

第一节 处理机调度概念

处理机调度是计算机操作系统中用来管理处理机执行能力的一部分资源的功能。本讲将介绍处理机调度是什么，以及处理机调度的算法：有单处理机的(只有 1 个 CPU)、实时调度算法(有更强的时间需求)、多处理机调度算法(解决多个 CPU 协调的问题)

首先介绍处理机调度的基本概念：

CPU资源的时分复用

- **进程切换：CPU资源的当前占用者切换**
 - ▣ 保存当前进程在PCB中的执行上下文(CPU状态)
 - ▣ 恢复下一个进程的执行上下文
- **处理机调度**
 - ▣ 从就绪队列中**挑选**下一个占用CPU运行的**进程**
 - ▣ 从多个可用CPU中**挑选**就绪进程可使用的**CPU资源**
- **调度程序：挑选就绪进程的内核函数**
 - ▣ **调度策略**
 - ▣ 依据什么原则挑选进程/线程？
 - ▣ **调度时机**
 - ▣ 什么时候进行调度？

进程切换即对 CPU 资源的当前占用者的切换，通过这种切换，可以实现 CPU 资源的时分复用。PCB：进程控制块。进程切换与 CPU 资源的时分复用相关，就是这里的处理机调度算法，调度算法的功能是：

- ①从就绪队列中挑选下一个占用 CPU 运行的进程
- ②如果是多处理机，还要解决从多个可用 CPU 中挑选就绪进程可使用的 CPU 资源的问题

对应处理机调度，有一个调度程序，这个程序是指，在内核当中挑选就绪进程的函数。如果是多处理机，还有挑选可用处理机的功能。

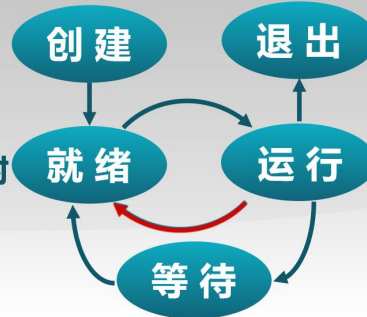
首先看调度时机：

调度时机

- 在进程/线程的生命周期中的什么时候进行调度？

调度时机

- 内核运行调度程序的条件
 - ▣ 进程从运行状态切换到等待状态
 - ▣ 进程被终结了
- 非抢占系统
 - ▣ 当前进程主动放弃CPU时
- 可抢占系统
 - ▣ 中断请求被服务例程响应完成时
 - ▣ 当前进程被抢占
 - ▣ 进程时间片用完
 - ▣ 进程从等待切换到就绪



内核运行调度程序的条件：

- ①有进程从运行状态切换到等待状态，这时把一个就绪进程放到 CPU 上
 - ②进程被终结了，即进程退出了，这时 CPU 资源空出，就又可以加载一个进程
- 以上两种情况对应非抢占系统，这种系统中操作系统不会主动剥夺进程对 CPU 的

占有。

若是抢占系统，还有两种情况：

- ①进程时间片用完，分配给它的执行时间结束，这时定时会有时钟中断，时钟中断的处理导致把当前正在运行进程重新放回就绪队列。然后再找一个新的进程来执行。
- ②有某一个处于等待状态的进程，它由等待变成就绪，这时若它更紧迫占用 CPU，就会抢占，把当前正在运行进程放回就绪队列中。

第二节 调度准则

调度策略

- 调度策略
 - ▣ 确定如何从就绪队列中选择下一个执行进程
- 调度策略要解决的问题
 - ▣ 挑选就绪队列中的哪一个进程？
 - ▣ 通过什么样的准则来选择？
- 调度算法
 - ▣ 在调度程序中实现的调度策略
- 比较调度算法的准则
 - ▣ 哪一个策略/算法较好？

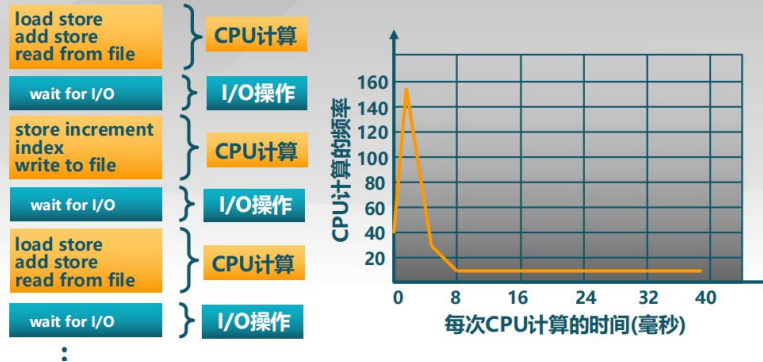
确定如何从就绪队列中选择下一个执行进程与调度目标有关。前面评价置换算法优劣的标准是在一类测试用例中，平均缺页次数少，但对调度策略来说，事情就没那么简单。

处理机资源的使用模式

■ 进程在CPU计算和I/O操作间交替

▶ 每次调度决定在下一个CPU计算时将哪个工作交给CPU

：▶ 在时间片机制下，进程可能在结束当前CPU计算前被迫放弃CPU



CPU 可指挥进行 I/O 操作，I/O 操作时，通常 CPU 处于等待状态。右图曲线为每次执行时间分配多长提供了依据。在时间片机制下，进程可能在结束当前 CPU 计算前被迫放弃 CPU，这对进程的执行是不利的。因此我们应选择一个合适的时间尺度来作为时间片的基本单位。

因此比较调度算法的准则如下：

比较调度算法的准则

■ CPU使用率

▶ CPU处于忙状态的**时间百分比**

■ 吞吐量

▶ 单位时间内完成的**进程数量**

■ 周转时间

▶ 进程从初始化到结束(包括等待)的**总时间**

■ 等待时间

▶ 进程在就绪队列中的**总时间**

■ 响应时间

▶ 从提交请求到产生响应所花费的**总时间**

CPU 使用率：CPU 处于忙状态的时间百分比，即不同调度算法在在进程执行等待 I/O 时，是否能及时找到另一个进程来占用 CPU 执行。若能，则 CPU 利用率提高。

等待时间不是指等待 I/O 的时间，这个应该注意。

吞吐量与延迟

- 调度算法的要求
 - ▣ 希望“更快”的服务
- 什么是更快？
 - ▣ 传输文件时的高带宽，调度算法的高吞吐量
 - ▣ 玩游戏时的低延迟，调度算法的低响应延迟
 - ▣ 这两个因素是独立的
- 与水管的类比
 - ▣ 低延迟：喝水的时候想要一打开水龙头水就流出来
 - ▣ 高带宽：给游泳池充水时希望从水龙头里同时流出大量的水，并且不介意是否存在延迟

①传输文件时的高带宽对应调度算法的高吞吐量(单位时间里执行更多的进程)

②玩游戏时的低延迟对应调度算法的低响应延迟(响应时间很短、很快)

这两个因素是独立的：延时低并不一定意味着高带宽，高带宽并不一定意味着低延时。

处理机调度策略的响应时间目标

- 减少响应时间
 - ▣ 及时处理用户的输入请求，尽快将输出反馈给用户
- 减少平均响应时间的波动
 - ▣ 在交互系统中，可预测性比高差异低平均更重要
- 低延迟调度改善了用户的交互体验
 - ▣ 如果移动鼠标时，屏幕中的光标没动，用户可能会重启电脑
- 响应时间是操作系统的计算延迟

平均响应时间的抖动要小，若在某些情况下响应时间很短，有些很长，这时用户体验是不稳定的。因此可预测的响应时间比高差异的低响应时间更重要。

处理机调度策略的吞吐量目标

- 增加吞吐量
 - ▣ 减少开销（操作系统开销，上下文切换）
 - ▣ 系统资源的高效利用（CPU，I/O设备）
- 减少等待时间
 - ▣ 减少每个进程的等待时间
- 操作系统需要保证吞吐量不受用户交互的影响
 - ▣ 操作系统必须不时进行调度，即使存在许多交互任务
- 吞吐量是操作系统的计算带宽

减少等待时间既可降低响应时间又可增大吞吐量。

处理机调度的公平性目标

■ 公平的定义

- ▣ 保证每个进程占用相同的CPU时间
 - ▣ 这公平么?
 - ▣ 一个用户比其他用户运行更多的进程时，怎么办?

保证每个进程占用相同的 CPU 时间不一定公平，若一用户比其他用户执行更多进程，则虽然每个进程占用 CPU 相同，但每个用户却不同。则这时：

- ▣ 保证每个进程的等待时间相同

■ 公平通常会增加平均响应时间

第三节 先来先服务、短进程优先和最高响应比优先调度算法

调度算法

■ 先来先服务算法

- ▣ FCFS: First Come, First Served

按先来后到。

■ 短进程优先算法

- ▣ SPN: Shortest Process Next
- ▣ SJF: Shortest Job First (短作业优先算法)
- ▣ SRT: Shortest Remaining Time (短剩余时间优先算法)

按作业执行时间长短排队。

■ 最高响应比优先算法

- ▣ HRRN: Highest Response Ratio Next

高响应比优先算法考虑的是进程在就绪队列里等待的时间，依据等待时间的长短来考虑。

■ 时间片轮转算法

- ▣ RR: Round Robin

让各个进程轮流占用一个基本的时间片，排到队列里时，仍按先来先服务进行排队。

■ 多级反馈队列算法

- ▣ MFQ: Multilevel Feedback Queues

把就绪队列排成多个子队列，不同子队列中可以有不同的算法，并且可以在多个队列之间调整一个进程所排的队列。

■ 公平共享调度算法

- ▣ FSS: Fair Share Scheduling

强调进程按照占用的资源的情况进行调度，保证每一个进程占用的资源相对公平。
下面具体看各种算法：

先来先服务算法(First Come First Served, FCFS)

- 依据进程进入就绪状态的先后顺序排列
 - ▣ 进程进入等待或结束状态时，就绪队列中的下一个进程占用CPU
- FCFS算法的周转时间
 - ▣ 示例：3个进程，计算时间分别为12,3,3

任务到达顺序：P₁, P₂, P₃



任务到达顺序：P₂, P₃, P₁



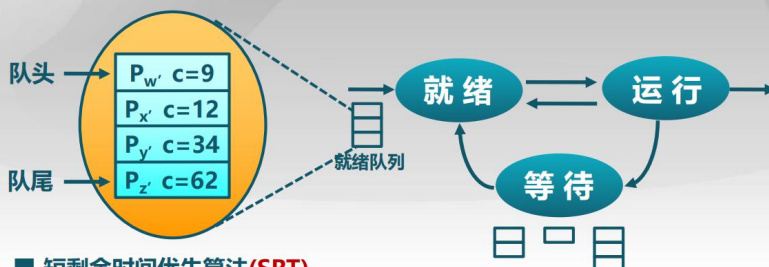
从上面可以看出，任务到达顺序为：P₁,P₂,P₃时，由于第一个进程执行时间比较长，造成后面两个等待时间较长，这样周转时间变长。若换到最下面的顺序，周转时间就短了。因此周转时间与进程到达的时间有很大的关系。

先来先服务算法的特征

- 优点
 - ▣ 简单
- 缺点
 - ▣ 平均等待时间波动较大
 - ▣ 短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

短进程优先算法(SPN)

- 选择就绪队列中执行时间最短进程占用CPU进入运行状态
 - ▣ 就绪队列按预期的执行时间来排序



- 短剩余时间优先算法(SRT)
 - ▣ SPN算法的可抢占改进

短进程优先算法考虑进程的执行时间特征。短进程优先算法有一问题：若一新进程到达，其预计时间比当前正在执行的还短怎么办？这时便有一变种，短剩余时间优先算法，若正在运行的进程的剩余时间比新来的进程要长，这时就允许新来进程抢先。

短进程优先算法具有最优平均周转时间

■ SPN算法中一组进程的平均周转时间



$$\text{周转时间} = (r_1 + r_2 + r_3 + r_4 + r_5 + r_6) / 6$$

修改进程执行顺序可能减少平均等待时间吗？



$$\begin{aligned} \text{周转时间} &= (r_1 + r_2 + r_4 - c_3 + r_5 - c_3 + r_3 + c_4 + c_5 + r_6) / 6 \\ &= (r_1 + r_2 + r_3 + r_4 + r_5 + r_6 + (c_4 + c_5 - 2c_3)) / 6 \end{aligned}$$

周转时间是进程从初始化到结束(包括等待)的总时间。 c 是进程执行所需时间。

短进程优先算法的特征：缺点

- 可能导致饥饿
 - ▣ 连续的短进程流会使长进程无法获得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？

若一台机器中，就绪进程不断产生，由于是按照由短到长的顺序排的，若短的很多，长的就会一直得不到 CPU 的资源，便导致饥饿。预测未来太难，于是利用过去预测未来，即执行时间预估，其中， α 是权重：

短进程优先算法的执行时间预估

■ 用历史的执行时间来预估未来的执行时间

```
process P
begin
  loop
    <read input from user>
    <process input>
  end loop
end P
```

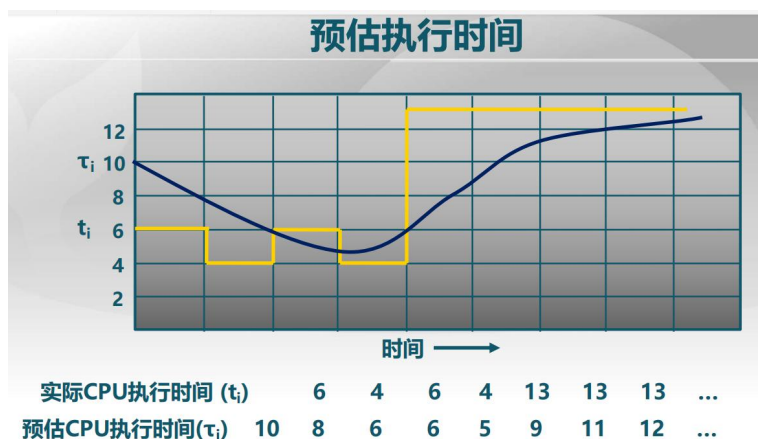
$$\tau_{n+1} = \alpha t_n + (1-\alpha) \tau_n, \text{ 其中 } 0 \leq \alpha \leq 1$$

t_n ——第 n 次的CPU计算时间

τ_{n+1} ——第 $n+1$ 次的CPU计算时间预估

$$\tau_{n+1} = \alpha t_n + (1-\alpha) \alpha t_{n-1} + (1-\alpha)(1-\alpha) \alpha t_{n-2} + \dots$$

下图是实际执行过程中，每一次占用 CPU 时，进入等待状态前的执行时间。



下面是最高响应比优先算法：

最高响应比优先算法(HRRN)

- 选择就绪队列中响应比R值最高的进程
$$R = (w + s) / s$$
 - w: 等待时间(waiting time)
 - s: 执行时间(service time)
- ▣ 在短进程优先算法的基础上改进
- ▣ 不可抢占
- ▣ 关注进程的等待时间
- ▣ 防止无限期推迟

短进程优先会因长进程前过多短进程产生而出现饥饿，这时将等待时间考虑进去就产生 HRRN 算法，防止无限期推迟(即饥饿)

第四节 时间片轮转、多级反馈队列、公平共享调度算法和 ucore 调度框架

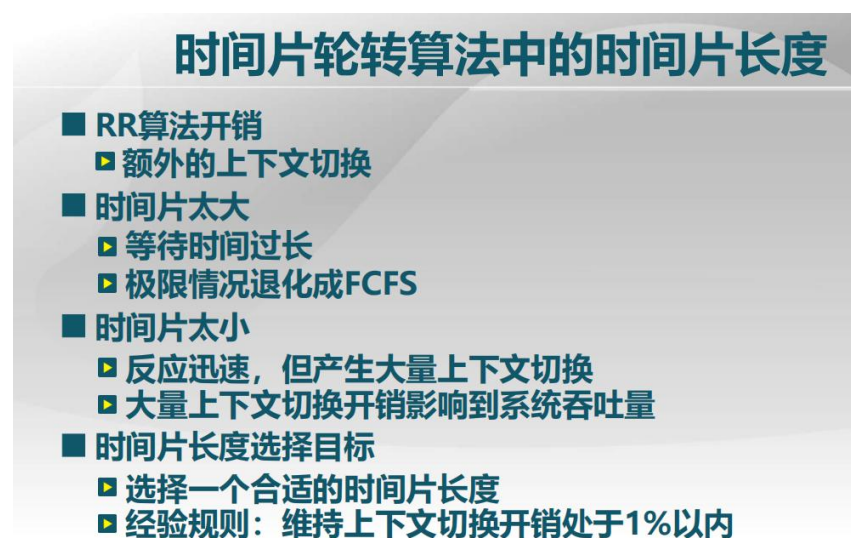


若有 n 个进程，则每隔 $n-1$ 个时间片执行一个时间片 q 。

下面是时间片算法的示例：



上面假定进程执行到达顺序为 P1,P2,P3,P4



时间片轮转算法靠时钟中断强行把正在执行的进程结束掉，所以它有额外的上下文切换开销。高吞吐量(单位时间里执行更多的进程)。10ms 左右一个时间片，可维持上下文切换开销处于 1%以内。

比较FCFS和RR

■ 示例: 4个进程的执行时间如下

P1	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

BestFCFS 假设进程到来顺序与短进程优先相同。WorstFCFS 相当于长进程优先。可见 RR 比较稳定, 而 FCFS 由于(假设)进程到来顺序不同, 对其影响很大, 抖动很大。

下面是多级队列调度算法, 前台交互要求时间片短, 用 RR, 后台计算时间长, 用 FCFS:

多级队列调度算法(MQ)

■ 就绪队列被划分成多个独立的子队列

- ▣ 如: 前台(交互)、后台(批处理)

■ 每个队列拥有自己的调度策略

- ▣ 如: 前台-RR、后台-FCFS

■ 队列间的调度

▣ 固定优先级

- ▣ 先处理前台, 然后处理后台
- ▣ 可能导致饥饿

▣ 时间片轮转

- ▣ 每个队列都得到一个确定的能够调度其进程的CPU总时间
- ▣ 如: 80%CPU时间用于前台, 20%CPU时间用于后台

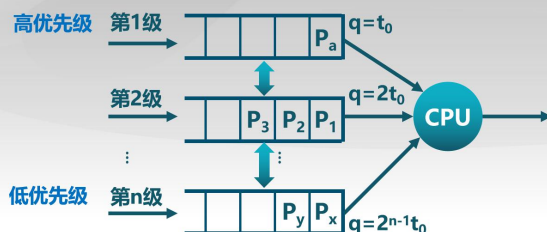
队列间的调度可以固定优先级, 但可能导致饥饿, 即后台长进程优先级低、计算时间长, 前台的交互的短进程数量很大。另一种是时间片轮转, 则前台有作业时可以得到很好的响应, 后台也不至于饥饿的状态。

多级队列算法各队列之间没有交互, 下面改进的多级反馈队列算法可实现队列间的进程交互。

多级反馈队列算法(MLFQ)

■ 进程可在不同队列间移动的多级队列算法

- ▣ 时间片大小随优先级级别增加而增加
- ▣ 如进程在当前的时间片没有完成，则降到下一个优先级



采用上图所用的优先级及时间片调整策略，得到 MLFQ 的特征，CPU 密集型的进程优先级会逐步降低，并且时间片会分的很大，这样切换的开销相对变小。而 I/O 密集型的进程，会停留在高优先级，因为每一次利用 CPU 算的时间都很短，其时间片没用完。

■ MLFQ算法的特征

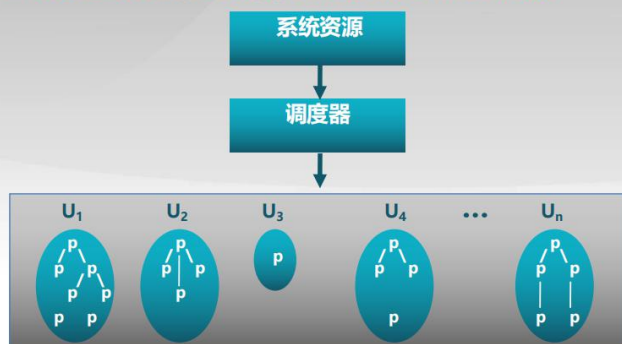
- ▣ CPU密集型进程的优先级下降很快
- ▣ I/O密集型进程停留在高优先级

在 MLFQ 算法之上，有公平共享调度算法，它强调资源访问的公平。

公平共享调度(FSS, Fair Share Scheduling)

■ FSS控制用户对系统资源的访问

- ▣ 一些用户组比其他用户组更重要
- ▣ 保证不重要的组无法垄断资源
- ▣ 未使用的资源按比例分配
- ▣ 没有达到资源使用率目标的组获得更高的优先级



重要的组中的用户分的时间更多。

传统调度算法总结

■ 先来先服务算法

- ▶ 不公平，平均等待时间较差

■ 短进程优先算法

- ▶ 不公平，平均周转时间最小
- ▶ 需要精确预测计算时间
- ▶ 可能导致饥饿

■ 最高响应比优先算法

- ▶ 基于SPN调度
- ▶ 不可抢占

最高响应比优先算法是基于短进程优先算法(SPN)改进的。

■ 时间片轮转算法

- ▶ 公平，但是平均等待时间较差

时间片轮转算法交互性很好。

■ 多级反馈队列

- ▶ 多种算法的集成

现在实际系统中用到的算法基本上都是多级反馈队列这种综合的算法，只是各个系统综合的方式不同。

■ 公平共享调度

- ▶ 公平是第一要素

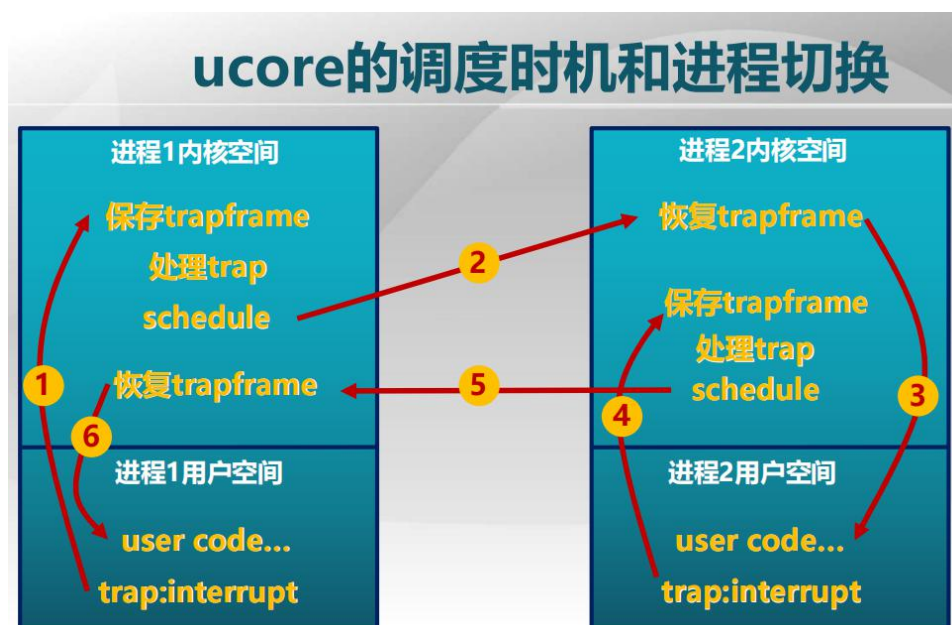
下面看，ucore 的调度队列 run_queue

```
struct run_queue {  
    list_entry_t run_list;  
    unsigned int proc_num;  
    int max_time_slice; //时间片  
    list_entry_t rq_link;  
};
```


下图是调度与进程状态的关联：



Ucore 只有两种基本状态：可执行(RUNNABLE)和等待(SLEEPING)。UNINT 是创建，ZOMBIE 是退出。



①执行过程中出现系统调用、中断或异常，这时切到内核，保存中断的现场，中断处理，调度，若还是调度到这个进程，进⑥，否则②。

②切到另一个进程，恢复中断现场

③回到用户态，执行过程中，还会出现中断，如时间片用完

④中断后进入内核态，保存中断现场，进行中断处理，然后调度，假设调度后切回上回执行进程。

⑤恢复现场。

⑥在用户态继续执行。

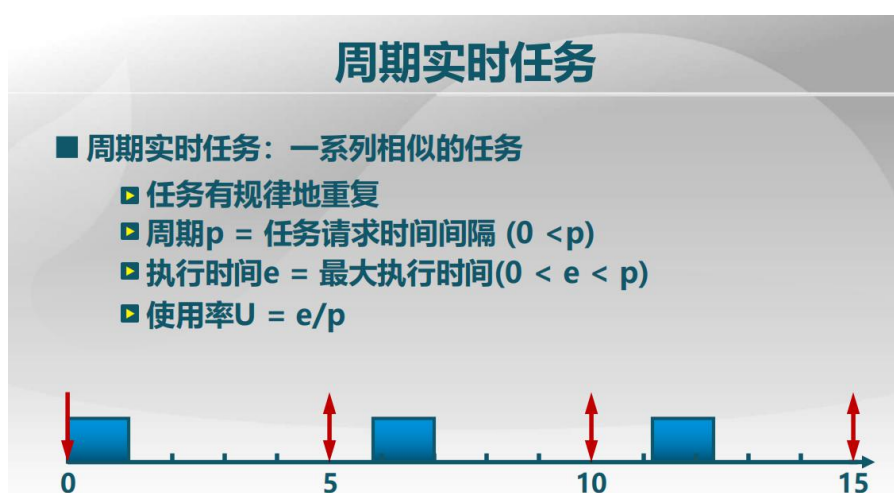
第五节 实时调度和多处理器调度

下面讨论实时和多处理机调度。实时调度是对时间有要求的调度算法。多处理器调度是在有多个处理器的系统里的调度算法。

实时操作系统

- 实时操作系统的定义
 - ▣ 正确性依赖于其**时间**和**功能**两方面的操作系统
- 实时操作系统的性能指标
 - ▣ **时间约束的及时性 (deadlines)**
 - ▣ 速度和平均性能相对不重要
- 实时操作系统的特性
 - ▣ 时间约束的**可预测性**

实时操作系统必须在约定的时间内完成约定的工作(时间约束的及时性), 最显著的特征是: 要求时间约束的可预测性, 即必须知道在什么情况下, 这些时间约束是能达到的。



使用率 U 若为 100%, 则很难保持实时性。

软时限和硬时限

■ 硬时限 (Hard deadline)

- ▣ 错过任务时限会导致灾难性或非常严重的后果
- ▣ 必须验证，在最坏情况下能够满足时限

■ 软时限(Soft deadline)

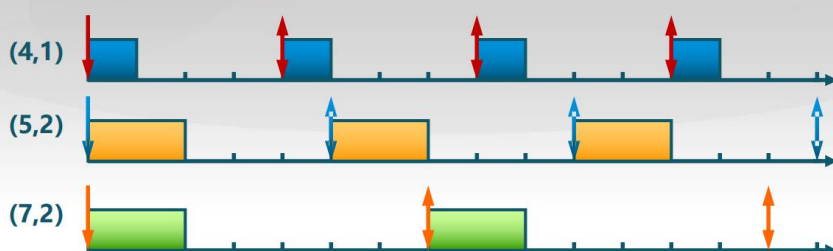
- ▣ 通常能满足任务时限
 - ▣ 如有时不能满足，则降低要求
- ▣ 尽力保证满足任务时限

硬时限又称硬实时，软时限又称软实时。

可调度性

■ 可调度表示一个实时操作系统能够满足任务时限要求

- ▣ 需要确定实时任务的执行顺序
- ▣ 静态优先级调度
- ▣ 动态优先级调度



上图中(4,1)表示出现频率和其执行时间，下面两个意思类似。

可调度表示一个实时操作系统能够满足任务时限要求，需要确定实时任务的执行顺序。

调度算法有两类：

- ①静态优先级调度：事先把执行顺序排出来调度即可
- ②动态优先级调度：无法事先给出执行顺序，执行的过程中确定执行顺序。

实时调度

■ 速率单调调度算法(RM, Rate Monotonic)

- ▣ 通过**周期**安排优先级
- ▣ 周期越短优先级越高
- ▣ 执行周期最短的任务

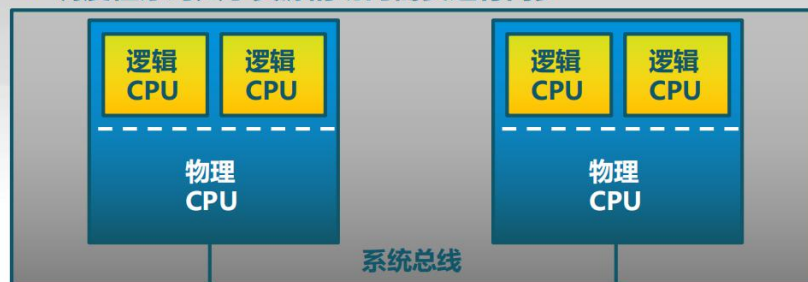
■ 最早截止时间优先算法 (EDF, Earliest Deadline First)

- ▣ 截止时间越早优先级越高
- ▣ 执行截止时间最早的任务

RM 是静态算法，频率越高，周期越短优先级越高。其难点在于，系统执行多少任务时是可调度的。EDF 是动态算法，与 RM 面对同样的难点。

多处理器调度

- 多处理机调度的特征
 - ▣ 多个处理机组成一个多处理机系统
 - ▣ 处理机间可负载共享
- 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



上图是多处理器调度。一条系统总线上连接了多个物理 CPU，每个 CPU 中可能有多个逻辑的核。

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

动态进程分配，对就绪队列的访问需要同步，开销就大。在实际系统中，动态和静态进程分配都有采用。

第六节 优先级反置

优先级反置(Priority Inversion)

- 操作系统中出现高优先级进程长时间等待低优先级进程所占用资源的现象
- 基于优先级的可抢占调度算法存在优先级反置



上图中下方应是：T3 的长时间运行导致高优先级的 T2 进行长时间等待。

上图中，T1 和 T2 需要占用临界资源，优先级方面：T1<T3<T2

优先级反置(Priority Inversion)指操作系统中出现高优先级进程长时间等待低优先级进程所占用资源而导致高优先级进程长期等待的现象。

优先级倒置，又称优先级反转、优先级逆转、优先级翻转，是一种不希望发生的任务调度状态。在该种状态下，一个高优先级任务间接被一个低优先级任务所抢先(preempted)，使得两个任务的相对优先级被倒置。

这往往出现在一个高优先级任务等待访问一个被低优先级任务正在使用的临界资源，从而阻塞了高优先级任务；同时，该低优先级任务被一个次高优先级的任务所抢先，从而无法及时地释放该临界资源。这种情况下，该次高优先级任务获得执行权。

在多数个案，发生优先级倒置并不导致直接伤害——高优先级任务的延迟运行不被察觉，最终，低优先级任务释放共享资源。虽然，亦存在很多情况优先级倒置会导致严重问题。

举例：

在操作系统中，一般情况下：

1.进程分优先级，高优先级进程需要执行时可打断现正在执行的低优先级进程。

2.普通的临界资源使用方法:如果一个临界资源被获取了，则其它想要获取此资源的程序被阻塞，直到此资源被释放。

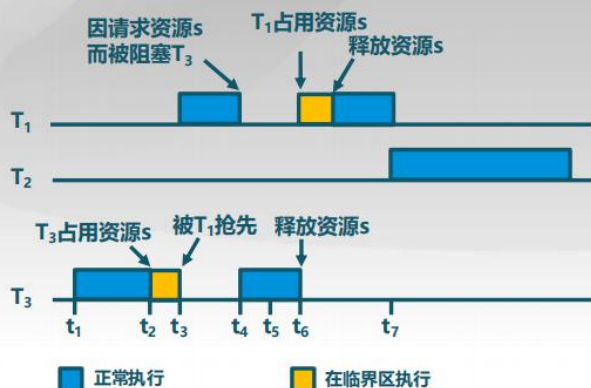
3.有三个进程（其优先级从高到低分别为 T1、T2、T3），有一个临界资源 CS（T1 与 T3 会用到）。这时，T3 先执行，获取了临界资源 CS。然后 T2 打断 T3。接着 T1 打断 T2，但由于 CS 已被 T3 获取，因此 T1 被阻塞，这样 T2 获得时间片。直到 T2 执行完毕后，T3 接着执行，其释放 CS 后，T1 才能获取 CS 并执行。这时，我们看 T1 与 T2，虽然 T1 优先级比 T2 高，但实际上 T2 优先于 T1 执行。这称之为优先级逆转。

优先级反置的处理方法有两种，第一种是优先级继承：

优先级继承 (Priority Inheritance)

■ 占用资源的低优先级进程继承申请资源的高优先级进程的优先级

▣ 只在占有资源的低优先级进程被阻塞时,才提高占有资源进程的优先级



上图中, 用线的位置表示优先级, $T_3 < T_2 < T_1$, 首先 T_3 进入执行状态, T_3 在 t_2 时刻占用资源 s 在临界区进行执行, 在临界区执行过程当中, 这时有一个优先级比它高的进程 T_1 进入执行状态, 这时 T_3 被抢先, T_1 开始执行, 在 T_1 执行的过程中, T_1 申请资源 s , T_1 的申请导致 T_3 的优先级提升, 提升后, T_3 继续执行, T_3 执行到释放资源 s , 这时 T_1 就可以占用资源继续其临界区执行, 然后释放资源执行。这时如果有 T_2 出现, 那么必须等到 T_1 结束之后才能占用 CPU 来继续执行。注意: 只在占有资源的低优先级进程被阻塞(抢占)时,才提高占有资源进程的优先级。

另一种做法是优先级天花板协议:

优先级天花板协议 (priority ceiling protocol)

■ 占用资源进程的优先级和所有可能申请该资源的进程的最高优先级相同

▣ 不管是否发生等待,都提升占用资源进程的优先级

▣ 优先级高于系统中所有被锁定的资源的优先级上限, 任务执行临界区时就不会被阻塞

这种做法会有优先级滥用的情况, 即各进程都把自己的优先级提得很高, 如果所有都提到最高, 那实际上没什么意义。