操作系统

2017年8月20日 16:40

1. **Windows/linux内存管理**

内存管理的必要性

出现在早期的计算机系统当中，程序是直接运行在物理内存中，每一个程序都能直接访问物理地址。如果这个系统只运行一个程序的话，并且这个程序所需的内存不要超过该机器的物理内存就不会出现问题。但是如今的系统都是支持多任务和多进程的，那么这时我们就要考虑如何将系统内有限的物理内存及时有效的分配给多个程序了，这就是内存管理的基本概念。顾名思义就是是记录哪些内存是正在使用的，哪些内存是空闲的；在进程需要时为其分配内存，在进程使用完后释放内存。

内存管理的例子

假如我们有三个程序，程序A，B，C，程序A运行的过程中需要10M内存，程序B运行的过程中需要100M内存，而程序C运行的过程中需要20M内存。如果系统同时需要运行程序A和B，那么早期的内存管理过程大概是这样的，将物理内存的前10M分配给A，接下来的10M-110M分配给B。这种内存管理的方法比较直接，好了，假设我们这个时候想让程序C也运行，同时假设我们系统的内存只有128M，显然按照这种方法程序C由于内存不够是不能够运行的。大家知道可以使用虚拟内存的技术，内存空间不够的时候可以将程序不需要用到的数据交换到磁盘空间上去，已达到扩展内存空间的目的。下面我们来看看这种内存管理方式存在的几个比较明显的问题。

1．进程地址空间不能隔离

由于程序直接访问的是物理内存，这个时候程序所使用的内存空间不是隔离的。举个例子，就像上面说的A的地址空间是0-10M这个范围内，但是如果A中有一段代码是操作10M-128M这段地址空间内的数据，那么程序B和程序C就很可能会崩溃（每个程序都可以访问系统的整个地址空间）。这样很多恶意程序或者是木马程序可以轻而易举地破快其他的程序，系统的安全性也就得不到保障了，这对用户来说也是不能容忍的。

2. 内存使用的效率低

如上面提到的，如果我们要像让程序A、B、C同时运行，那么唯一的方法就是使用虚拟内存技术将一些程序暂时不用的数据写到磁盘上，在需要的时候再从磁盘读回内存。这里程序C要运行，将A交换到磁盘上去显然是不行的，因为程序是需要连续的地址空间的，程序C需要20M的内存，而A只有10M的空间，所以需要将程序B交换到磁盘上去，而B足足有100M，可以看到为了运行程序C我们需要将100M的数据从内存写到磁盘，然后在程序B需要运行的时候再从磁盘读到内存，我们知道IO操作比较耗时，所以这个过程效率将会十分低下。

3. 程序运行的地址不能确定

程序每次需要运行时，都需要在内存中分配一块足够大的空闲区域，而问题是这个空闲的位置是不能确定的，这会带来一些重定位的问题，重定位的问题确定就是程序中引用的变量和函数的地址，如果有不明白童鞋可以去查查编译愿意方面的资料。

内存管理无非就是想办法解决上面三个问题，如何使进程的地址空间隔离，如何提高内存的使用效率，如何解决程序运行时的重定位问题？

基本内存管理方案

单一连续区

每次只运行一个用户程序，用户程序独占内存，总是被加载到同一个内存地址上。

特点：一段时间只有一个进程在内存，简单，内存利用率低

固定分区

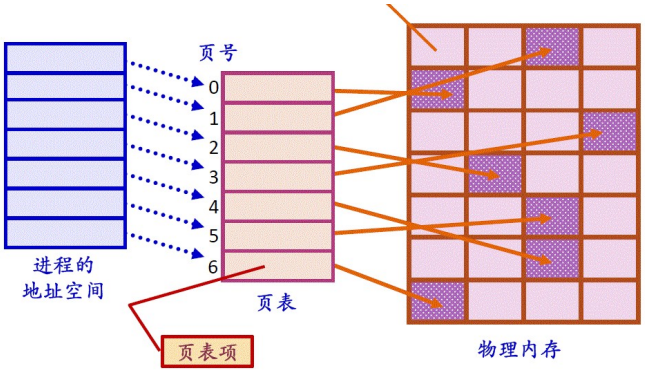
系统把内存空间分割成若干个连续区域，每个区域称为分区；每个分区的大小可以相同也可以不同，分区大小固定不变，每个分区有且只能装一个进程。

可变分区

根据进程的需求，把可分配的内存空间分割出一个分区，分配给进程。这种管理方案可能出现一些很小的，不易利用的空闲去区，从而导致内存利用率的下降，可以考虑利用紧缩技术来解决这个问题，也就是在内存移动程序中，将所有小的空闲区合并成较大的空闲区。

页式存储管理方案

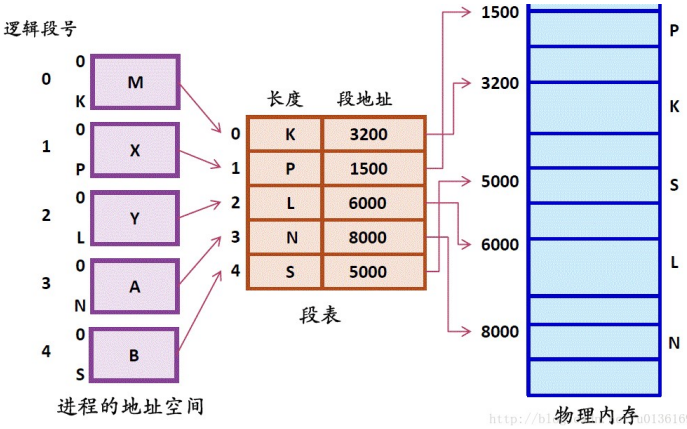
1. 用户进程地址空间被划分成大小相等的部分，称为页或者页面，从0开始编号。
2. 内存空间按同样的大小分成划分成大小相等的区域，称为页框。从0开始编号。
3. 以页为单位来进行内存分配，按照进程需要的页数来分配；逻辑上相邻的页，物理上不一定相邻。典型的页面尺寸是4K或4M。



从进程的地址空间转换到实际的内存空间是通过查页表项（记录了逻辑页与页框号的对应关系，每一个进程一个页表，存放在内存）来实现的。CPU取到逻辑地址，自动划分为页号和页内地址;用页号查页表，得到页框号，再与页内偏移拼接成物理地址。

段式存储管理方案

1. 用户进程地址空间按照程序自身的逻辑关系划分为若干个程序段，每个程序段都有一个段名。
2. 内存空间被动态划分成若干长度不相同的区域，称为物理段，每个物理段有起始地址和长度决定
3. 内存分配规则：以段为单位进行分配，每段在内存占据连续空间，但各段之间可以不相邻。
4. 逻辑地址为段号+段内地址。



段表记录了段号，段首地址和段长度之间的关系，每个进程一个段表，存在内存。地址转换规则为CPU取到逻辑地址，用段号查段表，得到该段在内存的起始地址，与段内偏移地址计算出物理地址。

段页式存储管理方案

1. 用户进程地址空间先按段进行划分，每一段再按页面进行划分。
2. 内存空间：同页式存储方案。
3. 内存分配规则：同页式存储方案。
4. 逻辑地址如下：

虚拟内存技术

基本思想：每个程序拥有自己的地址空间，这个空间被分割成很多个块，每一个块称为页面。每一页都有连续的地址范围。这些页被映射到物理内存，但并不是所有页都必须在内存中才能运行程序。当程序引用到一部分在物理内存中的地址空间时，由硬件立刻执行必要的

映射。当程序引用到一部分不再物理内存中的地址空间时，由**操作系统**（缺页异常）负责将缺失的部分装入物理内存并重新执行失败的指令。

缺页异常使用的常用页面置换算法

- 最佳页面置换算法（OPT）
- 先进先出算法（FIFO）
- 第二次机会算法（SCR）
- 时钟算法（clock）
- 最近未使用算法（NRU）
- 最近最少使用算法（LRU）
- 最不经常使用算法（NFU）
- 老化算法（aging）
- 工作集算法(working set)

算法	评价
OPT	不可实现，但可作为基准
NRU	LRU的很粗略的近似
FIFO	可能置换出重要的页面
Second Chance	比FIFO有很大的改善
Clock	实现的
LRU	很优秀，但很难实现
NFU	LRU的相对粗略的近似
Aging	非常近似LRU的有效算法
Working set	实现起来开销很大

来自 <http://blog.csdn.net/u013616945/article/details/77435607>

2. 内存池的理解

最近开始学习内存池技术，《高质量c++/c编程指南》在内存管理的第一句话就是：欢迎进入内存这片雷区，由此可见掌握内存管理对于c++/c程序员的重要性。使用内存池的优点有：降低动态申请内存的次数，提升系统性能，减少内存碎片，增加内存空间使用率。

内存池的分类：

一、不定长内存池：优点：不需要为不同的数据创建不同的内存池，缺点是分配出去的内存池不能回收到池中(?)。代表有apr_pool，obstack。

二、定长内存池：优点：使用完立即把内存归还池中。代表有Loki, Boost。

本次以sgi stl中实现的内存池作为学习对象。由于要实现的是一个**C语言**的内存池，所以这里用C的描述方式。喜欢C++的朋友可以直接看源文件或者《STL源码剖析》的讲解。sgi设计了二级配置机制，第一级配置器直接使用malloc()和free()。当配置区块超过128 bytes时，则采用第一级配置器；否则采用memory pool方式。

memory pool的整体思想是维护128/8 = 16个自由链表，这里的8是小型区块的上调边界。每个自由链表串接的区块大小如下：

序号	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
串接区块	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120	128
范围	1-8	9-16	17-24	25-32	33-40	41-48	49-56	57-64	65-72	73-80	81-88	89-96	97-104	105-112	113-120	121-128

几个过程的思路：

一、申请过程：

Code:

```
1. if (用户申请的内存不大于128 bytes)
2.     查找对应的链表
3.     if (对应链表拥有一块以上的区块)
4.         调整链表
5.         返回第一个区块地址
```

```
    else
        准备重新填充链表
            向内存池申请内存(指定数量的区块)
            if (内存池申请到一个区块)
                返回第一个区块地址
            else
                调整链表，将区块串接起来
                返回第一个区块地址
    else
        直接用malloc()申请内存
```

二、释放过程：

Code:

```
if (用户释放的内存块大于128 bytes)
    直接用free()释放
else
    查找对应的链表
    回收内存
```

三、向内存池申请内存过程：

Code:

```
if (内存池空间完全满足需求量)
    调整内存池起始位置
    返回空间地址
else if (内存池空间不能完全满足需求量，但能提供一个以上的区块)
    计算能够提供的最大内存
    调整内存池起始位置
    返回空间地址
else
    从内存池中压缩内存
    收集比size大的空间
    递归调用，修正nobjs
    再次申请内存，可能抛出异常
```

来自 <http://blog.csdn.net/houapple/article/details/6492735>

参考：http://www.cnblogs.com/findumars/p/6143020.html

3. 多线程如何同步

进程中线程同步的四种常用方式：

1、 临界区（CCriticalSection）

当多个线程访问一个独占性共享资源时，可以使用临界区对象。拥有临界区的线程可以访问被保护起来的资源或代码段，其他线程若想访问，则被挂起，直到拥有临界区的线程放弃临界区为止。具体应用方式：

1、 定义临界区对象CcriticalSection g_CriticalSection;

2、 在访问共享资源（代码或变量）之前，先获得临界区对象，g_CriticalSection.Lock（）；

3、 访问共享资源后，则放弃临界区对象，g_CriticalSection.Unlock（）；

2、 事件（CEvent）

事件机制，则允许一个线程在处理完一个任务后，主动唤醒另外一个线程执行任务。比如在某些网络应用程序中，一个线程如A负责侦听通信端口，另外一个线程B负责更新用户数据，利用事件机制，则线程A可以通知线程B何时更新用户数据。每个Cevent对象可以有两种状态：有信号状态和无信号状态。Cevent类对象有两种类型：人工事件和自动事件。

自动事件对象，在被至少一个线程释放后自动返回到无信号状态；

人工事件对象，获得信号后，释放可利用线程，但直到调用成员函数ReSet()才将其设置为无信号状态。在创建Cevent对象时，默认创建的是自动事件。

1、

```
CEvent(BOOL bInitiallyOwn=FALSE,
       BOOL bManualReset=FALSE,
```

```
    LPCTSTR lpszName=NULL,
    LPSECURITY_ATTRIBUTES lpsaAttribute=NULL);
```

- bInitiallyOwn:指定事件对象初始化状态，TRUE为有信号，FALSE为无信号；
- bManualReset：指定要创建的事件是属于人工事件还是自动事件。TRUE为人工事件，FALSE为自动事件；
- 后两个参数一般设为NULL，在此不作过多说明。

2、BOOL CEvent：：SetEvent();
将Cevent类对象的状态设置为有信号状态。如果事件是人工事件，则Cevent类对象保持为有信号状态，直到调用成员函数ResetEvent()将其重新设为无信号状态时为止。如果为自动事件，则在SetEvent（）后将事件设置为有信号状态，由系统自动重置为无信号状态。

3、BOOL CEvent：：ResetEvent();
将事件的状态设置为无信号状态，并保持该状态直至SetEvent（）被调用为止。由于自动事件是由系统自动重置，故自动事件不需要调用该函数。
一般通过调用WaitForSingleObject（）函数来监视事件状态。

3、 互斥量（CMutex）
互斥对象和临界区对象非常相似，只是其允许在进程间使用，而临界区只限制与同一进程的各个线程之间使用，
但是更节省资源，更有效率。
4、 信号量（CSemphore）
当需要一个计数器来限制可以使用某共享资源的线程数目时，可以使用“信号量”对象。CSemaphore类对象保存了对当前访问某一个指定资源的线程的计数值，该计数值是当前还可以使用该资源的线程数目。如果这个计数达到了零，则所有对这个CSemaphore类对象所控制的资源的访问尝试都被放入到一个队列中等待，直到超时或计数值不为零为止。
CSemaphore 类的构造函数原型及参数说明如下：

```
CSemaphore(
  LONG lInitialCount = 1,
  LONG lMaxCount = 1,
  LPCTSTR pstrName = NULL,
  LPSECURITY_ATTRIBUTES lpsaAttributes = NULL
);
```

- lInitialCount:信号量对象的初始计数值，即可访问线程数目的初始值；
- lMaxCount：信号量对象计数值的最大值，该参数决定了同一时刻可访问由信号量保护的资源的线程最大数目；
- 后两个参数在同一进程中使用一般为NULL，不作过多讨论；

一般是将当前可用资源计数设置为最大资源计数，每增加一个线程对共享资源的访问，当前可用资源计数就减1，只要当前可用资源计数大于0，就可以发出信号量信号。如果为0，则放入一个队列中等待。线程在处理完共享资源后，应在离开的同时通过ReleaseSemaphore（）函数将当前可用资源数加1。

```
BOOL ReleaseSemaphore(  HANDLE hSemaphore,       // hSemaphore:信号量句柄
LONG lReleaseCount,      // lReleaseCount：信号量计数值
LPLONG lpPreviousCount   // 参数一般为NULL);
```

来自 <http://www.cnblogs.com/lebronjames/archive/2010/08/11/1797702.html>

4. 内存泄露如何检测 解决

一.什么是内存泄漏:
内存泄露(memory leak)是指你向系统申请分配内存进行使用(new/alloc),可是使用完了却没有归还(delete),结果你申请到的那块内存你自己也不能再访问(也许你把他的地址给弄丢了),而系统将不能再把它分配给需要的程序,
相信大家都知道,一次内存泄漏的危害可以忽略,但是内存泄漏堆积后果很严重,无论多少内存,迟早会被占光.那么接下来就说说什么情况下会引起内存泄漏,
二.引起内存泄露的情况:
1>delegate(代理)如果声明为strong就会造成循环应用(比如类A强引用delegate属性,代理强

引用控制器,控制器又强引用类A),就会造成谁也释放不了(释放的原则是没有强引用,引用计数为0),进而造成内存泄漏,delegate也不能用assign声明,因为可能会造成野指针,应该声明为weak,

2>block下也可能造成内存泄漏,如果block中调用了self可能会造成类似上面代理中三方依次强引用的情况(但是也不全是,需要具体分析),这时候就会造成内存泄漏,解决办法是使self使用__weak修饰,变成弱引用.

3>NSTimer在VC释放之前,一定要调用[timer invalidate],不调用的后果就是NSTimer无法释放其target,如果target正好是self,则会导致循环引用,造成内存泄漏.

知道了在哪些情况下能引起内存泄漏了,但是很多情况下,我们还是很难主动的发现内存泄漏,这时候就需要内存泄漏检测了,

一: 内存泄漏

内存泄漏是编程中常常见到的一个问题，内存泄漏往往会一种奇怪的方式来表现出来,基本上每个程序都表现出不同的方式。 但是一般最后的结果只有两个，一个是程序当掉，一个是系统内存不足。 还有一种就是比较介于中间的结果程序不会当，但是系统的反映时间明显降低，需要定时的Reboot才会正常。

有一个很简单的办法来检查一个程序是否有内存泄漏。就是是用Windows的任务管理器(Task Manager)。运行程序，然后在任务管理器里面查看 “内存使用”和”虚拟内存大小”两项，当程序请求了它所需要的内存之后，如果虚拟内存还是持续的增长的话，就说明了这个程序有内存泄漏问题。 当然如果内存泄漏的数目非常的小，用这种方法可能要过很长时间才能看的出来。

当然最简单的办法大概就是用CompuWare的BoundChecker 之类的工具来检测了，不过这些工具的价格对于个人来讲稍微有点奢侈了。

如果是已经发布的程序，检查是否有内存泄漏是又费时又费力。所以内存泄漏应该在Code的生成过程就要时刻进行检查。

二: 原因

内存泄漏产生的原因一般是三种情况：

1. 分配完内存之后忘了回收；

2. 程序Code有问题，造成没有办法回收；

3. 某些API函数操作不正确，造成内存泄漏。

1. 内存忘记回收，这个是不应该的事情。但是也是在代码种很常见的问题。分配内存之后，用完之后，就一定要回收。如果不回收，那就造成了内存的泄漏，造成内存泄漏的Code如果被经常调用的话，那内存泄漏的数目就会越来越多的。从而影响整个系统的运行。比如下面的代码：

```
for (int =0;I<100;I++)
{
Temp = new BYTE[100];
}
```

就会产生 100*100Byte的内存泄漏。

2. 在某些时候，因为代码上写的有问题，会导致某些内存想回收都收不回来，比如下面的代码：

```
Temp1 = new BYTE[100];
Temp2 = new BYTE[100];
Temp2 = Temp1;
```

这样，Temp2的内存地址就丢掉了，而且永远都找不回了，这个时候Temp2的内存空间想回收都没有办法。

3. API函 数应用不当，在Windows提供API函数里面有一些特殊的API，比如FormatMessage。 如果你给它参数中有FORMAT_MESSAGE_ALLOCATE_BUFFER，它会在函数内部New一块内存Buffer出来。但是这个 buffer需要你调用LocalFree来释放。 如果你忘了，那就会产生内存泄漏。

来自 <http://blog.csdn.net/sunnylion1982/article/details/8186801>

5. windows平台下栈空间的大小

一般来说，我们所用的内存有栈和堆之分，其它的我们很少控制，栈的速度快，但是空间小、不灵活；而堆的空间几乎可以满足任何要求、灵活，但是相对的速度要慢了很多，并且在VC中堆是人为控制的，new了就要delete，否则很容易产生内存泄露等问题。

系统	栈的字节数	bits	digits	以兆为单位的值
Linux	8192K bytes	<=62407	<=18786	linux默认8M *(我的好像是10M)*
Windows	1024K bytes	<=10581	<=3185 (Release Version)	windows默认1M

一、VC++程序栈空间的大小

VC++默认的栈空间是1M，有两个方法更改:

a. link时用/STACK指定它的大小，或者在.def中使用STACKSIZE指定它的大小

【link选项】

b. 使用控制台命令“EDITBIN”更改exe的栈空间大小。

方法一：STACKSIZE　定义.def文件

语法：STACKSIZE reserve[,commit]

reserve：栈的大小；commit：可选项，与操作系统有关，在NT上指一次分配物理内存的大小

方法二：设定/STACK

打开工程，依次操作菜单如下：Project->Setting->Link，在Category 中选中Output，然后在Reserve中设定堆栈的最大值和commit。

注意：reserve默认值为1MB,最小值为4Byte；commit是保留在虚拟内存的页文件里面，它设置的较大会使栈开辟较大的值，可能增加内存的开销和启动时间。

二、Linux下程序栈空间的大小

linux下非编译器决定栈大小，而是由操作系统环境决定；而在Windows平台下栈的大小是被记录在可执行文件中的（由编译器来设置），即：windows下可以由编译器决定栈大小，而在Linux下是由系统环境变量来控制栈的大小的。

在Linux下通过如下命令可查看和设置栈的大小：

命令： ulimit -a　　# 显示当前栈的大小 （ulimit为系统命令，非编译器命令）

命令： ulimit -s 32768　　# 设置当前栈的大小为32M bytes

来自 <http://blog.csdn.net/sctq8888/article/details/9492285>

6. 进程和线程区别

1.定义

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.

2.关系

一个线程可以创建和撤销另一个线程;同一个进程中的多个线程之间可以并发执行.

相对进程而言，线程是一个更加接近于执行体的概念，它可以与同进程中的其他线程共享数据，但拥有自己的栈空间，拥有独立的执行序列。

3.区别

进程和线程的主要差别在于它们是不同的操作系统资源管理方式。进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。**但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。**

1）简而言之,一个程序至少有一个进程,一个进程至少有一个线程.

2）线程的划分尺度小于进程，使得多线程程序的并发性高。

3）另外，进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。

4）线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。**但是线程不能够独立执行，**必须依存在应用程序中，由应用程序提供多个线程执行控制。

5）从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。**这就是进程和线程的重要区别。**

4.优缺点

线程和进程在使用上各有优缺点：线程执行开销小，但不利于资源的管理和保护；而进程正相反。同时，线程适合于在SMP机器上运行，而进程则可以跨机器迁移。

来自 <http://blog.csdn.net/yaosiming2011/article/details/44280797/>

7. 线程之间的通信

1.全局变量

进程中的线程间内存共享，这是比较常用的通信方式和交互方式。

注：定义全局变量时最好使用volatile来定义，以防编译器对此变量进行优化。

2.Message消息机制

常用的Message通信的接口主要有两个：PostMessage和PostThreadMessage，

PostMessage为线程向主窗口发送消息。而PostThreadMessage是任意两个线程之间的通信接口。

2.1.PostMessage()

函数原型：

B00L PostMessage（HWND hWnd，UINT Msg，WPARAM wParam，LPARAM lParam）；

参数：

hWnd：其窗口程序接收消息的窗口的句柄。可取有特定含义的两个值：

HWND.BROADCAST：消息被寄送到系统的所有顶层窗口，包括无效或不可见的非自身拥有的窗口、被覆盖的窗口

和弹出式窗口。消息不被寄送到子窗口。

NULL：此函数的操作和调用参数dwThread设置为当前线程的标识符PostThreadMessage函数一样。

Msg：指定被寄送的消息。

wParam：指定附加的消息特定的信息。

IParam：指定附加的消息特定的信息。

返回值：如果函数调用成功，返回非零值：如果函数调用失败，返回值是零。

MS还提供了SendMessage方法进行消息间通讯，SendMessage(),他和PostMessage的区别是：

SendMessage是同步的，而PostMessage是异步的。SendMessage必须等发送的消息执行之后，才返回。

2.2.PostThreadMessage()

PostThreadMessage方法可以将消息发送到指定线程。

函数原型：BOOL PostThreadMessage(DWORD idThread,UINT Msg,WPARAM wParam, LPARAM lParam);

参数除了ThreadId之外，基本和PostMessage相同。

目标线程通过GetMessage()方法来接受消息。

注：使用这个方法时，目标线程必须已经有自己的消息队列。否则会返回ERROR_INVALID_THREAD_ID错误。可以用

PeekMessage()给线程创建消息队列。

3.CEvent对象

CEvent为MFC中的一个对象，可以通过对CEvent的触发状态进行改变，从而实现线程间的通信和同步。

来自 <http://www.cnblogs.com/dartagnan/archive/2011/11/21/2257607.html>

8. 操作系统自旋锁

线程被阻塞后便进入内核（Linux）调度状态，这个会导致系统在用户态与内核态之间来回切换，严重影响锁的性能。自旋锁的出现就是为了尽可能的避免线程阻塞。其原理是：当发生争用时，若Owner线程能在很短的时间内释放锁，则那些正在争用线程可以稍微等一等（自旋），在Owner线程释放锁后，争用线程可能会立即得到锁，从而避免了系统阻塞。但Owner运行的时间可能会超出了临界值，争用线程自旋一段时间后还是无法获得锁，这时争用线程则会停止自旋进入阻塞状态（后退）。基本思路就是自旋，不成功再阻塞，尽量降低阻塞的可能性，这对那些执行时间很短的代码块来说有非 常重要的性能提高。

线程自旋时做些啥？其实啥都不做，可以执行几次for循环，可以执行几条空的汇编指令，目的是占着CPU不放，等待获取锁的机 会。所以说，自旋是

把

双刃剑，如果旋的时间过长会影响整体性能，时间过短又达不到延迟阻塞的目的。显然，自旋的周期选择显得非常重要，但这与操作系统、硬件体系、系

统的负载等诸多场景相关，很难选择，如果选择不当，不但性能得不到提高，可能还会下降。

来自 <http://blog.csdn.net/halcyonbaby/article/details/8837298>

9. 线程的几种状态

线程在一定条件下，状态会发生变化。线程一共有以下几种状态：

1、新建状态(New)：新创建了一个线程对象。

2、就绪状态(Runnable)：线程对象创建后，其他线程调用了该对象的start()方法。该状态的线程位于“**可运行线程池**”中，变得可运行，只**等待获取CPU的使用权。即在就绪状态的进程除CPU之外，其它的运行所需资源都已全部获得。**

3、运行状态(Running)：就绪状态的线程获取了CPU，执行程序代码。

4、阻塞状态(Blocked)：阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。

阻塞的情况分三种：

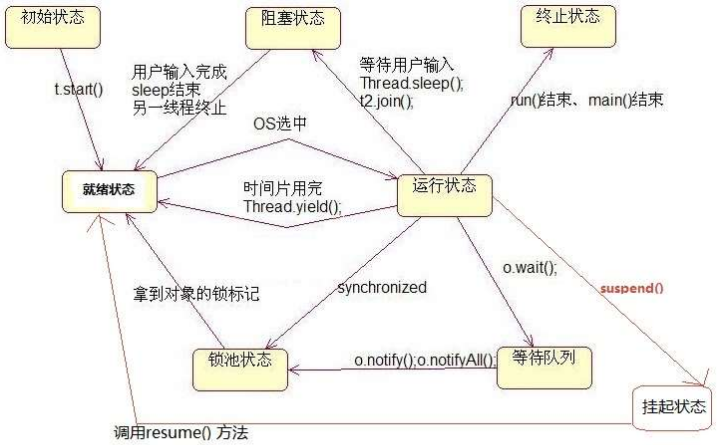
(1)、**等待阻塞**：运行的线程执行wait()方法，该线程会释放占用的所有资源，JVM会把该线程放入“**等待池**”中。进入这个状态后，是不能自动唤醒的，必须依靠其他线程调用notify()或notifyAll()方法才能被唤醒，

(2)、**同步阻塞**：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入“**锁池**”中。

(3)、**其他阻塞**：运行的线程执行sleep()或join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。

5、死亡状态(Dead)：线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

线程变化的状态转换图如下：



注：拿到对象的锁标记，即为获得了对该对象(**临界区**)的使用权限。即该线程获得了运行所需的资源，进入“就绪状态”，只需获得CPU，就可以运行。因为**当调用wait()后，线程会释放掉它所占有的“锁标志”，所以线程只有在此获取资源才能进入就绪状态。**

下面小小的作下解释：

1、线程的实现有两种方式，一是继承Thread类，二是实现Runnable接口，但不管怎样， 当我们new了这个对象后，线程就进入了初始状态；

2、当该对象调用了start()方法，就进入就绪状态；

3、进入就绪后，当该对象被**操作系统**选中，获得CPU时间片就会进入运行状态；

4、进入运行状态后情况就比较复杂了

4.1、run()方法或main()方法结束后，线程就进入终止状态；

4.2、当线程调用了自身的sleep()方法或其他线程的join()方法，进程让出CPU，然后就会进入阻塞状态（**该状态既停止当前线程，但并不释放所占有的资源即调用sleep ()函数后，线程不会释放它的“锁标志”。**）。当sleep()结束或join()结束后，该线程进入可运行状态，继续等待OS分配CPU时间片。**典型地，sleep()被用在等待某个资源就绪的情形：测试**发现条件不满足后，让线程阻塞一段时间后重新测试，直到条件满足为止。

4.3、线程调用了**yield()**方法，意思是放弃当前获得的CPU时间片，回到就绪状态，这时与其他进程处于同等竞争状态，OS有可能会接着又让这个进程进入运行状态； 调用 yield() 的效果等价于调度程序认为该线程已执行了足够的时间片从而需要转到另一个线程。yield()只是使当前线程重新回到可执行状态，所以执行yield()的线程有可能在进入到可执行状态后马上又被执行。

4.4、当线程刚进入可运行状态（注意，还没运行），发现将要调用的资源被synchroniza（同步），获取不到锁标记，将会立即进入**锁池状态**，等待获取锁标记（**这时的锁池里也许已经有了其他线程在等待获取锁标记，这时它们处于队列状态，既先到先得**），一旦线程获得锁标记后，就转入就绪状态，等待OS分配CPU时间片；

4.5. **suspend() 和 resume()方法**：两个方法配套使用，suspend()使得线程进入阻塞状态，并且不会自动恢复，必须其对应的resume()被调用，才能使得线程重新进入可执行状态。**典型地，suspend()和 resume() 被用在等待另一个线程产生的结果的情形：**测试发现结果还没有产生后，让线程阻塞，另一个线程产生了结果后，调用 resume()使其恢复。

4.6、wait()和 notify() 方法：当线程调用**wait()方法**后会进入等待队列（**进入这个状态会释放所占有的所有资源，与阻塞状态不同**），进入这个状态后，是不能自动唤醒的，必须依靠其他线程调用notify()或notifyAll()方法才能被唤醒（由于notify()只是唤醒一个线程，但我们由不能确定具体唤醒的是哪一个线程，也许我们需要唤醒的线程不能够被唤醒，因此在实际使用时，一般都用notifyAll()方法，唤醒有所线程），线程被唤醒后会进入锁池，等待获取锁标记。

wait() 使得线程进入阻塞状态，它有两种形式：

一种允许指定以毫秒为单位的一段时间作为参数；另一种没有参数。前者当对应的 notify()被调用或者超出指定时间时线程重新进入**可执行状态即就绪状态**，后者则必须对应的 notify()被调用。**当调用wait()后，线程会释放掉它所占有的“锁标志”，从而使线程所在对象中的其它synchronized数据可被别的线程使用。**waite()和notify()因为会对对象的“锁标志”进行操作，所以它们必须在synchronized函数或synchronizedblock中进行调

用。如果在non-synchronized函数或non-synchronizedblock中进行调用，虽然能编译通过，但在运行时会发生IllegalMonitorStateException的异常。

注意区别：初看起来wait() 和 notify() 方法与suspend()和 resume() 方法对没有什么分别，但是事实上它们是截然不同的。**区别的核心在于**，前面叙述的suspend()及其它所有方法在线程阻塞时都不会释放占用的锁（如果占用了的话），而wait() 和 notify() 这一对方法则相反。

上述的核心区别导致了一系列的细节上的区别

首先，前面叙述的所有方法都隶属于 Thread类，**但是wait() 和 notify() 方法这一对却直接隶属于 Object 类，也就是说，所有对象都拥有这一对方法。**初看起来这十分不可思议，但是实际上却是很自然的，因为这一对方法阻塞时要释放占用的锁，而锁是任何对象都具有的，调用任意对象的 wait() 方法导致线程阻塞，并且该对象上的锁被释放。而调用任意对象的notify()方法则导致因调用该对象的 wait()方法而阻塞的线程中随机选择的一个解除阻塞（但要等到获得锁后才真正可执行）。

其次，前面叙述的所有方法都可在任何位置调用，**但是wait() 和 notify() 方法这一对方法却必须在 synchronized 方法或块中调用，**理由也很简单，只有在synchronized方法或块中当前线程才占有锁，才有锁可以释放。同样的道理，调用这一对方法的对象上的锁必须为当前线程所拥有，这样才有锁可以释放。因此，这一对方法调用必须放置在这样的 synchronized方法或块中，该方法或块的上锁对象就是调用这一对方法的对象。若不满足这一条件，则程序虽然仍能编译，但在运行时会出现IllegalMonitorStateException异常。

wait() 和 notify()方法的上述特性决定了它们经常和synchronized方法或块一起使用，将它们和操作系统的进程间通信机制作一个比较就会发现它们的相似性：synchronized方法或块提供了类似于操作系统原语的功能，它们的执行不会受到多线程机制的干扰，而这一对方法则相当于 block和wake up 原语（这一对方法均声明为 synchronized）。它们的结合使得我们可以实现操作系统上一系列精妙的进程间通信的**算法**（如信号量算法），并用于解决各种复杂的线程间通信问题。

关于 wait() 和 notify() 方法最后再说明两点：

第一：调用notify() 方法导致解除阻塞的线程是从因调用该对象的 wait()方法而阻塞的线程中随机选取的，我们无法预料哪一个线程将会被选择，所以编程时要特别小心，避免因这种不确定性而产生问题。

第二：除了notify()，还有一个方法 notifyAll()也可起到类似作用，唯一的区别在于，调用 notifyAll()方法将把因调用该对象的 wait()方法而阻塞的所有线程一次性全部解除阻塞。当然，只有获得锁的那一个线程才能进入可执行状态。

谈到阻塞，就不能不谈一谈死锁，略一分析就能发现，suspend()方法和不指定超时期限的wait()方法的调用都可能产生死锁。遗憾的是，**Java**并不在语言级别上支持死锁的避免，我们在编程中必须小心地避免死锁。

来自 <http://blog.csdn.net/sinat_36042530/article/details/52565296>

10. **内存分页管理**
    http://www.cnblogs.com/edisonchou/p/5094066.html

11. **同步、异步、阻塞**

- **同步**

同步，**就是在发出一个功能调用时，在没有得到结果之前，该调用就不返回。**

要想实现同步操作，必须要获得线程的对象锁。获得它可以保证在同一时刻只有一个线能够进入临界区，并且在这个锁被释放之前，其他的线程都不能再进入这个临界区。如果其他线程想要获得这个对象的锁，只能进入等待队列等待。只有当拥有该对象锁的线程退出临界区时，锁才会被释放，等待队列中优先级最高的线程才能获得该锁。

实现同步的方式有两种：同步方法、同步代码块。

- **异步**

  当一个异步过程调用发出后，调用者不会立刻得到结果。实际处理这个吊用的部件是在调用发出后，**通过状态、通知来通知调用者，或通过回调函数处理这个调用。**

  由于每个线程都包含了运行时自身所需要的数据或方法，因此，在进行输入输出时，不必关系其他线程的状态或行为，也不必等到输入输出处理完毕才返回。当应用程序在对象上调用了一个需要花费很长时间来执行的方法，并且不希望让程序等待方法的返回时，就应该使用异步编程，异步能够提高程序的效率。

- **阻塞**

  阻塞调用是指调用结果返回之前，**当前线程会被挂起。**函数只有在得到结果之后才会返回。

- **非阻塞**

  指在不能立刻得到结果之前，该函数**不会阻塞当前线程**，而会立刻返回。

- **同步与阻塞**

  同步是个过程，阻塞是线程的一种状态。多个线程操作共享变量时可能会出现竞争。这时需要同步来防止两个以上的线程同时进入临界区，在这个过程中，后进入临界区的线程将阻塞，等待先进入的线程走出临界区。

- **线程同步一定发生阻塞吗？**

  线程同步不一定发生阻塞！！！线程同步的时候，需要协调推进速度，互相等待和互相唤醒会发生阻塞。

来自 <http://www.cnblogs.com/jiqianqian/p/6650680.html?utm_source=itdadao&utm_medium=referral>

12. **进程调度算法**

1．先来先服务调度算法

先来先服务(FCFS)调度**算法**是一种最简单的调度算法，该算法既可用于作业调度，也可用于进程调度。当在作业调度中采用该算法时，每次调度都是从后备作业队列中选择一个或多个最先进入该队列的作业，将它们调入内存，为它们分配资源、创建进程，然后放入就绪队列。在进程调度中采用FCFS算法时，则每次调度是从就绪队列中选择一个最先进入该队列的进程，为之分配处理机，使之投入运行。该进程一直运行到完成或发生某事件而阻塞后才放弃处理机。

2．短作业(进程)优先调度算法

短作业(进程)优先调度算法SJ(P)F，是指对短作业或短进程优先调度的算法。它们可以分别用于作业调度和进程调度。短作业优先(SJF)的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出一个估计运行时间最短的进程，将处理机分配给它，使它立即执行并一直执行到完成，或发生某事件而被阻塞放弃处理机时再重新调度。

二、高优先权优先调度算法

1．优先权调度算法的类型

为了照顾紧迫型作业，使之在进入系统后便获得优先处理，引入了最高优先权优先(FPF)调度算法。此算法常被用于批处理系统中，作为作业调度算法，也作为多种操作系统中的进程调度算法，还可用于实时系统中。当把该算法用于作业调度时，系统将从后备队列中选择若干个优先权最高的作业装入内存。当用于进程调度时，该算法是把处理机分配给就绪队列中优先权最高的进程，这时，又可进一步把该算法分成如下两种。

1) 非抢占式优先权算法

在这种方式下，系统一旦把处理机分配给就绪队列中优先权最高的进程后，该进程便一直执行下去，直至完成；或因发生某事件使该进程放弃处理机时，系统方可再将处理机重新分配给另一优先权最高的进程。这种调度算法主要用于批处理系统中；也可用于某些对实时性要求不严的实时系统中。

2) 抢占式优先权调度算法

在这种方式下，系统同样是把处理机分配给优先权最高的进程，使之执行。但在其执行期间，只要又出现了另一个其优先权更高的进程，进程调度程序就立即停止当前进程(原优先权最高的进程)的执行，重新将处理机分配给新到的优先权最高的进程。因此，在采用这种调度算法时，是每当系统中出现一个新的就绪进程i 时，就将其优先权Pi与正在执行的进程j 的优先权Pj进行比较。如果Pi≤Pj，原进程Pj便继续执行；但如果是Pi>Pj，则立即停止Pj的执行，做进程切换，使i 进程投入执行。显然，这种抢占式的优先权调度算法能更好地满足紧迫作业的要求，故而常用于要求比较严格的实时系统中，以及对性能要求较高的批处理和分时系统中。

2．高响应比优先调度算法

在批处理系统中，短作业优先算法是一种比较好的算法，其主要的不足之处是长作业的运行得不到保证。如果我们能为每个作业引入前面所述的动态优先权，并使作业的优先级随着等待时间的增加而以速率a 提高，则长作业在等待一定的时间后，必然有机会分配到处理机。该优先权的变化规律可描述为：

由于等待时间与服务时间之和就是系统对该作业的响应时间，故该优先权又相当于响应比RP。据此，又可表示为：

由上式可以看出：

(1) 如果作业的等待时间相同，则要求服务的时间愈短，其优先权愈高，因而该算法有利于短作业。

(2) 当要求服务的时间相同时，作业的优先权决定于其等待时间，等待时间愈长，其优先权愈高，因而它实现的是先来先服务。

(3) 对于长作业，作业的优先级可以随等待时间的增加而提高，当其等待时间足够长时，其优先级便可升到很高，从而也可获得处理机。简言之，该算法既照顾了短作业，又考虑了作业到达的先后次序，不会使长作业长期得不到服务。因此，该算法实现了一种较好的折衷。当然，在利用该算法时，每要进行调度之前，都须先做响应比的计算，这会增加系统开销。

三、基于时间片的轮转调度算法

1．时间片轮转法

1) 基本原理

在早期的时间片轮转法中，系统将所有的就绪进程按先来先服务的原则排成一个队列，每次调度时，把CPU 分配给队首进程，并令其执行一个时间片。时间片的大小从几ms 到几百ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证就绪队列中的所有进程在一给定的时间内均能获得一时间片的处理机执行时间。换言之，系统能在给定的时间内响应所有用户的请求。

2．多级反馈队列调度算法

前面介绍的各种用作进程调度的算法都有一定的局限性。如短进程优先的调度算法，仅照顾了短进程而忽略了长进程，而且如果并未指明进程的长度，则短进程优先和基于进程长度的抢占式调度算法都将无法使用。而多级反馈队列调度算法则不必事先知道各种进程所需的执行时间，而且还可以满足各种类型进程的需要，因而它是目前被公认的一种较好的进程调度算法。在采用多级反馈队列调度算法的系统中，调度算法的实施过程如下所述。

(1) 应设置多个就绪队列，并为各个队列赋予不同的优先级。第一个队列的优先级最高，第二个队列次之，其余各队列的优先权逐个降低。该算法赋予各个队列中进程执行时间片的大小也各不相同，在优先权愈高的队列中，为每个进程所规定的执行时间片就愈小。例如，第二个队列的时间片要比第一个队列的时间片长一倍，……，第i+1个队列的时间片要比第i个队列的时间片长一倍。

(2) 当一个新进程进入内存后，首先将它放入第一队列的末尾，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二队列的末尾，再同样地按FCFS原则等待调度执行；如果它在第二队列中运行一个时间片后仍未完成，再依次将它放入第三队列，……，如此下去，当一个长作业(进程)从第一队列依次降到第*n*队列后，在第*n* 队列便采取按时间片轮转的方式运行。

(3) 仅当第一队列空闲时，调度程序才调度第二队列中的进程运行；仅当第1～(i-1)队列均空时，才会调度第i队列中的进程运行。如果处理机正在第i队列中为某进程服务时，又有新进程进入优先权较高的队列(第1～(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队列的末尾，把处理机分配给新到的高优先权进程。

来自 <http://blog.csdn.net/luyafei_89430/article/details/12971171>

13. **进程间的通信方式**

管道(pipe)：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 (named pipe) ： 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(semophore) ： 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(message queue) ： 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 (sinal) ： 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(shared memory) ：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，如信号两，配合使用，来实现进程间的同步和通信。

套接字(socket) ： 套解口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同及其间的进程通信。

来自 <http://www.cnblogs.com/mydomain/archive/2010/09/23/1833369.html>

14. **自旋锁和互斥锁有什么区别**
15. **i++是不是原子操作**
16. **线程调度和进程调度的区别**
17. **死锁是怎么发生的**

（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，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，在未使用完之前，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。

18. **文件如何被加载到内存中（文件系统）**
19. 缺页中断时操作系统怎么做

缺页中断就是要访问的页不在主存，需要

操作系统

将其调入主存后再进行访问。

当进程执行过程中发生缺页中断时，需要进行页面换入，步骤如下：

<1> 首先硬件会陷入内核，在堆栈中保存程序计数器。大多数机器将当前指令的各种状态信息保存在CPU中特殊的寄存器中。

<2>启动一个汇编代码例程保存通用寄存器及其它易失性信息，以免被操作系统破坏。这个例程将操作系统作为一个函数来调用。

（在页面换入换出的过程中可能会发生上下文换行，导致破坏当前程序计数器及通用寄存器中本进程的信息）

<3>当操作系统发现是一个页面中断时，查找出来发生页面中断的虚拟页面（进程地址空间中的页面）。这个虚拟页面的信息通常会保存在一个硬件寄存器中，如果没有的话，操作系统必须检索程序计数器，取出这条指令，用软件分析该指令，通过分析找出发生页面中断的虚拟页面。

<4>检查虚拟地址的有效性及安全保护位。如果发生保护错误，则杀死该进程。

<5>操作系统查找一个空闲的页框(物理内存中的页面)，如果没有空闲页框则需要通过页面置换算法找到一个需要换出的页框。

<6>如果找的页框中的内容被修改了，则需要将修改的内容保存到磁盘上，此时会引起一个写磁盘调用，发生上下文切换（在等待磁盘写的过程中让其它进程运行）。

（注：此时需要将页框置为忙状态，以防页框被其它进程抢占掉）

<7>页框干净后，操作系统根据虚拟地址对应磁盘上的位置，将保持在磁盘上的页面内容复制到“干净”的页框中，此时会引起一个读磁盘调用，发生上下文切换。
<8>当磁盘中的页面内容全部装入页框后，向操作系统发送一个中断。操作系统更新内存中的页表项，将虚拟页面映射的页框号更新为写入的页框，并将页框标记为正常状态。
<9>恢复缺页中断发生前的状态，将程序指令器重新指向引起缺页中断的指令。
<10>调度引起页面中断的进程，操作系统返回汇编代码例程。
<11>汇编代码例程恢复现场，将之前保存在通用寄存器中的信息恢复。

来自 <http://blog.csdn.net/fengkuangshixisheng/article/details/26473385>

20.