# 锁机制

在多线程编程中，操作系统引入了锁机制。通过锁机制，能够**保证在多核多线程环境中，在某一个时间点上，只能有一个线程进入临界区代码，从而保证临界区中操作数据的一致性**。

## 死锁

是指**两个或两个以上的进程在执行过程中,因争夺资源而造成的一种互相等待的现象**,**若无外力作用,它们都将无法推进下去**。此时称系统处于死锁状态或系统产生了死锁,这些永远在互相等待的进程称为死锁进程。

## 产生死锁的原因

1. 因为**系统资源不足**。
2. 进程运行**推进的顺序不当**。
3. **资源分配不当**等。

如果系统资源充足，进程的资源请求都能够得到满足，死锁出现的可能性就很低，否则

就会因**争夺有限的资源而陷入死锁**。其次，**进程运行推进顺序与速度不同**，也可能产生死锁。

## 死锁的四个必要条件

1. **互斥条件**：一个资源每次只能被一个进程使用。

2. **请求与保持条件**：一个进程因请求资源而阻塞时，其它线程对已获得的资源保持不放。

3. **不剥夺条件**:进程已获得的资源，在末使用完之前，不能强行剥夺。

4. **循环等待条件**:若干进程之间形成一种头尾相接的循环等待资源关系。

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之

一不满足，就不会发生死锁。

## 死锁的解除与预防

<http://blog.csdn.net/abigale1011/article/details/6450845/>

理解了死锁的原因，尤其是产生死锁的四个必要条件，就可以最大可能地避免、预防和解除死锁。所以，在系统设计、进程调度等方面**注意如何不让这四个必要条件成立**，如何**确定资源的合理分配算法**，避免进程永久占据系统资源。此外，也要防止进程在处于等待状态

的情况下占用资源。因此，对资源的分配要给予合理的规划。

如何避免死锁：

<http://mp.weixin.qq.com/s?__biz=MjM5NzMyMjAwMA==&mid=2651479577&idx=2&sn=448e59a49ccc9dacff60e0dbd5ced113&chksm=bd2532668a52bb702e253175868aa479994aaa09363ed5cbf192cb00b4489c62b7c6d999a207&mpshare=1&scene=23&srcid=1207UAXOL8zDTAM9ln48x2Tr#rd>

### 加锁顺序

当多个线程需要相同的一些锁，但是按照不同的顺序加锁，死锁就很容易发生。

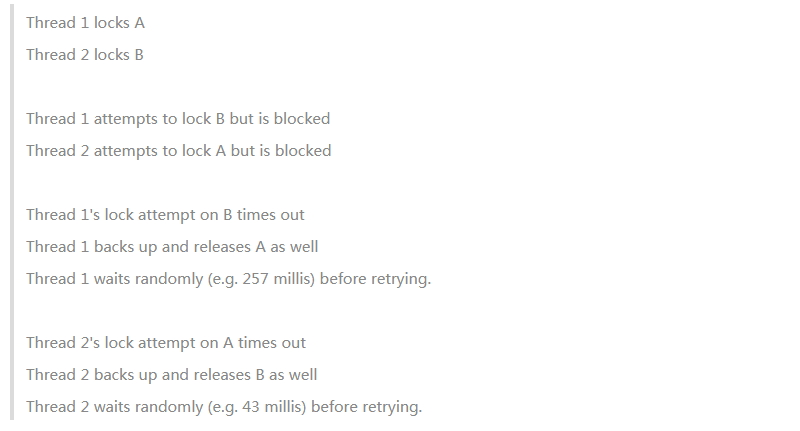
如果能确保所有的线程都是按照相同的顺序获得锁，那么死锁就不会发生。

按照顺序加锁是一种有效的死锁预防机制。但是，这种方式需要你事先知道所有可能会用到的锁(并对这些锁做适当的排序)，但总有些时候是无法预知的。



### 加锁时限

在尝试获取锁的时候加一个超时时间，这也就意味着在尝试获取锁的过程中若超过了这个时限该线程则放弃对该锁请求。若一个线程没有在给定的时限内成功获得所有需要的锁，则会进行回退并释放所有已经获得的锁，然后等待一段随机的时间再重试。这段随机的等待时间让其它线程有机会尝试获取相同的这些锁，并且让该应用在没有获得锁的时候可以继续运行(加锁超时后可以先继续运行干点其它事情，再回头来重复之前加锁的逻辑)。



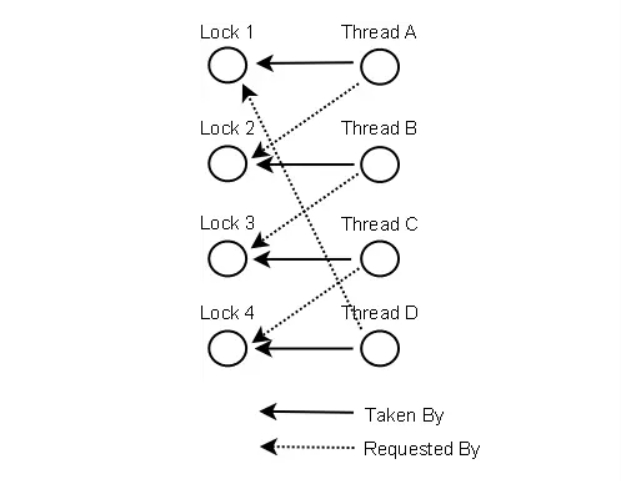
超时和重试机制是为了避免在同一时间出现的竞争，但是当线程很多时，其中两个或多个线程的超时时间一样或者接近的可能性就会很大，因此就算出现竞争而导致超时后，由于超时时间一样，它们又会同时开始重试，导致新一轮的竞争，带来了新的问题。

### 死锁检测

死锁检测是一个更好的死锁预防机制，它主要是针对那些不可能实现按序加锁并且锁超时也不可行的场景。

每当一个线程获得了锁，会在线程和锁相关的数据结构中（map、graph等等）将其记下。除此之外，每当有线程请求锁，也需要记录在这个数据结构中。

**当一个线程请求锁失败时，这个线程可以遍历锁的关系图看看是否有死锁发生**。



给这些线程设置优先级，让一个（或几个）线程回退，剩下的线程就像没发生死锁一样继续保持着它们需要的锁。如果赋予这些线程的优先级是固定不变的，同一批线程总是会拥有更高的优先级。为避免这个问题，可以在死锁发生的时候设置随机的优先级。

# 线程调度

在操作系统中，带权周转时间反映作业（或进程）长短问题，带权周转时间越大，作业（或进程）越短；带权周转时间越小，作业（或进程）越长。

**W=T/Ts：** //T：为周转时间，Ts为服务时间（运行时间）

## 先来先服务（FCFS）

FCFS调度算法是一种最简单的调度算法，该调度算法既可以用于作业调度也可以用于进程调度。在作业调度中，算法每次从后备作业队列中选择最先进入该队列的一个或几个作业，将它们调入内存，分配必要的资源，创建进程并放入就绪队列。

在进程调度中，FCFS调度算法每次从就绪队列中选择**最先进入该队列的进程**，将处理机分配给它，使之投入运行，直到完成或因某种原因而阻塞时才释放处理机。



FCFS调度算法属于不可剥夺算法。从表面上看，它对所有作业都是公平的，但若一个长作业先到达系统，就会使后面许多短作业等待很长时间，FCFS调度算法的特点是**算法简单，但效率低；对长作业比较有利，但对短作业不利**（相对SJF和高响应比）；有利于CPU繁忙型作业，而不利于I/O繁忙型作业。

## 短作业优先(SJF)

短作业（进程）优先调度算法是指对短作业（进程）优先调度的算法。短作业优先(SJF)调度算法是从后备队列中选择一个或若干个估计运行时间最短的作业，将它们调入内存运行。而短进程优先(SPF)调度算法，则是从就绪队列中选择一个**估计运行时间最短的进程**，将处理机分配给它，使之立即执行，直到完成或发生某事件而阻塞时，才释放处理机。



1. 该算法对长作业不利，如果有一长作业进入系统的后备队列，由于调度程序总是优先调度那些 (即使是后进来的）短作业，将导致长作业长期不被调度（“饥饿”现象，注意区分“死锁”。后者是系统环形等待，前者是调度策略问题）。
2. 该算法完全未考虑作业的紧迫程度，因而不能保证紧迫性作业会被及时处理。
3. 由于作业的长短只是根据用户所提供的估计执行时间而定的，而用户又可能会有意或无意地缩短其作业的估计运行时间，致使该算法不一定能真正做到短作业优先调度。

注意，SJF调度算法的平均等待时间、平均周转时间最少。

## 优先级调度算法

优先级调度算法又称优先权调度算法，该算法既可以用于作业调度，也可以用于进程调度，该算法中的优先级用于描述作业运行的紧迫程度。

在作业调度中，优先级调度算法每次从后备作业队列中选择优先级最髙的一个或几个作业，将它们调入内存，分配必要的资源，创建进程并放入就绪队列。在进程调度中，优先级调度算法**每次从就绪队列中选择优先级最高的进程**，将处理机分配给它，使之投入运行。

该调度算法分为：

1. 非剥夺式优先级调度算法。当某一个进程正在处理机上运行时，即使有某个更为重要或紧迫的进程进入就绪队列，仍然让正在运行的进程继续运行，直到由于其自身的原因而主动让出处理机时（任务完成或等待事件），才把处理机分配给更为重要或紧迫的进程。
2. 剥夺式优先级调度算法。当一个进程正在处理机上运行时，若有某个更为重要或紧迫的进程进入就绪队列，则立即暂停正在运行的进程，将处理机分配给更重要或紧迫的进程。

## 时间片轮转

时间片轮转调度算法主要适用于分时系统。在这种算法中，系统将所有就绪进程按到达时间的先后次序排成一个队列，进程调度程序总是选择就绪队列中第一个进程执行，即先来先服务的原则，但仅能运行一个时间片，如100ms。在使用完一个时间片后，即使进程并未完成其运行，它也必须释放出（被剥夺）处理机给下一个就绪的进程，而被剥夺的进程返回到就绪队列的末尾重新排队，等候再次运行。

时间片的长短通常由以下因素确定：系统的响应时间、就绪队列中的进程数目和系统的处理能力。

# 页面置换算法

地址映射过程中，若在页面中发现所要访问的页面不在内存中，则产生缺页中断。当发生缺页中断时，如果操作系统内存中没有空闲页面，则操作系统必须在内存选择一个页面将其移出内存，以便为即将调入的页面让出空间。而用来选择淘汰哪一页的规则叫做页面置换算法。

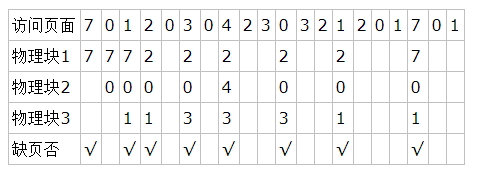
假定系统为某进程分配了**三个物理块**，并考虑有以下页面号引用串：

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

## 最佳置换算法（OPT）（理想置换算法）

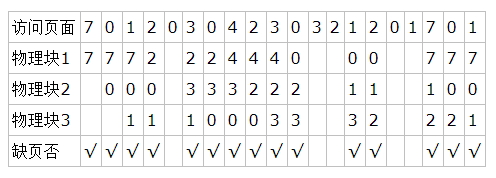
从主存中移出永远不再需要的页面；如无这样的页面存在，则选择最长时间不需要访问的页面。即选择的被淘汰页面将是以后永不使用的，或者是在最长时间内不再被访问的页面，这样可以保证获得最低的缺页率。

可以看到，发生缺页中断的次数为9，页面置换的次数为6。



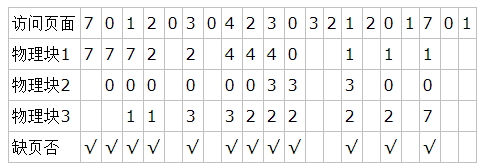
## 先进先出置换算法（FIFO）

是最简单的页面置换算法。这种算法的基本思想是：当需要淘汰一个页面时，**总是选择驻留主存时间最长的页面进行淘汰**，即先进入主存的页面先淘汰。其理由是：最早调入主存的页面不再被使用的可能性最大。



## 最近最久未使用（LRU）算法

利用局部性原理，根据一个作业在执行过程中过去的页面访问历史来推测未来的行为。它认为过去一段时间里不曾被访问过的页面，在最近的将来可能也不会再被访问。所以，这种算法的实质是：当需要淘汰一个页面时，总是选择**在最近一段时间内最久不用的页面予以淘汰**。



LRU算法根据各页以前的情况，是“向前看”的，而最佳置换算法则根据各页以后的使用情况，是“向后看”的。

# 银行家算法

每个进程所能获取的每种资源数量是多少[MAX]

每个进程当前所分配到的每种资源的数量是多少[ALLOCATED]

系统当前可分配的每种的资源数量是多少[AVAILABLE]

只有当资源满足以下条件，资源才会被分配：

1. request <= max, 也可设置错误条件，当进程所请求的资源超过最大的要求
2. request <= available, 或者进程一直等直到资源可分配

银行家算法名字源于该算法实际上是用于确保银行系统不会用尽系统资源，因为当银行系统不再满足所有客户的需求，系统将不会分配钱（看作资源）给客户，银行必须确保对钱的请求不会导致银行系统处于不安全状态。如果上述情况不会发生，则该情况下请求是被允许的，否则，客户必须等到其他客户往银行存进足够银行分配的资金。