第5章 CPU调度

孙承杰

E-mail： sunchengjie@hit.edu.cn
哈工大计算学部人工智能教研室
2023年秋季学期

主要内容

- 调度基本概念
- 调度准则
- 调度算法
- 案例分析

5.1 基本概念

（1）什么是CPU调度？

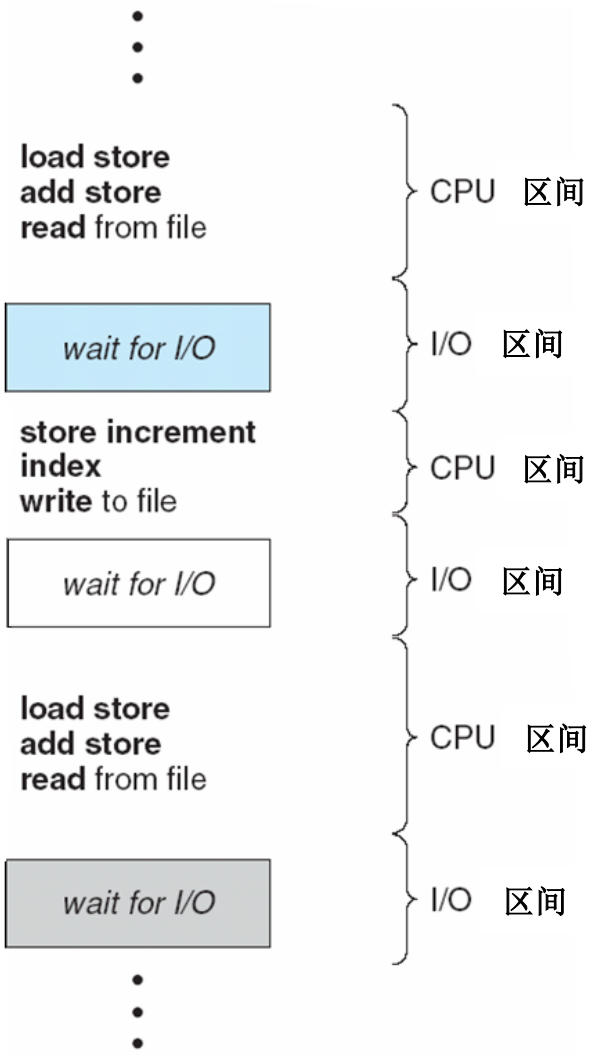
- 每当CPU空闲时，操作系统必须按照一定的策略从就绪队列当中选择一个进程来执行。
- 调度的对象：进程或线程（任务），其方式与原则是一样的。故经常以进程来说明：进程调度 <==> CPU调度

（2）CPU调度遵循什么原则？

- 总原则：资源高效利用、保证公平合理、满足应用时间需求
- 一般包括如下度量指标：
  - 提高CPU利用率
  - 提高系统运算的吞吐量
  - 缩短进程的周转时间
  - 缩短进程的等待时间
  - 提高用户的响应满意度
  - ……

5.1 基本概念

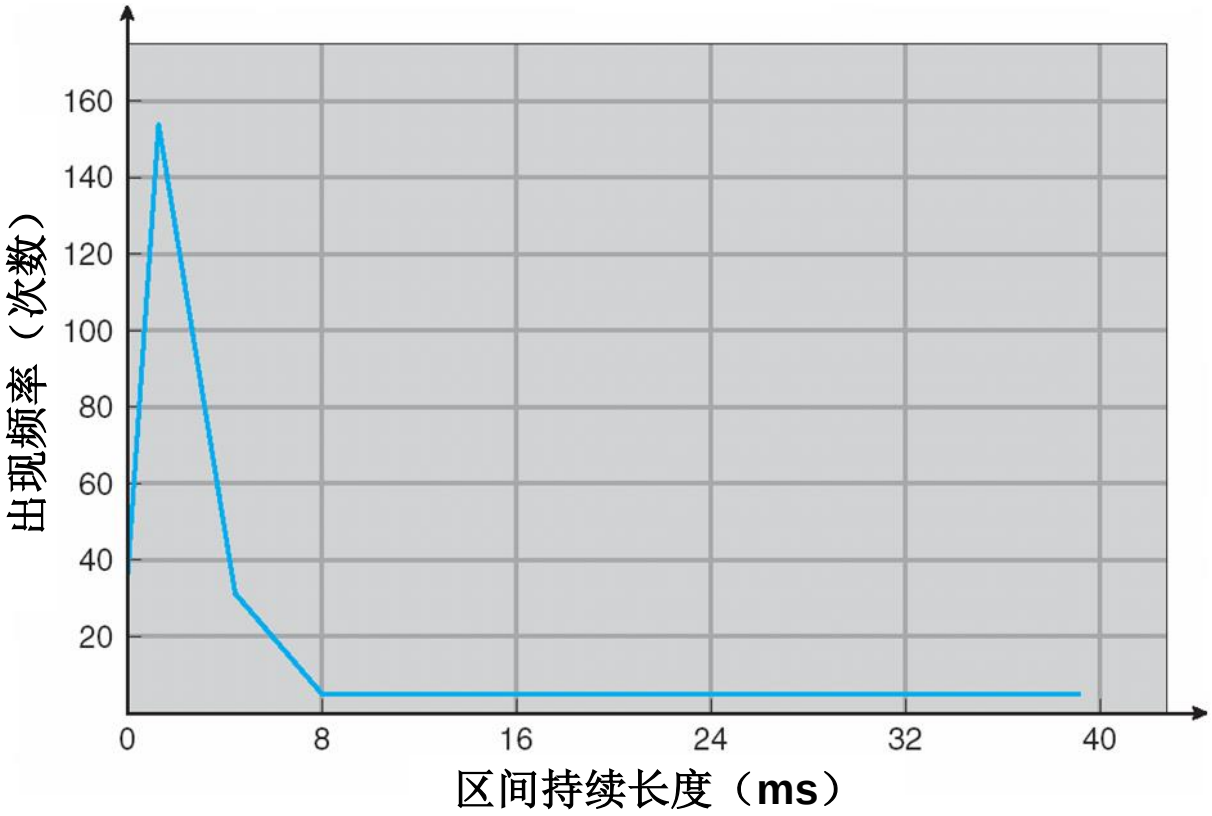
（3）CPU-I/O执行周期



- 程序代码可以分为计算类代码和I/O类代码
- 进程执行过程由CPU执行和I/O等待周期组成CPU区间和I/O区间
- CPU约束型程序以计算为主，CPU区间会较多，还会有少量长的CPU区间
- I/O约束型程序以I/O为主，但配合I/O处理会有大量短的CPU区间

5.1 基本概念

（3）CPU-I/O执行周期（续）

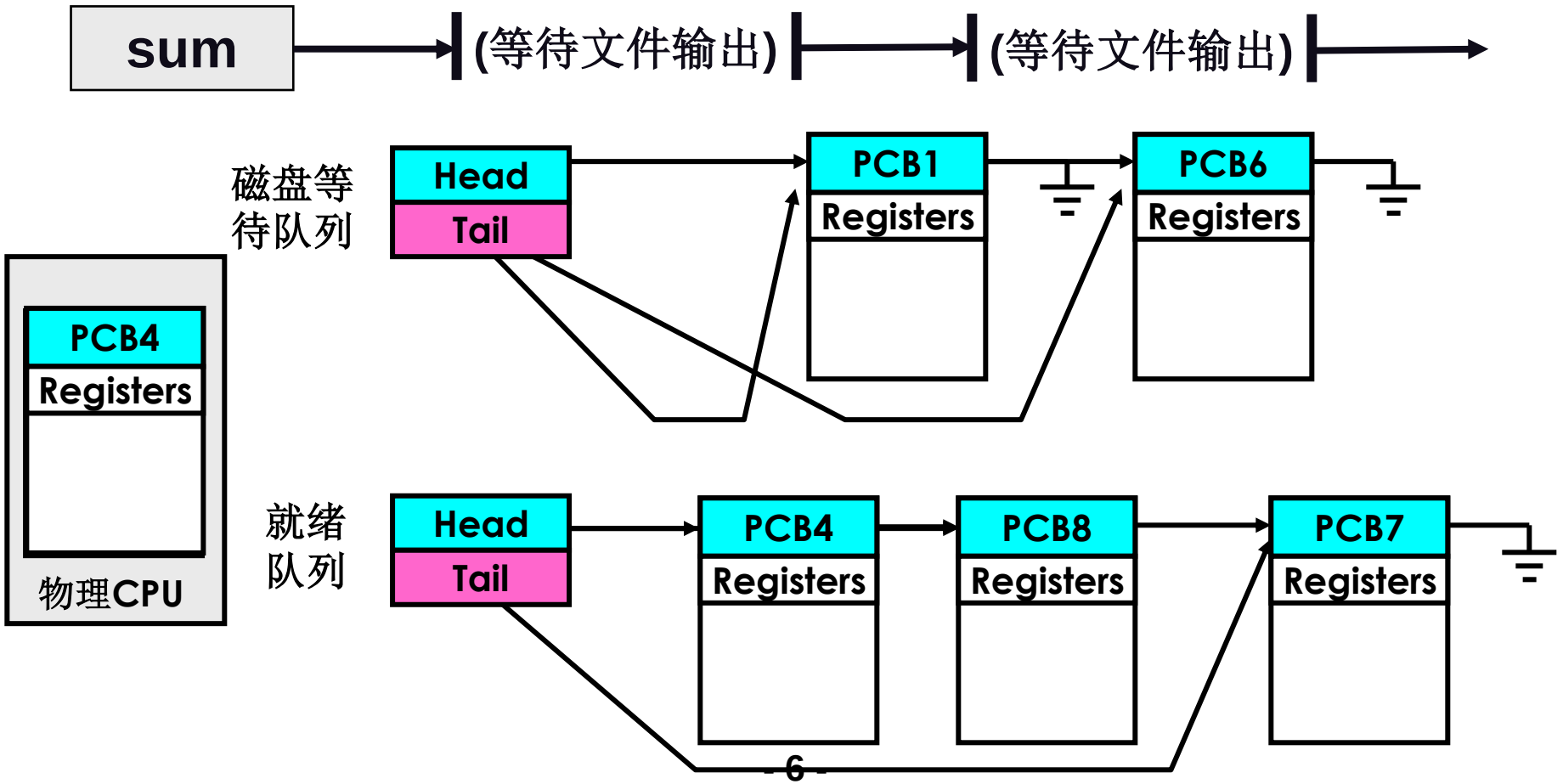


- 实验证明：进程中短的CPU区间出现的几率极高

5.1 基本概念

（4）非抢占式调度与抢占式调度

调度情况1： 因等待某些事件而让出CPU

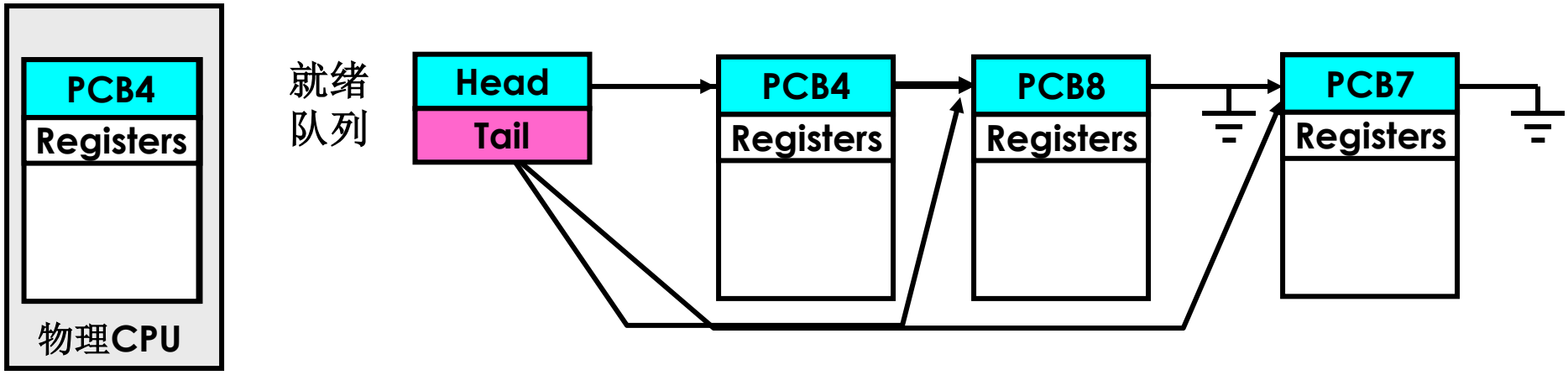


5.1 基本概念

（4）非抢占式调度与抢占式调度（续）

调度情况2：规定的时间片到了

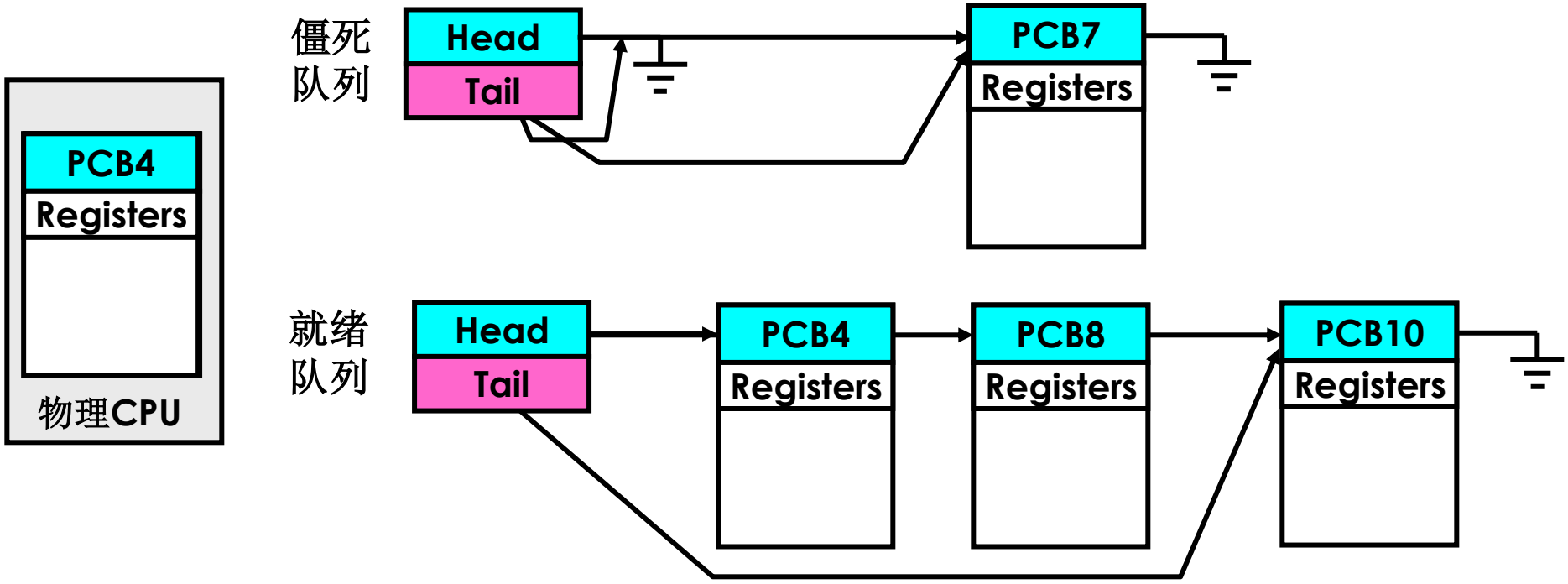
调度情况3：出现了优先级更高的进程



5.1 基本概念

（4）非抢占式调度与抢占式调度（续）

调度情况4：进程的任务完成了，自动终止退出



5.1 基本概念

（4）非抢占式调度与抢占式调度（续）

调度情况1：因等待某些事件而让出CPU

调度情况4：进程的任务完成了，自动终止退出

非抢占式调度：由于主动让出CPU的情况发生，致使调度程序将CPU分配给某就绪进程的调度方式

早期的多道批处理操作系统；Windows3.X版等

调度情况2：规定的时间片到了

调度情况3：出现了优先级更高的进程

抢占式调度：操作系统将正在运行的进程强行暂停，由调度程序将CPU分配给其他就绪进程的调度方式

大多数现代操作系统：1、2、3、4情况都存在且支持

Windows95及之后版本；MacOS-X；Linux等

5.1 基本概念

（5）分派程序（dispatcher）

- 将CPU的控制交给由短期调度程序选择的进程的模块
- 功能
  - 切换上下文
  - 切换到用户模式
  - 跳转到用户程序的而合适位置，以重新启动程序
- 每次进程切换时都要使用，所以应尽可能快
- 分派延迟（dispatch latency）
  - 分派程序停止一个进程而启动另一个所要花的时间称为

让出CPU的具体实现

```
extern Queue ReadyQueue;
extern Queue DiskWaitQueue;
...
DiskWaitQueue.EnQueue(pCur);
Dispatch();
...
Dispatch()
{
    pNew = PickNext(ReadyQueue);
    Switch(pCur,pNew);
}
```

分派程序调用“进程选择函数”完成调度

该函数就是CPU调度，调度就是下一步该选择哪一个进程或线程来执行(分配CPU资源)!

进程和线程都可能是调度单位，有时又称为任务

PickNext()的直观思考

- 简单想一想! 应该有很多种策略
- FIFO?
  - FIFO显然是公平的策略
  - FIFO显然没有考虑进程(线程)执行的任务的区别
- Priority?
  - 优先级该怎么设定?

5.2 调度准则

许多算法：评价准则是什么?

CPU调度策略的设计准则

- 公平性：“合理”的分配CPU
- 响应时间短(交互式)，周转时间短(批处理)
  - 响应时间: 从用户输入到产生反应的时间
  - 周转时间: 从任务提交到任务结束的时间
  - 周转时间短 ⇔ 等待时间(在就绪队列中的时间)短
- 吞吐量大
  - 吞吐量: 单位时间完成的任务数量
  - 吞吐量大 ⇔ CPU使用率高 + 上下文切换代价小

存在矛盾的目标集合

- 响应时间短与公平性之间的矛盾
  - 响应时间短 ⇒ 前台任务的优先级高
  - 前台任务优先调度 ⇒ 后台任务得不到CPU
- 吞吐量和响应时间之间的矛盾
  - 吞吐量大 ⇒ 上下文切换代价小 ⇒ 时间片大
  - 时间片大 ⇒ 响应时间长(100 × 10ms vs. 2 × 500ms)
- ……

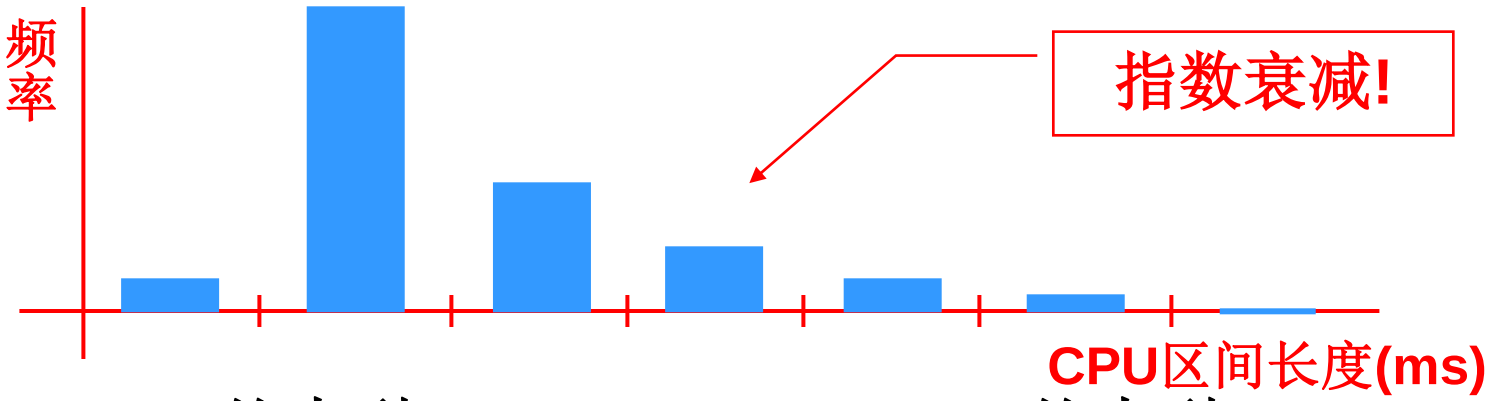
协调多个目标是操作系统之所以复杂的
一个重要原因，也是复杂系统的一个基本特点

CPU调度应综合考虑 – 任务特点

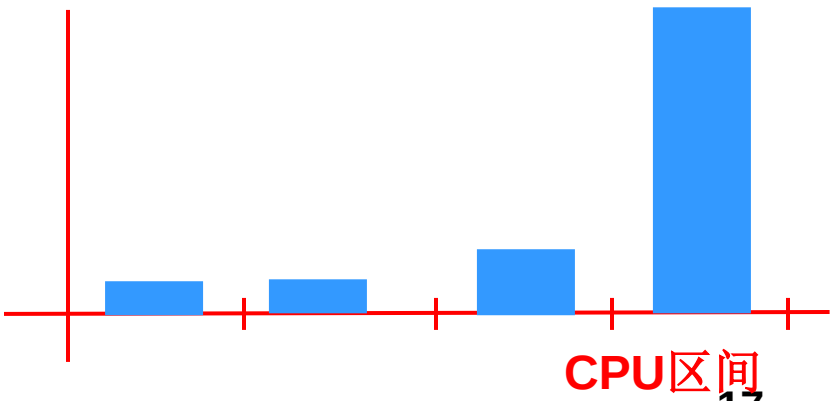
- 任务可以分为交互式任务和批处理任务
  - 交互式任务注重对用户的响应：如WORD
  - 批处理任务注重对任务吞吐量：如gcc
- 有趣的问题是许多系统中既有交互式任务，又有批处理任务
  - 前台任务 + 后台任务

CPU调度应综合考虑 – 任务特点(续)

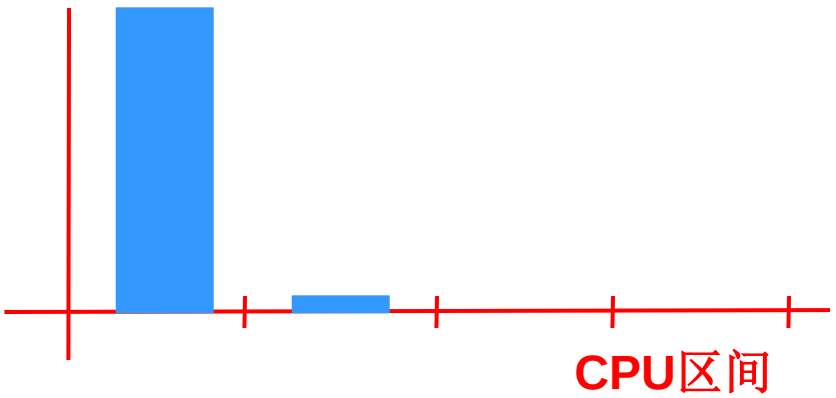
- 任务的CPU-IO区间的周期特性
  - 一般任务生存期: **I/O(载入), CPU, I/O, .., CPU (exit())**



- **CPU-约束型**



- **I/O-约束型**



CPU调度应综合考虑 – 调度算法的实现

- 复杂调度算法 vs. 调度程序执行时间
  - 兼顾许多特点会造成复杂的调度算法
  - 调度程序执行时间应尽量短
  - 就绪队列的数据结构: 队列、多级队列、树、无序链表

- **CPU调度应综合考虑……**

不可能设计完美的调度算法，只能根据应用的特征进行折中权衡。这是操作系统等复杂系统的设计精髓!

5.3 调度算法

CPU调度算法

（1）先到先服务调度 FCFS

（2）最短作业优先调度 SJF

（3）优先级调度

（4）转轮法调度 RR

（5）多级队列调度

（6）多级反馈队列调度

（7）多核/多处理器调度

(1) FIFO或First Come, First Served (FCFS)

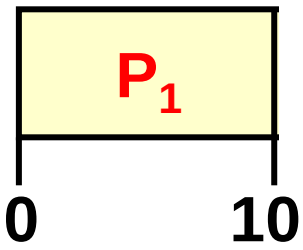
- 调度的顺序就是任务到达就绪队列的顺序

比如：同优先级任务按照进入就绪队列先后顺序执行

- 一个实例
  - 假定任务的到达顺序为:
    P_1，P_2，P_3，P_4，P_5;
    到达时刻都为0。

任务	CPU区间(ms)
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

- 调度结果



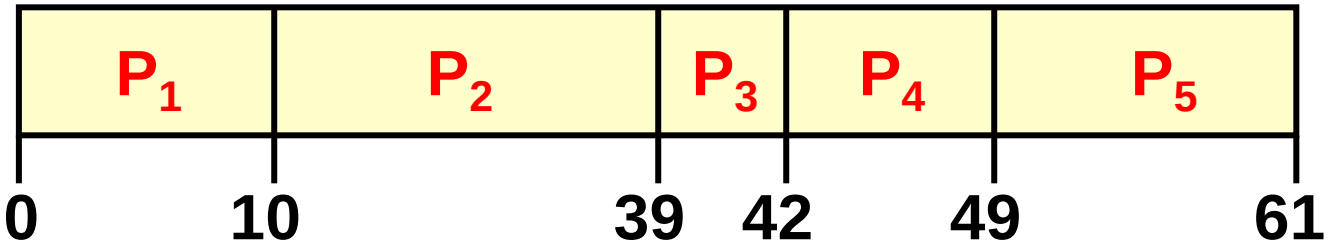
(1) FIFO或First Come, First Served (FCFS)

- 调度的顺序就是任务到达就绪队列的顺序

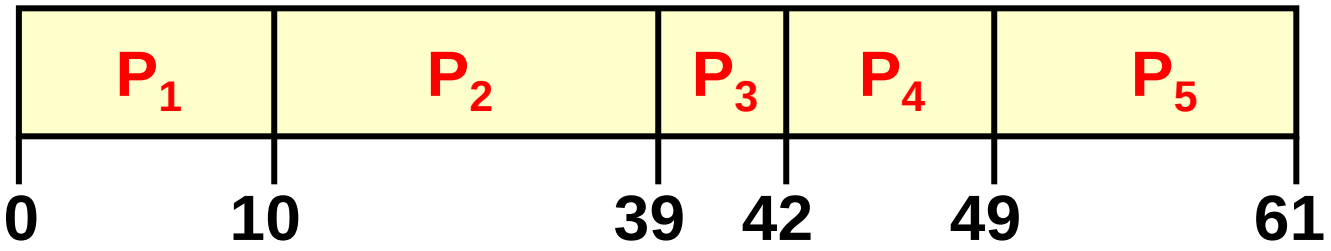
- 一个实例
  - 假定任务的到达顺序为:
    P_1，P_2，P_3，P_4，P_5;
    到达时刻都为0。

任务	CPU区间(ms)
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

- 调度结果



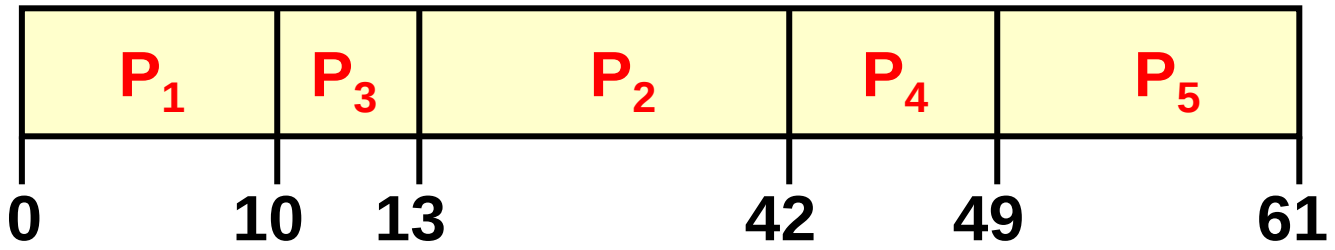
FCFS的分析



- 平均等待时间: (0+10+39+42+49)/5 = 28

公平、简单(FIFO队列)、非抢占、不适合交互式

- 未考虑任务特性，平均等待时间可以缩短



- 平均等待时间: (0+10+13+42+49)/5 = 22.8

(2) Shortest Job First (SJF)

- 最短的作业(CPU区间长度最小)最先调度

- 一个实例
  - 假定任务的到达顺序为:
    P_1，P_2，P_3，P_4，P_5;
    到达时刻都为0。

任务	CPU区间(ms)
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

- 调度结果

(2) Shortest Job First (SJF)

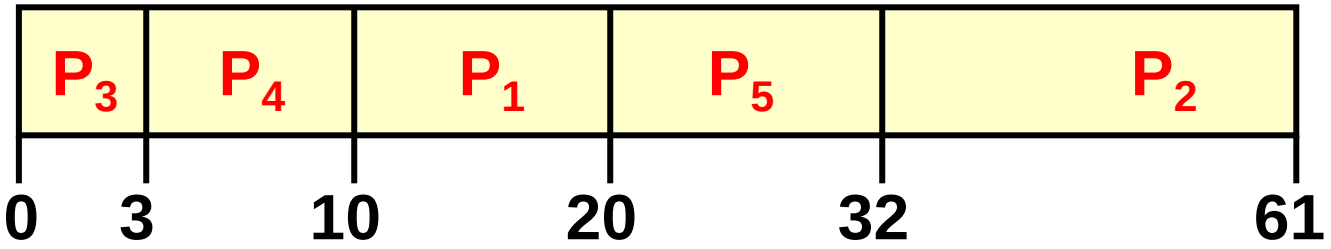
- 最短的作业(CPU区间长度最小)最先调度

根据历史信息判定某类应用的执行时间，优先执行较短的。

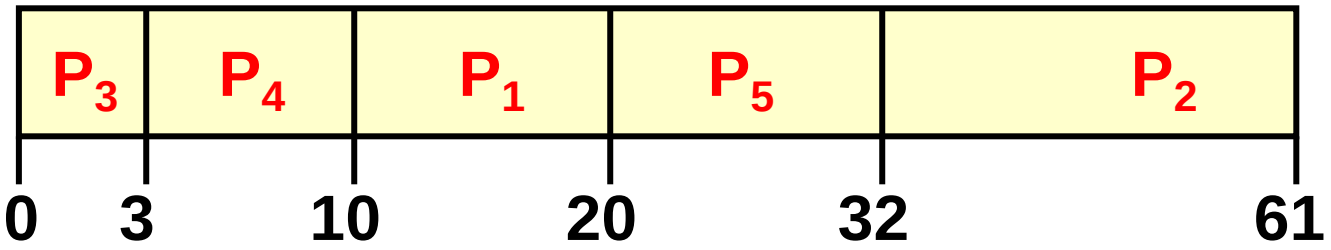
- 一个实例
  - 假定任务的到达顺序为:
    P_1，P_2，P_3，P_4，P_5;
    到达时刻都为0。

任务	CPU区间(ms)
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

- 调度结果



SJF的分析



- 平均等待时间: (0+3+10+20+32)/5 = 13

同时到达作业SJF可以保证最小的平均等待时间

- 证明: 设$P_1P_2...P_n$是调度结果序列，p_i是任务CPU区间大小。显然该调度序列的平均等待时间为:

  $(0 + p_1 + p_1+p_2+ p_1+p_2+p_3 + \ldots.)/n$

  如果存在$i<j$而$p_i>p_j$，交换p_i,p_j调度顺序会减小上值。

SJF: 任务到达的时间有先后怎么办?

- Shortest Remaining Job First (SRJF)
- SJF的可抢占版本

- 一个实例
  - 假定任务P_1，P_2，P_3，P_4，P_5的到达顺序为:

任务	到达时间 (ms)	CPU区间 (ms)
P_1	0	10
P_2	0	29
P_3	5	3
P_4	5	7
P_5	30	12

- 调度结果

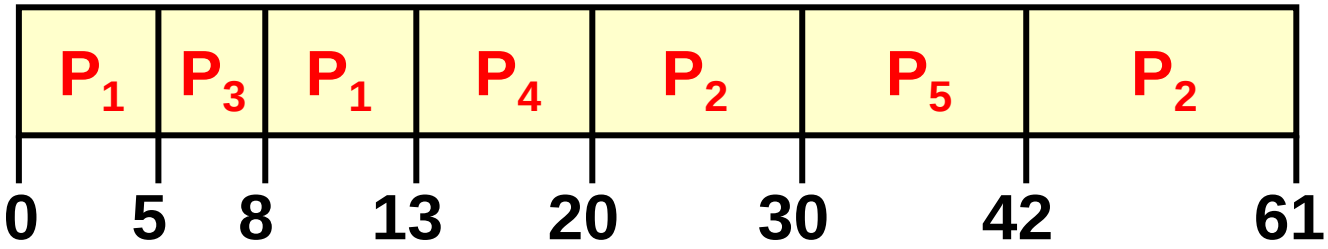
SJF: 任务到达的时间有先后怎么办?

- Shortest Remaining Job First (SRJF)
- SJF的可抢占版本

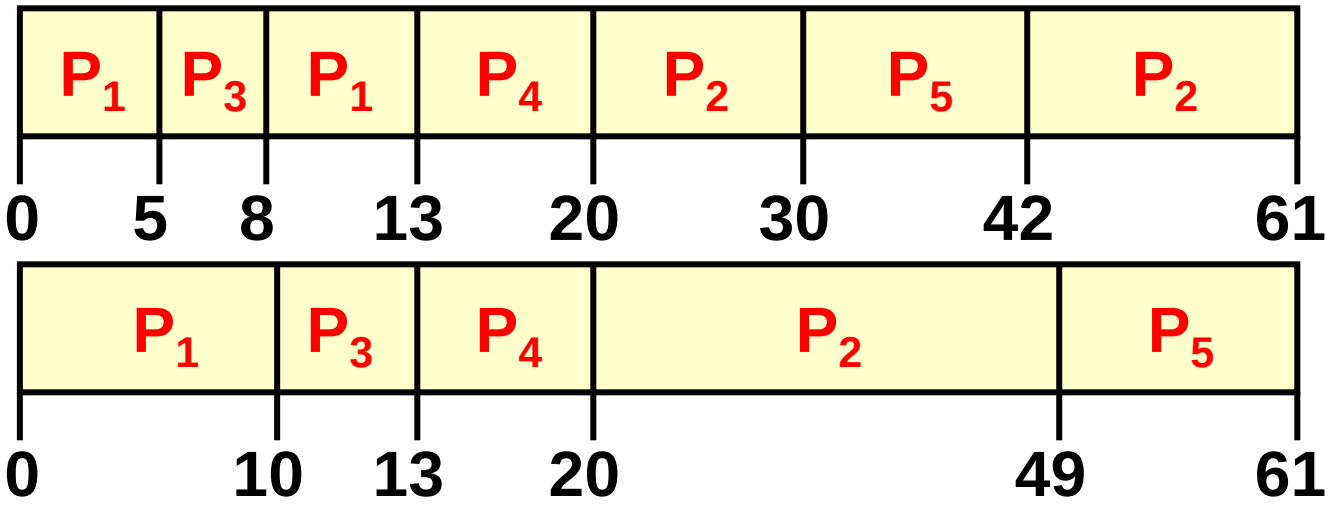
- 一个实例
  - 假定任务P_1，P_2，P_3，P_4，P_5的到达顺序为:

任务	到达时间 (ms)	CPU区间 (ms)
P_1	0	10
P_2	0	29
P_3	5	3
P_4	5	7
P_5	30	12

- 调度结果



SJF vs. SRJF



- 平均等待时间(SRJF):

  (3+32+0+8+0)/5 = 8.6

- 平均等待时间(SJF):

  (0+20+5+8+19)/5 = 10.4

- 抢占显然具有优点

CPU区间必须是已知!

任务	到达时间 (ms)	CPU区间 (ms)
P_1	0	10
P_2	0	29
P_3	5	3
P_4	5	7
P_5	30	12

SJF(SRJF): 如何知道下一CPU区间大小

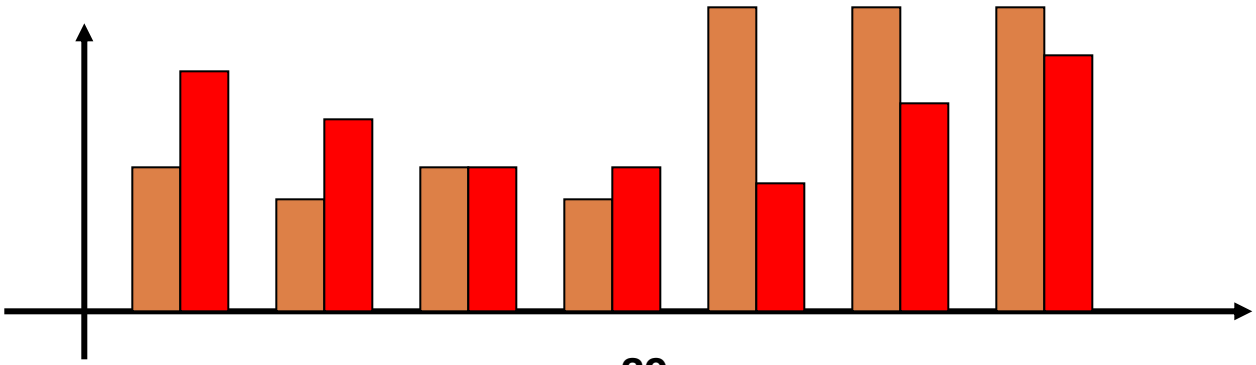
- 根据历史进行预测: 指数平均法

$\tau_{n+1} = \alpha t_n + (1- \alpha)\tau_n$ | τ_{n+1}是预测值，t_n是当前CPU区间

  - $\tau_{n+1} = \alpha t_n + (1- \alpha)\alpha t_n + \ldots + (1- \alpha)^j \alpha t_{n-j} + \ldots + (1-\alpha)^{n+1}\tau_0$
  - α用来控制最近信息和历史对预测的作用

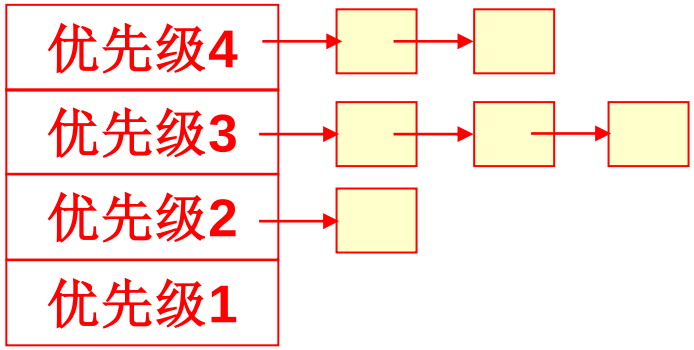
    α=0-忽略近期，关注历史； α=1-只跟当前有关；通常α=0.5

CPU区间(t_i):	6	4	6	4	13	13	13	
预测结果(τ_i):	10	8	6	6	5	9	11	12



(3) SJF的一般化: 优先权调度

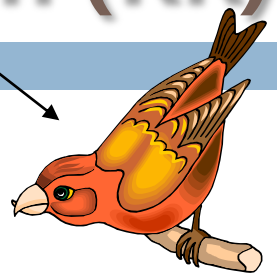
- 每个任务关联一个优先权，调度优先权最高的任务
  - 实例1: I/O bound任务应获得更大的优先权，使得I/O尽量忙，并和CPU并行工作。优先级设为1/f，f为CPU区间所占的比重。
  - 实例2: 定义多个优先队列: 前台任务、后台任务。只有高优先级队列为空时才调度其他任务。



优先权调度引起的一个有趣问题

- 优先权太低的任务一直就绪，得不到运行: 饥饿
  - 一个有趣故事: 1973年关闭的MIT的IBM 7094时，发现有一个进程在1967年提交但一直未运行。
  - 处理办法: 优先级动态调整。
  - 最常见的方法是“老化”(aging)技术: 任务的优先级会随等待时间增长而不断增高。
- FCFS是RR的特例，SJF是优先权调度的特例。这些调度算法都不适合于交互式系统。

(4) 适合交互式的调度: Round-Robin (RR)



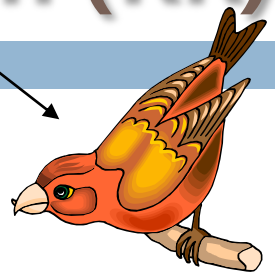
- RR: 按时间片来轮转调度（“轮叫”算法）

- 一个实例
  - 假定任务的到达顺序为:
    P_1，P_2，P_3，P_4，P_5;
    到达时刻都为0，时间片为10。

任务	CPU区间(ms)
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

- 调度结果

(4) 适合交互式的调度: Round-Robin (RR)

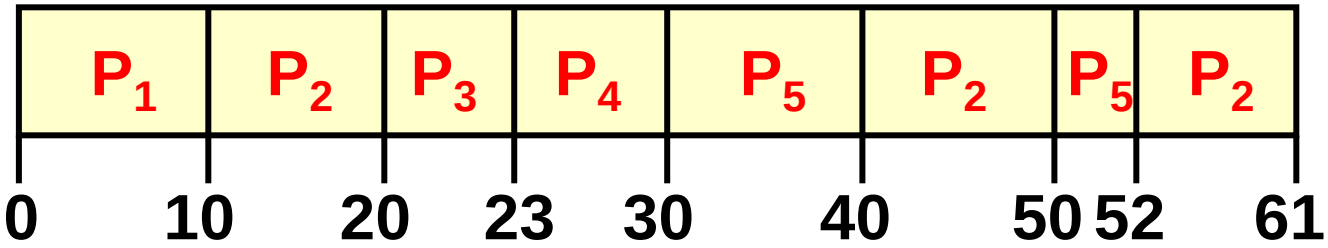


- RR: 按时间片来轮转调度（“轮叫”算法）

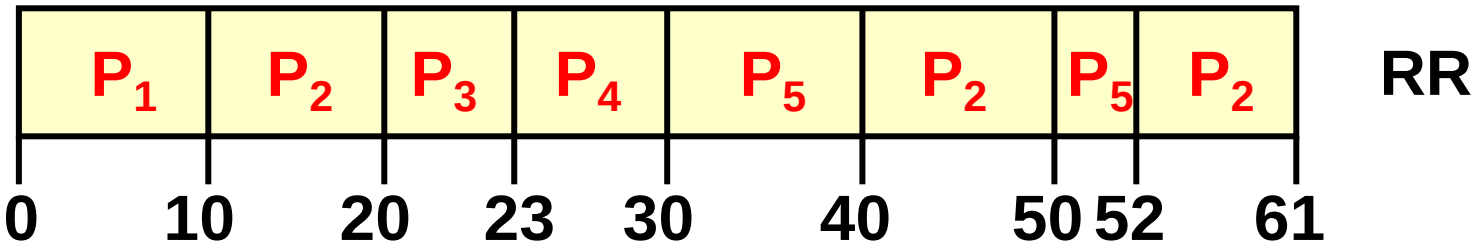
- 一个实例
  - 假定任务的到达顺序为:
    P_1，P_2，P_3，P_4，P_5;
    到达时刻都为0，时间片为10。

任务	CPU区间(ms)
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

- 调度结果

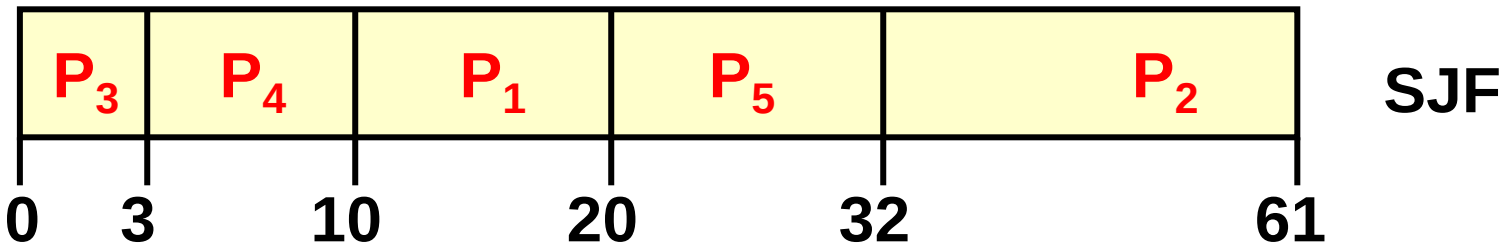


RR的分析



- 平均等待时间？ (0+32+20+23+40)/5 = 23
- SJF的平均等待时间: 13; FCFS的平均等待时间: 28

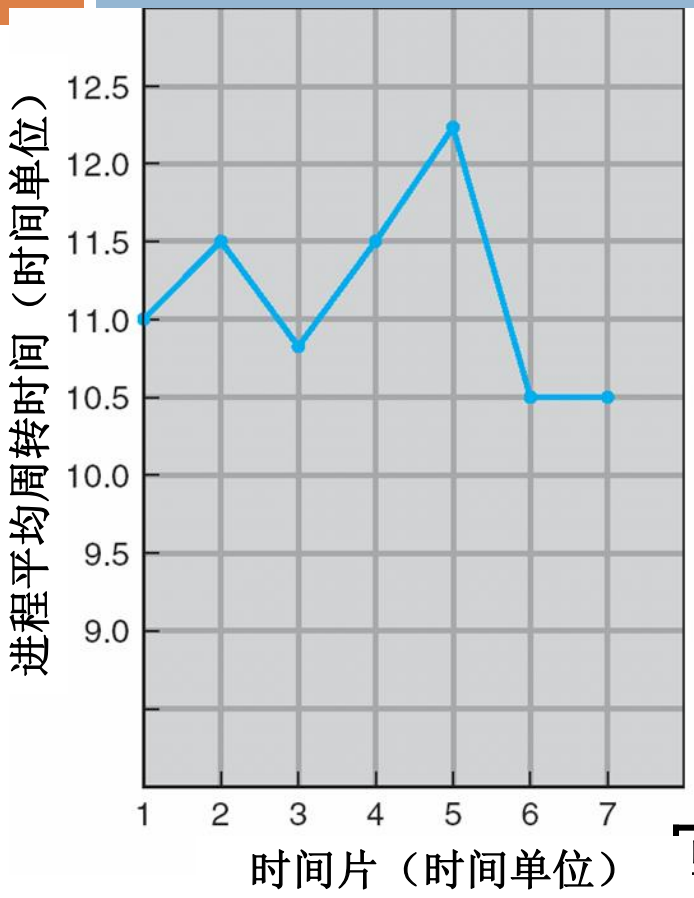
- RR优点: 定时有响应，等待时间较短
- RR缺点: 上下文切换次数较多(7次 vs. 4次)



RR中的时间片该如何设定?

- 时间片太大
  - 响应时间太长
  - 如时间片500ms × 10任务，响应需要5秒
- 时间片太小
  - 吞吐量变小，周转时间变长
  - 如时间片20ms，上下文切换5ms，20%的切换代价
- 折中: 时间片10-100ms，切换时间0.1-1ms(1%)

RR调度例子：周转时间随着时间片大小而变化



process	time
P_1	6
P_2	3
P_3	1
P_4	7

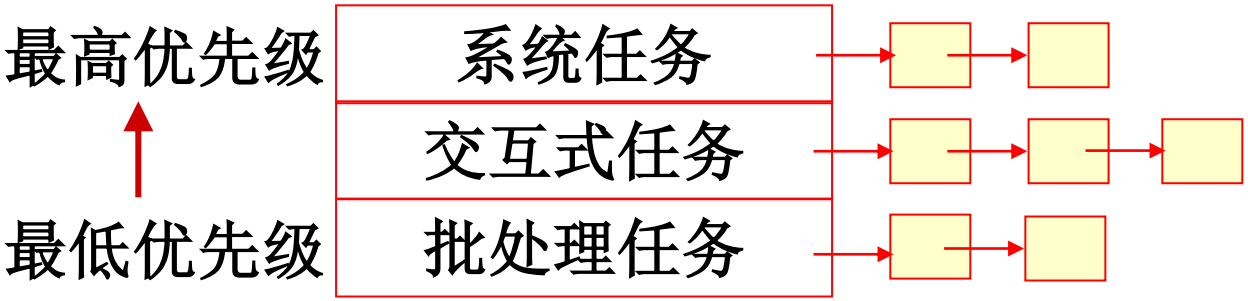
当时间片≥7时等同FCFS！

时间片	进程切换过程																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	P1	P2	P3	P4	P1	P2	P4	P1	P2	P4	P1	P4	P1	P4	P1	P4	P4
2	P1		P2		P3	P4		P1		P2	P4		P1		P4		P4
3	P1			P2			P3	P4			P1			P4			P4
4	P1				P2			P3	P4				P1		P4		
5	P1					P2			P3	P4					P1	P4	
6	P1						P2			P3	P4						P4
7	P1						P2			P3	P4						

时间片	P1周转时间	P2周转时间	P3周转时间	P4周转时间	平均周转时间
1	15	9	3	17	44/4=11.0
2	14	10	5	17	46/4=11.5
3	13	6	7	17	43/4=10.75
4	14	7	8	17	46/4=11.5
5	15	8	9	17	49/4=12.25
6	6	9	10	17	42/4=10.5
7	6	9	10	17	42/4=10.5

(5) 混合多种调度算法 – 多级队列调度

- 按照一定的规则建立多个进程队列
- 不同的队列有固定的优先级（高优先级有抢占权）
- 不同的队列可以给不同的时间片和采用不同的调度方法

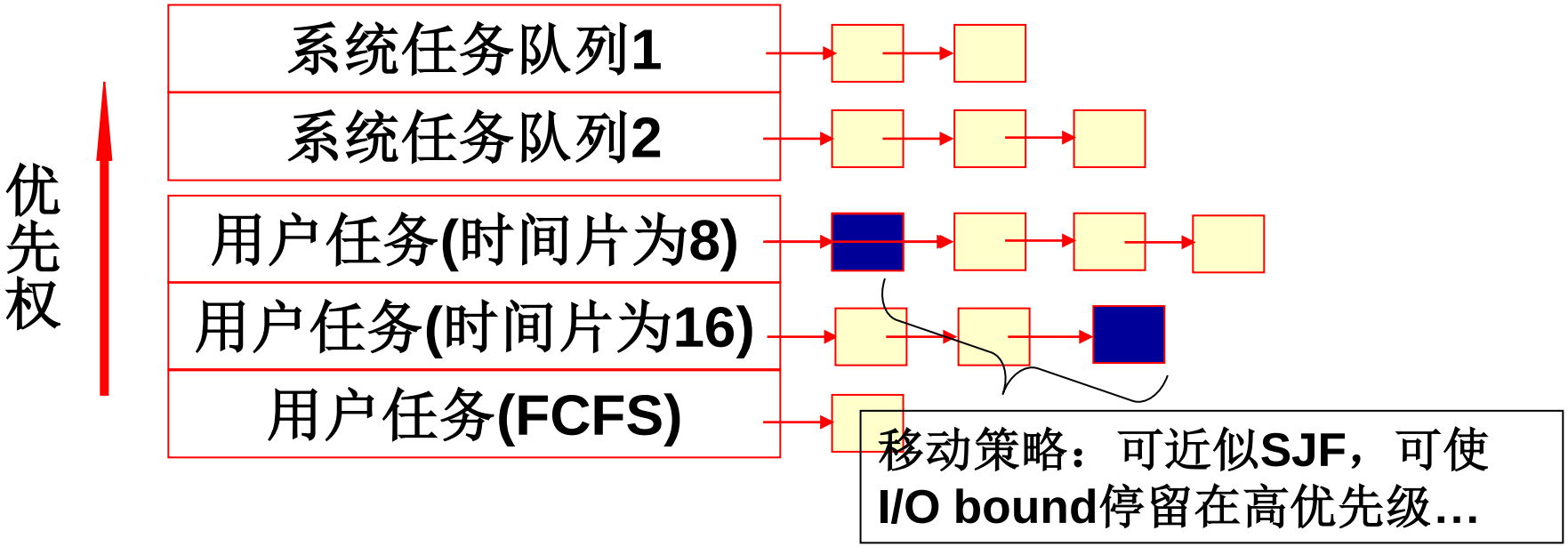



```
if(!IsEmpty(KernelQ)){ next=Pri(); return;}
if(!IsEmpty(ResponseQ)){ next=RR(); return;}
if(!IsEmpty(BatchQ)){ next=SJF(); return;}
...
```

- 存在问题：也存在一定程度的“饥饿”现象

(6) 更成熟的多级队列调度 – 多级反馈队列

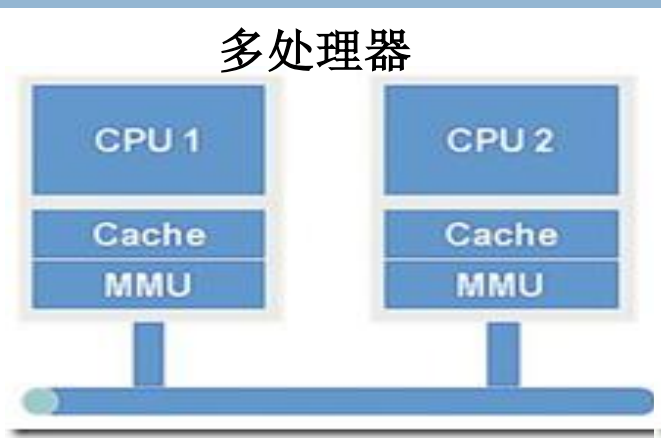
- 任务可以在队列之间移动，更细致的区分任务

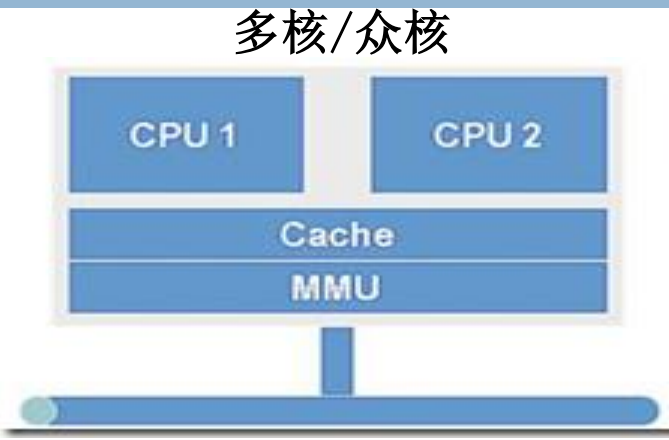


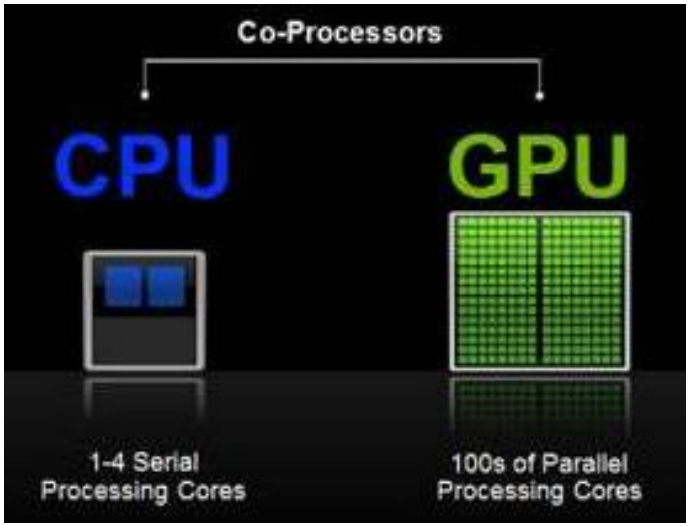
- 可根据占用CPU时间多少来移动队列，阻止“饥饿”，可在PCB中记录其占用的CPU区间，作为移动依据
- 最通用的调度算法，多数OS都使用该方法或其变形，如UNIX、Windows等。

(7) 多核CPU调度算法

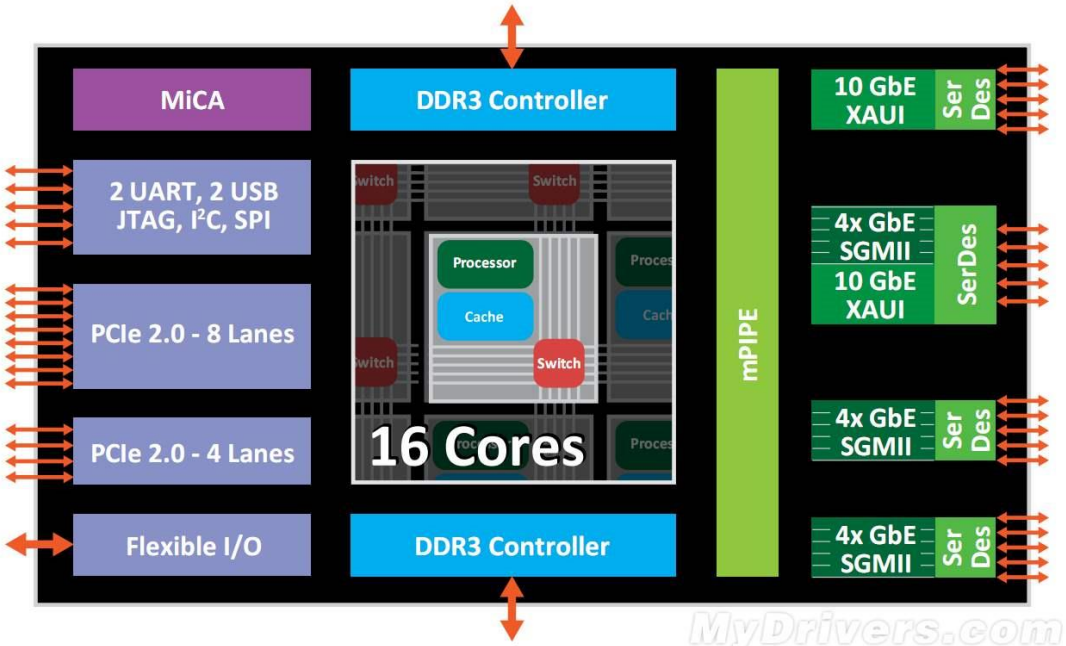
- 当系统CPU只有一个核心或一个处理器时，上述的算法策略已经较为充分。
- 多核/多处理器会给调度带来什么问题？
  负载、竞争、Cache





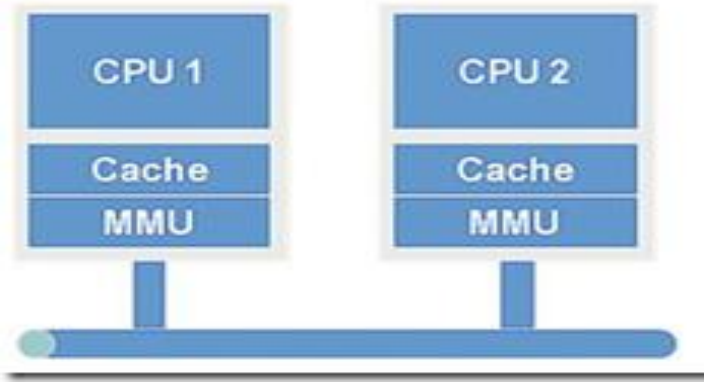


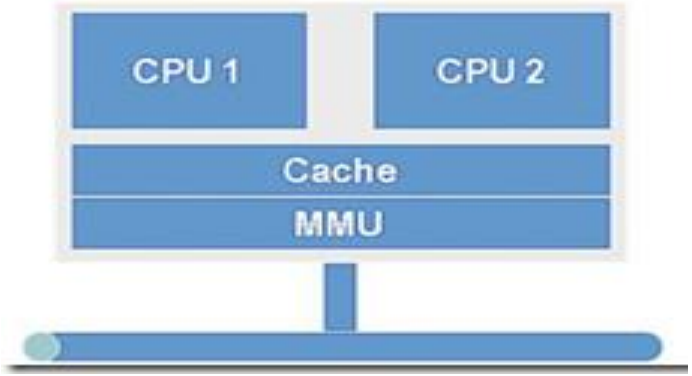
图形处理器(Graphics Processing Unit：GPU）



(7) 多核CPU调度算法

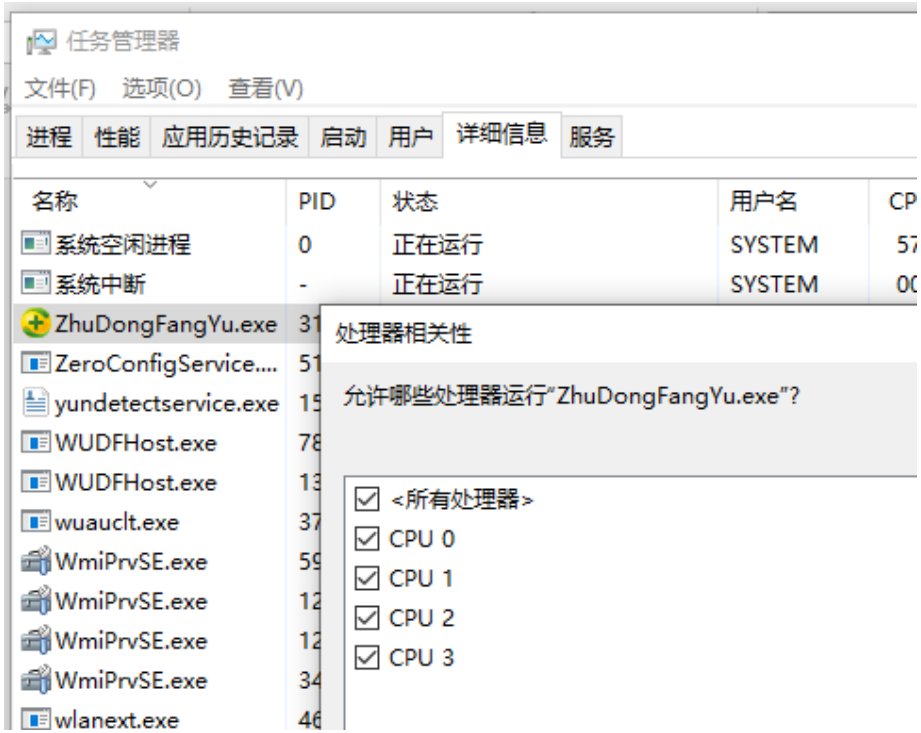
- CPU的亲和性affinity ： 就是进程/线程要在指定的 CPU 上尽量长时间地运行而不被迁移到其他处理器，也称为CPU关联性；
- 多核处理器，每个CPU（或特定几个CPU共享）有缓存，缓存着进程使用内存或寄存器等信息，当进程被OS调度到其他CPU时，CPU cache命中率可能就会降低。





（7）多核CPU调度算法

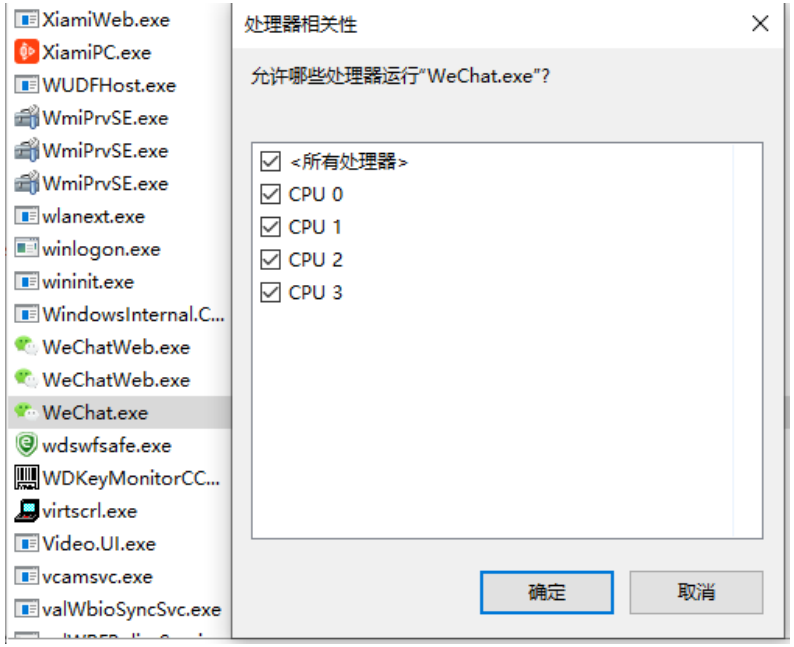
- 软亲和性：进程要在指定的 CPU 上尽量保持长时间。当前Linux 内核进程调度器具有软 CPU 亲和性，这意味着进程通常不会在处理器之间频繁迁移。
- 硬亲和性：简单来说就是利用内核提供给用户的API，强行将进程或者线程绑定到某一个指定的cpu核运行。



(7) 多核CPU调度算法

windows软关联（软亲和性）：

- SetProcessAffinityMask 用于设置进程的CPU集合，具有对应的Get函数；
- SetThreadAffinityMask 用于设置线程的CPU集合，具有对应的Get函数。



(7) 多核CPU调度算法

Linux**多核调度器的亲和性**

- 在 Linux 内核中，进程控制块 task_struct中与亲和性（affinity）相关是 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。
- 具有 4 个物理 CPU 的系统可以有 4 位。如果这些 CPU 都启用了超线程，那么这个系统就有一个 8 位的位掩码。
- Linux 中通过set_cpus_allowed_ptr函数设置特定CPU

（7）多核CPU调度算法

Linux多核调度器的亲和性

- 如果为给定的进程设置了给定的位，那么这个进程就可以在相关的 CPU 上运行。如果一个进程可以在任何 CPU 上运行，可在处理器之间进行迁移，那么位掩码就全是 1（缺省状态）。
- Linux 内核 API 提供了：

① sched_set_affinity()（用来修改位掩码）

② sched_get_affinity()（用来查看当前的位掩码）

③ cpu_affinity 会被传递给子线程

（7）多核CPU调度算法

- linux 2.4 版本O(n)调度器

① 采用一个runqueue运行队列来管理所有就绪进程，调度器schedule中会选择一个优先级最高（时间片最大）进程运行，同时对睡眠的进程增加一些时间片。

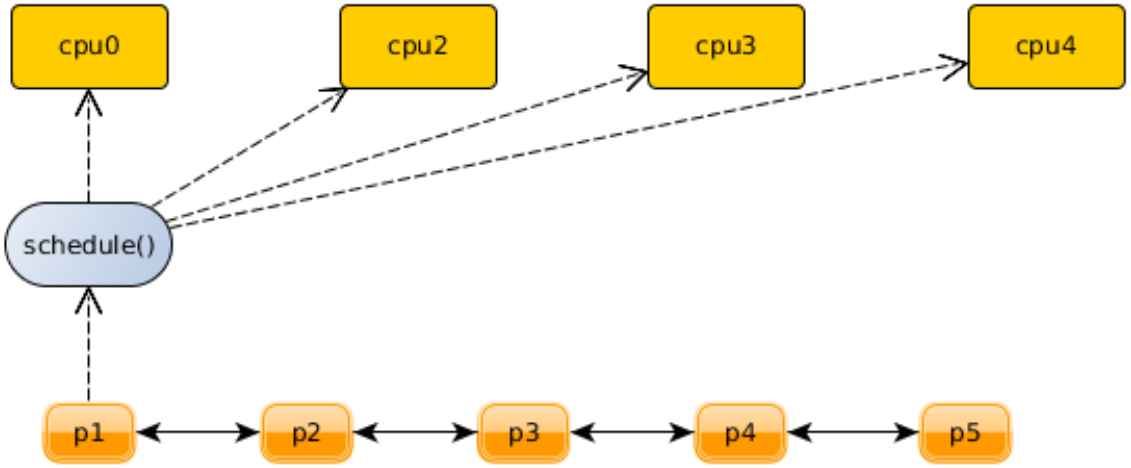
② 当runqueue运行队列中无进程可选择时，则会对系统中所有的进程进行一次重新计算时间片的操作，同时也会对剩余时间片的进程做一次补偿。

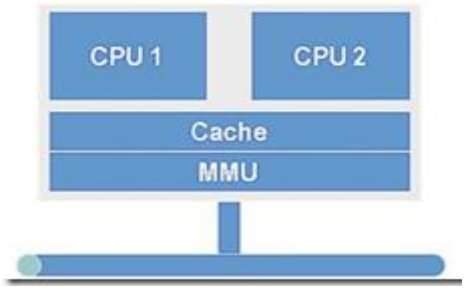
```
long alarm;    // 报警定时值
long utime, stime, cutime, cstime, start_time;
     // utime 用户态运行时间
     // stime 内核态运行时间
     // cutime 子进程用户态运行时间
     // cstime 子进程内核态运行时间
     // start_time 进程开始运行时刻
unsigned short used_math;    // 标志，是否使用了 387 协处理器
```

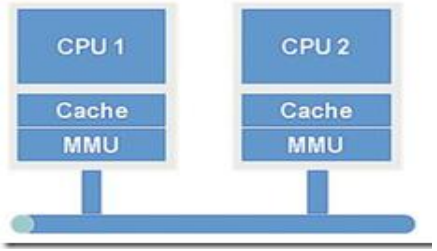
(7) 多核CPU调度算法

O(n)调度器的缺陷：

- 时间复杂度是O(n)
- SMP系统扩展不好，各CPU空闲访问runqueue需要加锁
- 实时、交互进程不能及时调度
- CPU空转的现象存在（等待锁，多CPU访问共享数据）
- 进程在各个CPU之间迁移。如果任务被调度到 CPU-2 上执行，那么在CPU-1中 cache数据将无效，并将其缓存到 CPU-2 的cache中。







（7）多核CPU调度算法

Linux O(1) 调度器

- Linux调度器发展过程:O(n)->O(1)->O(log(n))

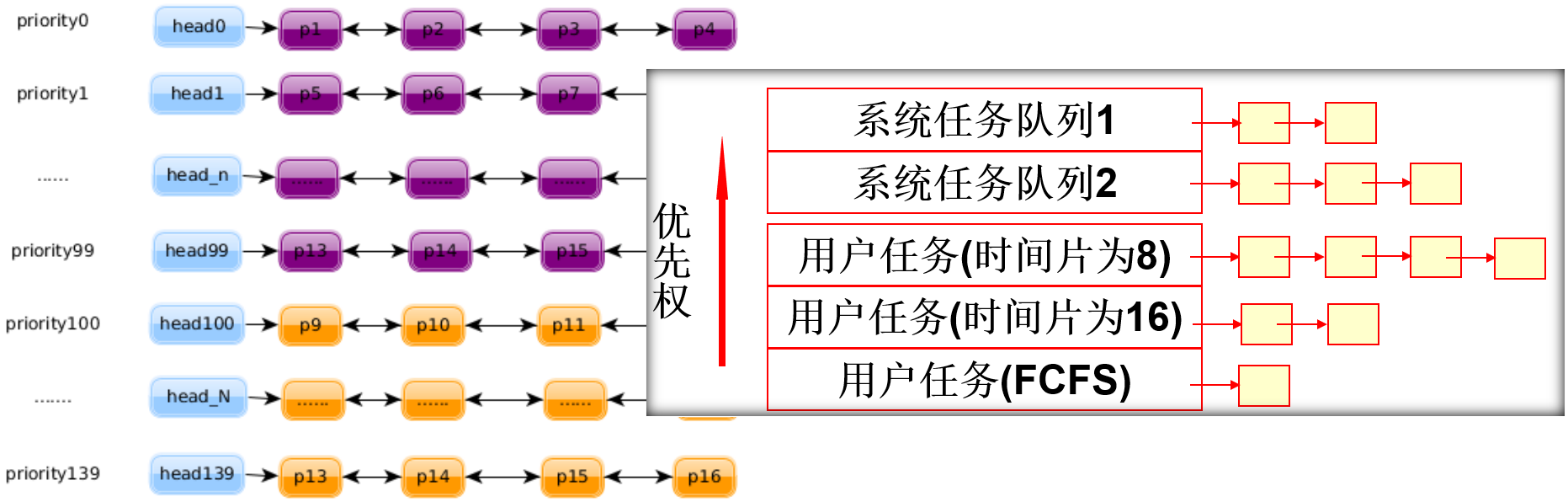
字段	版本
O(n)的始调度算法	linux-0.11~2.4
O(1)调度器	linux-2.5
CFS调度器	linux-2.6~至今

- **Linux O(1) 调度器**中引入了per-CPU runqueue的结构。系统中所有的可运行状态的进程首先经过**负载均衡模块**挂入各CPU的runqueue
- 每隔 200ms检查 CPU 的负载是否均衡，如果不均衡，**负载均衡模块**在 CPU 之间进行一次任务均衡操作。然后由各CPU调度器调度。

(7) 多核CPU调度算法

Linux O(1) 调度器

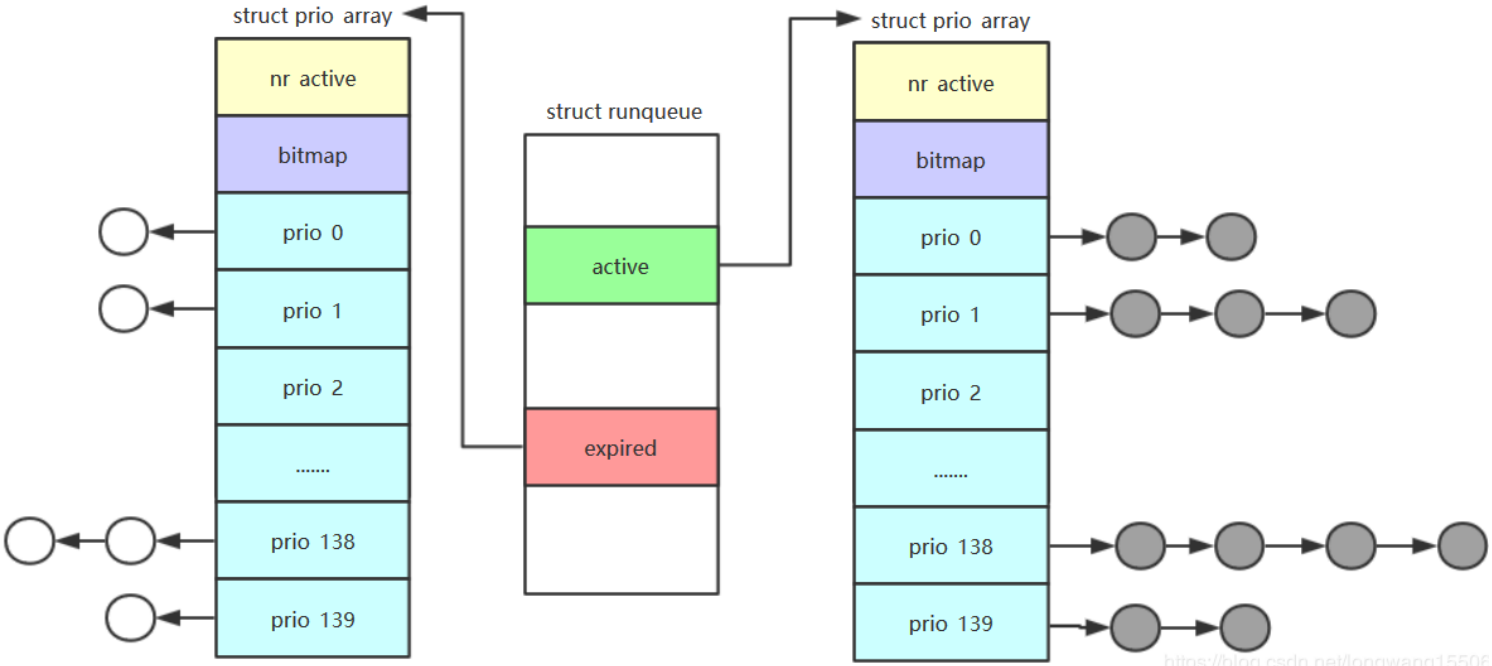
- O(1)调度有140个优先级级别，由0～139, 0～99 为实时优先级，而100～139为非实时优先级。
- 每个cpu runqueue维护自己的active队列与expried队列。当前cpu上运行进程的时间片用完后就会被放入expired队列中。
- 当active队列中所有进程的时间片都用完，进程执行完毕后，交换active队列和expried。
- expried队列就成为了active队列。这样做只需要指针的交换。



（7）多核CPU调度算法

Linux O(1) 调度器

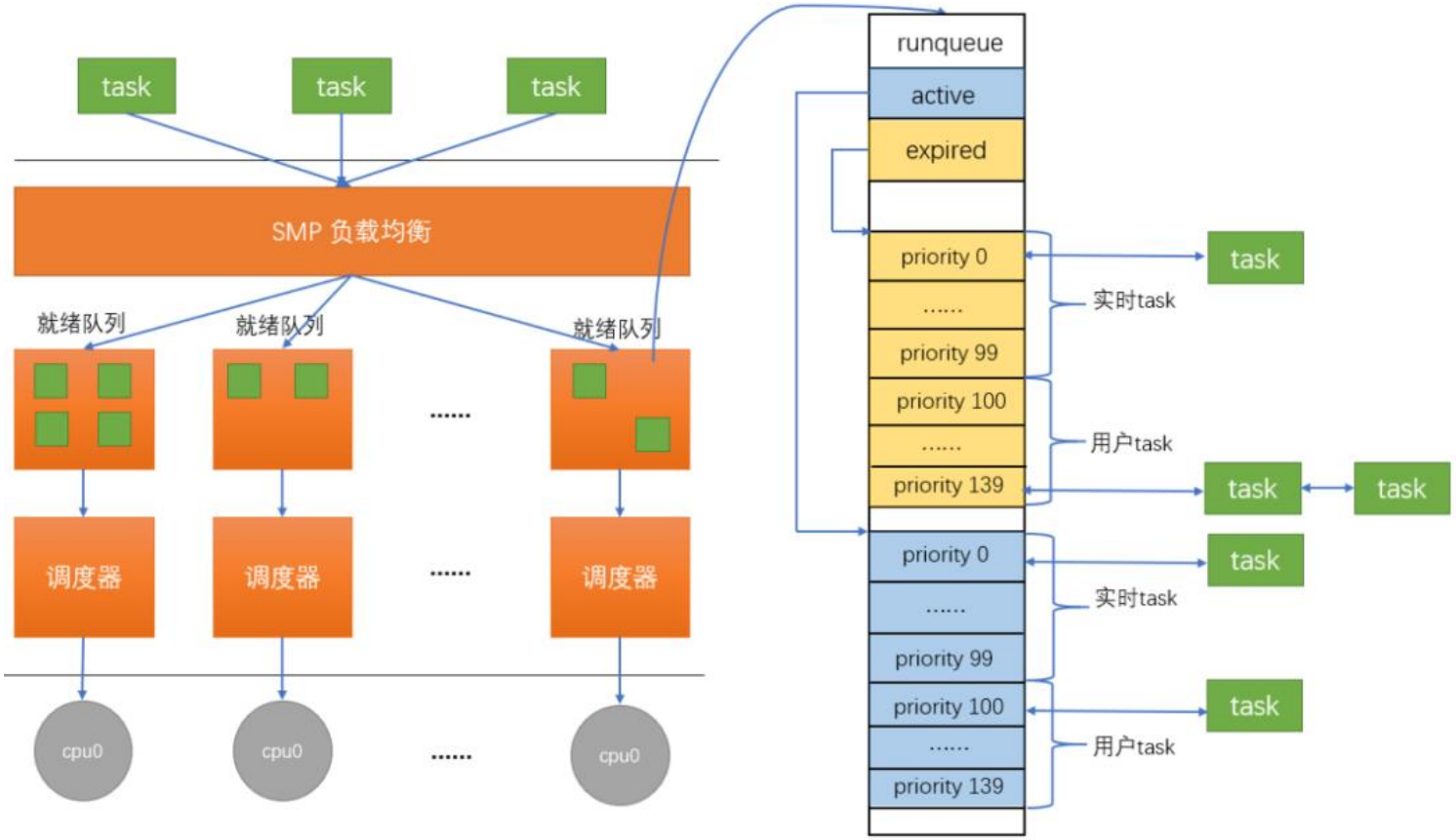
- runqueue是一个PER_CPU变量，SMP系统可有效的避免多个CPU去访问同一个runqueue（避免竞争）。
- 为了解决O(n)中所有的进程都无序排列在runqueue中，O(1)算法中将进程按照优先级排列，而且相同优先级的都挂在同优先级的队列中。
- 同时提供了一个bitmap结构，用来存放哪些优先级中有可以运行的进程。当每次picknext的时候，只需要访问bitmap，然后去对应的优先级队列中按照优先级策略选择进程。



（7）多核CPU调度算法

Linux O(1) 调度器

- Task 与负载均衡和 runqueue 以及对应调度器之间的关系：
- 每个 runqueue 里又会分为active和expired队列，每个队列中可挂载140个优先级不同的 task 。关于调度器在 runqueue 里的算法实现我们看下面一张图：



(7) 多核CPU调度算法

Linux O(1) 调度器

- Linux 2.6 kernel 里有 140 种优先
每个优先级下面用一个 FIFO queu
- 如何找到当前CPU runqueue最高
- 如果从 0 开始一直遍历下去，算法
关的 O(M)，也不能算作 O(1)。
- 在bitarray中为每种优先级分配一
process，那么就对相应的 bit 染色，置为 1，否则置为 0。问题就简化成寻找一个 bitarray 里面最高位是 1 的 bit（left-most bit）。可以通过位操作实现。

图5.10 位扫描指令的功能示意图

a) 正向扫描指令BSF(Bit Scan Forward)从右向左扫描，即：从低位向高位扫描；

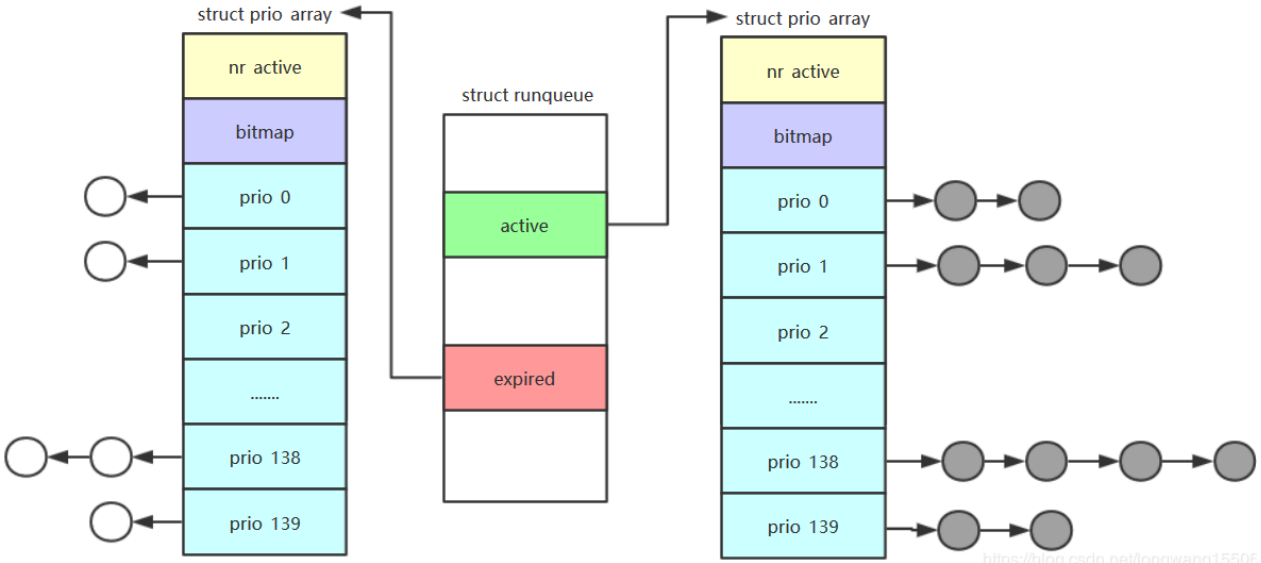
b) 逆向扫描指令BSR(Bit Scan Reverse)从左向右扫描，即：从高位向低位扫描。

（7）多核CPU调度算法：

Linux O(1) 调度器

O(1)算法基本步骤：Picknext

1. 在 active bitarray 里，寻找 left-most bit 1 的位置 x。
2. 在 active priority array（APA）中，找到对应队列 APA[x]。
3. 从 APA[x] 中 dequeue 一个 process，dequeue （出队）后，如果 APA[x] 的 queue 为空，那么将 active bitarray 里第 x bit置为 0。
4. 对于当前执行完的 process，重新计算其 priority，然后 enqueue（入队） 到 expired priority array（EPA）相应的队里 EPA[priority]。
5. 如果 priority 在 expired bitarray 里对应的 bit 为 0，将其置 1。
6. 如果 active bitarray 全为零，将 active bitarray 和 expired bitarray 交换。

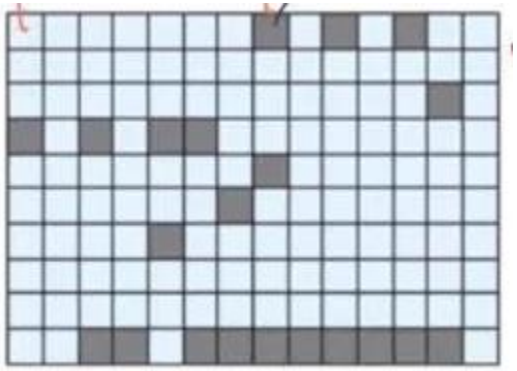


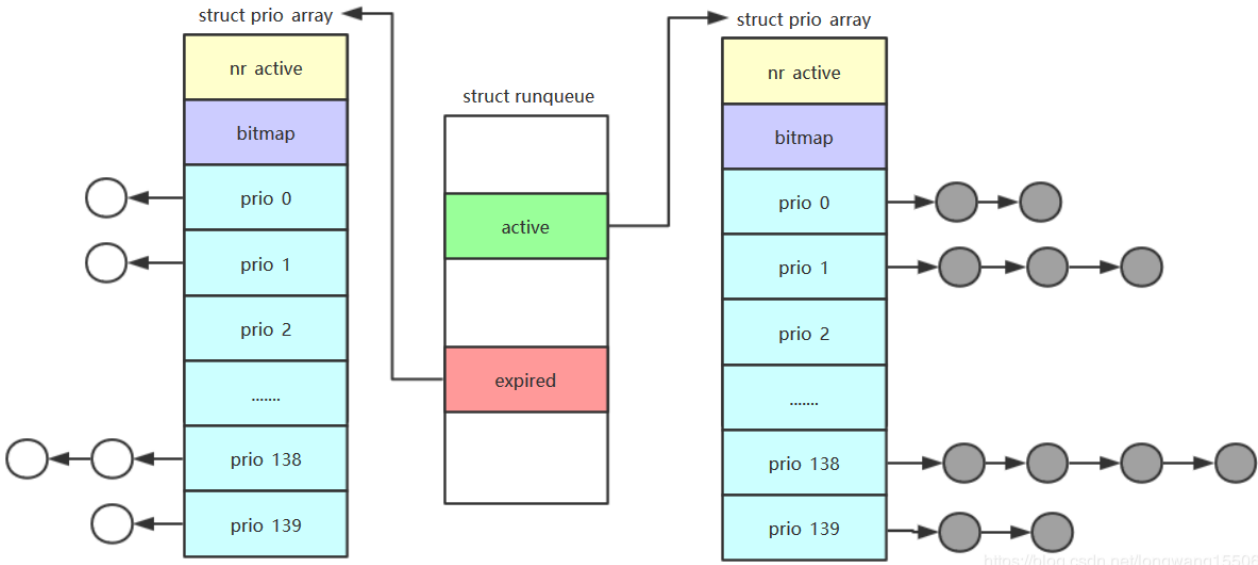
（7）多核CPU调度算法

Linux O(1) 调度器

O(1)调度器时间复杂度分析：

- linux O(1)调度器使用了2个(bitarray+queue)数据结构, 整个调度开销如下: 针对 [140]bit+[140]queue
- Search,insert,delete都是bitarray的O(1)+queue的O(1)来完成的。





（7）多核CPU调度算法

- 全局队列调度
  - 操作系统维护一个全局的任务等待队列。
  - 当系统中有一个CPU核心空闲时，操作系统就从全局任务等待队列中选取就绪任务开始在此核心上执行。
  - 这种方法的优点是CPU核心利用率较高。
- 局部队列调度。
  - 操作系统为每个CPU内核维护一个局部的任务等待队列。
  - 当系统中有一个CPU内核空闲时，便从该核心的任务等待队列中选取恰当的任务执行。
  - 这种方法的优点是任务基本上无需在多个CPU核心间切换，有利于提高CPU核心局部Cache命中率。

（7）多核CPU调度算法

多核、多线程与任务类型

- a.多核CPU——计算密集型任务。尽量使用多线程，例如加密、解密，数据压缩解压缩（视频、音频、普通数据）。
- b.单核CPU——计算密集型任务。任务已经把CPU资源消耗了，使用多线程提升CPU利用率空间很小；如果人机交互多，尽量用多线程。

- c.单核CPU——IO密集型任务，使用多线程还是为了人机交互，
- d.多核CPU——IO密集型任务，跟单核原因一样。

Linux中（分时OS）的调度时机（调度点）

- 由用户引起的调度: 如创建进程、调用wait等让出CPU。此时的调度时机是系统调用返回，fork等都是系统调用

```
在linux/kernel/system_call.s中
_system_call:
   call _sys_call_table(,%eax,4) //执行系统调用
   pushl %eax  //将返回值压栈
   movl _current,%eax //取出当前进程指针
   cmpl $0,state(%eax) //看当前进程状态是否为0(就绪)
   jne reschedule//不是0，所以想调度只需修改该域为非0即可
reschedule: pushl $ret_from_sys_call
   jmp _schedule  //转去执行schedule()，调度
ret_from_sys_call: pop xx    iret//切换回用户态执行
```

- 由内核引起的调度: 如时钟中断，给资源上锁等。此时只需在内核的适当位置(如do_timer())调用schedule()

（实时）嵌入式操作系统调度时机

- 基于优先级抢占调度策略：
- 引起任务阻塞或使阻塞任务变为就绪的系统调用返回点；
- 对于同优先级采用时间片轮转，此时时间片定时中断发生时进行调度。

案例分析

Linux调度算法分析

- 目前Linux支持三种进程调度策略，分别是SCHED_FIFO 、SCHED_RR和SCHED_NORMAL；而Linux支持两种类型的进程，实时进程和普通进程。
- 实时进程可以采用SCHED_FIFO 和SCHED_RR调度策略；普通进程则采用SCHED_NORMAL调度策略O(1)\CFS。
- 从Linux2.6.23内核版本以后，普通进程（采用调度策略SCHED_NORMAL的进程）采用了绝对公平调度算法CFS，不再跟踪进程的睡眠时间，也不区分是否为交互式进程，是完全公平的含义。

Linux调度算法分析

CFS调度器基本原理

① 为每一个进程都设置一个虚拟时钟-virtual runtime (vruntime)。

② 如果一个进程随着执行时间的不断增长，其vruntime也将不断增大，没有得到执行的进程vruntime将保持不变。

③ 调度器将会选择最小的vruntime那个进程来执行。这就是所谓的“完全公平”。

④ 不同优先级的进程其vruntime增长速度不同，优先级高的进程vruntime增长得慢，所以它可能得到更多的运行机会。

Windows调度算法分析

Windows 的调度器基本原理

- 是一个抢占式的、支持多处理器的优先级调度算法，它为每个处理器定义了一个链表数组，相同优先级的线程挂在同一个队列中。
- 调度程序采用 32 级的优先级确定线程执行顺序。优先级分为两大类：可变类包括优先级从 1～15 的线程（还有一个线程运行在优先级 0，它用于内存管理），实时类优先级从 16～31 的线程。
- 当一个线程满足了执行条件时，它首先被挂到当前处理器的一个待分配的队列（称为延迟的就绪链表）中，然后调度器会在适当的时候（当它获得了控制权时）把待分配队列上的线程分配到当前处理器对应优先级的线程队列中。
- Windows 中线程的优先级调整考虑到了很多因素，如前台线程，等待I/O完成后的线程也有轻微的优先级提升，这是一些来自实践经验的设计，使得 Windows 操作系统对于交互式应用程序有更好的性能表现。

Windows调度算法分析

- Windows API 定义了一个进程可能属于的一些优先级类型。它们包括：
  - real-time　　REALTIME_PRIORITY_CLASS
  - high　　HIGH_PRIORITY_CLASS
  - above normal　　ABOVE_NORMAL_PRIORITY_CLASS
  - normal　　NORMAL_PRIORITY_CLASS
  - below_normal　　BELOW_NORMAL_PRIORITY_CLASS
  - idle　　IDLE_PRIORITY_CLASS
- 进程通常属于类 NORMAL_PRIORITY_CLASS。除非进程的父进程属于类 IDLE_PRIORITY_CLASS，或者在创建进程时指定了某个类。
- 通过 Windows API 的函数 SetPriorityClass()，进程的优先级的类可以修改。除了 REALTIME_PRIORITY_CLASS外，所有其他类的优先级都是可变的。

Windows调度算法分析

- 具有给定优先级类的一个线程也有一个相对优先级。这个相对优先级的值包括：
  - IDLE
  - LOWEST
  - BELOW_NORMAL
  - NORMAL
  - ABOVE_NORMAL
  - HIGHEST
  - TIME_CRITICAL

进程优先级

线程优先级	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

- 每个线程的优先级基于它所属的进程优先级类型和它在该类型中的相对优先级，图中说明了这种关系。

Windows调度算法分析

- 每个线程在所属进程优先级类型中有一个优先级基值：
  REALTIME_PRIORITY_CLASS — 24
  HIGH_PRIORITY_CLASS — 13
  ABOVE_NORMAL_PRIORITY—CLASS — 10
  NORMALJPRIORITY_CLASS — 8
  BELOW_NORMAL_PRIORITY_CLASS — 6
  IDLE_PRIORITY_CLASS — 4
- 线程的优先级初值通常为线程所属进程的优先级基值，通过 Windows API 的函数 SetThreadPriority() 也可修改线程的优先级基值。
- 当一个线程的时间片用完时，该线程被中断。如果线程属于可变的优先级类型，那么它的优先级就被降低。不过，该优先级不能低于优先级基值。降低优先级可以限制计算密集型线程的 CPU 消耗。

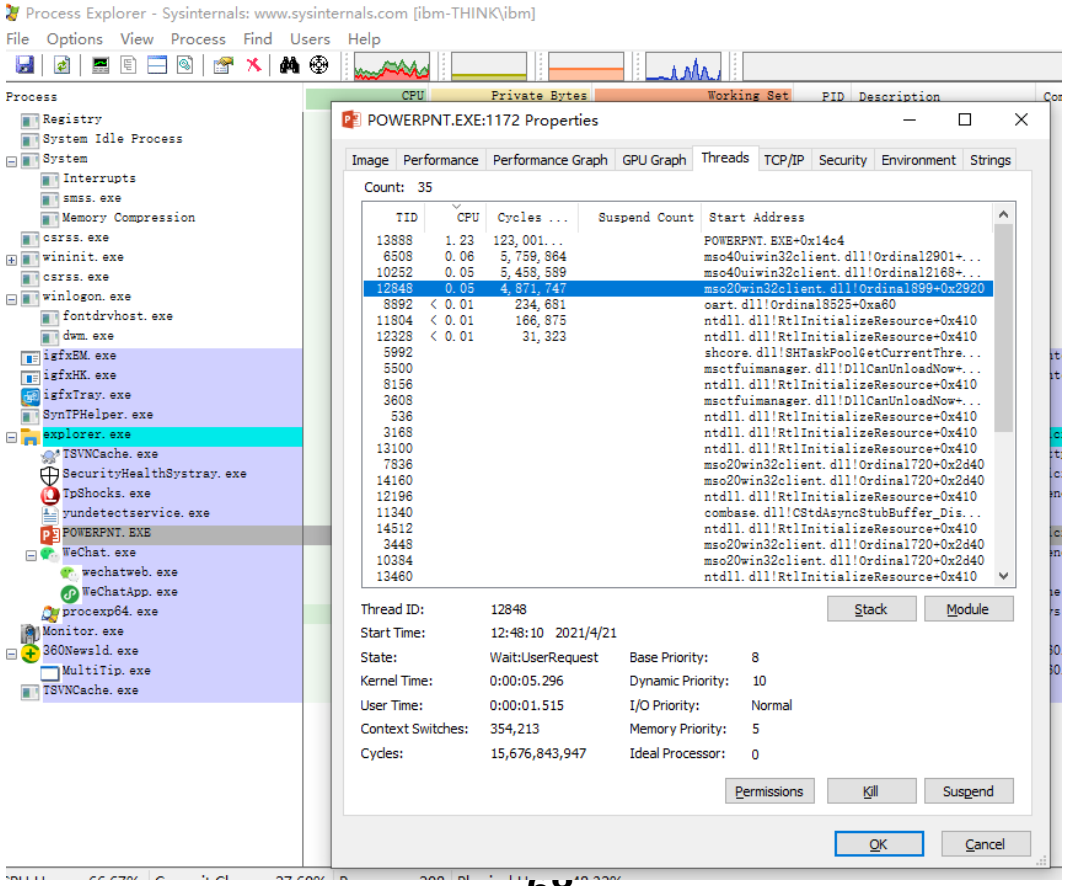
Windows调度算法分析

- 当一个可变优先级的线程从等待事件队列中释放，调度程序会提升其优先级。提升数量取决于线程等待什么。
- 当用户运行一个交互程序时，系统需要提供特别好的表现。对于类 NORMAL PRIORITY CLASS 的进程，Windows 将这类进程分成两种：前台进程，屏幕上已选的进程；后台进程，屏幕上未选的进程。
- 当进程从后台移到前台，Windows 增加它的时间片，通常是原来的 3 倍。

- 任务可以分为交互式任务和批处理任务
  - 交互式任务注重对用户的响应：如WORD
  - 批处理任务注重对任务吞吐量：如gcc
- 有趣的问题是许多系统中既有交互式任务，又有批处理任务
  - 前台任务 + 后台任务

Windows调度算法分析

- 例如，等待键盘 I/O 的线程将得到一个较高优先级提升；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。这也使得 I/O 密集型线程保持 I/O 设备忙碌，同时允许计算密集型线程更好利用 CPU 。



CPU调度的总结

- 并发能提高效率 ⇒ 并发的核心是进程能让出CPU
- 进程让出CPU ⇒ 下一个进程使用CPU ⇒ 这个选择就是调度
- 进程、线程(内核级、用户级)都能调度 ⇒ 任务调度
- 调度任务分类: 交互式，批处理
- 调度时机分类: 抢占式、非抢占式
- CPU调度算法: FCFS, SJF, Priority(批处理); RR(交互式)
- CPU调度算法: 多级队列, 多级反馈队列(混合)
- 多核调度：亲和性与处理器分组（缓存）、局部队列、动态优先级调整
- Linux和windows支持多线程、多处理器、动态优先级抢占且分时