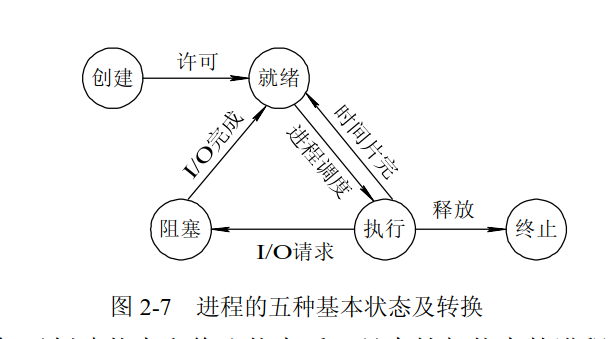
## 进程的有哪几种状态，状态转换图，及导致转换的事件

* 状态转换图



* 状态：

（1）**就绪状态**：进程已获得除处理机CPU外的所需资源，等待分配处理机资源，只要分配到CPU就可执行。在某一时刻，可能有若干个进程处于该状态。

（2）**运行状态**：占用处理机资源运行，处于此状态的进程的数目小于等于CPU的数目。

（3）**阻塞状态**：由于进程等待某种条件（如I/O操作或进程同步），在条件满足之前无法继续执行，只要主动放弃CPU，将自己阻塞起来，等待所需的时间发生。该事件发生前即使把处理机分配给该进程，也无法运行。

* 转换解释及导致转换的时间：从状态转换图中，存在四种状态转换。

|  |  |  |
| --- | --- | --- |
| 状态转换 | 解释 | 导致状态转换的事件 |
| 就绪🡪运行 | 进程调度程序按照某种算法，从就绪队列中选取一个进程投入运行 | 进程调度 |
| 运行—>就绪 | 正在执行的进程如因时间片用完或被强占而被暂停执行 | 时间片到了；被高优先级进程抢占 |
| 运行🡪阻塞 | 正在执行的进程因等待的事件尚未发生而无法执行 | 进程请求I/O；同步信号；数据 |
| 阻塞🡪就绪 | 当进程等待的事件发生时 | I/O完成；同步信号到达；数据到达 |

## 进程与线程的区别

* 基本概念

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位.

线程是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位.线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源.

* 进程和线程的区别：

1、进程是资源分配的基本单位，线程是cpu调度，或者说是程序执行的最小单位。因此，实现并发功能的单位是线程。

　 2、进程有独立的地址空间（比如在linux下面启动一个新的进程，系统必须分配给它独立的地址空间，建立众多的数据表来维护它的代码段、堆栈段和数据段，这是一种非常昂贵的多任务工作方式）。而运行一个进程中的线程，它们之间共享大部分数据，使用相同的地址空间，因此启动一个线程，切换一个线程远比进程操作要快，花费也要小得多。当然，线程是拥有自己的局部变量和临时栈区。

　　3、线程之间的通信比较方便。同一个进程下的线程共享数据（比如全局变量，静态变量），通过这些数据来通信不仅快捷而且方便，当然如何处理好这些访问的同步与互斥正是编写多线程程序的难点。而进程之间的通信只能通过[进程通信](http://baike.baidu.com/view/549640.htm)的方式进行。

4、进程是死的，只是一些资源的集合，真正的程序执行都是线程来完成的，程序启动的时候[**操作系统**](http://lib.csdn.net/base/operatingsystem)就帮你创建了一个主线程。

## 3、Java中内存存在各种变量

|  |  |  |
| --- | --- | --- |
| 内存区域 | 存放的内容 | 其他说明 |
| 寄存器 | 程序中无法控制 |  |
| 栈 | 存放基本类型的数据和对象的引用，但是对象本身放在堆中（new 出来的对象） | 1. 栈位于通用RAM(随机访问存储器)中，栈由编译器自动分配和释放。 2. 栈的优势是存取速度比堆快，仅次于CPU中的寄存器。 3. 栈还有一个很大的特点就是栈中的数据可以共享 |
| 堆 | 存放new出来的数据 | 也位于RAM中 |
| 静态区 | 存放使用static修饰的静态变量 |  |
| 常量区 | 1. 包含基本数据类型如int、long等以final声明的常量值‘ 2. String字符串 | 特别注意的是对于方法运行期位于栈中的局部变量String常量的值可以通过 String.intern()方法将该值置入到常量池中。 |

解释1：

比如定义两个int类型的变量：int a = 3; int b = 3;这里a和b是一个指向int型的引用，指向"3"这个字面值。编译器先处理int a = 3;这句语句的时候，先在栈中创建一个变量为a的引用，然后查找有没字面值为3的地址，如果没有就开辟出一个存储3的地址，然后将a指向这个3对应的地址。接着处理int b = 3;，也是先创建一个变量b的引用，由于栈中已经有字面量3了，于是就把b也指向3对应的这个地址，所以a和b都指向了一个地址。当我们执行 b = 4;的时候，首先还是去栈中查找有没字面量值为4对应的地址，如果没有就开辟个，然后将b指向这个新开辟的地址。如果已经有了就直接将b指向这个地址，此时a还是指向3，但b指向4了，而且他俩不再指向同一个地址了。实际上形参，局部变量都存储在栈中。

示例网址：http://blog.csdn.net/xingzhemoluo/article/details/40050025

## 4、[操作系统常用调度算法](http://www.cnblogs.com/kxdblog/p/4798401.html)

在操作系统中存在多种调度算法，其中有的调度算法适用于作业调度，有的调度算法适用于进程调度，有的调度算法两者都适用。下面介绍几种常用的调度算法。（周转时间 = 程序结束时间--开始服务时间、带权周转时间 = 周转时间/要求服务时间）

* 先来先服务(FCFS)调度算法

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。  
 下面通过一个实例来说明FCFS调度算法的性能。假设系统中有4个作业，它们的提交时间分别是8、8.4、8.8、9，运行时间依次是2、1、0.5、0.2，系统釆用FCFS调度算法，这组作业的平均等待时间、平均周转时间和平均带权周转时间见表2-3。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 表2-3 FCFS调度算法的性能 | | | | | | | |
| **作业号** | **提交时间** | **运行时间** | **开始时间** | **等待时间** | **完成时间** | **周转时间** | **带权周转时间** |
| 1 | 8 | 2 | 8 | 0 | 10 | 2 | 1 |
| 2 | 8.4 | 1 | 10 | 1.6 | 11 | 2.6 | 2.6 |
| 3 | 8.8 | 0.5 | 11 | 2.2 | 11.5 | 2.7 | 5.4 |
| 4 | 9 | 0.2 | 11.5 | 2.5 | 11.7 | 2.7 | 13.5 |

平均等待时间 t = (0+1.6+2.2+2.5)/4=1.575  
平均周转时间 T = (2+2.6+2.7+2.7)/4=2.5  
平均带权周转时间 W = (1+2.6+5.4+13.5)/4=5.625  
  
 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，因此它不能作为分时系统和实时系统的主要调度策略。但它常被结合在其他调度策略中使用。例如，在使用优先级作为调度策略的系统中，往往对多个具有相同优先级的进程按FCFS原则处理。  
  
 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/SPF)调度算法

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。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 表2-4 SJF调度算法的性能 | | | | | | | |
| **作业号** | **提交时间** | **运行时间** | **开始时间** | **等待时间** | **完成时间** | **周转时间** | **带权周转时间** |
| 1 | 8 | 2 | 8 | 0 | 10 | 2 | 1 |
| 2 | 8,4 | 1 | 10.7 | 2.3 | 11.7 | 3.3 | 3.3 |
| 3 | 8.8 | 0.5 | 10.2 | 1.4 | 10.7 | 1.9 | 3.8 |
| 4 | 9 | 0.2 | 10 | 1 | 10.2 | 1.2 | 6 |

平均等待时间 t = (0+2.3+1.4+1)/4=1.175  
平均周转时间 T = (2+3.3+1.9+1.2)/4=2.1  
平均带权周转时间 W = (1+3.3+3.8+6)/4=3.525  
  
SJF调度算法也存在不容忽视的缺点：

该算法对长作业不利，由表2-3和表2-4可知，SJF调度算法中长作业的周转时间会增加。更严重的是，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）。

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。

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。  
  
根据新的更高优先级进程能否抢占正在执行的进程，可将该调度算法分为：

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。

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。

而根据进程创建后其优先级是否可以改变，可以将进程优先级分为以下两种：

静态优先级。优先级是在创建进程时确定的，且在进程的整个运行期间保持不变。确定静态优先级的主要依据有进程类型、进程对资源的要求、用户要求。

动态优先级。在进程运行过程中，根据进程情况的变化动态调整优先级。动态调整优先级的主要依据为进程占有CPU时间的长短、就绪进程等待CPU时间的长短。

* 高响应比优先（HRN）调度算法

高响应比优先调度算法主要用于作业调度，该算法是对FCFS调度算法和SJF调度算法的一种综合平衡，同时考虑每个作业的等待时间和估计的运行时间。在每次进行作业调度时，先计算后备作业队列中每个作业的响应比，从中选出响应比最高的作业投入运行。  
  
响应比的变化规律可描述为：（等待时间 + 运行时间） / 运行时间  
  
  
根据公式可知：

当作业的等待时间相同时，则要求运行时间越短，其响应比越高，有利于短作业。

当要求服务时间相同时，作业的响应比由其等待时间决定，等待时间越长，其响应比越高，因而它实现的是先来先服务。

对于长作业，作业的响应比可以随等待时间的增加而提高，当其等待时间足够长时，其响应比便可升到很高，从而也可获得处理机。克服了饥饿状态，兼顾了长作业。

* 时间片轮转调度算法

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。  
  
 在时间片轮转调度算法中，时间片的大小对系统性能的影响很大。如果时间片足够大，以至于所有进程都能在一个时间片内执行完毕，则时间片轮转调度算法就退化为先来先服务调度算法。如果时间片很小，那么处理机将在进程间过于频繁切换，使处理机的开销增大，而真正用于运行用户进程的时间将减少。因此时间片的大小应选择适当。  
  
 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。

* 多级反馈队列调度算法（集合了前几种算法的优点）

多级反馈队列调度算法是时间片轮转调度算法和优先级调度算法的综合和发展，如图2-5 所示。通过动态调整进程优先级和时间片大小，多级反馈队列调度算法可以兼顾多方面的系统目标。例如，为提高系统吞吐量和缩短平均周转时间而照顾短进程；为获得较好的I/O设备利用率和缩短响应时间而照顾I/O型进程；同时，也不必事先估计进程的执行时间。算法思想如下：

1. 应设置多个就绪队列，并为各个队列赋予不同的优先级，第1级队列的优先级最高，第2级队列次之，其余队列的优先级逐次降低。
2. 赋予各个队列中进程执行时间片的大小也各不相同，在优先级越高的队列中，每个进程的运行时间片就越小。例如，第2级队列的时间片要比第1级队列的时间片长一倍， ……第i+1级队列的时间片要比第i级队列的时间片长一倍。（都是倍数关系）
3. 当一个新进程进入内存后，首先将它放入第1级队列的末尾，按FCFS原则排队等待调度。当轮到该进程执行时，如它能在该时间片内完成，便可准备撤离系统；如果它在一个时间片结束时尚未完成，调度程序便将该进程转入第2级队列的末尾，再同样地按FCFS 原则等待调度执行；如果它在第2级队列中运行一个时间片后仍未完成，再以同样的方法放入第3级队列……如此下去，当一个长进程从第1级队列依次降到第 n 级队列后，在第 n 级队列中便釆用时间片轮转的方式运行。
4. 仅当第1级队列为空时，调度程序才调度第2级队列中的进程运行；仅当第1至(i-1)级队列均为空时，才会调度第i级队列中的进程运行。如果处理机正在执行第i级队列中的某进程时，又有新进程进入优先级较高的队列（第 1至(i-1)中的任何一个队列），则此时新进程将抢占正在运行进程的处理机，即由调度程序把正在运行的进程放回到第i级队列的末尾，把处理机分配给新到的更高优先级的进程。

多级反馈队列的优势有：

* + 终端型作业用户：短作业优先。
  + 短批处理作业用户：周转时间较短。
  + 长批处理作业用户：经过前面几个队列得到部分执行，不会长期得不到处理。

## 5、进程间通信（IPC）的方式

|  |  |
| --- | --- |
| 通信方式 | 解释 |
| 管道 | 管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。 |
| 有名管道 | 有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。 |
| 信号量 | 信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。 |
| 消息队列 | 消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。 |
| 信号 | 信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。 |
| 共享内存 | 共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的IPC方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号量，配合使用，来实现进程间的同步和通信。 |
| 套接字 | 套解字也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同级其间的进程通信。 |

## 5、进程同步方式的代码 (一定要会写生产者、消费者问题，完全消化理解)

临界资源：是一种一次只能为一个进程服务的资源，独占资源。（互斥访问、不产生死锁、不能有饥饿进程）

临界区（CCriticalSection）：是进程中访问临界资源的程序段,每个使用该资源的进程都要包含一个临界区。

通过对多进程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。

进程（线程）同步常用的方法：互斥锁、条件变量、信号量。

事件（CEvent）：为协调共同对一个共享资源的单独访问而设计的。

互斥量（CMutex）：为控制一个具有有限数量用户资源而设计。

信号量（CSemaphore）：用来通知线程有一些事件已发生，从而启动后继任务的开始。

  生产者、消费者问题

    Var Mutex,Sin,Sout : semaphore : = 1,n,0;  // 定义三个信号量

    Var in,out : integer : = 0,0;

    Parbegin

        process proceducer // 生产者

            begin

While(true) do

begin

                    生产一个产品;

                    P(Sin);   //先同步后互斥

                    P(Mutex);  //申请缓冲池的使用权

                    Buffer(in):= nextp;  // 将产品放入缓冲池中

                    In : = (in+1) mod n;  // 下一个空缓冲区地址

                    V(Mutex);   //释放缓冲池使用权

                    P(Sout);   // 释放一个满缓冲区

                  end;

                end

         process consumer //消费者

begin

while(true) do

                 begin

                    P(Sout);

                    P(Mutex);

                    Nextc := Buffer(out);

                    Out := (out+1) mod n;

                    V(mutex);

                    V(Sin);

                    消费一个数据；

                  end

               end

        parend

解释：

Mutex:对缓冲区实现进程之间（生产者和消费者）的互斥访问。

Sin: 表示当前缓冲区中空闲的空间大小，初始值为N

Sout:表示当前缓冲区中数据的个数，初始值为0

in 和 out分别为存入数据和读取数据时使用的数组下标。

信号量：只支持两种操作,P()和V()，也叫做P、V操作，这两个操作是原子操作，不会被打断。

还有读者—写者问题

## 7、线程的实现方式. (也就是用户线程与内核线程的区别)

|  |  |
| --- | --- |
| 用户线程 | 内核线程 |
| 定义：根据操作系统内核是否对线程可感知，可以把线程分为内核线程和用户线程。 | |
| 用户线程指不需要内核支持而在用户程序中实现的线程，其不依赖于操作系统核心，用户进程利用线程库提供创建、同步、调度和管理线程的函数来控制用户线程。 | 内核线程建立和销毁都是由操作系统负责、通过系统调用完成的，操作系统在调度时，参考各进程内的线程运行情况做出调度决定，如果一个进程中没有就绪态的线程，那么这个进程也不会被调度占用CPU。 |
| 一个线程阻塞将使得整个进程（包括它的所有线程）阻塞。  原因：操作系统内核不知道多线程的存在，内核的管理和控制仍然是以进程为基本单位（处理器时间片分配或调度）。 | 当一个线程执行了一个系统调度后被阻塞起来，同一个进程中的其他线程仍可能会被系统调度投入运行。  原因：内核级线程对系统可见，所以是以线程为基本调度单位。 |
| 优点：  （1）可以在不支持线程的操作系统中实现。  （2）创建和销毁线程、线程切换代价等线程管理的代价比内核线程少得多。  （3）允许每个进程定制自己的调度算法，线程管理比较灵活。  （4）线程能够利用的表空间和堆栈空间比内核级线程多。 | 内核线程的优缺点刚好跟用户线程相反。 |
| 缺点：  1、同一进程中在同一时刻只有有一个线程在运行；  2、在一个进程中，如果有一个线程使用了系统调用而阻塞，那么整个进程都会被挂起。 |

实际上，操作系统可以使用混合的方式来实现线程。

## 8、Java实现线程的方式有三种方式

* 继承Thread类



* 实现Runnable接口



* 实现Callable接口，重写call函数

Callable是类似于Runnable的接口，实现Callable接口的类和实现Runnable的类都是可被其它线程执行的任务。

Callable和Runnable的不同:

|  |  |
| --- | --- |
| Callable接口 | Runnable接口 |
| 方法是call() | 方法是run() |
| 任务执行后有返回值 | 任务执行后不能返回值 |
| call()方法可抛出异常 | run()方法是不能抛出异常的 |
| 运行Callable任务可拿到一个Future对象，Future表示异步计算的结果。它提供了检查计算是否完成的方法,以等待计算的完成,并检索计算的结果.通过Future对象可了解任务执行情况,可取消任务的执行,还可获取任务执行的结果。 |  |

## 9、用户态和核心态的区别。

核心态：当一个任务（进程）执行系统调用而进入内核代码中执行时，我们就称进程处于内核运行态（或简称为内核态）。此时处理器处于特权级最高的（0级）内核代码中执行。当进程处于内核态时，执行的内核代码会使用当前进程的内核栈。每个进程都有自己的内核栈。

用户态：当进程在执行用户自己的代码时，则称其处于用户运行态（用户态）。即此时处理器在特权级最低的（3级）用户代码中运行。当正在执行用户程序而突然被中断程序中断时，此时用户程序也可以象征性地称为处于进程的内核态。因为中断处理程序将使用当前进程的内核栈。这与处于内核态的进程的状态有些类似。

用户态切换到内核态的3种方式：系统调用、异常、外围设备中断。

## 10、用户栈和内核栈的概念以及区别

* **内核栈和用户栈的概念**：

intel的cpu分为四个运行级别ring0~ring3，内核创建进程，创建进程的同时创建进程控制块，创建进程自己的堆栈。一个进程有两个堆栈，用户栈和系统栈。用户堆栈的空间指向用户地址空间，内核堆栈的空间指向内核地址空间。

有个CPU堆栈指针寄存器，进程运行的状态有用户态和内核态，当进程运行在用户态时。CPU堆栈指针寄存器指向的是用户堆栈地址，使用的是用户堆栈；当进程运行在内核态时，CPU堆栈指针寄存器指向的是内核堆栈地址，使用的是内核堆栈。

* **堆栈切换**

当系统因为系统调用（软中断）或硬件中断，CPU切换到特权工作模式，进程陷入内核态，进程使用的栈也要从用户栈转向系统栈。从用户态到内核态要两步骤，首先是将用户堆栈地址保存到内核堆栈中，然后将CPU堆栈指针寄存器指向内核堆栈。

当由内核态转向用户态，步骤首先是将内核堆栈中得用户堆栈地址恢复到CPU堆栈指针寄存器中。

* **内核栈和用户栈区别**

1. **使用的状态不同：**

内核栈是系统运行在内核态的时候使用的栈，用户栈是系统运行在用户态时候使用的栈。

当进程由于中断进入内核态时，系统会把一些用户态的数据信息保存到内核栈中，当返回到用户态时，取出内核栈中得信息恢复出来，返回到程序原来执行的地方。用户栈就是进程在用户空间时创建的栈，比如一般的函数调用，将会用到用户栈。

**2、保存内容不同：**

内核栈是属于操作系统空间的一块固定区域，可以用于保存中断现场、保存操作系统子程序间相互调用的参数、返回值等。

用户栈是属于用户进程空间的一块区域，用户保存用户进程子程序间的相互调用的参数、返回值等。

* 为何要设置两个不同的栈?

1、共享原因：内核的代码和数据是为所有的进程共享的，如果不为每一个进程设置对应的内核栈，那么就不能实现不同的进程执行不同的代码。

2、安全原因：如果只有一个栈，那么用户就可以修改栈内容来突破内核安全保护。

## 11、死锁的概念，导致死锁的原因

（1）、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陷入无休止的等待。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。

（2）、原因: 1、进程对系统资源的使用存在竞争关系。2、系统资源不足。3、进程运行推进的顺序不合适。 4、资源分配不当。

（3）、产生死锁的四个必要条件（切记：不是充分条件）：

* 互斥条件： 一个资源每次只能被一个进程使用。
* 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
* 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。
* 环路条件: 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。

1. 处理死锁的三个方式

* 通过破除死锁四个必要条件之一，来防止死锁产生。（事前—>预防策略）
* 仔细地对资源进行动态分配，以避免死锁。 （事中—>避免策略）
* 检测死锁并且恢复。 （事后—>检测与解除策略）

## 12. 预防死锁的方法

通过破除死锁四个必要条件之一，来预防死锁产生。有两种方法：

方法一：当其申请的资源得不到满足时，也必须放弃其原先占有的资源；（破坏了请求和保持）

方法二：只适用于申请资源的进程优先级比占有该资源的进程优先级高时，如果一个进程申请的资源被其它进程占用，而申请进程的优先级较高，那么它可以强迫占有资源的进程放弃。（破坏了不可剥夺）

## 13、避免死锁的方法

概述：避免是指进程在每次申请资源时判断这些操作是否安全，例如，使用银行家算法。死锁避免算法的执行会增加系统的开销。

所谓银行家算法，是指在分配资源之前先试探性分配，然后判断是否会导致系统死锁：**如果会死锁，则不分配，否则就分配。**

**按照银行家算法的思想，当进程请求资源时，系统将按如下原则分配系统资源：**

(1) 进程可以分期请求资源，请求的总数量不能超过最大需求量。

(2) 进程本次对资源的请求量不超过系统中的资源数时：可以接纳该进程；当系统现有的资源不能满足进程本次需资源数时，对进程的请求可以推迟分配。

(3) 当系统现有的资源能满足进程尚需资源数时，先试探性把资源分配给此进程；

（4）调用安全算法safe(),[测试](http://lib.csdn.net/base/softwaretest)系统是否安全。如果返回true，表明安全，可以正式分配；如果返回false，表明不安全，要推迟分配。

## 14、死锁检测和解除

死锁检测：死锁预防和避免都是事前措施，而死锁的检测则是判断系统是否处于死锁状态，如果是，则执行死锁解除策略。

化简系统S的资源分配图。

死锁解除：这是与死锁检测结合使用的，它使用的方式就是剥夺。即将某进程所拥有的资源强行收回，分配给其他的进程。

* 撤销进程法：撤销部分死锁的进程，用释放出来的资源来救活其他死锁的进程。
* 资源剥夺法：将一个阻塞进程挂起后，剥夺该进程所占有的所有资源，并保存它在挂起点前的状态（为恢复做准备）。

## 15、Windows内存管理的方式(块式、页式、段式、段页式)

内存管理是操作系统中的重要部分（两三句话恐怕谁也说不清楚吧）

* **概述：**当程序运行时需要从内存中读出这段程序的代码。代码的位置必须在物理内存中才能被运行，由于现在的操作系统中有非常多的程序运行着，内存中不能够完全放下，所以引出了虚拟内存的概念。

（1）、把那些不常用的程序片断就放入虚拟内存，当需要用到它的时候在load入主存（物理内存）中。这个就是内存管理所要做的事。

（2）、内存管理还有另外一件事需要做：计算程序片段在主存中的物理位置，以便CPU调度。

**现在常用段页式管理**

（1）、块式管理（连续空间）

把主存分为一大块、一大块的，当所需的程序片断不在主存时就分配一块主存空间，把程序片断load入主存，就算所需的程序片度只有几个字节也只能把这一块分配给它。这样会造成很大的浪费，平均浪费了50％的内存空间，但是易于管理。（内碎片）

为了避免内碎片的浪费，可以采用动态分块的方式：产生外碎片。

（2）、页式管理（离散空间）：把主存分为一页一页的，每一页的空间要比一块一块的空间小很多，显然这种方法的空间利用率要比块式管理高很多。（内碎片） 《页号，页内偏移》

（3）、分段式管理（段间离散空间，段内是连续的）：（外碎片。可以通过浮动来解决）

段：指的是一组逻辑信息，如主程序、过程、函数或数组等。每个进程按照其地址空间的内在逻辑关系划分为若干段，每个段赋予一个段名。段的长度可以是任意的，有逻辑信息的大小来决定。 《段号，段内偏移》

一个进程的地址空间可以包含以下不同的段：代码段、数据段、堆栈段、内存共享段。

（3）、段页式管理：结合了段式管理和页式管理的优点。将程序分成若干段，每个段分成若干页。

* 内存上仍然采用分页式，分成大小相等的帧，使用位示图记录内存的使用情况。
* 段表：一个进程有一张。
* 段内页表：一个进程中的每一个段对应一张页表，记录每个段分为了几页，每页分到内存中的哪几帧中。

《段号，页号，页内偏移》

二维逻辑地址：段号+段内地址  
分页与分段的主要区别：  
1）、段是信息的逻辑单位，它是根据用户的需要划分的，因此段对用户是可见的；页是信息的物理单位，是为了管理主存的方便而划分的，对用户是透明的。  
2）、页的大小固定不变，由系统决定。段的大小是不固定的，它由其完成的功能决定。  
3）、段式向用户提供的是二维地址空间，页式向用户提供的是一维地址空间，其页号和页内偏移是机器硬件的功能。  
4）、由于段是信息的逻辑单位，因此便于存贮保护和信息的共享，页的保护和共享受到限制。

分页与分段存储管理系统虽然在很多地方相似，但从概念上讲，两者是完全不同的，它们之间的区别如下：  
  ①页是信息的物理单位。分页的目的是实现离散分配，减少外部碎片，提高内存利用率。段是信息的逻辑单位。每一段在逻辑上是一组相对完整的信息集合。  
  ②分页式存储管理的作业地址空间是一维的，而分段式存储管理的作业地址空间是二维的。  
  ③页的大小固定且由系统确定，是等长的。而段的长度不定。  
  ④分页的优点体现在内存空间的管理上，而分段的优点体现在地址空间的管理上。

## 16、内存连续分配方式采用的几种算法及各自优劣

* 单一连续分配：是一种最简单的存储管理方式，其优点是软件处理简单，最大缺点是存储器不能充分利用。多用于单用户微机操作系统中。
* 分区分配：是多道程序环境下各种存储管理方式中最简单的一种。它将内存划分成若干个分区，在每个分区中按照连续分配方式分配给一个作业。分区有如下两种形式:

**（1）、固定分区分配：**指内存在处理作业前已被划分成若干个大小不等的分区，存储管理程序根据每个作业步的最大存储量分配一个足够大的分区给它。当找不到一个足够大的分区时，则通知作业调度挑选另一作业，这种方式分配和回收内存简单，但内存利用不充分，会产生“碎片”空间。

**（2）、可变分区分配**：指内存事先并未被分区，只有当作业进入内存时，才根据作业的大小建立分区。其特点是分区的个数和大小都是可变的，但需要建立分配区表和空白区表，且表的长度是不固定的。

* 可重定位分区分配：为使各分区中的用户程序能移到内存的一端，使外碎片集中于另一端成为一个大的空闲分区。

动态重地位技术是允许进程在内存中浮动的。当内存中出现一些外碎片后，可以将各个进程进行“搬家”，让他们移动到内存的一端，使所有的外部碎片移动到另一端，这样就可以使若干个零星的碎片合并成一个大的空闲区了，这一过程也称为进程浮动。

这种分配方法的优点是清除碎片，更大程度地利用内存空间，但必须硬件的支持，且要花费时间和加大系统开销。

## 17、虚拟存储器

（1）、管理方式

* 请求式分页存储管理

在进程开始运行之前,不是装入全部页面,而是装入一个或零个页面,之后根据进程运行的需要,动态装入其他页面;当内存空间已满,而又需要装入新的页面时,则根据某种算法淘汰某个页面,以便装入新的页面

* 请求式分段存储管理

为了能实现虚拟存储,段式逻辑地址空间中的程序段在运行时并不全部装入内存,而是如同请求式分页存储管理,首先调入一个或若干个程序段运行,在运行过程中调用到哪段时,就根据该段长度在内存分配一个连续的分区给它使用.若内存中没有足够大的空闲分区,则考虑进行段的紧凑或将某段或某些段淘汰出去,这种存储管理技术称为请求式分段存储管理

## 18.页面置换算法，会算所需换页数

  地址映射过程中，若在页面中发现所要访问的页面不再内存中，则产生缺页中断。当发生缺页中断时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。常见的置换算法有：

（1）最佳置换算法（OPT）（理想置换算法）

这是一种理想情况下的页面置换算法，但实际上是不可能实现的。该算法的基本思想是：每次选择的淘汰页面总是不再使用额，或者最长时间不再使用的页面，尽量避免刚调出去的页面又要调入。虽然这个算法不可能实现，但是最佳页面置换算法可以用于对可实现算法的性能进行衡量比较。

（2）、先进先出置换算法（FIFO）

最简单的页面置换算法是先入先出（FIFO）法。这种算法的实质是，总是选择在主存中停留时间最长（即最老）的一页置换，即先进入内存的页，先退出内存。

这种算法的效率并不高，原因是在内存中驻留时间长的页不一定是最近不使用的页。反而那些常被访问的页，往往在主存中也停留得最久，结果它们因变“老”而不得不被置换出去。

FIFO的另一个缺点是，它有一种异常现象，即在增加存储块的情况下，反而使缺页中断率增加了。

（3）、最近最久未使用（LRU）算法：强调时间

FIFO算法和OPT算法之间的主要差别是，FIFO算法利用页面进入内存后的时间长短作为置换依据，而OPT算法的依据是页面没有被访问的时间。它的实质是，当需要置换一页时，选择在最近一段时间里最久没有使用过的页面予以置换。这种算法就称为最久未使用算法（Least Recently Used，LRU）。LRU算法是经常采用的页面置换算法，并被认为是相当好的。

（4）Clock置换算法（LRU算法的近似实现）

（5）最少使用（LFU）置换算法：强调次数

2. 什么是线程安全？ 如果多线程的程序运行结果是可预期的，而且与单线程的程序运行结果一样，那么说明是“线程安全”的。