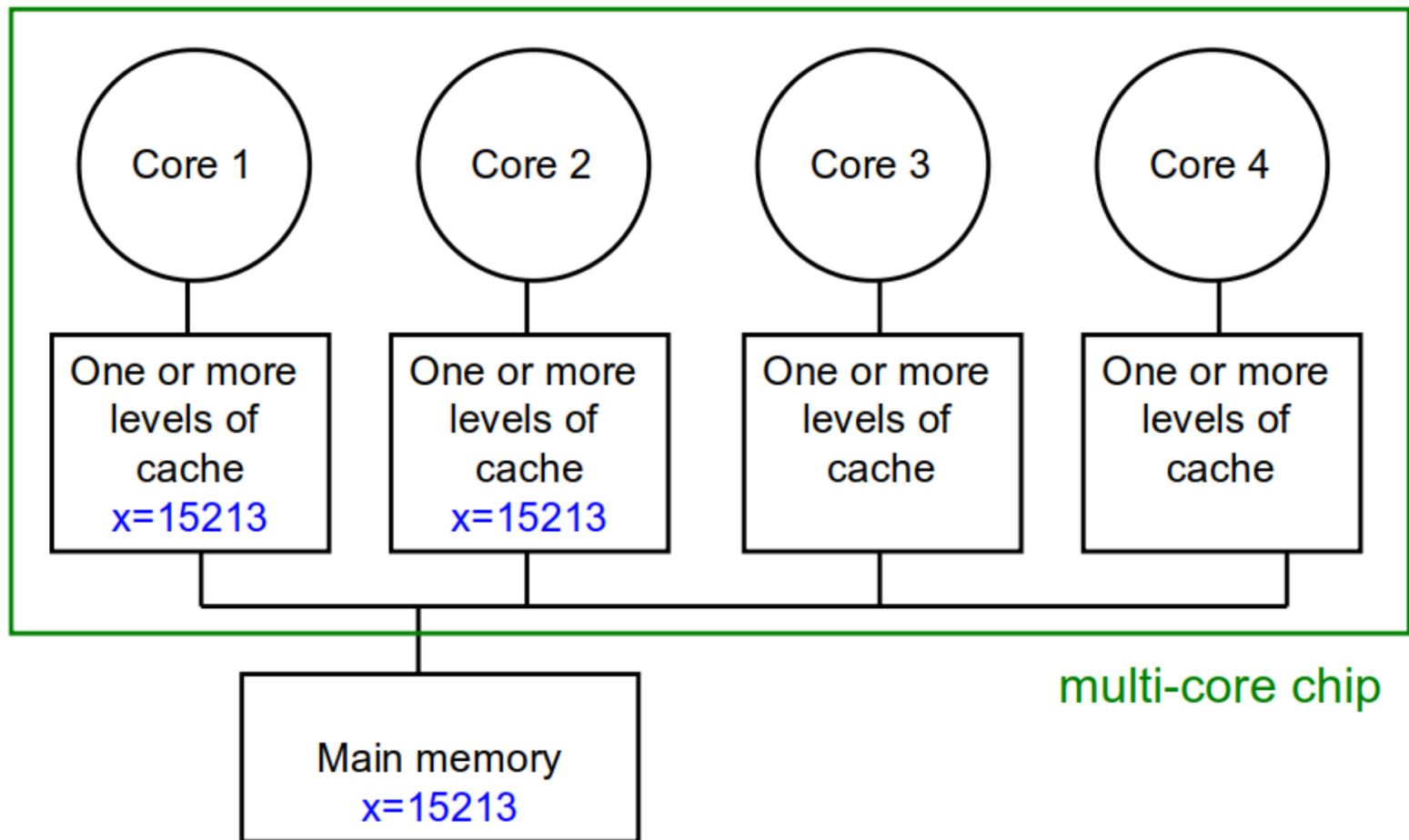
带缓存的单CPU



两个有缓存的CPU共享内存



# Cache一致性问题







# 处理器的亲和性(Processor Affinity)

- 定义:

- **处理器亲和性**是指进程或线程**倾向于在特定的**处理器或处理器集合上运行的特性。

- 原因:

- 由于该进程或线程之前在这些处理器上运行过，因此可能有一些数据或状态仍然缓存在那里，如果再次在这些处理器上运行，则可以利用这些缓存数据，从而提高性能。





# 静态亲和性 (Static Affinity)

- 定义：静态亲和性是在任务分配时事先确定哪些任务应该在特定的处理器上运行。这种方法通常在系统初始化时就设置好，不会在运行时改变。
- 优点：
  - 减少了任务迁移的开销。
  - 更易于预测系统性能。
- 缺点：
  - 可能出现负载不均衡的情况。
  - 不能适应负载变化的情况。





# 动态亲和性(Dynamic Affinity)

- 定义：动态亲和性是在运行时根据当前负载情况调整任务和处理器之间的分配。这种方法可以灵活应对系统的变化和负载波动。
- 优点：
  - 能有效地平衡负载，提高系统资源的利用率。
  - 更能适应动态变化的工作负载。
- 缺点：
  - 可能导致更高的任务迁移开销。
  - 增加系统的复杂性。





# 单队列多处理器调度 SQMS

## ■ 负载分担(load sharing)策略

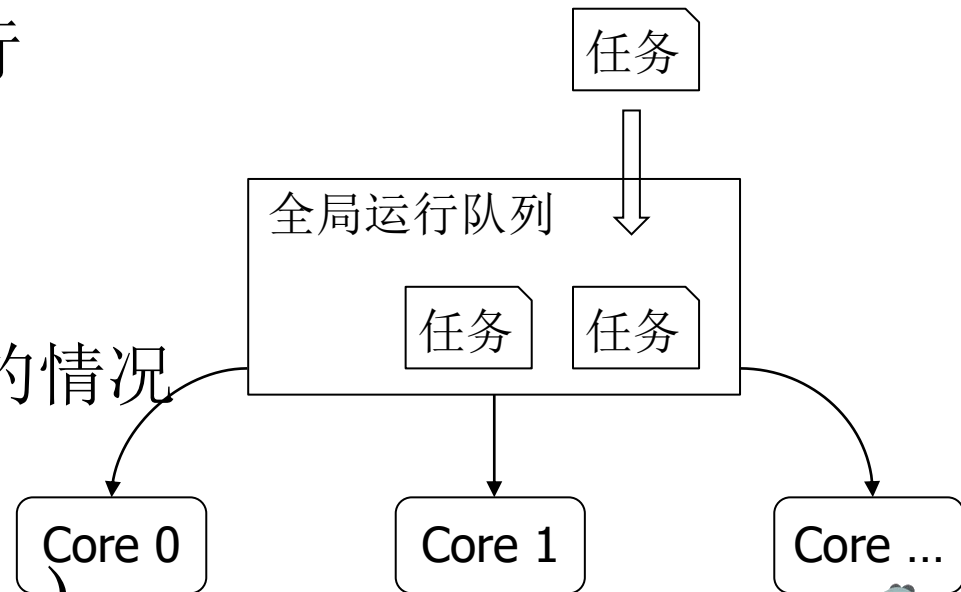
- 多处理器（核）共享一个全局运行队列
- 当某个CPU核心空闲时，根据调度策略从全局运行队列中选择一个任务运行

## ■ 优点：

- 设计简单
- 不会出现CPU资源浪费的情况

## ■ 缺点：

- 缺乏可扩展性 (scalability)
- 缓存亲和性 (cache affinity) 弱

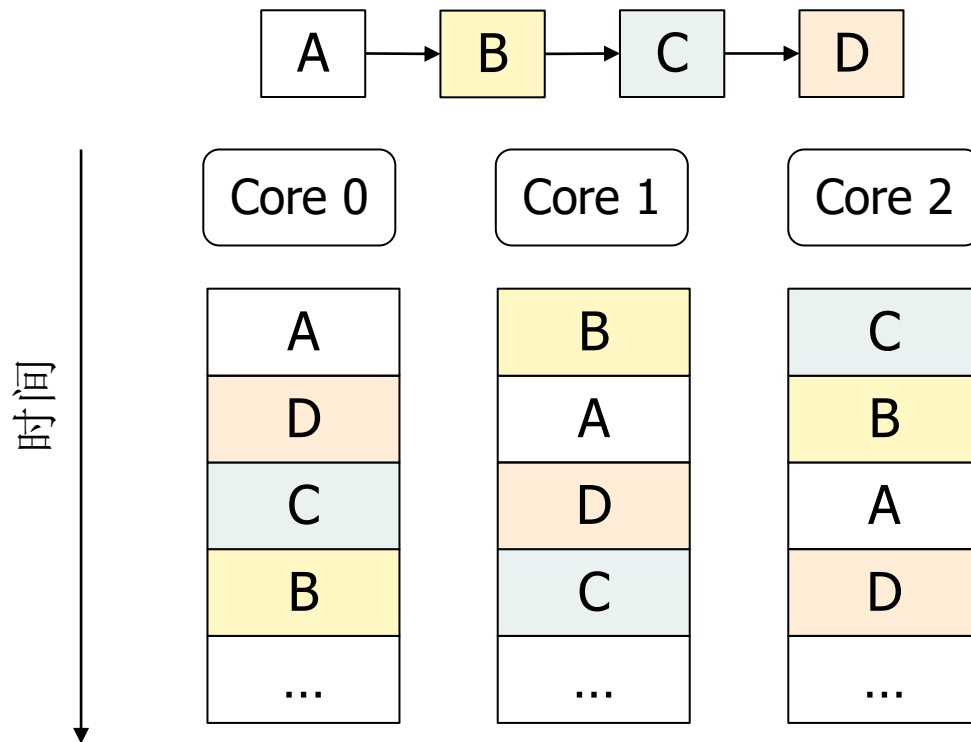




# 负载分担(load sharing)策略

## ■ 存在的问题

- 多处理器（核）共享一个全局运行队列带来的同步开销
- 任务在多个处理器（核）间来回切换的开销



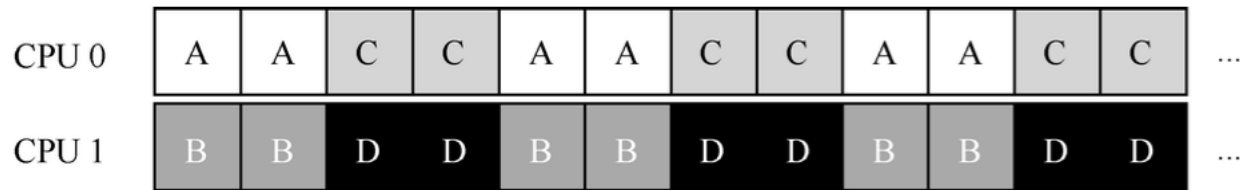
负载分担带来的问题





# 多队列处理器调度MQMS

- 假设系统中有两个CPU 0, CPU 1, 有四个进程A, B, C, D



- 可能存在负载不均的问题

迁移





# 同构/异构多处理器任务调度

- 调度算法的目标：是将任务有效地分配给各个处理器，以达到负载均衡，提高系统的性能。
- 常见的同构多处理器调度算法包括：
  - 轮转（Round-Robin）
  - 最短作业优先（Shortest Job First）
  - 优先级调度（Priority Scheduling）
- 常见的异构多处理器调度算法包括：
  - 最小化完成时间（Min-Min）
  - 最大化完成时间（Max-Min）
  - 负载均衡（Load Balancing）
  - 动态调度算法





# 练习题

---

- 5.2 Explain the difference between preemptive and nonpreemptive scheduling.







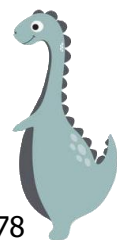
# 练习题

- 5.4 Consider the following set of processes, with the length of the CPU burst time given in milliseconds:

Process	Burst Time	Priority
P <sub>1</sub>	2	2
P <sub>2</sub>	1	1
P <sub>3</sub>	8	4
P <sub>4</sub>	4	2
P <sub>5</sub>	5	3

The processes are assumed to have arrived in the order P<sub>1</sub>, P<sub>2</sub>, P<sub>3</sub>, P<sub>4</sub>, P<sub>5</sub>, all at time 0

- Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, nonpreemptive priority (a larger priority number implies a higher priority), and RR (quantum = 2).
- What is the turnaround time of each process for each of the scheduling algorithms in part a?
- What is the waiting time of each process for each of these scheduling algorithms?
- Which of the algorithms results in the minimum average waiting time (over all processes)?





# 练习题

- 在RR策略的设计实现中，通常会维护一个运行队列，队列中的元素是对任务的引用（指针），每次被调度的任务会运行固定长度的时间片。如果在实施RR策略的队列中加入一个功能，可以在该任务的后面插入多个对同一任务的引用，那么这样的设计会带来什么样的影响？这样的设计使得RR策略与什么类型的调度相似？
- 什么是处理器亲和性（Processor Affinity）？它在多处理器调度中起什么作用？

