**1.程序什么时候使用线程，什么时候单线程效率高？**

使用线程可以把占据长时间的程序中的任务放到后台去处理，如用户在界面点击，让一个线程取处理该项操作，在一些等待的任务实现上如用户输入、文件读写和网络收发数据等的情况下，使用线程可以释放掉一些资源如内存占用等等。

单线程效率高的情况：对于频繁创建和切换线程可能会造成大的时间开销，这样多线程带来的时间代价将会拖慢整个程序，由此不如单线程来的方便。

**2.线程惊群问题**

什么是惊群？

        举一个很简单的例子，当你往一群鸽子中间扔一块食物，虽然最终只有一个鸽子抢到食物，但所有鸽子都会被惊动来争夺，没有抢到食物的鸽子只好回去继续睡觉， 等待下一块食物到来。这样，每扔一块食物，都会惊动所有的鸽子，即为惊群。对于操作系统来说，多个进程/线程在等待同一资源是，也会产生类似的效果，其结果就是每当资源可用，所有的进程/线程都来竞争资源，造成的后果：

1）系统对用户进程/线程频繁的做无效的调度、上下文切换，系统系能大打折扣。

2）为了确保只有一个线程得到资源，用户必须对资源操作进行加锁保护，进一步加大了系统开销。

关于“线程池中惊群问题”：

在任务队列加入新的任务后，使用pthread\_cond\_signa而不是使用pthread\_cond\_broadcast()函数来通知等待的线程队列，这样可靠的保证了，每次只通知到一个等待线程，避免了多个线程同时抢占资源的问题。

**3.C++函数内的静态变量初始化以及线程安全问题**

函数内静态类对象初始化非线程安全（C++11之前）

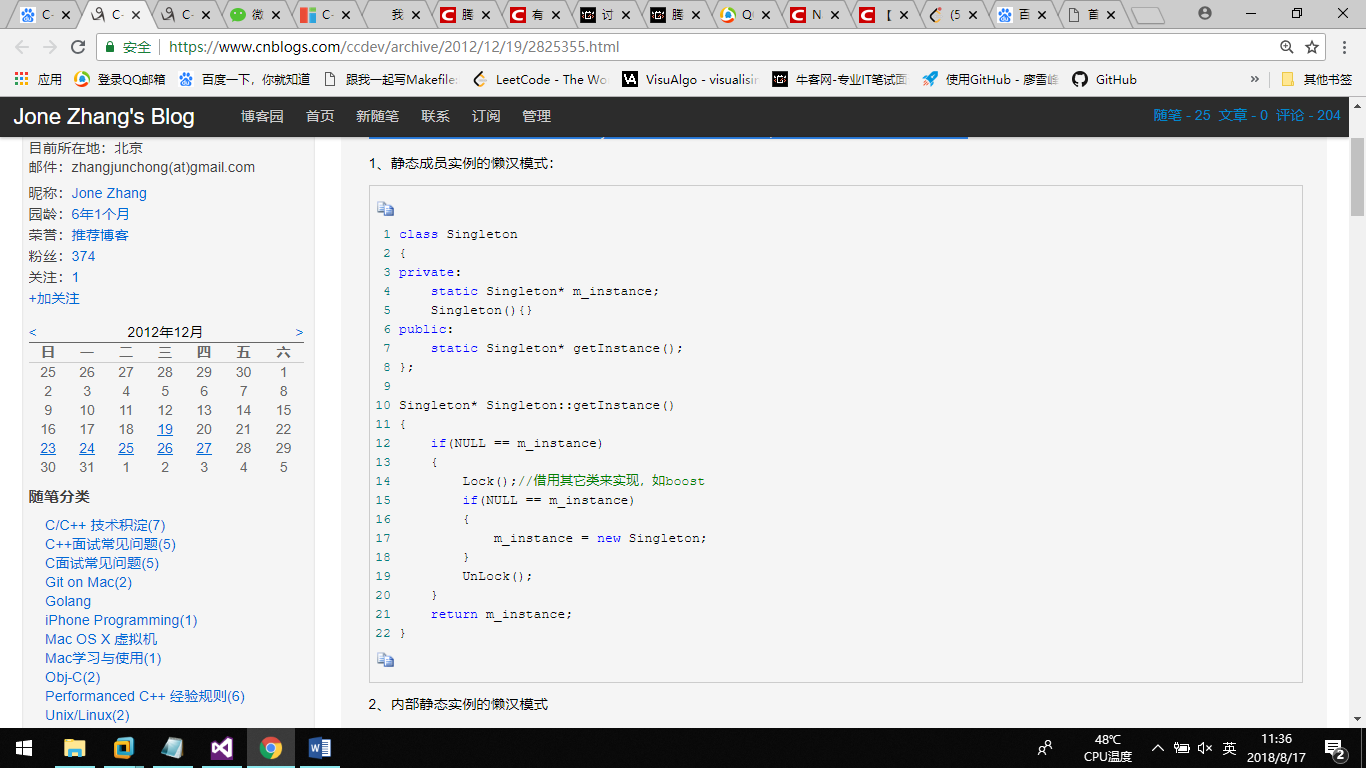


以上代码简单来说，就是返回一个TestObject的类对象。TestFunction中永远返回一个静态对象obj。 那么现在重点来了，你必须知道两点：   
1)obj是在函数TestFunction第一次被调用的时候才会调用构造函数   
2)obj在应用程序启动的时候，obj对象内存中的值都为0。并且这里的obj在初始化的时候（这里可以认为调用构造函数）是非线程安全的。（这个问题网上资料有限，个人认知程度短浅，请大家谅解）

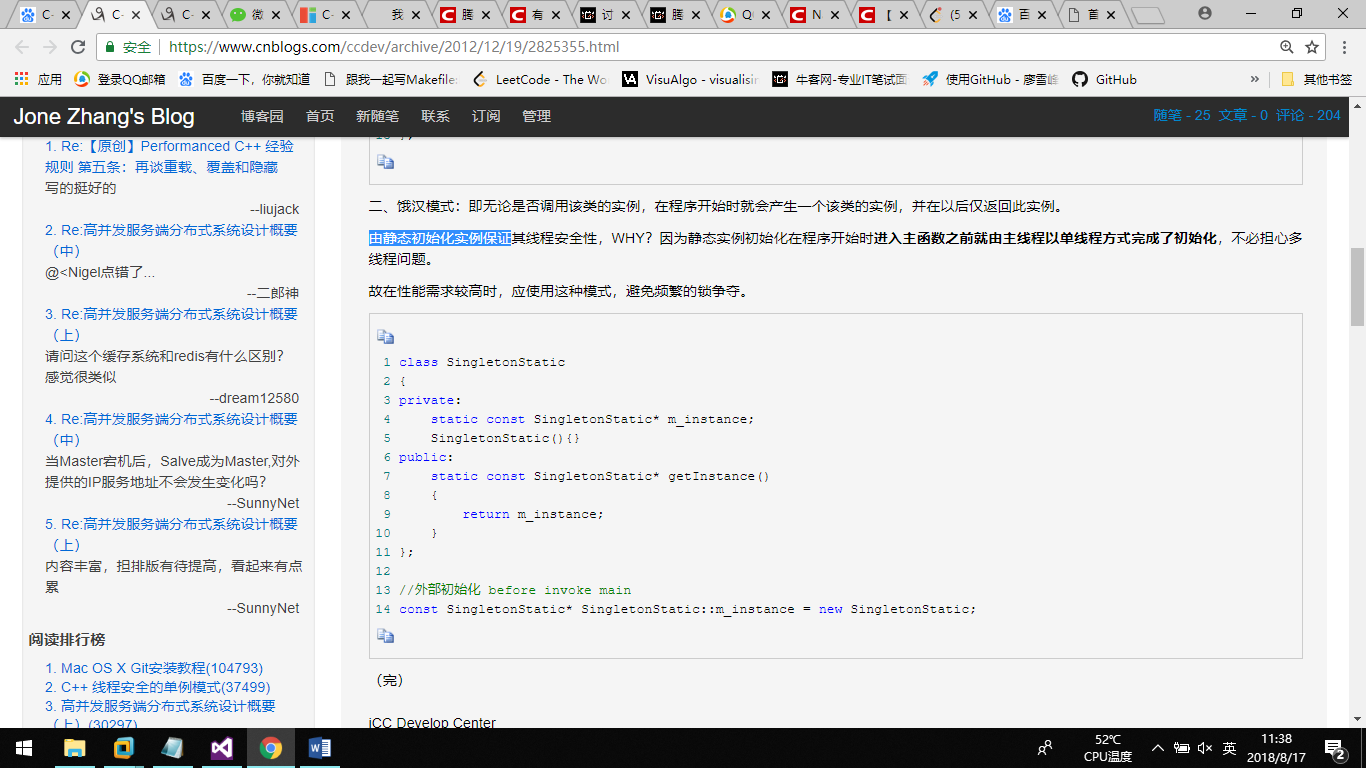
**4.C++线程安全的单例类**

在某些应用环境下面，一个类只允许有一个实例，这就是著名的单例模式。单例模式分为懒汉模式，跟饿汉模式两种：

1、懒汉模式：即第一次调用该类实例的时候才产生一个新的该类实例，并在以后仅返回此实例。需要用锁，来保证其线程安全性：原因：多个线程可能进入判断是否已经存在实例的if语句，从而non thread safety.使用double-check来保证thread safety.但是如果处理大量数据时，该锁才成为严重的性能瓶颈。



饿汉模式：



**5.多线程环境带有状态的对象的讨论**

1)线程安全

要搞清楚有状态对象和无状态对象，首先需要弄清楚线程安全的问题。如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，那么就是线程安全的。

或者说，一个类或者程序所提供的接口对于线程来说是原子操作或者多个线程之间的切换不会导致该接口的执行结果存在二义性，也就是说我们不用考虑同步的问题。

　　线程安全问题都是由全局变量及静态变量引起的。 若每个线程中对全局变量、静态变量只有读操作，而无写操作，一般来说，这个全局变量是线程安全的；若有多个线程同时执行写操作，一般都需要考虑线程同步，否则就可能影响线程安全。

2)关于线程安全

常量始终是线程安全的，因为只存在读操作。

每次调用方法前都新建一个实例是线程安全的，因为不会访问共享的资源（共享堆内存）。

局部变量是线程安全的。因为每执行一个方法，都会在独立的空间创建局部变量，它不是共享的资源。局部变量包括方法的参数变量和方法内变量。

3)有状态和无状态对象

有状态就是有数据存储功能。有状态对象(Stateful Bean)，就是有实例变量的对象 ，可以保存数据，是非线程安全的。在不同方法调用间不保留任何状态。其实就是有数据成员的对象。

无状态就是一次操作，不能保存数据。无状态对象(Stateless Bean)，就是没有实例变量的对象。不能保存数据，是不变类，是线程安全的。具体来说就是只有方法没有数据成员的对象，或者有数据成员但是数据成员是可读的对象。

4)代码示例

/\*\*

\* 有状态bean,有state,user等属性，并且user有存偖功能，是可变的。

\*/

public class StatefulBean {

public int state;

// 由于多线程环境下，user是引用对象，是非线程安全的

public User user;

public int getState() {

return state;

}

public void setState(int state) {

this.state = state;

}

public User getUser() {

return user;

}

public void setUser(User user) {

this.user = user;

}

}

/\*\*

\* 无状态bean,不能存偖数据。因为没有任何属性，所以是不可变的。只有一系统的方法操作。

\*/

public class StatelessBeanService {

// 虽然有billDao属性，但billDao是没有状态信息的，是Stateless Bean.

BillDao billDao;

public BillDao getBillDao() {

return billDao;

}

public void setBillDao(BillDao billDao) {

this.billDao = billDao;

}

public List<User> findUser(String Id) {

return null;

}

}

**6.C++多线程加volatile的错误认识**

volatile解释：

变量可以被某些编译器的未知因素修改，操作系统、其他线程、硬件等，用volatile修饰的变量，编译器无法进行优化，只能访问改变量的内存地址，表明系统总是从该变量的内存中读取。

多线程下使用volatile：

当多个线程使用某一个变量且变量会改变时，应将变量用volatile声明，可以防止编译器将变量从内存中放入寄存器中，保证系统每次真正从内存中取出变量：

volatile bool bstop=false;

线程1：

while(!bstop){}//若没有volatile关键字，则编译器的优化做法将false值装入寄存器，方便每次读取，但这样会陷入死循环，其他线程改变了bstop值，线程1也觉察不到。

bstop=false;

return;

线程2：

bstop = true;

while(bstop);

//若没有volatile，线程2本来该终止线程1循环，但由于编译器优化做法，导致线程1进入死循环。

**7.并行编程中多进程和多线程，什么情况下多进程能解决的多线程无法解决**

多进程程序，一个进程崩溃不会影响到其他进程，但进程的切换和通信成本高，因此不适合太多数据交互的程序。

多线程程序，一个线程崩溃导致整个进程死掉，但线程之间的切换简单，通信方便，适合需要频繁交互数据频繁对一个对象进行不同处理的程序。

通过引进线程，一个任务可以分成多个子任务进行，提高了CPU利用率。

**8.如何证明一个数据结构是线程安全的**

一个不论运行时（Runtime）如何调度线程都不需要调用方提供额外的同步和协调机制还能正确地运行的类是线程安全的多线程的场景很多很复杂，难以穷尽地说那些条件下是或者不是线程安全的，但是有一些常用的肯定线程安全的场景：1)无状态的一定是线程安全的。这个很好理解，因为所谓线程不安全也就是一个线程修改了状态，而另一个线程的操作依赖于这个被修改的状态。2)只有一个状态，而且这个状态是由一个线程安全的对象维护的，那这个类也是线程安全的。比如你在数据结构里只用一个AtomicLong来作为计数器，那递增计数的操作都是线程安全的，不会漏掉任何一次计数，而如果你用普通的long做++操作则不一样，因为++操作本身涉及到取数、递增、赋值 三个操作，某个线程可能取到了另外一个线程还没来得及写回的数就会导致上一次写入丢失。3)有多个状态的情况下，维持不变性（invariant）的所有可变（mutable）状态都用同一个锁来守护的类是线程安全的。这一段有些拗口，首先类不变性的意思是指这个类在多线程状态下能正确运行的状态，其次用锁守护的意思是所有对该状态的操作都需要获取这个锁，而用同一个锁守护的作用就是所有对这些状态的修改实际最后都是串行的，不会存在某个操作中间状态被其他操作可见，继而导致线程不安全。所以这里的关键在于如何确定不变性，可能你的类的某些状态对于类的正确运行是无关紧要的，那就不需要用和其他状态一样的锁来守护。因此我们常可以看到有的类里面会创建一个新的对象作为锁来守护某些和原类本身不变性无关的状态。

**9、lock-free的实现方式**

（1）lock-free定义：多线程中不会导致线程间相互阻塞，称之为lock-free。不使用锁结构，可以降低线程间互相阻塞的机会。

（2）所谓lock-free的实现，实际上就是在不使用锁结构的条件下，实现线程安全。实现方式可以采用C++11中的Atomic，。

（3）Atomic：

Atomic一系列原子操作类，它们提供的方法能保证具有原子性。这些方法是不可再分的，获取这些变量的值时，永远获得修改前的值或修改后的值，不会获得修改过程中的中间数值。

这些类都禁用了拷贝构造函数，原因是原子读和原子写是2个独立原子操作，无法保证2个独立的操作加在一起仍然保证原子性。

atomic<T>提供了常见且容易理解的方法：

1）store： store是原子写操作

2）load： load则是对应的原子读操作

3）exchange：允许2个数值进行交换，并保证整个过程是原子的。

4）compare\_exchange\_weak

5）compare\_exchange\_strong

compare\_exchange\_weak和compare\_exchange\_strong要求在参数中传入期待的数值和新的数值。它们对比变量的值和期待的值是否一致，如果是，则替换为用户指定的一个新的数值。如果不是，则将变量的值和期待的值交换。

weak版本允许偶然出乎意料的返回（比如在字段值和期待值一样的时候却返回了false），不过在一些循环算法中，这是可以接受的。通常它比起strong有更高的性能。

（4）Atomic的简单使用：

定义一个具有原子性操作的链表并在头节点前面插入一个节点：

std::atomic<node\*> head;

node\* const new\_node=new node(data);

new\_node->next=head.load();

（5）利用Atomic实现一个lock-free的栈：

参考博文：https://blog.csdn.net/alien\_taiji/article/details/53404176 https://www.cnblogs.com/dengzz/p/5686866.html

**10、锁的实现方式（不太明确该怎么回答）**

在硬件层面，CPU提供了原子操作、关中断、锁内存总线的机制；OS基于这几个CPU硬件机制，就能够实现锁；再基于锁，就能够实现各种各样的同步机制。

所谓的锁，说白了就是内存中的一个整型数，拥有两种状态：空闲状态和上锁状态。加锁时，判断锁是否空闲，如果空闲，修改为上锁状态，返回成功；如果已经上锁，则返回失败。解锁时，则把锁状态修改为空闲状态。

OS是怎样保证这个锁操作本身的原子性呢？举个例子，在多核环境中，两个核上的代码同时申请一个锁，两个核同时取出锁变量，同时判断说这个锁是空闲状态，然后又同时修改为上锁状态，同时返回成功。这样的话就可能出现两个核同时获取到了锁，OS是怎么避免这种情况的呢？

为了弄明白锁的实现原理，首先看看如果OS不采用任何其他手段，什么情况下会导致上锁失败？假如把加锁过程用如下伪码表示：

1）read lock；

2）判断lock状态；

3）如果已经加锁，失败返回；

4）把锁状态设置为上锁；

5）返回成功。

上述每一步都对应到一条汇编语句，所以我们可以认为每一步本身是原子的。

那么什么情况能够导致两个线程同时获取到锁呢？

1）中断：假设线程A执行完第一步，发生中断，中断返回后，OS调度线程B，线程B也来加锁并且加锁成功，这时OS调度线程A执行，线程从第二步开始执行，也加锁成功。

2）多核：当然了，想想上面举的例子，描述的就是两个核同时获取到锁的情况。

既然明白锁失败的原因，解决手段就很明确了：

先考虑单核场景：

1）既然只有中断才能把上锁过程打断，造成多线程操作失败。我先关中断不就得了，在加锁操作完成后再开中断。

2）上面这个手段太笨重了，能不能硬件做一种加锁的原子操作呢？能，大名鼎鼎的“test and set”指令就是做这个事情的。

针对多核环境，“test and set”指令可能同时被两个核执行，从而破坏其原子性，所以不能采用这种方式进行。怎么办呢？首先我们得明白这个地方的关键点，关键点是两个核会并行操作内存而且从操作内存这个调度来看“test and set”不是原子的，需要先读内存然后再写内存，如果我们保证这个内存操作是原子的，就能保证锁的正确性了。确实，硬件提供了锁内存总线的机制，我们在锁内存总线的状态下执行test and set操作，就能保证同时只有一个核来test and set，从而避免了多核下发生的问题。

参考博文：https://blog.csdn.net/Onlyonefate/article/details/52183908

**11、多线程编程的时候，使用无锁结构会不会比有锁结构更加快**

具体情况需具体分析。

无论有锁（mutex）还是无锁（lock-free）总是只有一个线程在执行任务，只不过对于lock-free所有线程都可以进临界区。所以从这点上看其实有锁和无锁在性能上应该是一样的。

但是从另一点上是有区别的，他们的不同体现在拿不到锁的态度：有锁的情况就是睡觉，无锁的情况就不断spin。

睡觉这个动作会陷入内核，发生context switch，这个是有开销的，但是这个开销能有多大呢，当你的临界区很小的时候，这个开销的比重就非常大。这也是为什么临界区很小的时候，换成lockfree性能通常会提高很多的原因。

再来看lockfree的spin，一般都遵循一个固定的格式：先把一个不变的值X存到某个局部变量A里，然后做一些计算，计算/生成一个新的对象，然后做一个CAS操作，判断A和X还是不是相等的，如果是，那么这次CAS就算成功了，否则再来一遍。如果上面这个loop里面“计算/生成一个新的对象”非常耗时并且contention很严重，那么lockfree性能有时会比mutex差。另外lockfree不断地spin引起的CPU同步cacheline的开销也比mutex版本的大。

lockfree的意义不在于绝对的高性能，它比mutex的优点是使用lockfree可以避免死锁/活锁，优先级翻转等问题。但是因为ABA problem、memory order等问题，使得lockfree比mutex难实现得多。

**12、linux线程是如何进行切换的**

（1）一般的进程切换分为两步 ：

1）切换页目录使用新的地址空间

2）切换内核栈和硬件上下文

对于Linux来讲，地址空间是线程和进程的最大区别，如果是线程切换的话，不需要做第一步，也就是切换页目录使用新的地址空间。但是切换内核栈和硬件上下文则是线程切换和进程切换都需要做的。

（2）切换进程上下文：

进程上下文可以分为三个部分:

用户级上下文: 正文、数据、用户堆栈以及共享存储区；

寄存器上下文: 通用寄存器、程序寄存器(IP)、处理器状态寄存器(EFLAGS)、栈指针(ESP)；

系统级上下文: 进程控制块task\_struct、内存管理信息(mm\_struct、vm\_area\_struct、pgd、pte)、内核栈。

系统中的每一个进程都有自己的上下文。一个正在使用处理器运行的进程称为当前进程(current)。当前进程因时间片用完或者因等待某个事件而阻塞时，进程调度需要把处理器的使用权从当前进程交给另一个进程，这个过程叫做进程切换。此时，被调用进程成为当前进程。在进程切换时系统要把当前进程的上下文保存在指定的内存区域（该进程的任务状态段TSS中），然后把下一个使用处理器运行的进程的上下文设置成当前进程的上下文。当一个进程经过调度再次使用CPU运行时，系统要恢复该进程保存的上下文。所以，进程的切换也就是上下文切换。

（3）线程切换：

Linux下的线程实质上是轻量级进程(light weighted process),线程生成时会生成对应的进程控制结构，只是该结构与父线程的进程控制结构共享了同一个进程内存空间。同时新线程的进程控制结构将从父线程（进程）处复制得到同样的进程信息，如打开文件列表和信号阻塞掩码等。创建线程比创建新进程成本低，因为新创建的线程使用的是当前进程的地址空间。相对于在进程之间切换，在线程之间进行切换所需的时间更少，因为后者不包括地址空间之间的切换。

线程切换上下文切换的原理与此类似，只是线程在同一地址空间中，不需要MMU等切换，只需要切换必要的CPU寄存器，因此，线程切换比进程切换快的多。

**13、Linux 开发，使用多线程还是用 IO 复用 select/epoll（此问题存在一定争论，并未找到更好的答案，希望各位大佬指正）**

多线程模型适用于处理短连接，且连接的打开关闭非常频繁的情形，但不适合处理长连接。多线程模型默认情况下，（在Linux）每个线程会开8M的栈空间，假定有10000个连接，开这么多个线程需要80G的内存空间！即使调整每个线程的栈空间，也很难满足更多的需求。攻击者可以利用这一点发动DDoS，只要一个连接连上服务器什么也不做，就能吃掉服务器几M的内存，这不同于多进程模型，线程间内存无法共享，因为所有线程处在同一个地址空间中。内存是多线程模型的软肋。

在UNIX平台下多进程模型擅长处理并发长连接，但却不适用于连接频繁产生和关闭的情形。Windows平台忽略此项。 同样的连接需要的内存数量并不比多线程模型少，但是得益于操作系统虚拟内存的Copy on Write机制，fork产生的进程和父进程共享了很大一部分物理内存。但是多进程模型在执行效率上太低，接受一个连接需要几百个时钟周期，产生一个进程 可能消耗几万个CPU时钟周期，两者的开销不成比例。而且由于每个进程的地址空间是独立的，如果需要进行进程间通信的话，只能使用IPC进行进程间通 信，而不能直接对内存进行访问。在CPU能力不足的情况下同样容易遭受DDos，攻击者只需要连上服务器，然后立刻关闭连接，服务端则需要打开一个进程再关闭。

同时需要保持很多的长连接，而且连接的开关很频繁，最高效的模型是非阻塞、异步IO模型。而且不要用select/poll，这两个API的有着O(N)的时间复杂度。在Linux用epoll，BSD用kqueue，Windows用IOCP，或者用libevent封装的统一接口（对于不同平台libevent实现时采用各个平台特有的API），这些平台特有的API时间复杂度为O(1)。然而在非阻塞，异步I/O模型下的编程是非常痛苦的。由于I/O操作不再阻塞，报文的解析需要小心翼翼，并且需要亲自管理维护每个链接的状态。并且为了充分利用CPU，还应结合线程池，避免在轮询线程中处理业务逻辑。

参考知乎：http://www.zhihu.com/question/20114168/answer/14024115

**14、异步，多线程和并行的区别**

（1）并发：在[操作系统](http://baike.baidu.com/view/880.htm)中，是指一个时间段中有几个程序都处于已启动运行到运行完毕之间，且这几个程序都是在同一个[处理机](http://baike.baidu.com/view/2107226.htm)上运行。其中两种并发关系分别是同步和互斥

其中并发又有伪并发和真并发，伪并发是指单核处理器的并发，真并发是指多核处理器的并发。

（2）并行：在单处理器中多道程序设计系统中，进程被交替执行，表现出一种并发的外部特种；在多处理器系统中，进程不仅可以交替执行，而且可以重叠执行。在多处理器上的程序才可实现并行处理。从而可知，并行是针对多处理器而言的。并行是同时发生的多个并发事件，具有并发的含义，但并发不一定并行，也亦是说并发事件之间不一定要同一时刻发生。

（3）互斥：进程间相互排斥的使用临界资源的现象，就叫互斥。

（4）同步：进程之间的关系不是相互排斥临界资源的关系，而是相互依赖的关系。进一步的说明：就是前一个进程的输出作为后一个进程的输入，当第一个进程没有输出时第二个进程必须等待。具有同步关系的一组并发进程相互发送的信息称为消息或事件。

（5）异步：异步和同步是相对的，同步就是顺序执行，执行完一个再执行下一个，需要等待、协调运行。异步就是彼此独立,在等待某事件的过程中继续做自己的事，不需要等待这一事件完成后再工作。线程就是实现异步的一个方式。异步是让调用方法的主线程不需要同步等待另一线程的完成，从而可以让主线程干其它的事情。

（6）多线程：多线程是程序设计的逻辑层概念，它是进程中并发运行的一段代码。多线程可以实现线程间的切换执行。

**15、Linux 下多线程和多进程程序的优缺点，各自适合什么样的业务场景**

（1）多线程优点：

1）多线程更快捷：每个线程共享一个虚拟地址空间，因此从调度开销方面考虑多线程占优。

2）数据共享方便：由于同一进程下的线程之间共享数据空间，所以一个线程的数据可以直接为其它线程所用，快捷而且方便；

3）使多CPU系统更加有效。操作系统会保证当线程数不大于CPU数目时，不同的线程运行于不同的CPU上；

4）调度开销更小，不用切换地址空间。

（2）多线程缺点：

1）稳定性差：一个线程挂掉整个程序挂。

2）数据脏：线程间由于共享内存空间而导致数据脏，需要加锁实现数据同步，导致性能低下。

3）线程数量收到限制：每个线程与主程序共用地址空间，受限于4GB地址空间，此时再增加CPU也不能提高性能。

（3）多进程优点：

1）多进程更稳定：一个进程挂掉不会影响其它进程。

2）数据干净：进程之间的内存空间相对独立，各个进程之间的变量不会相互影响。

3）更好的多核可伸缩性：每个子进程都有4GB地址空间和相关资源，因此通过增加CPU，就可以容易扩充性能；

（4）多进程缺点：

1）开销更大：每个进程都维护专属的虚拟地址空间，需要额外的开销

2）数据共享困难：各进程间的数据相对独立，不易共享。

3）调度开销大：每次调度都需要切换地址空间，切换进程上下文

（5）适用场景：

1）不同任务间需要大量共享数据或频繁通信时，采用多线程。

2）需要提供非均质的服务（有优先级任务处理）事件响应有优先级，采用多线程

3）与人有IO交互的应用，良好的用户体验（键盘鼠标的输入，立刻响应）

4）如果工作集较大，就用多线程，避免cpu cache频繁的换入换出；比如memcached缓存系统；

5）需要频繁创建销毁的优先用线程

6）需要进行大量计算的优先使用线程

7）强相关的处理用线程，弱相关的处理用进程

8）可能要扩展到多机分布的用进程，多核分布的用线程

注：RCU的发明者，Paul McKenny说过：能用多进程方便的解决问题的时候不要使用多线程。

16、开发多线程的程序应该注意哪些问题

多线程的主要是需要处理大量的IO操作或者处理的情况需要花大量的时间等等，比如读写文件，网络数据接收，视频图像的采集，处理显示保存等操作缓慢的情形和需大幅度的提高性能的程序中使用。

但也不是都使用多线程，因为多线程过多的线程一般会导致数据共享问题，太多多线程切换也是会影响性能的，所以一般不须采用多线程的不用多线程效果更好。

# 1 线程间通信

线程间通信主要涉及在线程间传递数据，或相互通知某些事件的完成。可采用的方法包括：

方法1：全局变量

方法2：发送消息

# **2 线程间的同步与互斥**

多线程会涉及对共享资源或独占资源的访问，也就引入了同步与互斥的问题。

假设多个线程都要修改同一个全局变量：

互斥——同一时刻只能有一个线程修改该变量，但谁先处理谁后处理无所谓。

同步——同一时刻只能有一个线程修改该变量，但其中一个线程要等待另一个线程处理完之后才能处理。

即，同步是有先后顺序要求的互斥。

3.线程使用中要注意，如何控制线程的调度和阻塞，例如利用事件的触发来控制线程的调度和阻塞，也有用消息来控制的。   
4、线程中如果用到公共资源，一定要考虑公共资源的线程安全性。一般用LOCK锁机制来控制线程安全性。一定要保证不要有死锁机制。

5、线程的终止一般要使线程体在完成一件工作的情况下终止，一般不要直接使用抛出线程异常的方式终止线程。   
6、线程的优先级一定根据程序的需要要有个整体的规划。

7、注意条件返回时互斥锁的解锁问题

8、正确处理 Linux 平台下的线程结束问题   
在 Linux 平台下，当处理线程结束时需要注意的一个问题就是如何让一个线程善始善终，让其所占资源得到正确释放。在 Linux 平台默认情况下，虽然各个线程之间是相互独立的，一个线程的终止不会去通知或影响其他的线程。但是已经终止的线程的资源并不会随着线程的终止而得到释放，我们需要调用 pthread\_join() 来获得另一个线程的终止状态并且释放该线程所占的资源

9等待的绝对时间问题   
超时是多线程编程中一个常见的概念。例如，当你在 Linux 平台下使用 pthread\_cond\_timedwait() 时就需要指定超时这个参数，以便这个 API 的调用者最多只被阻塞指定的时间间隔。

17、如何测试线程池的性能

18、死锁的原因和避免

### 一、死锁的概念

所谓死锁，是指多个进程在运行过程中因争夺资源而照成的一种僵局。当进程处于这种僵持状态时，若无外力作用，它们都将无法再向前推进。

### 二、产生死锁的原因

（1）竞争资源。当系统中供多个进程共享的资源如打印机、公用队列等，其数目不足以满足诸进程的需要时，会引起诸进程对资源的竞争而产生死锁。   
（2）进程间推进顺序非法。进程在运行过程中，请求和释放资源的顺序不当，也同样会产生进程死锁。

### 三、产生死锁的必要条件

（1）互斥条件：指进程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求该资源，则请求者只能等待，直至占有该资源的进程用毕释放。   
（2）请求和保持条件：指进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源又被其它进程占有，此时请求进程阻塞，但又对自己获得的其它资源保持不放。   
（3）不剥夺条件：指进程已获得资源，在使用完之前，不能被剥夺，只能在使用完时由自己释放。   
（4）环路等待条件：指在发生死锁时，必然存在一个进程—资源的环形链，即进程集合（P0，P1,P2,…,Pn）中的P0正在等待一个P1占用的资源；P1正在等待一个P2占用的资源，……,Pn正在等待已被P0占用的资源。

处理死锁的策略

1、忽略该问题。例如**鸵鸟算法**。

 2、检测死锁并且恢复。  
3、仔细地对资源进行动态分配，以避免死锁。  
4、通过破除死锁四个必要条件之一，来防止死锁产生。

**鸵鸟算法：**

该算法可以应用在极少发生死锁的的情况下。为什么叫鸵鸟算法呢，因为传说中鸵鸟看到危险就把头埋在地底下，可能鸵鸟觉得看不到危险也就没危险了吧。跟掩耳盗铃有点像。

**银行家算法：**

        所谓银行家算法，是指在分配资源之前先看清楚，资源分配后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) 当一个进程对资源的最大需求量不超过系统中的资源数时可以接纳该进程。

(2) 进程可以分期请求资源，当请求的总数不能超过最大需求量。

(3) 当系统现有的资源不能满足进程尚需资源数时，对进程的请求可以推迟分配，但总能使进程在有限的时间里得到资源。

(4) 当系统现有的资源能满足进程尚需资源数时，必须测试系统现存的资源能否满足该进程尚需的最大资源数，若能满足则按当前的申请量分配资源，否则也要推迟分配。

**解决死锁的策略**

对待死锁的策略主要有：

(1) 死锁预防：破坏导致死锁必要条件中的任意一个就可以预防死锁。例如，要求用户申请资源时一次性申请所需要的全部资源，这就破坏了保持和等待条件；将资源分层，得到上一层资源后，才能够申请下一层资源，它破坏了环路等待条件。预防通常会降低系统的效率。

(2) 死锁避免：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。

(3) 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。

(4) 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。

**死锁的避免：**死锁的预防是通过破坏产生条件来阻止死锁的产生，但这种方法破坏了系统的并行性和并发性。  
死锁产生的前三个条件是死锁产生的必要条件，也就是说要产生死锁必须具备的条件，而不是存在这3个条件就一定产生死锁，那么只要在逻辑上回避了第四个条件就可以避免死锁。  
避免死锁采用的是允许前三个条件存在，但通过合理的资源分配算法来确保永远不会形成环形等待的封闭进程链，从而避免死锁。该方法支持多个进程的并行执行，为了避免死锁，系统动态的确定是否分配一个资源给请求的进程。方法如下：  
1.如果一个进程的当前请求的资源会导致死锁，系统拒绝启动该进程；  
2.如果一个资源的分配会导致下一步的死锁，系统就拒绝本次的分配；  
显然要避免死锁，必须事先知道系统拥有的资源数量及其属性

19、如何理解互斥锁，条件锁，读写锁以及自旋锁

**读写锁特点：**

**1）多个读者可以同时进行读**

**2）写者必须互斥（只允许一个写者写，也不能读者写者同时进行）**

**3）写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）**

**互斥锁特点：**

**一次只能一个线程拥有互斥锁，其他线程只有等待**

**互斥锁是在抢锁失败的情况下主动放弃CPU进入睡眠状态直到锁的状态改变时再唤醒，而操作系统负责线程调度，为了实现锁的状态发生改变时唤醒阻塞的线程或者进程，需要把锁交给操作系统管理，所以互斥锁在加锁操作时涉及上下文的切换。互斥锁实际的效率还是可以让人接受的，加锁的时间大概100ns左右，而实际上互斥锁的一种可能的实现是先自旋一段时间，当自旋的时间超过阀值之后再将线程投入睡眠中，因此在并发运算中使用互斥锁（每次占用锁的时间很短）的效果可能不亚于使用自旋锁。**

**条件变量的特点：**

**互斥锁一个明显的缺点是他只有两种状态：锁定和非锁定。而条件变量通过允许线程阻塞和等待另一个线程发送信号的方法弥补了互斥锁的不足，他常和互斥锁一起使用，以免出现竞态条件。当条件不满足时，线程往往解开相应的互斥锁并阻塞线程然后等待条件发生变化。一旦其他的某个线程改变了条件变量，他将通知相应的条件变量唤醒一个或多个正被此条件变量阻塞的线程。总的来说互斥锁是线程间互斥的机制，条件变量则是同步机制。**

**自旋锁的特点：**

**如果进线程无法取得锁，进线程不会立刻放弃CPU时间片，而是一直循环尝试获取锁，直到获取为止。如果别的线程长时期占有锁那么自旋就是在浪费CPU做无用功，但是自旋锁一般应用于加锁时间很短的场景，这个时候效率比较高。**

20、互斥锁，同步锁，临界区，互斥量，信号量，自旋锁之间联系是什么

1、临界区：通过对多线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。   
2、互斥量：为协调共同对一个共享资源的单独访问而设计的，互斥对象只有一个。   
3、信号量：为控制一个具有有限数量用户资源而设计，只能在进程上下文中使用，适合长时间访问共享资源的情况   
4、自旋锁：适合短时间访问共享资源的情况，如果锁被长时间持有，等待线程会消耗大量资源   
5、事件：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。

21、pthread\_cond\_wait 为什么需要传递 mutex 参数

pthread\_cond\_wait 函数用于等待目标条件变量。mutex参数用于保护条件变量的互斥锁，以确保pthread\_cond\_wait 操作的原子性，在调用pthread\_cond\_wait前，必须确保互斥锁mutex已经加锁，否则将导致不可预期的结果。pthread\_cond\_wait函数执行时，首先把调用线程放入条件变量的等待队列中，然后将互斥锁mutex解锁。可见，从ptread\_cond\_wait开始执行到其调用线程被放入条件变量的等待队列之间的这段时间内，ptread\_cond\_signal 和pthread\_cond\_broadcast 函数不会修改条件变量，换言之，pthread\_cond\_wait 函数不会错过目标条件变量的任何变化，当pthread\_cond\_wait 函数成功返回时，互斥锁mutex将被再次锁上。

22、多线程网络编程中如何合理地选择线程数

# 确定最佳线程数量

首先确定应用是CPU密集型 （例如分词，加密等），还是耗时io（ 网络，文件操作等）

CPU密集型：最佳线程数等于cpu核心数或稍微小于cpu核心数.

耗时io型：最佳线程数一般会大于cpu核心数很多倍。一般是io设备延时除以cpu处理延时，得到一个倍数，我的经验数值是20–50倍\*cpu核心数,保证线程空闲可以衔接上。

最佳线程数量也与机器配置（内存，磁盘速度）有关，如果cpu，内存，磁盘任何一个达到顶点，就需要适当减少线程数。

**23.malloc和free是线程安全的吗，在多线程开发用这两个函数应该注意什么**

是线程安全的，但是不可重入的。

如果并发量高，分配频繁，可以考虑使用tcmalloc。tcmalloc是用于优化C++写的多线程应用。可以使得程序在高并发下内存占用更加稳定。

在多线程高并发时候最好使用线程池，或者是直接一次性分配好内存，后面复用。

malloc/free会导致系统用户态/核心态切换，消耗大。

Malloc/free线程安全意味着它们要加锁，可以看到任务管理器的锯齿形状

不断的malloc/free运行久了会有内存碎片。

**用到的概念，线程安全，可重入函数**

**线程安全：**在拥有共享数据的多条线程并行执行的程序中，线程安全的代码会通过同步机制保证各个线程都可以正常并且正确的执行，不会出现数据污染等情况。如果每次运行的结果和单线程运行的结果是一样的，而且其他变量的值和预期也是一样的，就是线程安全的。

**可重入函数：**可重入函数可以有多于一个任务并发使用，而不必担心数据错误。不可充数函数不能超过一个数据共享，除非能确保函数的互斥，或者使用信号量，或者在代码的关键部分禁用中断。可重入函数可以在任何时候被中断，稍后继续运行，不会丢失数据。可重入函数要么使用本地，要么使用全局变量保护数据。

**24.僵尸进程和孤儿进程有什么区别，如何处理**

**区别**：僵尸进程占用一个进程ID号，占用资源，危害系统。但孤儿进程与僵尸进程不同的是，由于父进程已经死亡，子系统会帮助父进程回收处理孤儿进程。所以孤儿进程实际上是不占用资源的，因为它最终是被系统回收了，不会像僵尸进程那样占用ID，损害运行系统。

**僵尸进程**：一个进程使用fork创建子进程，如果子进程退出，而父进程没有调用wait或者waitpid获取子进程的状态，那么子进程的进程描述符仍然保存在系统中，这种进程称为僵死进程。

**孤儿进程**：一个父进程退出，而他的一个或者多个子进程还在运行，那么那些子进程称为孤儿进程。孤儿进程将被init（进程号为1）收养，并由init进程对它们完成状态收集的工作。子进程的死亡需要父进程来处理，所以正常的进程应该是子进程先于父进程死亡，当父进程先于子进程死亡时，子进程死亡没有父进程处理，这个死亡的子进程就是孤儿进程。

**25.Linux系统中进程，线程，时间片的关系**

在Linux系统中，对于用户创建的进程（线程）来说，CPU分配时间片的单位是线程，线程使实际工作的单元，进程只是一个容器，用来管理一个或多个线程。

所以在理论上应该尽量使用多线程并发，这样可以抢到更多的时间片，但是实际问题是当线程数过多时，操作系统必须进行调度，也就是分配时间片。线程被调度的时候需要进行上下文的切换，这种操作是一种额外的开销。线程数量过多时，上下文切换会产生额外的开销，对系统效率造成负面的影响。

线程和进程有优先级，在抢占式的调度中，优先级高的线程可以冲优先级低的线程那里抢占CPU。另外，在多CPU平台上，调度算法还要考虑缓存的关联性。

26、在Linux系统中，对于用户创建的进程(线程)来说，CPU分配时间片的单位是线程还是进程？

是线程。线程是实际工作的单元，进程只是一个容器，用来管理一个或多个线程。

拓展1：这是不是就意味着尽量使用多线程并发，这样可以抢到更多的时间片。

理论上是的，多线程的一种用途就是能同时做好几件事情，以提高效率。但实际问题是，CPU的数量（核心数）是有限的，而且并不多。如果你的CPU有8个CPU，并且整个系统中有8个线程的话，不考虑中断等因素，每个线程理论上能一直执行下去。然而多于8个线程以后，操作系统就必须进行调度，也就是分配时间片。具体的调度算法有很多种。如果一个进程创建了很多线程的话，最多也只有8个能够处于执行的状态，其余的线程必须等待调度。线程被调度的时候需要进行上下文切换，这个操作是一种额外的开销。线程数量过多的时候，上下文切换产生的额外开销会对系统的效率造成负面影响。

拓展2：操作系统对于拥有多线程的进程，是否会减少其每个线程的时间片，或做其他处理来保证公平性？

这就是调度算法需要考虑和优化的问题。比如线程和进程有优先级，在抢占式的调度中，优先级高的线程可以从优先级低的线程那里抢占CPU，同样优先级的线程才会轮转CPU。另外，在多CPU平台上，调度算法还要考虑缓存的关联性等。在一个进程中的多个线程要注意在可能的情况下将本线程阻塞，将剩余的时间片让给兄弟线程。

在主流的操作系统实现里，一般进程是不能直接控制自己的线程的执行顺序的。也就是说，把一个线程挂起并不能保证另一个线程一定能够被执行。

Ps：Linux内核其实不区分进程和线程，内核把执行单元叫做任务(task)。操作系统实际上调度的是进程，进程通过fork()来创建同样的另一个进程。每个进程有一个PID，同一组进程中最先启动的那个进程还有一个TGID。严格来说前者应该叫线程ID，后者应该叫进程ID。Linux里的线程实际上是共享一些资源的一系列进程而已。

27、内核级调度和用户级调度**（我把题目理解成内核级线程和用户级线程来作答）**

内核线程：由操作系统内核创建和撤销，内核空间实现还为每个内核支持线程设置了一个线程控制块，内核是根据该控制块而感知某个线程是否存在，并加以控制。线程切换也由内核控制，切换的时候，要从用户态进入内核态，切换完毕要从内核态返回用户态。可以很好的利用多CPU

用户级线程：仅存在于用户空间中。线程的创建、撤销、线程之间的同步和通信等功能，都无需系统调用来实现。对于同一进程的线程之间切换仍然是不需要内核支持的。因此，内核也不知道用户级线程的存在，少了进出内核态的消耗，但不能很好的利用多CPU。

详解：

在传统的操作系统中，拥有资源和独立调度的基本单位都是进程。在引入线程的操作系统中，线程是独立调度的基本单位，进程是资源拥有的基本单位。在同一进程中，线程的切换不会引起进程切换。在不同进程中进行线程切换,如从一个进程内的线程切换到另一个进程中的线程时，会引起进程切换。

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。在多线程操作系统中，各个系统的实现方式并不相同，在有的系统中实现了用户级线程，有的系统中实现了内核级线程，有的系统是混合型的。

用户级线程(User-LevelThread, ULT)：由应用程序所支持的线程实现, 内核意识不到用户级线程的实现

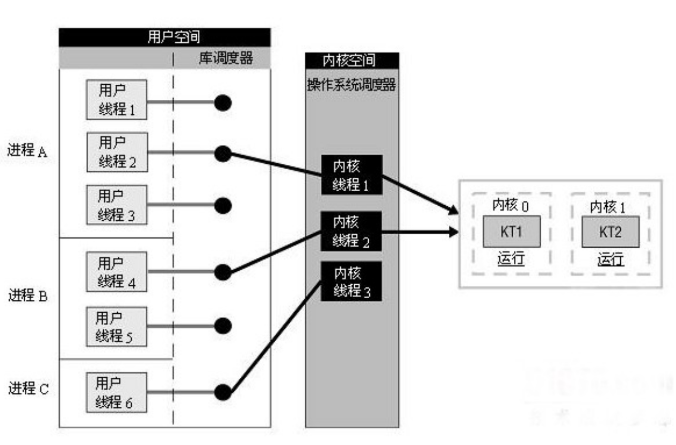
内核级线程(Kemel-LevelThread, KLT)：内核级线程又称为内核支持的线程

1>用户级线程

有关线程管理的所有工作都由应用程序在用户空间中完成，内核意识不到线程的存在。应用程序可以通过使用线程库设计成多线程程序。通常，应用程序从单线程运行开始，在其运行的任何时刻，可以通过调用线程库中的派生例程创建一个在相同进程中运行的新线程。此类线程的创建、撤销、线程之间的同步与通信功能，都无须利用系统调用来实现。

用户级线程对于操作系统是不可见的，因此无法被调度到处理器内核。每个线程并不具有自身的线程上下文。因此，就线程的同时执行而言，任意给定时刻每个进程只能够有一个线程在运行，而且只有一个处理器内核会被分配给该进程。对于一个进程，可能有成千上万个用户级线程，但是它们对系统资源没有影响。运行时库调度并分派这些线程。

下图说明了用户级线程的实现方式。库调度器从进程的多个线程中选择一个线程，然后该线程和该进程允许的一个内核线程关联起来。内核线程将被操作系统调度器指派到处理器内核。用户级线程是一种”多对一”的线程映射。



用户线程的优点：

* 可以在不支持线程的操作系统中实现。
* 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, 因为保存线程状态的过程和调用程序都只是本地过程。
* 允许每个进程定制自己的调度算法，线程管理比较灵活。这就是必须自己写管理程序，与内核线程的区别。
* 线程能够利用的表空间和堆栈空间比内核级线程多。
* 不需要陷阱，不需要上下文切换，也不需要对内存高速缓存进行刷新，使得线程调用非常快捷。
* 线程的调度不需要内核直接参与，控制简单。

用户线程的缺点：

* 线程发生I/O或页面故障引起的阻塞时，如果调用阻塞系统调用则内核由于不知道有多线程的存在，而会阻塞整个进程从而阻塞所有线程, 因此同一进程中只能同时有一个线程在运行。
* 页面失效也会产生类似的问题。
* 一个单独的进程内部，没有时钟中断，所以不可能用轮转调度的方式调度线程。
* 资源调度按照进程进行，多个处理机下，同一个进程中的线程只能在同一个处理机下分时复用。

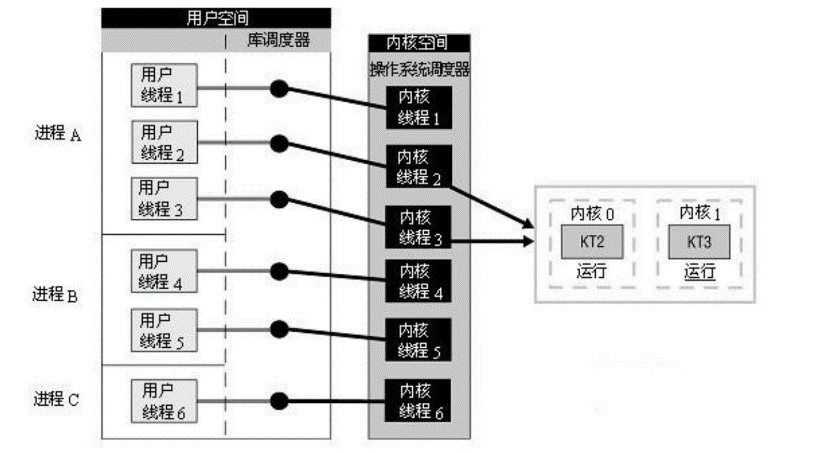
2>内核级线程

在内核级线程中，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。在内核的支持下运行，无论是用户进程的线程，或者是系统进程的线程，他们的创建、撤销、切换都是依靠内核实现的。线程管理的所有工作由内核完成，应用程序没有进行线程管理的代码，只有一个到内核级线程的编程接口。内核为进程及其内部的每个线程维护上下文信息，调度也是在内核基于线程架构的基础上完成。

内核线程驻留在内核空间，它们是内核对象。有了内核线程，每个用户线程被映射或绑定到一个内核线程。用户线程在其生命期内都会绑定到该内核线程。一旦用户线程终止，两个线程都将离开系统。这被称作一对一线程映射，线程的创建、撤销和切换等，都需要内核直接实现，即内核了解每一个作为可调度实体的线程。

这些线程可以在全系统内进行资源的竞争内核空间内为每一个内核支持线程设置了一个线程控制块（TCB），内核根据该控制块，感知线程的存在，并进行控制。

如图所示，即内核级线程的实现方式,每个用户线程都直接与一个内核线程相关联。操作系统调度器管理、调度并分派这些线程。运行时库为每个用户级线程请求一个内核级线程。操作系统的内存管理和调度子系统必须要考虑到数量巨大的用户级线程。您必须了解每个进程允许的线程的最大数目是多少。操作系统为每个线程创建上下文。进程的每个线程在资源可用时都可以被指派到处理器内核。

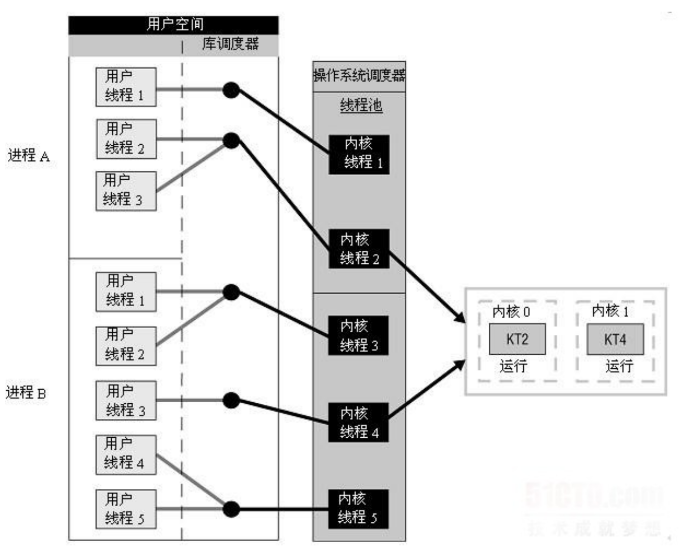


内核线程的优点:

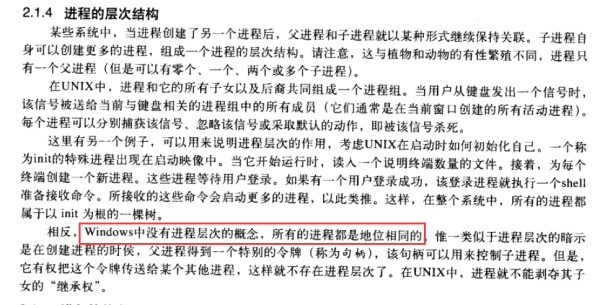
* 多处理器系统中，内核能够并行执行同一进程内的多个线程。
* 如果进程中的一个线程被阻塞，能够切换同一进程内的其他线程继续执行（用户级线程的一个缺点）。
* 所有能够阻塞线程的调用都以系统调用的形式实现，代价可观。
* 当一个线程阻塞时，内核根据选择可以运行另一个进程的线程，而用户空间实现的线程中，运行时系统始终运行自己进程中的线程。
* 信号是发给进程而不是线程的，当一个信号到达时，应该由哪一个线程处理它？线程可以“注册”它们感兴趣的信号。

3>组合型

在一些系统中，使用组合方式的多线程实现，线程创建完全在用户空间中完成，线程的调度和同步也在应用程序中进行。一个应用程序中的多个用户级线程被映射到一些（小于或等于用户级线程的数目）内核级线程上。下图说明了用户级与内核级的组合实现方式，在这种模型中，每个内核级线程有一个可以轮流使用的用户级线程集合。



28、Linux中进程具有父子层次结构，Windows中没有进程层次，这两种设计各有什么优劣？**（优劣没有找到）**



windows进程独立性比较强，每个进程都是独立的个体。虽然事实上每个进程记录了自己的父进程，创建时也继承了父进程的一些信息，但是一旦开跑之后，大家都是平等的关系了。

在windows上，a创建了b，b创建了c，b退出之后，c无法找到自己的爷爷进程。进程链就断了，c成为了一个顶端进程。linux则不同，一般情况下，父进程死掉会影响到子进程。

29、linux用户级进程跟内核线程（进程）有什么差别？**（我把题目理解成内核级线程和用户级线程的区别来作答）**

内核线程：由操作系统内核创建和撤销，内核空间实现还为每个内核支持线程设置了一个线程控制块，内核是根据该控制块而感知某个线程是否存在，并加以控制。线程切换也由内核控制，切换的时候，要从用户态进入内核态，切换完毕要从内核态返回用户态。可以很好的利用多CPU

用户级线程：仅存在于用户空间中。线程的创建、撤销、线程之间的同步和通信等功能，都无需系统调用来实现。对于同一进程的线程之间切换仍然是不需要内核支持的。因此，内核也不知道用户级线程的存在，少了进出内核态的消耗，但不能很好的利用多CPU。

区别：

* 内核级线程是OS内核可感知的，而用户级线程是OS内核不可感知的。
* 用户级线程的创建、撤消和调度不需要OS内核的支持，是在语言（如Java）这一级处理的；而内核支持线程的创建、撤消和调度都需OS内核提供支持，而且与进程的创建、撤消和调度大体是相同的。
* 用户级线程执行系统调用指令时将导致其所属进程被中断，而内核支持线程执行系统调用指令时，只导致该线程被中断。
* 在只有用户级线程的系统内，CPU调度还是以进程为单位，处于运行状态的进程中的多个线程，由用户程序控制线程的轮换运行；在有内核支持线程的系统内，CPU调度则以线程为单位，由OS的线程调度程序负责线程的调度。
* 用户级线程的程序实体是运行在用户态下的程序，而内核支持线程的程序实体则是可以运行在任何状态下的程序。

##### 30、为什么要区分用户态和内核态

在CPU的所有指令中，有些指令是非常危险的，如果错用，将导致系统崩溃，比如清内存、设置时钟等。

如果所有的程序都能使用这些指令，那么系统死机的概率将大大增加。

所以，CPU将指令分为特权指令和非特权指令，对于那些危险的指令，只允许操作系统及其相关模块使用，普通应用程序只能使用那些不会造成灾难的指令。

Intel的CPU将特权等级分为4个级别：Ring0~Ring3

Linux使用Ring3级别运行用户态，Ring0作为内核态。

Linux的内核是一个有机整体，每个用户进程运行时都好像有一份内核的拷贝。每当用户进程使用系统调用时，都自动地将运行模式从用户级转为内核级（成为陷入内核），此时，进程在内核的地址空间中运行。

##### 31、从用户空间到内核空间有以下触发手段

1. 系统调用：用户进程通过系统调用申请使用操作系统提供的服务程序来完成工作，比如read()、fork()等。系统调用的机制其核心还是使用了操作系统为用户特别开放的一个中断来实现的。

2. 中断：当外围设备完成用户请求的操作后，会向CPU发送中断信号。这时CPU会暂停执行下一条指令（用户态）转而执行与该中断信号对应的中断处理程序（内核态）

3. 异常：当CPU在执行运行在用户态下的程序时，发生了某些事先不可知的异常，这时会触发由当前运行进程切换到处理此异常的内核相关程序中，也就转到了内核态，比如缺页异常。

##### 32、进程的内存空间布局

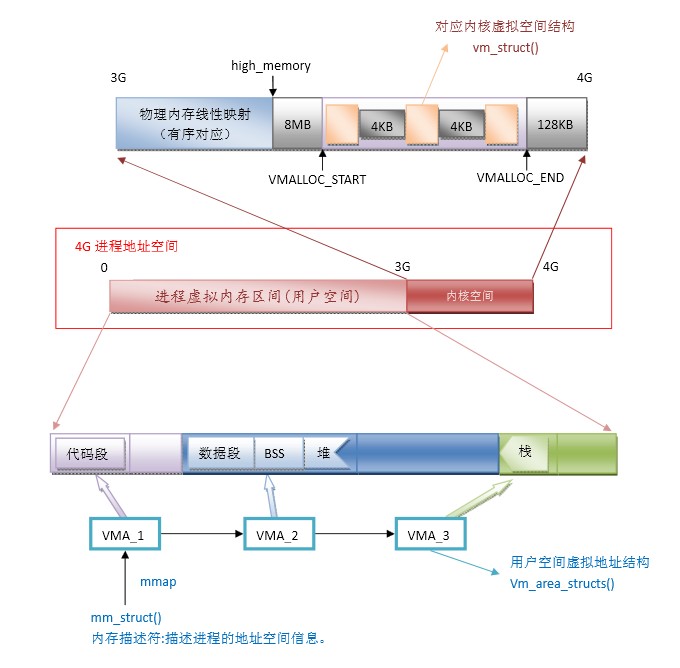
\*进程寻址空间0~4G

\*进程在用户态只能访问0~3G，只有进入内核态才能访问3G~4G

\*进程通过系统调用进入内核态

\*每个进程虚拟空间的3G~4G部分是相同的

\*进程从用户态进入内核态不会引起CR3的改变但会引起堆栈的改变



##### 33、进程间通信（IPC）方式

IPC: 管道、FIFO、信号、信号量、消息队列、共享内存、套接字

1. 管道：速度慢，容量有限，只有父子进程能通讯

2 有名管道通信

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。

3 消息队列通信

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。

4 信号量通信

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。

5 信号

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。

6 共享内存通信

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。

7 套接字通信

套接字( socket ) ： 套接口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同机器间的进程通信.

##### 34、进程空间和内核空间对内存的管理不同（不是很明白问题）

进程内存管理的对象是进程线性地址空间上的内存镜像(虚拟内存)，这些内存镜像其实就是进程使用的虚拟内存区域（memory region）。进程虚拟空间是个32或64位的“平坦”（独立的连续区间）地址空间（空间的具体大小取决于体系结构）。要统一管理这么大的平坦空间可绝非易事，为了方便管理，虚拟空间被划分为许多大小可变的(但必须是4096的倍数)内存区域，这些区域在进程线性地址中像停车位一样有序排列。这些区域的划分原则是“将访问属性一致的地址空间存放在一起”，所谓访问属性在这里无非指的是“可读、可写、可执行等”。

Linux内核管理物理内存是通过分页机制实现的，它将整个内存划分成无数个4k（在i386体系结构中）大小的页，从而分配和回收内存的基本单位便是内存页了。利用分页管理有助于灵活分配内存地址，因为分配时不必要求必须有大块的连续内存，系统可以东一页、西一页的凑出所需要的内存供进程使用。虽然如此，但是实际上系统使用内存时还是倾向于分配连续的内存块，因为分配连续内存时，页表不需要更改，因此能降低TLB的刷新率（频繁刷新会在很大程度上降低访问速度）。

##### 35、虚拟内存的作用

1. 内存访问保护

我们就可以通过设置段界限或页表项来设定软件运行时的访问空间，确保软件运行不越界，完成内存访问保护的功能。

2. 按需分页（lazy load技术）

通过内存地址虚拟化，可以使得软件在没有访问某虚拟内存地址时不分配具体的物理内存，而只有在实际访问某虚拟内存地址时，操作系统再动态地分配物理内存，建立虚拟内存到物理内存的页映射关系

3. 页换入换出（page swap in/out）

把不经常访问的数据所占的内存空间临时写到硬盘上，这样可以腾出更多的空闲内存空间给经常访问的数据；当CPU访问到不经常访问的数据时，再把这些数据从硬盘读入到内存中

4. 写时复制（copy on write）

两个虚拟页的数据内容相同时，可只分配一个物理页框，这样如果对两个虚拟页的访问方式是只读方式，这这两个虚拟页可共享页框，节省内存空间；如果CPU对其中之一的虚拟页进行写操作，则这两个虚拟页的数据内容会不同，需要分配一个新的物理页框，并将物理页框标记为可写，这样两个虚拟页面将映射到不同的物理页帧，确保整个内存空间的正确访问。

##### 36、虚拟内存的实现

虚拟内存中，允许将一个作业分多次调入内存。釆用连续分配方式时，会使相当一部分内存空间都处于暂时或“永久”的空闲状态，造成内存资源的严重浪费，而且也无法从逻辑上扩大内存容量。因此，虚拟内存的实现需要建立在离散分配的内存管理方式的基础上。虚拟内存的实现有以下三种方式：

请求分页存储管理。

请求分段存储管理。

请求段页式存储管理。

不管哪种方式，都需要有一定的硬件支持。一般需要的支持有以下几个方面：

一定容量的内存和外存。

页表机制（或段表机制），作为主要的数据结构。

中断机构，当用户程序要访问的部分尚未调入内存，则产生中断。

地址变换机构，逻辑地址到物理地址的变换。

**37、Linux的slab层**

答：SLAB层:  
              想必大多数人一提起linux内存管理,第一反应是"啊,特么的怎么这么复杂",哈哈,其实就是看书时蒙蔽了,别急,兄弟我陪你捋捋.  
正经的来吧:  
 (1)             先说说平时编程吧,实现个链表,给每个元素申请空间,用的时候添加链表,不用是删除,释放空间,俗话说的增删改查嘛,可是这个效率方面到底高不高呢?答案是:我特么的怎么知道,又不是搞导弹路线计算,需要那么搞吗,这个要看项目的具体需要了,但是需不需要都要先搞明白嘛,开源能留下来的东西,你我就别争论了.  
 (2)          言归正传吧.............. 分配和释放数据结构是linux内核最普遍的操作之一.为了便于数据的频繁分配和释放,编程人员都会实现空链表,哈哈,这会有人郁闷了,这和空链表有什么关系,别急,往下看,这里所说的空链表就是先申请好许多对象在内存,这样在用的时候直接从链表里面取出来就可以用了,不用的时候,在给它放入链表中就可以了,这就是所谓的特么的空链表.         原谅我的语言怎么这么脏,兄弟就是最近心情不好,就是特么的那种特别的不好,搞技术的我就只能在文章里表现了哈哈!!!!  
(3)            综上所述,空链表就相当于计算机中的高速缓存了,但是这样还是远远不够,否则的话,linux的命运岂不是掌握在了驱动开发工程师的手里面了,这样的话内核面临的问题是不能全局的控制,当可用的内存变的紧张的时候,内核无法通知每个空闲链表区收缩一下自己的缓存,特么的能通知的到才见鬼了,切切切,可是这样内核不爽了,让你收缩一下自己的缓存你还听不见,反了你了,基于这种情况,linux决定把空链表的特权回收,就应该这样治治你,不然特么的翻天呀你还,于是linux决定提出slab层(也就是所谓的slab分配器).  slab分配器扮演着通用数据结构缓存层的角色.丫的,让你牛逼!!  
(4)            slab分配器的概念在linux操作系统中得以实现,linux数据结构缓冲层具有同样的名字和基本设计思想:  
                  1,频繁使用的数据结构也会频繁的分配和释放,所以应该特么的缓存它们.  
                  2, 频繁分配和释放势必会导致内存碎片,(难以找到大块的内存).为了避免,空闲链表的缓存会连续的存放.因为以释放的数据结构又会放回空闲链表,因此不会导致碎片,  
                  3,回收的对象可以立即投入到下一次的分配中,因此频繁分配和释放的数据结构使用空链表可以提高其性能.  
                  4,如果slab分配器知道对象数据结构,页大小还有缓冲区的大小的话,会做出更加明智的决策.  
                  5,如果让部分缓存专属于特定的CPU处理器,那么就特么的不用考虑SMP,更不用考虑加锁了.  
                  6,对存放的对象进行着色,以防止多个对象映射到相同的高速缓存行  
                 linux   slab分配器在设计和实现上也确实考虑到了以上原则.  
(5)            slab层把不同的对象划分为不同的所谓的高速缓存组,其中每个高速缓存组都存放不同类型的对象,真特么的废话,不然怎么给你讲呢哈哈!!!  
                 例如:一个高速缓存组里面存放( task\_struct )进程描述符,而另一个高速缓存组存放(inode)描述符,更特么搞笑的是居然我们平时驱动中使用的kmalloc()也是在slab层之上使用了一组高速缓存,真特么的逆天吧,对于此种原因我就不解释了吧,记住,这就是特么的实现,也是现实.然后这些高速缓存又被划分为slab.明白了吗?,slab由一个或者多个物理上连续的页组成,一般条件下,slab也就只有一个页面的大小,每个高速缓存可以由多个slab组成,  
(6)            每个slab都包含一些成员对象,也就是被缓存的数据结构,每个slab处于三种状态之一:空,部分满,满.   当内核需要一个新的对象时,优先从部分满的slab中分配,其次是从空的slab中分配,再特么的不行就创建一个slab,这种策略可以减少碎片.  
  
  
                 高速缓存  ------>slab         ----->     [对象]   [对象] [对象]  
                                   |  
                                   |------->slab        ----->     [对象]  
(7)             每个高速缓存都使用struct  kmem\_cache 结构表示.该结构包含三个链表:slab\_full,slabs\_partial, slabs\_empty,均存放在kmem\_list3结构内,该结构在<mm/slab.c>中定义.这些链表包含高速缓存中的所有slab,我特么的没看错吧,是的你没有看错,是所有的slab,  slab描述符struct slab 用来描述每个slab  
  
                          struct slab{  
                                      struct list\_head          list;  
                                      unsigned long           colourful //对象的着色偏移  
                                      void                             \*s\_mem   //slab中的第一个对象  
                                     unsigned  int               inuse        //slab已分配的对象数  
                                     kmem\_bufctl\_t            free            //第一个空闲对象,如果有的话  
                          }  
(8)          slab分配器可以创建新的slab,这是通过\_get\_free\_pages()低级内核页分配器分配的,举个简单的例子吧.(包你看的明白)  
               static inline void\* kmem\_getpages(struct kmem\_cache \*cachp, gfp\_t flags)  
               {  
                               void\* addr;  
                               flags |= cachp->gfpflags;  
                               addr = (void\*) \_get\_free\_pages(flags, cachp->gfporder);  
                               return addr;  
                 }  
                接着调用kmem\_freepages()释放内存,最终调用的是free\_pages().  
                当然slab层的关键是避免频繁分配和释放页,由此可知,slab层只有当给定的高速缓存部分既没有满也没有空的slab时才会调用页分配函数,而只有在下列情况下才会真正的调用释放函数:(1)当可用的内存变得紧缺时,系统会释放出更多的内存以供使用(2)或当高速缓存被显示的撤销时才会调用函数释放.  
                  slab层的管理是在每个高速缓冲区的基础上,并提供给内核一个简单的接口来完成的.通过接口可以创建和撤销新的高速缓存,并在高速缓存上分配和释放对象,高速缓存及其内slab的复杂管理完全通过slab层的内部机制来处理,  
  
(9)          slab分配器的接口:  
              1, 一个新的高速缓存通过以下函数创建:  
               struct kmem\_cache \* kmem\_cache\_create( const char\* name, size\_t size, size\_t  align,  unsigned long flags, void (\*ctor)(void\*));  
               第一个参数是高速缓存区的名字;  
               第二个参数是高速缓存区的每个元素的大小;  
               第三个参数是slab内第一个对象的偏移,一般情况下为0;  
               第四个参数是flags是可选项  
               第五个参数为NULL;  
               2,撤销一个高速缓存则调用  
                int kmem\_cache\_destory(struct mem\_cache \*cachep);    return 成功返回0,失败返回非0值;  
                调用该函数时必须满足两个条件:(1) 高速缓存区中的所有slab都必须为空,哪怕只有一个对象被分配出去并正在使用的话,就不能去撤销这个高速缓存区.  
                                                                          (2) 调用此函数的过程中不能再访问这个高速缓存区了.  
 (10)     从缓冲区中分配对象  
             创建高速缓冲区后可以从中获取对象  
              void\*  kmem\_cache\_calloc(struct mem\_cache \*cachep, gfp\_t  flags);   //flags 为GFP\_KERNEL或者GFP\_ATOMIC  
(11)     释放对象到缓冲区中  
              void   kmem\_cache\_free(struct mem\_cache \*cachep,void \* objp);  
             这样就能把它返回到原先的slab中,意思就是把cachep中的对象objp标记为空闲  
  
exmple:  
                                                 struct kmem\_cache \*task\_struct\_cachep;  
                                                 task\_struct\_cachep = kmem\_cache\_create("task\_struct", sizeof(struct task\_struct), 0, SLAB\_PANIC| SLAB\_NOTRACK, NULL);  
                                                   
                                                 struct task\_struct  \*tsk;  
                                                 tsk = kmem\_cache\_calloc(cachep, GFP\_KERNEL);  
                                                if(!tsk)  
                                                     return NULL;  
                                               int err;  
                                               err = kmem\_cache\_destory(task\_struct\_cachep);  
                                               if(err)  
                                                    /\*出错,撤销高速缓存区\*/

**38、fork与vfork区别**

答：fork（）与vfock（）都是创建一个进程，那他们有什么区别呢？总结有以下三点区别：   
1.  fork  （）：子进程拷贝父进程的数据段，代码段   
    vfork （ ）：子进程与父进程共享数据段   
2.  fork （）父子进程的执行次序不确定   
    vfork 保证子进程先运行，在调用exec 或exit 之前与父进程数据是共享的,在它调用exec  
     或exit 之后父进程才可能被调度运行。   
3.  vfork （）保证子进程先运行，在她调用exec 或exit 之后父进程才可能被调度运行。如果在  
   调用这两个函数之前子进程依赖于父进程的进一步动作，则会导致死锁。   
下面通过几个例子加以说明：   
第一：子进程拷贝父进程的代码段的例子：

1. #include<sys/types.h>
2. #include<unistd.h>
3. #include<stdio.h>
5. int main()
6. {
7. pid\_t pid;
8. pid = fork();
9. if(pid<0)
10. printf("error in fork!\n");
11. else if(pid == 0)
12. printf("I am the child process,ID is %d\n",getpid());
13. else
14. printf("I am the parent process,ID is %d\n",getpid());
15. return 0;
17. }

运行结果：

1. [root@localhost fork]# gcc -o fork fork.c
2. [root@localhost fork]# ./fork
3. I am the child process,ID is 4711
4. I am the parent process,ID is 4710

为什么两条语 都会打印呢？这是因为fork（）函数用于从已存在的进程中创建一个新的进   
程，新的进程称为子进程，而原进程称为父进程，fork （）的返回值有两个，子进程返回0，  
父进程返回子进程的进程号，进程号都是非零的正整数，所以父进程返回的值一定大于零，  
在pid=fork();语句之前只有父进程在运行，而在pid=fork();之后，父进程和新创建的子进程   
都在运行，所以如果pid==0，那么肯定是子进程，若pid ！=0 （事实上肯定大于0），那么是   
父进程在运行。而我们知道fork()函数子进程是拷贝父进程的代码段的，所以子进程中同样   
有   
if(pid<0)   
         printf("error in fork!");   
     else if(pid==0)   
         printf("I am the child process,ID is %d\n",getpid());   
     else   
         printf("I am the parent process,ID is %d\n",getpid());   
}   
这么一段代码，所以上面这段代码会被父进程和子进程各执行一次，最终由于子进程的pid= =0,

而打印出第一句话，父进程的pid>0，而打印出第二句话。于是得到了上面的运行结果。   
再来看一个拷贝数据段的例子：

1. #include<sys/types.h>
2. #include<unistd.h>
3. #include<stdio.h>
5. int main()
6. {
7. pid\_t pid;
8. int cnt = 0;
9. pid = fork();
10. if(pid<0)
11. printf("error in fork!\n");
12. else if(pid == 0)
13. {
14. cnt++;
15. printf("cnt=%d\n",cnt);
16. printf("I am the child process,ID is %d\n",getpid());
17. }
18. else
19. {
20. cnt++;
21. printf("cnt=%d\n",cnt);
22. printf("I am the parent process,ID is %d\n",getpid());
23. }
24. return 0;
25. }

大家觉着打印出的值应该是多少呢？是不是2 呢？先来看下运行结果吧

1. [root@localhost fork]# ./fork2
2. cnt=1
3. I am the child process,ID is 5077
4. cnt=1
5. I am the parent process,ID is 5076

为什么不是2 呢？因为我们一次强调fork （）函数子进程拷贝父进程的数据段代码段，所以   
cnt++;   
    printf("cnt= %d\n",cnt);

    return 0   
将被父子进程各执行一次，但是子进程执行时使自己的数据段里面的（这个数据段是从父进   
程那copy 过来的一模一样）count+1，同样父进程执行时使自己的数据段里面的count+1，   
他们互不影响，与是便出现了如上的结果。

那么再来看看vfork （）吧。如果将上面程序中的fork （）改成vfork（），运行结果是什么   
样子的呢？

1. [root@localhost fork]# gcc -o fork3 fork3.c
2. [root@localhost fork]# ./fork3
3. cnt=1
4. I am the child process,ID is 4711
5. cnt=1
6. I am the parent process,ID is 4710
7. 段错误

本来vfock（）是共享数据段的，结果应该是2，为什么不是预想的2 呢？先看一个知识点：   
vfork 和fork 之间的另一个区别是：vfork 保证子进程先运行，在她调用exec 或exit 之   
后父进程才可能被调度运行。如果在调用这两个函数之前子进程依赖于父进程的进一步动   
作，则会导致死锁。   
这样上面程序中的fork （）改成vfork（）后，vfork （）创建子进程并没有调用exec 或exit，  
所以最终将导致死锁。   
怎么改呢？看下面程序：

1. #include<sys/types.h>
2. #include<unistd.h>
3. #include<stdio.h>
5. int main()
6. {
7. pid\_t pid;
8. int cnt = 0;
9. pid = vfork();
10. if(pid<0)
11. printf("error in fork!\n");
12. else if(pid == 0)
13. {
14. cnt++;
15. printf("cnt=%d\n",cnt);
16. printf("I am the child process,ID is %d\n",getpid());
17. \_exit(0);
18. }
19. else
20. {
21. cnt++;
22. printf("cnt=%d\n",cnt);
23. printf("I am the parent process,ID is %d\n",getpid());
24. }
25. return 0;
27. }

如果没有\_exit（0）的话，子进程没有调用exec 或exit，所以父进程是不可能执行的，在子   
进程调用exec 或exit 之后父进程才可能被调度运行。   
所以我们加上\_exit(0);使得子进程退出，父进程执行，这样else 后的语句就会被父进程执行，   
又因在子进程调用exec 或exit之前与父进程数据是共享的,所以子进程退出后把父进程的数   
据段count改成1 了，子进程退出后，父进程又执行，最终就将count变成了2，看下实际   
运行结果：

1. [root@localhost fork]# gcc -o fork3 fork3.c
2. [root@localhost fork]# ./fork3
3. cnt=1
4. I am the child process,ID is 4711
5. cnt=2
6. I am the parent process,ID is 4710

网上抄的一段，可以再理解理解：   
为什么会有vfork，因为以前的fork 很傻， 它创建一个子进程时，将会创建一个新的地址   
空间，并且拷贝父进程的资源，而往往在子进程中会执行exec 调用，这样，前面的拷贝工   
作就是白费力气了，这种情况下，聪明的人就想出了vfork，它产生的子进程刚开始暂时与   
父进程共享地址空间（其实就是线程的概念了），因为这时候子进程在父进程的地址空间中   
运行，所以子进程不能进行写操作，并且在儿子 霸占”着老子的房子时候，要委屈老子一   
下了，让他在外面歇着（阻塞），一旦儿子执行了exec 或者exit 后，相 于儿子买了自己的   
房子了，这时候就相 于分家了。

**39、exit()与\_exit()区别**

## 答：高级I/O函数

在Linux标准库中，有一套称为**高级I/O函数**,例如我们所熟知的printf,fopen,fread,fwrite都在此列，他们也被称为缓冲I/O。其特征是对应每一个打开的文件，都存在一个缓冲区，**在每次读文件时**会多读若干条记录，这样下次读文件时就可以直接从内存的缓冲区去读。**在每次写文件时**也会先写入缓冲区，当缓冲区写满，或者我们手动的刷新缓冲区，或者遇到\n,EOF这样的结束符，才会把对应的数据从缓冲区写入到文件中。这样的好处是大大增加的文件的读写的速度，因为我们都知道磁盘上的读写速度是很慢的，但这也为我们编程带来了一点麻烦，例如有些数据，我们认为已经写入了文件，但实际上它们很可能存在在缓冲区中。

关于printf的缓冲区问题推荐大家看下面的博客：   
<http://blog.csdn.net/bit_clearoff/article/details/53998476>

## exit和exit

首先我们来看看下面的实例：   
  
从图片中我们可以看出，通过exit(0)来结束掉main函数的程序可以输出结果，通过调用\_exit(0)函数的程序不能输出结果。

这时为什么呢？？？？   
我们来看看exit()和\_exit()各组定义在哪个头文件中吧：   
\_exit定义在unistd.h中的第273行:

而exit()定义在stdlib.h文件中：   
exit函数：定义在C标准库stdlib.h中；当我们调用exit(0)时，表示正常退出当前进程，当我们调用eixt(1)时表示非正常退出当前进程。   
具体可以查看stdlib.h文件中的宏定义：

其中exit在结束进程之前要做以下的事情：

### 调用atexit()注册的函数（出口函数）

atexit()函数定义在stdlib中，它的函数定义为:**int atexit(void (\*)(void))**,形参为一个返回值为void，参数为void的函数指针，很多时候我们需要在程序退出的时候做一些诸如释放资源的操作，但程序退出的方式有很多种，比如main()函数运行结束、在程序的某个地方用exit()结束程序、用户通过Ctrl+C或Ctrl+break操作来终止程序等等，因此需要有一种与程序退出方式无关的方法来进行程序退出时的必要处理。方法就是用atexit()函数来注册程序正常终止时要被调用的函数。   
  
在一个程序中最多可以用atexit()注册32个处理函数，这些处理函数的调用顺序与其注册的顺序相反，也即最先注册的最后调用，最后注册的最先调用。   
我们可以使用atexit()函数在main函数结束时对整个进程的内存空间进行销毁，作用相当于C++中的析构函数.

### 调用cleanup()关闭所有的流

这一步操作导致所有的缓冲被输出

### 最后调用\_exit()函数终止进程

\_exit()函数主要做了清理内存空间，结束进程调用等工作。

## 案例分析

这下我们应该了解了出现前面例子中情况的原因了，是因为exit函数在结束进程时用cleanup()刷新了缓冲区，使缓冲区中的数据被输出,而\_exit函数只是exit函数中的一部分,它调用了系统调用函数exit().

所以eixt和\_exit函数我们可以用下面这张图来总结一下：

**40、Linux是如何避免内存碎片的**

## 答：内存碎片问题

* 频繁地请求和释放不同大小的内存，必然导致内存碎片问题的产生，结果就是当再次要求分配连续的内存时，即使整体内存是足够的，也无法满足连续内存的需求。该问题也称之为外碎片(external fragmentation)。
* 解决方案：
* 避免外碎片的方法有两种:   
  1、利用分页单元把一组非连续的空闲页框映射到连续的线性地址   
  2、开发一种适当的技术来记录现存的空闲的连续页框块的情况，以尽量避免为满足对小块的请求而分割大的空闲快
* 第一种方案的意思是，我们使用地址转换技术，把非连续的物理地址转换成连续的线性地址。
* 第二种方案的意思是，开发一种特有的分配技术来记录下来空闲内存的情况，从而解决内存碎片问题。
* Linux采用了第二种方案，因为在某些情况下，系统的确需要连续的物理地址（DMA处理器可以直接访问总线）。

## 这里先对Linux内存管理做一个简单介绍

* linux kernel 通过把整个物理内存划分成以一个个page进行管理，管理器就是伙伴系统，它的最小分配单元就是page。但是对于小于page的内存分配，如果直接分配一个page，是一个很大的浪费。linux kernel 通过slab来实现对小于page大小的内存分配。slab把page按2的m次幂进行划分一个个字节块，当kmalloc申请内存时，通过slab管理器返回需要满足申请大小的最小空闲内存块。
* slub主要是针对slab的对象管理数据的优化版本，相比于slab，slub提供更小的管理成本开销。而且slub对多核系统的支持也更加友好。细节这里就不展开讲。
* 所以kernel的内存管理是个2层分层系统，从下往上依次为：
* 第一层为全部物理内存：其管理器为伙伴系统，最小管理单位为page；
* 第二层为slab page：其管理器为slab/slub，最小管理单位为2的m次幂的字节块；

## 伙伴系统(buddy system)

* Linux采用著名的伙伴系统(buddy system)算法来解决外碎片问题。把所有的空闲页框分组为11个块链表，每个链表分别包含大小为1,2,4,8,16,32,64,128,256,512,1024个连续的页框，对1024个页框的最大请求对应着4MB大小的连续RAM（每页大小为4KB），每个块的第一个页框的物理地址是该块大小的整数倍，例如，大小为16个页框的块，其起始地址是16\*2^12的倍数。   
  我们通过一个例子来说明伙伴算法的工作原理，假设现在要请求一个256个页框的块（1MB），算法步骤如下：   
  • 在256个页框的链表中检查是否有一个空闲快，如果没有，查找下一个更大的块，如果有，请求满足。   
  • 在512个页框的链表中检查是否有一个空闲块，如果有，把512个页框的空闲块分为两份，第一份用于满足请求，第二份链接到256个页框的链表中。如果没有空闲块，继续寻找下一个更大的块。   
  下图比较形象地描述了该过程。   
     
  页的请求   
  以上过程的逆过程，就是页框块的释放过程，也是该算法名字的由来，内核试图把大小为B的一对空闲伙伴块合并为一个2B的单独块，满足以下条件的两个块称之为伙伴：   
  • 两个块具有相同的大小   
  • 他们的物理地址是连续的   
  第一块的第一个页框的物理地址是2 \* B \* 2^12   
  该算法是递归的，如果它成功合并了B，就会试图去合并2B，以再次试图形成更大的块。

## 高速缓存Slab层

* slab是Linux操作系统的一种内存分配机制。其工作是针对一些经常分配并释放的对象，如进程描述符等，这些对象的大小一般比较小，如果直接采用伙伴系统来进行分配和释放，不仅会造成大量的内存碎片，而且处理速度也太慢。
* 而slab分配器是基于对象进行管理的，相同类型的对象归为一类(如进程描述符就是一类)，每当要申请这样一个对象，slab分配器就从一个slab列表中分配一个这样大小的单元出去，而当要释放时，将其重新保存在该列表中，而不是直接返回给伙伴系统，从而避免这些内碎片。slab分配器并不丢弃已分配的对象，而是释放并把它们保存在内存中。当以后又要请求新的对象时，就可以从内存直接获取而不用重复初始化。
* 对象高速缓存的组织如右下图所示，高速缓存的内存区被划分为多个slab，每个slab由一个或多个连续的页框组成，这些页框中既包含已分配的对象，也包含空闲的对象。
* 在cache和object中加入slab分配器，是在时间和空间上的折中方案。
* 另外为了解决多核和NUMA架构下效率问题，slab管理器kmem\_cache又把slab page对象分为2层结构，从下往上依次为：
* 第一层为NUMA node下cpu共享page：管理器为kmem\_cache\_node，管理node下的slab对象，解决NUMA架构的内存访问效率问题。当本层的空闲page不足时，从伙伴系统申请空闲page；
* 第二层为per-cpu专属page：管理器为kmem\_cache\_cpu，管理cpu专属的slab对象，解决多核竞争问题。当本层的空闲page不足时，从第一层申请空闲page；

### slab分配算法

* slab分配算法采用cache 存储内核对象。当创建cache 时，起初包括若干标记为空闲的对象。对象的数量与slab的大小有关。开始，所有对象都标记为空闲。当需要内核数据结构的对象时，可以直接从cache 上直接获取，并将对象初始化为使用。
* 下面考虑内核如何将slab分配给表示进程描述符的对象。在Linux系统中，进程描述符的类型是struct task\_struct ，其大小约为1.7KB。当Linux 内核创建新任务时，它会从cache 中获得struct task\_struct 对象所需要的内存。Cache 上会有已分配好的并标记为空闲的struct task\_struct 对象来满足请求。
* Linux 的slab 可有三种状态：   
  满的：slab 中的所有对象被标记为使用。   
  空的：slab 中的所有对象被标记为空闲。   
  部分：slab 中的对象有的被标记为使用，有的被标记为空闲。
* slab 分配器首先从部分空闲的slab 进行分配。如没有，则从空的slab 进行分配。如没有，则从物理连续页上分配新的slab，并把它赋给一个cache ，然后再从新slab 分配空间。

41、共享内存的实现原理

答：共享内存可以说是最有用的进程间通信方式，也是最快的IPC形式。两个不同进程A、B共享内存的意思是，同一块物理内存被映射到进程A、B各自的进程地址空间。进程A可以即时看到进程B对共享内存中数据的更新，反之亦然。由于多个进程共享同一块内存区域，必然需要某种同步机制，互斥锁和信号量都可以。  
  
采用共享内存通信的一个显而易见的好处是效率高，因为进程可以直接读写内存，而不需要任何数据的拷贝。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝，而共享内存则只拷贝两次数据[1]：一次从输入文件到共享内存区，另一次从共享内存区到输出文件。实际上，进程之间在共享内存时，并不总是读写少量数据后就解除映射，有新的通信时，再重新建立共享内存区域。而是保持共享区域，直到通信完毕为止，这样，数据内容一直保存在共享内存中，并没有写回文件。共享内存中的内容往往是在解除映射时才写回文件的。因此，采用共享内存的通信方式效率是非常高的。  
  
Linux的2.2.x内核支持多种共享内存方式，如mmap()系统调用，Posix共享内存，以及系统V共享内存。linux发行版本如Redhat 8.0支持mmap()系统调用及系统V共享内存，但还没实现Posix共享内存，本文将主要介绍mmap()系统调用及系统V共享内存API的原理及应用。  
  
一、内核怎样保证各个进程寻址到同一个共享内存区域的内存页面  
  
1、page cache及swap cache中页面的区分：一个被访问文件的物理页面都驻留在page cache或swap cache中，一个页面的所有信息由struct page来描述。struct page中有一个域为指针mapping ，它指向一个struct address\_space类型结构。page cache或swap cache中的所有页面就是根据address\_space结构以及一个偏移量来区分的。  
  
2、文件与address\_space结构的对应：一个具体的文件在打开后，内核会在内存中为之建立一个struct inode结构，其中的i\_mapping域指向一个address\_space结构。这样，一个文件就对应一个address\_space结构，一个address\_space与一个偏移量能够确定一个page cache 或swap cache中的一个页面。因此，当要寻址某个数据时，很容易根据给定的文件及数据在文件内的偏移量而找到相应的页面。  
  
3、进程调用mmap()时，只是在进程空间内新增了一块相应大小的缓冲区，并设置了相应的访问标识，但并没有建立进程空间到物理页面的映射。因此，第一次访问该空间时，会引发一个缺页异常。  
  
4、对于共享内存映射情况，缺页异常处理程序首先在swap cache中寻找目标页（符合address\_space以及偏移量的物理页），如果找到，则直接返回地址；如果没有找到，则判断该页是否在交换区(swap area)，如果在，则执行一个换入操作；如果上述两种情况都不满足，处理程序将分配新的物理页面，并把它插入到page cache中。进程最终将更新进程页表。  
注：对于映射普通文件情况（非共享映射），缺页异常处理程序首先会在page cache中根据address\_space以及数据偏移量寻找相应的页面。如果没有找到，则说明文件数据还没有读入内存，处理程序会从磁盘读入相应的页面，并返回相应地址，同时，进程页表也会更新。  
  
5、所有进程在映射同一个共享内存区域时，情况都一样，在建立线性地址与物理地址之间的映射之后，不论进程各自的返回地址如何，实际访问的必然是同一个共享内存区域对应的物理页面。  
注：一个共享内存区域可以看作是特殊文件系统shm中的一个文件，shm的安装点在交换区上。  
  
上面涉及到了一些数据结构，围绕数据结构理解问题会容易一些。  
  
二、mmap()及其相关系统调用  
  
mmap()系统调用使得进程之间通过映射同一个普通文件实现共享内存。普通文件被映射到进程地址空间后，进程可以向访问普通内存一样对文件进行访问，不必再调用read()，write（）等操作。  
  
注：实际上，mmap()系统调用并不是完全为了用于共享内存而设计的。它本身提供了不同于一般对普通文件的访问方式，进程可以像读写内存一样对普通文件的操作。而Posix或系统V的共享内存IPC则纯粹用于共享目的，当然mmap()实现共享内存也是其主要应用之一。  
  
1、mmap()系统调用形式如下：  
  
void\* mmap ( void \* addr , size\_t len , int prot , int flags , int fd , off\_t offset )  
参数fd为即将映射到进程空间的文件描述字，一般由open()返回，同时，fd可以指定为-1，此时须指定flags参数中的MAP\_ANON，表明进行的是匿名映射（不涉及具体的文件名，避免了文件的创建及打开，很显然只能用于具有亲缘关系的进程间通信）。len是映射到调用进程地址空间的字节数，它从被映射文件开头offset个字节开始算起。prot 参数指定共享内存的访问权限。可取如下几个值的或：PROT\_READ（可读） , PROT\_WRITE （可写）, PROT\_EXEC （可执行）, PROT\_NONE（不可访问）。flags由以下几个常值指定：MAP\_SHARED , MAP\_PRIVATE , MAP\_FIXED，其中，MAP\_SHARED , MAP\_PRIVATE必选其一，而MAP\_FIXED则不推荐使用。offset参数一般设为0，表示从文件头开始映射。参数addr指定文件应被映射到进程空间的起始地址，一般被指定一个空指针，此时选择起始地址的任务留给内核来完成。函数的返回值为最后文件映射到进程空间的地址，进程可直接操作起始地址为该值的有效地址。这里不再详细介绍mmap()的参数，读者可参考mmap()手册页获得进一步的信息。  
  
2、系统调用mmap()用于共享内存的两种方式：  
  
（1）使用普通文件提供的内存映射：适用于任何进程之间；此时，需要打开或创建一个文件，然后再调用mmap()；典型调用代码如下：   
  
        fd=open(name, flag, mode);  
if(fd<0)  
        ...  
          
ptr=mmap(NULL, len , PROT\_READ|PROT\_WRITE, MAP\_SHARED , fd , 0); 通过mmap()实现共享内存的通信方式有许多特点和要注意的地方，我们将在范例中进行具体说明。   
  
（2）使用特殊文件提供匿名内存映射：适用于具有亲缘关系的进程之间；由于父子进程特殊的亲缘关系，在父进程中先调用mmap()，然后调用fork()。那么在调用fork()之后，子进程继承父进程匿名映射后的地址空间，同样也继承mmap()返回的地址，这样，父子进程就可以通过映射区域进行通信了。注意，这里不是一般的继承关系。一般来说，子进程单独维护从父进程继承下来的一些变量。而mmap()返回的地址，却由父子进程共同维护。  
对于具有亲缘关系的进程实现共享内存最好的方式应该是采用匿名内存映射的方式。此时，不必指定具体的文件，只要设置相应的标志即可，参见范例2。  
  
3、系统调用munmap()  
  
int munmap( void \* addr, size\_t len )  
该调用在进程地址空间中解除一个映射关系，addr是调用mmap()时返回的地址，len是映射区的大小。当映射关系解除后，对原来映射地址的访问将导致段错误发生。  
  
4、系统调用msync()  
  
int msync ( void \* addr , size\_t len, int flags)  
一般说来，进程在映射空间的对共享内容的改变并不直接写回到磁盘文件中，往往在调用munmap（）后才执行该操作。可以通过调用msync()实现磁盘上文件内容与共享内存区的内容一致。  
  
三、mmap()范例  
  
下面将给出使用mmap()的两个范例：范例1给出两个进程通过映射普通文件实现共享内存通信；范例2给出父子进程通过匿名映射实现共享内存。系统调用mmap()有许多有趣的地方，下面是通过mmap（）映射普通文件实现进程间的通信的范例，我们通过该范例来说明mmap()实现共享内存的特点及注意事项。  
  
范例1：两个进程通过映射普通文件实现共享内存通信  
  
范例1包含两个子程序：map\_normalfile1.c及map\_normalfile2.c。编译两个程序，可执行文件分别为map\_normalfile1及map\_normalfile2。两个程序通过命令行参数指定同一个文件来实现共享内存方式的进程间通信。map\_normalfile2试图打开命令行参数指定的一个普通文件，把该文件映射到进程的地址空间，并对映射后的地址空间进行写操作。map\_normalfile1把命令行参数指定的文件映射到进程地址空间，然后对映射后的地址空间执行读操作。这样，两个进程通过命令行参数指定同一个文件来实现共享内存方式的进程间通信。  
  
下面是两个程序代码：  
  
/\*-------------map\_normalfile1.c-----------\*/  
#include <sys/mman.h>;  
#include <sys/types.h>;  
#include <fcntl.h>;  
#include <unistd.h>;  
typedef struct{  
        char name[4];  
        int  age;  
}people;  
  
main(int argc, char\*\* argv) // map a normal file as shared mem:  
{  
        int fd,i;  
        people \*p\_map;  
        char temp;  
          
        fd=open(argv[1],O\_CREAT|O\_RDWR|O\_TRUNC,00777);  
        lseek(fd,sizeof(people)\*5-1,SEEK\_SET);  
        write(fd,"",1);  
          
        p\_map = (people\*) mmap( NULL,sizeof(people)\*10,PROT\_READ|PROT\_WRITE,MAP\_SHARED,fd,0 );  
        close( fd );  
        temp = 'a';  
        for(i=0; i<10; i++)  
        {  
                temp += 1;  
                memcpy( ( \*(p\_map+i) ).name, &temp,2 );  
                ( \*(p\_map+i) ).age = 20+i;  
        }  
        printf(" initialize over /n ")；  
        sleep(10);  
  
        munmap( p\_map, sizeof(people)\*10 );  
        printf( "umap ok /n" );  
}  
  
/\*-------------map\_normalfile2.c-----------\*/  
#include <sys/mman.h>;  
#include <sys/types.h>;  
#include <fcntl.h>;  
#include <unistd.h>;  
typedef struct{  
        char name[4];  
        int  age;  
}people;  
  
main(int argc, char\*\* argv)        // map a normal file as shared mem:  
{  
        int fd,i;  
        people \*p\_map;  
        fd=open( argv[1],O\_CREAT|O\_RDWR,00777 );  
        p\_map = (people\*)mmap(NULL,sizeof(people)\*10,PROT\_READ|PROT\_WRITE,MAP\_SHARED,fd,0);  
        for(i = 0;i<10;i++)  
        {  
        printf( "name: %s age %d;/n",(\*(p\_map+i)).name, (\*(p\_map+i)).age );  
  
        }  
        munmap( p\_map,sizeof(people)\*10 );  
}  
  
map\_normalfile1.c首先定义了一个people数据结构，（在这里采用数据结构的方式是因为，共享内存区的数据往往是有固定格式的，这由通信的各个进程决定，采用结构的方式有普遍代表性）。map\_normfile1首先打开或创建一个文件，并把文件的长度设置为5个people结构大小。然后从mmap()的返回地址开始，设置了10个people结构。然后，进程睡眠10秒钟，等待其他进程映射同一个文件，最后解除映射。  
  
map\_normfile2.c只是简单的映射一个文件，并以people数据结构的格式从mmap()返回的地址处读取10个people结构，并输出读取的值，然后解除映射。  
  
分别把两个程序编译成可执行文件map\_normalfile1和map\_normalfile2后，在一个终端上先运行./map\_normalfile2 /tmp/test\_shm，程序输出结果如下：  
  
initialize over  
umap ok  
  
在map\_normalfile1输出initialize over 之后，输出umap ok之前，在另一个终端上运行map\_normalfile2 /tmp/test\_shm，将会产生如下输出(为了节省空间，输出结果为稍作整理后的结果)：  
  
name: b        age 20;        name: c        age 21;        name: d        age 22;        name: e        age 23;        name: f        age 24;  
name: g        age 25;        name: h        age 26;        name: I        age 27;        name: j        age 28;        name: k        age 29;  
  
在map\_normalfile1 输出umap ok后，运行map\_normalfile2则输出如下结果：  
  
name: b        age 20;        name: c        age 21;        name: d        age 22;        name: e        age 23;        name: f        age 24;  
name:        age 0;        name:        age 0;        name:        age 0;        name:        age 0;        name:        age 0;  
  
从程序的运行结果中可以得出的结论  
  
1、 最终被映射文件的内容的长度不会超过文件本身的初始大小，即映射不能改变文件的大小；  
  
2、 可以用于进程通信的有效地址空间大小大体上受限于被映射文件的大小，但不完全受限于文件大小。打开文件被截短为5个people结构大小，而在map\_normalfile1中初始化了10个people数据结构，在恰当时候（map\_normalfile1输出initialize over 之后，输出umap ok之前）调用map\_normalfile2会发现map\_normalfile2将输出全部10个people结构的值，后面将给出详细讨论。  
注：在linux中，内存的保护是以页为基本单位的，即使被映射文件只有一个字节大小，内核也会为映射分配一个页面大小的内存。当被映射文件小于一个页面大小时，进程可以对从mmap()返回地址开始的一个页面大小进行访问，而不会出错；但是，如果对一个页面以外的地址空间进行访问，则导致错误发生，后面将进一步描述。因此，可用于进程间通信的有效地址空间大小不会超过文件大小及一个页面大小的和。  
  
3、 文件一旦被映射后，调用mmap()的进程对返回地址的访问是对某一内存区域的访问，暂时脱离了磁盘上文件的影响。所有对mmap()返回地址空间的操作只在内存中有意义，只有在调用了munmap()后或者msync()时，才把内存中的相应内容写回磁盘文件，所写内容仍然不能超过文件的大小。  
  
范例2：父子进程通过匿名映射实现共享内存  
  
#include <sys/mman.h>;  
#include <sys/types.h>;  
#include <fcntl.h>;  
#include <unistd.h>;  
typedef struct{  
        char name[4];  
        int  age;  
}people;  
main(int argc, char\*\* argv)  
{  
        int i;  
        people \*p\_map;  
        char temp;  
        p\_map=(people\*)mmap(NULL,sizeof(people)\*10,PROT\_READ|PROT\_WRITE,MAP\_SHARED|MAP\_ANONYMOUS,-1,0);  
        if(fork() == 0)  
        {  
                sleep(2);  
                for(i = 0;i<5;i++)  
                        printf("child read: the %d people's age is %d/n",i+1,(\*(p\_map+i)).age);  
                (\*p\_map).age = 100;  
                munmap(p\_map,sizeof(people)\*10); //实际上，进程终止时，会自动解除映射。  
                exit();  
        }  
        temp = 'a';  
        for(i = 0;i<5;i++)  
        {  
                temp += 1;  
                memcpy((\*(p\_map+i)).name, &temp,2);  
                (\*(p\_map+i)).age=20+i;  
        }  
  
        sleep(5);  
        printf( "parent read: the first people,s age is %d/n",(\*p\_map).age );  
        printf("umap/n");  
        munmap( p\_map,sizeof(people)\*10 );  
        printf( "umap ok/n" );  
}  
  
考察程序的输出结果，体会父子进程匿名共享内存：  
  
child read: the 1 people's age is 20  
child read: the 2 people's age is 21  
child read: the 3 people's age is 22  
child read: the 4 people's age is 23  
child read: the 5 people's age is 24  
  
parent read: the first people,s age is 100  
umap  
umap ok  
  
四、对mmap()返回地址的访问  
  
前面对范例运行结构的讨论中已经提到，linux采用的是页式管理机制。对于用mmap()映射普通文件来说，进程会在自己的地址空间新增一块空间，空间大小由mmap()的len参数指定，注意，进程并不一定能够对全部新增空间都能进行有效访问。进程能够访问的有效地址大小取决于文件被映射部分的大小。简单的说，能够容纳文件被映射部分大小的最少页面个数决定了进程从mmap()返回的地址开始，能够有效访问的地址空间大小。超过这个空间大小，内核会根据超过的严重程度返回发送不同的信号给进程。可用如下图示说明：  
  
  
  
  
  
注意：文件被映射部分而不是整个文件决定了进程能够访问的空间大小，另外，如果指定文件的偏移部分，一定要注意为页面大小的整数倍。下面是对进程映射地址空间的访问范例：  
  
#include <sys/mman.h>;  
#include <sys/types.h>;  
#include <fcntl.h>;  
#include <unistd.h>;  
typedef struct{  
        char name[4];  
        int  age;  
}people;  
  
main(int argc, char\*\* argv)  
{  
        int fd,i;  
        int pagesize,offset;  
        people \*p\_map;  
          
        pagesize = sysconf(\_SC\_PAGESIZE);  
        printf("pagesize is %d/n",pagesize);  
        fd = open(argv[1],O\_CREAT|O\_RDWR|O\_TRUNC,00777);  
        lseek(fd,pagesize\*2-100,SEEK\_SET);  
        write(fd,"",1);  
        offset = 0;        //此处offset = 0编译成版本1；offset = pagesize编译成版本2  
        p\_map = (people\*)mmap(NULL,pagesize\*3,PROT\_READ|PROT\_WRITE,MAP\_SHARED,fd,offset);  
        close(fd);  
          
        for(i = 1; i<10; i++)  
        {  
                (\*(p\_map+pagesize/sizeof(people)\*i-2)).age = 100;  
                printf("access page %d over/n",i);  
                (\*(p\_map+pagesize/sizeof(people)\*i-1)).age = 100;  
                printf("access page %d edge over, now begin to access page %d/n",i, i+1);  
                (\*(p\_map+pagesize/sizeof(people)\*i)).age = 100;  
                printf("access page %d over/n",i+1);  
        }  
        munmap(p\_map,sizeof(people)\*10);  
}  
  
如程序中所注释的那样，把程序编译成两个版本，两个版本主要体现在文件被映射部分的大小不同。文件的大小介于一个页面与两个页面之间（大小为：pagesize\*2-99），版本1的被映射部分是整个文件，版本2的文件被映射部分是文件大小减去一个页面后的剩余部分，不到一个页面大小(大小为：pagesize-99)。程序中试图访问每一个页面边界，两个版本都试图在进程空间中映射pagesize\*3的字节数。  
  
版本1的输出结果如下：  
  
pagesize is 4096  
access page 1 over  
access page 1 edge over, now begin to access page 2  
access page 2 over  
access page 2 over  
access page 2 edge over, now begin to access page 3  
Bus error                //被映射文件在进程空间中覆盖了两个页面，此时，进程试图访问第三个页面  
  
版本2的输出结果如下：  
  
pagesize is 4096  
access page 1 over  
access page 1 edge over, now begin to access page 2  
Bus error                //被映射文件在进程空间中覆盖了一个页面，此时，进程试图访问第二个页面  
  
结论：采用系统调用mmap()实现进程间通信是很方便的，在应用层上接口非常简洁。内部实现机制区涉及到了linux存储管理以及文件系统等方面的内容，可以参考一下相关重要数据结构来加深理解。在本专题的后面部分，将介绍系统v共享内存的实现。

在共享内存（上）中，主要围绕着系统调用mmap()进行讨论的，本部分将讨论系统V共享内存，并通过实验结果对比来阐述两者的异同。系统V共享内存指的是把所有共享数据放在共享内存区域（IPC shared memory region），任何想要访问该数据的进程都必须在本进程的地址空间新增一块内存区域，用来映射存放共享数据的物理内存页面。  
  
系统调用mmap()通过映射一个普通文件实现共享内存。系统V则是通过映射特殊文件系统shm中的文件实现进程间的共享内存通信。也就是说，每个共享内存区域对应特殊文件系统shm中的一个文件（这是通过shmid\_kernel结构联系起来的），后面还将阐述。  
  
1、系统V共享内存原理  
  
进程间需要共享的数据被放在一个叫做IPC共享内存区域的地方，所有需要访问该共享区域的进程都要把该共享区域映射到本进程的地址空间中去。系统V共享内存通过shmget获得或创建一个IPC共享内存区域，并返回相应的标识符。内核在保证shmget获得或创建一个共享内存区，初始化该共享内存区相应的shmid\_kernel结构注同时，还将在特殊文件系统shm中，创建并打开一个同名文件，并在内存中建立起该文件的相应dentry及inode结构，新打开的文件不属于任何一个进程（任何进程都可以访问该共享内存区）。所有这一切都是系统调用shmget完成的。  
  
注：每一个共享内存区都有一个控制结构struct shmid\_kernel，shmid\_kernel是共享内存区域中非常重要的一个数据结构，它是存储管理和文件系统结合起来的桥梁，定义如下：  
  
struct shmid\_kernel /\* private to the kernel \*/  
{          
        struct kern\_ipc\_perm        shm\_perm;  
        struct file \*                shm\_file;  
        int                        id;  
        unsigned long                shm\_nattch;  
        unsigned long                shm\_segsz;  
        time\_t                        shm\_atim;  
        time\_t                        shm\_dtim;  
        time\_t                        shm\_ctim;  
        pid\_t                        shm\_cprid;  
        pid\_t                        shm\_lprid;  
};  
  
该结构中最重要的一个域应该是shm\_file，它存储了将被映射文件的地址。每个共享内存区对象都对应特殊文件系统shm中的一个文件，一般情况下，特殊文件系统shm中的文件是不能用read()、write()等方法访问的，当采取共享内存的方式把其中的文件映射到进程地址空间后，可直接采用访问内存的方式对其访问。  
  
这里我们采用[1]中的图表给出与系统V共享内存相关数据结构：  
  
  
  
  
  
正如消息队列和信号灯一样，内核通过数据结构struct ipc\_ids shm\_ids维护系统中的所有共享内存区域。上图中的shm\_ids.entries变量指向一个ipc\_id结构数组，而每个ipc\_id结构数组中有个指向kern\_ipc\_perm结构的指针。到这里读者应该很熟悉了，对于系统V共享内存区来说，kern\_ipc\_perm的宿主是shmid\_kernel结构，shmid\_kernel是用来描述一个共享内存区域的，这样内核就能够控制系统中所有的共享区域。同时，在shmid\_kernel结构的file类型指针shm\_file指向文件系统shm中相应的文件，这样，共享内存区域就与shm文件系统中的文件对应起来。  
  
在创建了一个共享内存区域后，还要将它映射到进程地址空间，系统调用shmat()完成此项功能。由于在调用shmget()时，已经创建了文件系统shm中的一个同名文件与共享内存区域相对应，因此，调用shmat()的过程相当于映射文件系统shm中的同名文件过程，原理与mmap()大同小异。  
  
2、系统V共享内存API  
  
对于系统V共享内存，主要有以下几个API：shmget()、shmat()、shmdt()及shmctl()。  
  
#include <sys/ipc.h>;  
#include <sys/shm.h>;  
  
shmget（）用来获得共享内存区域的ID，如果不存在指定的共享区域就创建相应的区域。shmat()把共享内存区域映射到调用进程的地址空间中去，这样，进程就可以方便地对共享区域进行访问操作。shmdt()调用用来解除进程对共享内存区域的映射。shmctl实现对共享内存区域的控制操作。这里我们不对这些系统调用作具体的介绍，读者可参考相应的手册页面，后面的范例中将给出它们的调用方法。  
  
注：shmget的内部实现包含了许多重要的系统V共享内存机制；shmat在把共享内存区域映射到进程空间时，并不真正改变进程的页表。当进程第一次访问内存映射区域访问时，会因为没有物理页表的分配而导致一个缺页异常，然后内核再根据相应的存储管理机制为共享内存映射区域分配相应的页表。  
  
3、系统V共享内存限制  
  
在/proc/sys/kernel/目录下，记录着系统V共享内存的一下限制，如一个共享内存区的最大字节数shmmax，系统范围内最大共享内存区标识符数shmmni等，可以手工对其调整，但不推荐这样做。  
  
在[2]中，给出了这些限制的测试方法，不再赘述。  
  
4、系统V共享内存范例  
  
本部分将给出系统V共享内存API的使用方法，并对比分析系统V共享内存机制与mmap()映射普通文件实现共享内存之间的差异，首先给出两个进程通过系统V共享内存通信的范例：  
  
/\*\*\*\*\* testwrite.c \*\*\*\*\*\*\*/  
#include <sys/ipc.h>;  
#include <sys/shm.h>;  
#include <sys/types.h>;  
#include <unistd.h>;  
typedef struct{  
        char name[4];  
        int age;  
} people;  
main(int argc, char\*\* argv)  
{  
        int shm\_id,i;  
        key\_t key;  
        char temp;  
        people \*p\_map;  
        char\* name = "/dev/shm/myshm2";  
        key = ftok(name,0);  
        if(key==-1)  
                perror("ftok error");  
        shm\_id=shmget(key,4096,IPC\_CREAT);          
        if(shm\_id==-1)  
        {  
                perror("shmget error");  
                return;  
        }  
        p\_map=(people\*)shmat(shm\_id,NULL,0);  
        temp='a';  
        for(i = 0;i<10;i++)  
        {  
                temp+=1;  
                memcpy((\*(p\_map+i)).name,&temp,1);  
                (\*(p\_map+i)).age=20+i;  
        }  
        if(shmdt(p\_map)==-1)  
                perror(" detach error ");  
}  
/\*\*\*\*\*\*\*\*\*\* testread.c \*\*\*\*\*\*\*\*\*\*\*\*/  
#include <sys/ipc.h>;  
#include <sys/shm.h>;  
#include <sys/types.h>;  
#include <unistd.h>;  
typedef struct{  
        char name[4];  
        int age;  
} people;  
main(int argc, char\*\* argv)  
{  
        int shm\_id,i;  
        key\_t key;  
        people \*p\_map;  
        char\* name = "/dev/shm/myshm2";  
        key = ftok(name,0);  
        if(key == -1)  
                perror("ftok error");  
        shm\_id = shmget(key,4096,IPC\_CREAT);          
        if(shm\_id == -1)  
        {  
                perror("shmget error");  
                return;  
        }  
        p\_map = (people\*)shmat(shm\_id,NULL,0);  
        for(i = 0;i<10;i++)  
        {  
        printf( "name:%s/n",(\*(p\_map+i)).name );  
        printf( "age %d/n",(\*(p\_map+i)).age );  
        }  
        if(shmdt(p\_map) == -1)  
                perror(" detach error ");  
}  
  
testwrite.c创建一个系统V共享内存区，并在其中写入格式化数据；testread.c访问同一个系统V共享内存区，读出其中的格式化数据。分别把两个程序编译为testwrite及testread，先后执行./testwrite及./testread 则./testread输出结果如下：  
  
name: b        age 20;        name: c        age 21;        name: d        age 22;        name: e        age 23;        name: f        age 24;  
name: g        age 25;        name: h        age 26;        name: I        age 27;        name: j        age 28;        name: k        age 29;  
  
通过对试验结果分析，对比系统V与mmap()映射普通文件实现共享内存通信，可以得出如下结论：  
  
1、 系统V共享内存中的数据，从来不写入到实际磁盘文件中去；而通过mmap()映射普通文件实现的共享内存通信可以指定何时将数据写入磁盘文件中。注：前面讲到，系统V共享内存机制实际是通过映射特殊文件系统shm中的文件实现的，文件系统shm的安装点在交换分区上，系统重新引导后，所有的内容都丢失。  
  
2、 系统V共享内存是随内核持续的，即使所有访问共享内存的进程都已经正常终止，共享内存区仍然存在（除非显式删除共享内存），在内核重新引导之前，对该共享内存区域的任何改写操作都将一直保留。  
  
3、 通过调用mmap()映射普通文件进行进程间通信时，一定要注意考虑进程何时终止对通信的影响。而通过系统V共享内存实现通信的进程则不然。注：这里没有给出shmctl的使用范例，原理与消息队列大同小异。  
  
结论：  
  
共享内存允许两个或多个进程共享一给定的存储区，因为数据不需要来回复制，所以是最快的一种进程间通信机制。共享内存可以通过mmap()映射普通文件（特殊情况下还可以采用匿名映射）机制实现，也可以通过系统V共享内存机制实现。应用接口和原理很简单，内部机制复杂。为了实现更安全通信，往往还与信号灯等同步机制共同使用。  
  
共享内存涉及到了存储管理以及文件系统等方面的知识，深入理解其内部机制有一定的难度，关键还要紧紧抓住内核使用的重要数据结构。系统V共享内存是以文件的形式组织在特殊文件系统shm中的。通过shmget可以创建或获得共享内存的标识符。取得共享内存标识符后，要通过shmat将这个内存区映射到本进程的虚拟地址空间。

**42、银行家算法**

答：**死锁的定义>**

   如果一组进程中的每一个进程都在等待仅由该组进程中的其他进程才能引发的事件，那仫该组进程就是死锁的.

**产生死锁的必要条件>**

    1).互斥条件:进程对所分配到的资源进行排它性使用，即在一段时间内，某资源只能被一个进程占用。如果此时还有其他进程请求该资源，则请求资源只能等待，直至占有该资源的进程用毕释放.

    2).请求和保持条件:进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源已经被其他进程占有，此时请求进程被保持，但对自己所获得的资源又保持不放.

    3).不可抢占条件:进程已获得的资源在未使用完之前不可被抢占，只能在使用完成后自己释放.

    4).环路等待:在发生死锁时，必然存在一个进程，资源的循环链.

**死锁产生的原因>**

   1).竞争可重用不可抢占式的资源

   2).竞争可消耗资源

   3).进程推进顺序不当.

**可重用性资源**:可供重复使用多次的资源.

**不可抢占性资源**:一旦系统把某资源分配给该进程后，就不能将它强行收回，只能在进程使用完后自动释放.

**可消耗资源**:又叫临时性资源，它是在进程运行期间，由进程动态的创建和消耗的.

 利用银行家算法解决死锁>

1).银行家算法中的数据结构

      (1).可利用资源向量Available

      (2).最大需求矩阵Max

      (3).分配矩阵Allocation

      (4).需求矩阵Need

2).银行家算法

Request请求向量，

     (1).如果Request[i] <= Need[i][j]转下步，否则它所需要的资源数已超过它所需要的最大值

     (2).如果Request[i] <= Available[i][j]转下步，否则尚无足够资源，进程需等待

     (3).系统试分配给进程p,并修改Available,Allocation和Need

                   Available[j] -= Request[j]

                   Allocation[i][j] += Request[j]

                   Need[i][j] -= Request[j]

     (4)系统执行安全性算法，检查此次资源分配后系统是否处于安全状态.若安全，才正式分配；否则恢复原来的分配状态，让该进程等待

3).安全性算法

    (1).设置两个向量,工作向量Work,在执行安全性算法开始时 Work=Available;Finish:表示有足够的资源分配给进程，使之运行完成,Finish[i]=false;当有足够资源分配给进程时，再另Finish[i]=false

   (2).从进程集合中找到一个满足该条件的进程:

           Finish[i]=false

           Need[i][j] <= Work[j]

   (3).当进程获得资源后，可顺利执行，并修改Work向量和Finsh向量

          Work[i] += Allocation[i][j]

          Finish[i]=true

   (4).如果所有进程的Finish[i]=true说明系统处于安全状态，否则系统处于不安全状态.

   在实现这份代码的时候陷入了一个误区那就是当你找到了一个安全序列之后，它查找过的Finish必定会被修改为true,而Finish这个数组又是在全局定义的，所以只要找到一次正确的安全序列这个Finish所找到的位就会被置为true会一直出现找到安全序列的，所以在找到安全序列之后一定要将Finish重新赋初值false,这个问题让我排错了半天，在定义变量的时候最好不要定义为全局的。。。

1. const int MAXPROCESS=100;
2. const int MAXRESOURCE=100;
4. int Available[MAXRESOURCE]; *//可用资源向量*
5. int Max[MAXPROCESS][MAXRESOURCE]; *//最大需求矩阵*
6. int Allocation[MAXPROCESS][MAXRESOURCE]; *//分配矩阵*
7. int Need[MAXPROCESS][MAXRESOURCE]; *//需求矩阵*
9. int Request[MAXRESOURCE]; *//请求向量*
11. int Work[MAXRESOURCE]; *//工作向量*
12. bool Finish[MAXPROCESS];
13. int SafeSeries[MAXPROCESS]; *//安全序列*
15. int n; *//进程数*
16. int m; *//资源数*
18. void Init()
19. {
20. cout<<"请输入进程总数>";
21. cin>>n;
22. cout<<"请输入资源种类数>";
23. cin>>m;
24. int i,j;
25. printf("请输入%d类资源的当前可利用资源数目>\n",m);
26. for( i = 0; i < m; ++i )
27. {
28. cin >> Available[i];
29. }
30. printf("最大需求矩阵(%d\*%d输入)>\n",n,m);
31. for( i = 0; i < n; ++i )
32. {
33. for( j = 0; j < m; ++j )
34. {
35. cin >> Max[i][j];
36. }
37. }
38. printf("分配矩阵(%d\*%d输入)>\n",n,m);
39. for( i = 0; i < n; ++i )
40. {
41. for( j = 0; j < m; ++j )
42. {
43. cin >> Allocation[i][j];
44. }
45. }
46. printf("需求矩阵(%d\*%d输入)\n",n,m);
47. for( i = 0; i < n; ++i )
48. {
49. for( j = 0; j < m; ++j )
50. {
51. cin >> Need[i][j];
52. }
53. }
54. }
56. bool IsSafe()
57. {
58. int i=0;
59. for (i=0;i<n;++i)
60. {
61. if(Finish[i] == true)
62. continue;
63. else
64. break;
65. }
66. if(i == n)
67. return true; *//安全*
68. else
69. return false; *//不安全*
70. }
72. bool Select(int &tmp,int tmpNeed[][MAXRESOURCE])
73. {
74. *//选择一个Finish[i]=false,Need[i]<=Work[i]*
75. int j=0;
76. for (int i=0;i<n;++i)
77. {
78. if(Finish[i])
79. continue;
80. for (j=0;j<m;++j)
81. {
82. if(tmpNeed[i][j] > Work[j])
83. break;
84. }
85. if(j == m)
86. {
87. tmp=i;
88. return true;
89. }
90. }
91. return false;
92. }
94. bool Safe(int \*tmpAvail,int tmpAlloc[][MAXRESOURCE],int tmpNeed[][MAXRESOURCE])
95. {
96. for(int i = 0; i < n; ++i)
97. {
98. Finish[i] = false;
99. }
100. for (int j = 0; j < m; ++j)
101. {
102. Work[j] = tmpAvail[j];
103. }
104. int tmp=0;
105. int index = 0;
106. while(Select(tmp,tmpNeed))
107. {
108. Finish[tmp] = true;
109. for (int k = 0; k < m; ++k)
110. {
111. Work[k] += tmpAlloc[tmp][k];
112. }
113. SafeSeries[index++] = tmp;
114. }
115. if(IsSafe())
116. return true; *//安全*
117. else
118. return false; *//不安全*
119. }
121. void Display()
122. {
123. int i=0;
124. int j=0;
125. cout<<"当前可利用的资源数目"<<endl;
126. for(i = 0; i < m; ++i)
127. {
128. cout << Available[i]<<" ";
129. }
130. cout<<endl;
131. cout<<"最大需求矩阵"<<endl;
132. for(i = 0; i < n; ++i )
133. {
134. for( j = 0; j < m; ++j)
135. {
136. cout<<Max[i][j]<<" ";
137. }
138. cout<<endl;
139. }
140. cout<<"分配矩阵"<<endl;
141. for( i = 0; i < n; ++i )
142. {
143. for( j = 0; j < m; ++j )
144. {
145. cout<<Allocation[i][j]<<" ";
146. }
147. cout<<endl;
148. }
149. cout<<"需求矩阵"<<endl;
150. for( i = 0; i < n; ++i )
151. {
152. for( j = 0; j < m; ++j )
153. {
154. cout<<Need[i][j]<<" ";
155. }
156. cout<<endl;
157. }
158. }
160. void BankA()
161. {
162. int i=0;
163. int index=0;
164. cout<<"请输入请求资源的进程下标>";
165. cin>>index;
166. assert(index < n && index >= 0);
167. cout<<"请输入当前的请求资源>"<<endl;
168. for (i=0;i<m;++i)
169. {
170. cin>>Request[i];
171. }
172. for (i=0;i<m;++i)
173. {
174. if(Request[i] <= Need[index][i])
175. {
176. continue;
177. }
178. else
179. {
180. cout<<"第一次试分配失败"<<endl;
181. break;
182. }
183. }
184. if(i < 3) *//如果第一次试分配失败则不执行后面的语句*
185. {
186. return ;
187. }
188. for(i=0;i<m;++i)
189. {
190. if(Request[i] <= Available[i])
191. {
192. continue;
193. }
194. else
195. {
196. cout<<"第二次试分配失败"<<endl;
197. break;
198. }
199. }
200. if(i < 3) *//如果第二次试分配失败则不同执行后面的语句*
201. {
202. return ;
203. }
204. *//开始试分配*
205. int tmpAvail[MAXRESOURCE]={0};
206. int tmpAlloc[MAXPROCESS][MAXRESOURCE]={0};
207. int tmpNeed[MAXPROCESS][MAXRESOURCE]={0};
208. memmove(tmpAvail,Available,sizeof(int)\*MAXRESOURCE);
209. memmove(tmpAlloc,Allocation,sizeof(int)\*MAXPROCESS\*MAXRESOURCE);
210. memmove(tmpNeed,Need,sizeof(int)\*MAXPROCESS\*MAXRESOURCE);
211. for (int i=0;i<m;++i)
212. {
213. tmpAvail[i] -= Request[i];
214. tmpAlloc[index][i] += Request[i];
215. tmpNeed[index][i] -= Request[i];
216. }
217. *//开始执行安全性算法*
218. bool ret=Safe(tmpAvail,tmpAlloc,tmpNeed);
219. if(ret == true)
220. {
221. *//如果试分配成功则更新Available,Allocation,Allocation的状态*
222. memmove(Available,tmpAvail,sizeof(int)\*MAXRESOURCE);
223. memmove(Allocation,tmpAlloc,sizeof(int)\*MAXPROCESS\*MAXRESOURCE);
224. memmove(Need,tmpNeed,sizeof(int)\*MAXPROCESS\*MAXRESOURCE);
225. cout<<"进程p"<<index<<"请求资源允许"<<endl;
226. }
227. else
228. {
229. *//只要试分配失败则将Finish置为false*
230. for(int i = 0; i < n; ++i)
231. {
232. Finish[i] = false;
233. }
234. cout<<"第三次试分配失败"<<endl;
235. }
236. }
238. void Menu()
239. {
240. cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*银行家算法\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;
241. cout<<"\*\*\*\*\*\*\*\*\*\*1.测试安全性代码\*\*\*\*\*\*\*\*\*\*"<<endl;
242. cout<<"\*\*\*\*\*\*\*\*\*\*2.测试银行家算法\*\*\*\*\*\*\*\*\*\*"<<endl;
243. cout<<"\*\*\*\*\*\*\*\*\*\*3.初始化\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;
244. cout<<"\*\*\*\*\*\*\*\*\*\*4.打印矩阵\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;
245. cout<<"\*\*\*\*\*\*\*\*\*\*0.退出\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;
246. }

   test.cpp

1. void testBank()
2. {
3. int index=0;
4. int flag=1;
5. while (flag)
6. {
7. Menu();
8. cout<<"请输入您的选择"<<endl;
9. cin>>index;
10. switch(index)
11. {
12. case 1:
13. Safe(Available,Allocation,Need);
14. if(IsSafe())
15. {
16. cout<<"该时刻是安全的，安全序列为>";
17. for (int i = 0; i < n ; ++i)
18. {
19. printf("p%d->",SafeSeries[i]);
20. }
21. cout<<"end"<<endl;
22. *//分配成功将Finish的值置为false*
23. for(int i = 0; i < n; ++i)
24. {
25. Finish[i] = false;
26. }
27. }
28. else
29. {
30. cout<<"该时刻是不安全的"<<endl;
31. }
32. break;
33. case 2:
34. BankA();
35. if(IsSafe())
36. {
37. cout<<"安全序列为>";
38. for (int i = 0; i < n ; ++i)
39. {
40. printf("p%d->",SafeSeries[i]);
41. }
42. cout<<"end"<<endl;
43. *//分配成功将Finish的值置为false*
44. for(int i = 0; i < n; ++i)
45. {
46. Finish[i] = false;
47. }
48. }
49. else
50. {
51. cout<<"进程请求资源不被允许"<<endl;
52. }
53. break;
54. case 3:
55. Init();
56. break;
57. case 4:
58. Display();
59. break;
60. case 0:
61. flag=0;
62. break;
63. default:
64. cout<<"您的输入错误,请重新输入"<<endl;
65. break;
66. }
67. }
68. }

  虽然搞了一下午写完整了这个银行家算法，整体效果还算满意，但是作为网络工程专业的学生今年虽然学了java的GUI但是仍然不会用图形用户界面(GUI)来对这个程序做一个界面出来，感觉好悲催哭,以后一定努力学习这方面的知识

  实现效果

**43、linux中断响应机制**

答：根据中断来源分为：内部中断（来自于CPU内部，如中断指令、溢出等）和外部中断（来自于外设，如读写完成中断）

根据是都可屏蔽分成：可屏蔽中断和不屏蔽中断（NMI），可屏蔽中断通过中断屏蔽字被屏蔽，屏蔽后，该中断不再的导向响应；不屏蔽中断不能被屏蔽。

根据中断入口跳转方法的不同，分成向量中断和非向量中断

采用向量中断的CPU通常给不同的中断号分配不同的中断号，当检查到某中断号的中断到来后，就自动跳转到与该中断号对应的地址执行。不同中断号的中断有不同的入口地址。非向量中断的多个中断共享一个入口地址，进入该入口地址后再通过软件判断中断标志来识别具体是哪个中断。也就是说，向量中断由硬件提供中断服务程序的入口地址，非向量中断由软件提供中断服务程序的入口地址。

定时器在硬件上也依赖中断实现，嵌入式可编程定时器（PIT）工作原理如下：

-----时钟输入--→预设计数值==？当前计数值-------Y-----→输出中断（并复位当前计数值变量）

工作原理：当定时器接收到一个时钟输入，当时钟脉冲到来时，将当前计数值+1并与预先设置的计数值（计数比较）比较，若相等，证明计数周期满，产生定时器中断并复位当前计数值。

Linux中断处理程序架构

设备的中断会打断内核中进程的正常调度和运行。

Linux内核的中断机制：为了在中断执行时间尽可能段和中断处理完成大量工作之间找到一个平衡点，Linux将中断处理程序分成两个半部，顶半部（top half）和底半部（bottom half）。

中断-----→上半部（紧急的硬件操作）----调度------→下半部（延缓的耗时操作）

顶半部完成尽可能少的比较紧急的功能，往往只是简单读取寄存器的中断状态并清除中断标志后就进行“登记中断”的工作。“登记中断”意味着将底半部处理程序挂到该设备的底半部执行队列中。这样，顶半部执行的速度会很快，可以服务更多的中断请求。

所以底半部来完成中断时间的绝大多数任务（几乎是中断要做的所有事情），而且可以被新的中断的打断，这是底半部和顶半部最大的区别，因为顶半部往往设计成不可中断。底半部一般不再硬件中断服务程序中执行。但是也不能僵化的认为Linux设备的驱动程序一定就是分成两部分，要是中断要处理的工作本身很少，则完全可以直接在顶半部全部完成。

Linux中断编程

中断申请（申请IRQ）-----request\_irq()

释放IRQ------free\_irq()

使能和屏蔽IRQ

disable\_irq()       disable\_irq\_nosync()     enable\_irq()

底半部机制：Linux实现底半部机制主要方式有   tasklet 工作队列  软中断

44、如何实现守护进程

守护进程（Daemon）是运行在后台的一种特殊进程。它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。守护进程是一种很有用的进程。

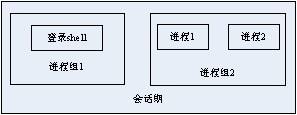
1、守护进程最重要的特性是后台运行。  
2、守护进程必须与其运行前的环境隔离开来。这些环境包括未关闭的文件描述符，控制终端，会话和进程组，工作目录以及文件创建掩模等。这些环境通常是守护进程从执行它的父进程（特别是shell）中继承下来的。  
3、守护进程的启动方式有其特殊之处。它可以在Linux系统启动时从启动脚本/etc/rc.d中启动，可以由作业规划进程crond启动，还可以由用户终端（shell）执行。

　　总之，除开这些特殊性以外，守护进程与普通进程基本上没有什么区别。因此，编写守护进程实际上是把一个普通进程按照上述的守护进程的特性改造成为守护进程。如果对进程有比较深入的认识就更容易理解和编程了。

守护进程之编程规则  
**（1）首先要做的是调用umask将文件模式创建屏蔽字设置为0。**  
　　文件权限掩码：是指屏蔽掉文件权限中的对应位。例如，有个文件权限掩码是050，它就屏蔽了文件组拥有者的可读与可执行权限（对应二进制为，rwx, 101）。由于fork函数创建的子进程继承了父进程的文件权限掩码，这就给子进程使用文件带来了诸多的麻烦。因此，把文件权限掩码设置为0（即，不屏蔽任何权限），可以增强该守护进程的灵活性。设置文件权限掩码的函数是umask。通常的使用方法为umask(0)。

**（2）调用fork，然后使父进程退出（exit）**。if(pid=fork()) exit(0);

**（3）调用setsid以创建一个新会话，脱离控制终端和进程组**。setsid函数作用：用于创建一个新的会话，并担任该会话组的组长。



　　调用setsid有3个作用：(a) 让进程摆脱原会话的控制；(b) 让进程摆脱原进程组的控制；(c) 让进程摆脱原控制终端的控制； setsid()  
　　使用setsid函数的目的：由于创建守护进程的第一步调用了fork函数来创建子进程再将父进程退出。由于在调用fork函数时，子进程拷贝了父进程的会话期、进程组、控制终端等，虽然父进程退出了，但会话期、进程组、控制终端等并没有改变，因此，这还不是真正意义上的独立开了。使用setsid函数后，能够使进程完全独立出来，从而摆脱其他进程的控制。

**（4）将当前工作目录更改为根目录。**#define NOFILE 256 for(i=0;i<NOFILE;i++) close(i);

**（5）关闭不再需要的文件描述符。**这使守护进程不再持有从其父进程继承来的某些文件描述符（父进程可能是shell进程，或某个其他进程）。

**（6）某些守护进程打开/dev/null使其具有文件描述符0、1和2**，这样，任何一个试图读标准输入、写标准输出和标准出错的库例程都不会产生任何效果。因为守护进程并不与终端设备相关联，所以不能在终端设备上显示其输出，也无处从交互式用户那里接受输入。

#include <sys/types.h>

#include<sys/stat.h>

#include<sys/time.h>

#include<sys/resource.h>

#include<unistd.h>

#include<fcntl.h>

#include<stdlib.h>

#include<syslog.h>

void daemonize(const char \*cmd)

{

int i, fd0, fd1, fd2;

pid\_t pid;

struct rlimit rl;

struct sigaction sa;

umask(0); // Clear file creation mask.

if (getrlimit(RLIMIT\_NOFILE, &rl) < 0) { // Get maximum number of file descriptors.

err\_quit("%s: can't get file limit", cmd);

}

if ((pid = fork()) < 0) { //这一步fork保证进程不是进程组组长进程

err\_quit("%s: can't fork", cmd);

}

else if (pid != 0) { /\* parent \*/

exit(0);

}

setsid(); // 创建一个回话，会话只包含子进程，且子进程是会话首进程

/\*

会话首进程的退出会出发SIGHUP信号

默认此信号的操作会终止进程

\*/

sa.sa\_handler = SIG\_IGN;

sigemptyset(&sa.sa\_mask);

sa.sa\_flags = 0;

if (sigaction(SIGHUP, &sa, NULL) < 0) {

err\_quit("%s: can't ignore SIGHUP", cmd);

}

/\*

再次创建子进程，退出父进程，保证守护进程不是会话首进程，这样open的时候就不会被分配终端

\*/

if ((pid = fork()) < 0) {

err\_quit("%s: can't fork", cmd);

}

else if (pid != 0) { /\* parent \*/

exit(0);

}

if (chdir("/") < 0) { // 改变当前工作路径为根目录

err\_quit("%s: can't change directory to /", cmd);

}

if (rl.rlim\_max == RLIM\_INFINITY) { //关闭所有打开的文件描述符

rl.rlim\_max = 1024;

}

for (i = 0; i < rl.rlim\_max; i++) {

close(i);

}

/\*

因为前面关闭了所有的文件描述符，此时open返回的必定是最小的0，后面两次dup返回的依次是1、2，

也就完成了对标准输入、标准输出、标准错误重定向至/dev/null的操作

\*/

fd0 = open("/dev/null", O\_RDWR);

fd1 = dup(0);

fd2 = dup(0);

/\*

\* Initialize the log file.

\*/

openlog(cmd, LOG\_CONS, LOG\_DAEMON);

if (fd0 != 0 || fd1 != 1 || fd2 != 2)

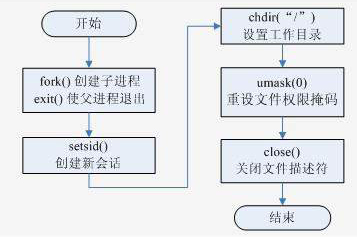
{

syslog(LOG\_ERR, "unexpected file descriptors %d %d %d",fd0, fd1, fd2);

exit(1);

}

}



45、32位系统一个进程最多有多少堆内存

理论上是4G. Linux实现的是 虚拟地址的前3G供给用户态的进程. 后1G是内核的部分. 也就是用户态的进程不能访问0xc0000000以上的虚拟地址.

46、线程安全和不安全的讨论

**线程安全**就是多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问直到该线程读取完，其他线程才可使用。不会出现数据不一致或者数据污染。

**线程不安全**就是不提供数据访问保护，有可能出现多个线程先后更改数据造成所得到的数据是脏数据

线程安全问题都是由[全局变量](http://baike.baidu.com/view/261041.htm)及[静态变量](http://baike.baidu.com/view/675642.htm)引起的。若每个线程中对[全局变量](http://baike.baidu.com/view/261041.htm)、[静态变量](http://baike.baidu.com/view/675642.htm)只有读操作，而无写操作，一般来说，这个全局变量是线程安全的；若有多个线程同时执行写操作，一般都需要考虑[线程同步](http://baike.baidu.com/view/6501899.htm)，否则的话就可能影响线程安全。

47、可重入函数与线性安全的区别与联系

当谈及可重入函数与线性安全的区别与关系时，先说一下可重入函数与线性安全它们各自的概念。

1、什么是可重入函数？   
一个函数被不同控制流调用，有可能在第一次调用还没返回时就再次进入该函数，这称为重入；   
如果一个函数只访问自己的局部变量或参数，则可称为可重入函数；

2、线性安全   
多个线程访问时，采用了枷锁机制，当一个线程访问该类的某个数据时，进行保护，其他进程不能进行访问直到该进程读取完，其它线程才可使用，不会出现数据不一致或者数据污染；

不提供数据访问保护，有可能出现多个线程先后更改数据造成，所得到的数据是脏数据；

3.可重入函数与线性安全的区别与关系   
1）线程安全是在多个线程情况下引发的，而可重入函数可以在只有一个线程的情况下来说；   
2）线程安全不一定是可重入的，而可重入函数则一定是线程安全的；   
3）如果一个函数中有全局变量，那么这个函数既不是线程安全也不是可重入的；   
4）如果将对临界资源的访问加上锁，则这个函数线程安全的；但如果这个重入函数若加锁未释放则会产生死锁，因此是不可重入的；   
5）线程安全函数能够是不同的线程访问同一块地址空间，而可重入函数要求不同的执行流对数据的操作互不影响，使结果是相同的；   
6）如果一个函数当中全是自身占栈空间的，那么既是线程安全又是可重入的

48、双重检查锁定模式的风险

DCLP的关键之处在于我们观察到的这一现象：调用者在调用instance()时，pInstance在大部分时候都是非空的，因此没必要再次初始化。所以，DCLP在加锁之前先做了一次pInstance是否为空的检查。只有判断结果为真（即pInstance还未初始化），加锁操作才会进行，然后再次检查pInstance是否为空（这就是该模式被命名为双重检查的原因）。第二次检查是必不可少的，因为，正如我们之前的分析，在第一次检验pInstance和加锁之间，可能有另一个线程对pInstance进行初始化。

49、内存屏障详解

内存屏障，在x86 上是”sfence”指令，强迫所有的、在屏障指令之前的 存储指令在屏障以前发生，并且让 store buffers 刷新到发布这个指令的 CPU cache。这将使程序状态对其他 CPU 可见，这样，如果需要它们可以对它做出响应。一个实际的好例子是下面的、简化的、来自Disruptor的类[BatchEventProcessor](http://code.google.com/p/disruptor/source/browse/trunk/code/src/main/com/lmax/disruptor/BatchEventProcessor.java)。当sequence被更新后，其他消费者和生产者知道这个消费者的进展，并进行适当的响应。所有在屏障之前对内存的更新现在都可见了

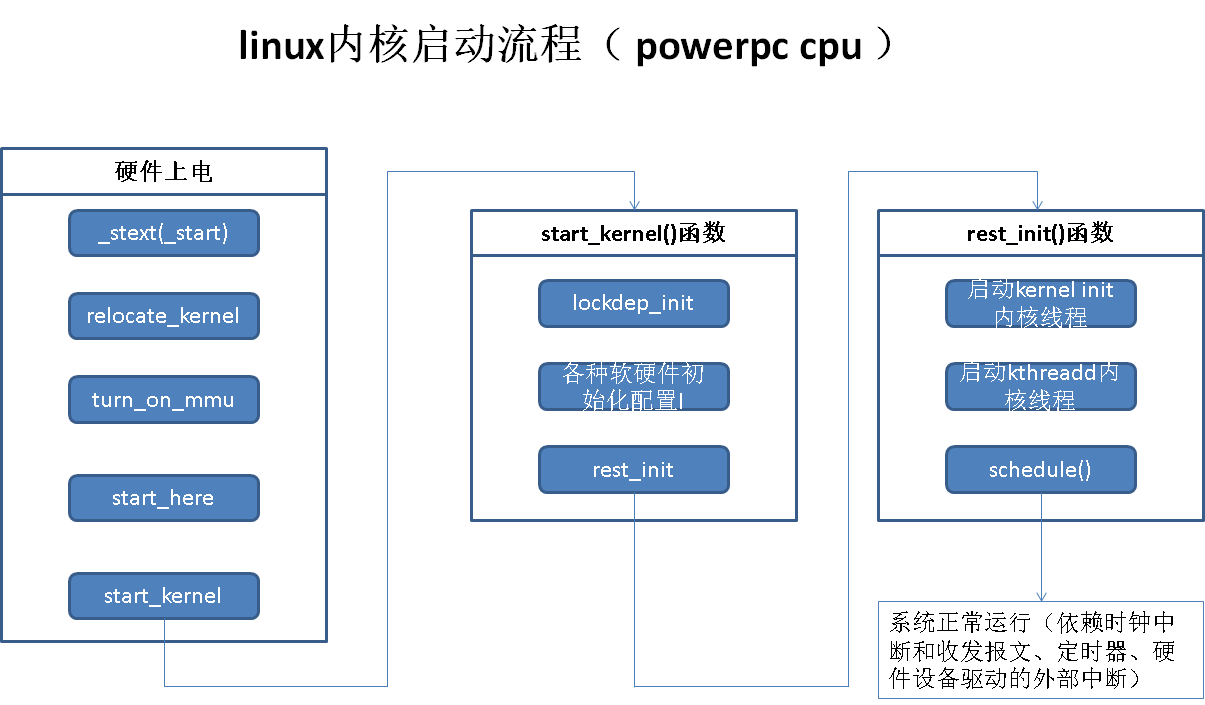
50、原子操作原理

32位IA-32处理器使用基于对缓存加锁或总线加锁的方式来实现多处理器之间的原子操作。首先处理器会自动保证基本的内存操作的原子性。处理器保证从系统内存中读取或者写入一个字节是原子的，意思是当一个处理器读取一个字节时，其他处理器不能访问这个字节的内存地址。Pentium 6和最新的处理器能自动保证单处理器对同一个缓存行里进行16/32/64位的操作是原子的，但是复杂的内存操作处理器是不能自动保证其原子性的，比如跨总线宽度、跨多个缓存行和跨页表的访问。但是，处理器提供总线锁定和缓存锁定两个机制来保证复杂内存操作的原子性。

51.Linux的内核线程

1、内核线程，只是一个称呼，实际上就是一个进程，有自己独立的TCB，参与内核调度，也参与内核抢占。这个进程的特别之处有两点，第一、该进程没有前台。第二、永远在内核态中运行

2、创建内核线程有两种方法，一种是 kthread\_create() ，一种是 kernel\_thread()



**52.使用线程是如何防止出现大的波峰**

意思是如何防止同时产生大量的线程，方法是使用线程池，线程池具有可以同时提高调度效率和限制资源使用的好处，线程池中的线程达到最大数时，其他线程就会排队等候。

**53.操作系统进程调试策略**

1、FCFS（First come first serve)，或者称为FIFO算法，先来先处理。这个算法的优点是简单，实现容易，并且似乎公平；缺点在于短的任务有可能变的非常慢，因为其前面的任务占用很长时间，造成了平均响应时间非常慢。

2、时间片轮询算法，这是对FIFO算法的改进，目的是改善短程序（运行时间短）的响应时间，其方法就是周期性地进行进程切换。这个算法的关键点在于时间片的选择，时间片过大，那么轮转就越接近FIFO，如果太小，进程切

换的开销大于执行程序的开销，从而降低了系统效率。因此选择合适的时间片就非常重要。选择时间片的两个需要考虑的因素：一次进程切换所使用的系统消耗以及我们能接受的整个系统消耗、系统运行的进程数。时间片轮询看上起非常公平，并且响应时间非常好，然而时间片轮转并不能保证系统的响应时间总是比FIFO短，这很大程度上取决于时间片大小的选择，以及这个大小与进程运行时间的相互关系。

3、STCF算法（Short time to complete first)，顾名思义就是短任务优先算法。这种算法的核心就是所有的程序都有一个优先级，短任务的优先级比长任务的高，而OS总是安排优先级高的进程运行。

STCF又分为两类：非抢占式和抢占式。非抢占式STCF就是让已经在CPU上运行的程序执行到结束或者阻塞，然后在所有的就绪进程中选择执行时间最短的来执行；而抢占式STCF就不是这样，在每进来一个新的进程时，就对所有进程（包括正在CPU上执行的进程）进行检查，谁的执行时间短，就运行谁。

STCF总是能提供最优的响应时间，然而它也有缺点，第一可能造成长任务的程序无法得到CPU时间而饥饿，因为OS总是优先执行短任务；其次，关键问题在于我们怎么知道程序的运行时间，怎么预测某个进程需要的执行时间？通常有两个办法：使用启发式方法估算（例如根据程序大小估算），或者将程序执行一遍后记录其所用的CPU时间，在以后的执行过程中就可以根据这个测量数据来进行STCF调度。

4、优先级调度，STCF遇到的问题是长任务的程序可能饥饿，那么优先级调度算法可以通过给长任务的进程更高的优先级来解决这个问题；优先级调度遇到的问题可能是短任务的进程饥饿，这个可以通过动态调整优先级来解决。实际上动态调整优先级（称为权值）+时间片轮询的策略正是linux的进程调度策略之一的 SCHED\_OTHER分时调度策略,它的调度过程如下：

（1）创建任务指定采用分时调度策略，并指定优先级nice值(-20~19)。

（2）将根据每个任务的nice值确定在cpu上的执行时间(counter)。

（3）如果没有等待资源，则将该任务加入到就绪队列中。

（4）调度程序遍历就绪队列中的任务，通过对每个任务动态优先级的计算(counter+20-nice)结果，选择计算结果最大的一个去运行，当这个时间片用完后(counter减至0)或者主动放弃cpu时，该任务将被放在就绪队列末尾(时间片用完)或等待队列(因等待资源而放弃cpu)中。

（5）此时调度程序重复上面计算过程，转到第4步。

（6）当调度程序发现所有就绪任务计算所得的权值都为不大于0时，重复第2步。

linux还有两个实时进程的调度策略：FIFO和RR,实时进程会立即抢占非实时进程。

5、显然，没有什么调度算法是毫无缺点的，因此现代OS通常都会采用混合调度算法。例如将不同的进程分为几个大类，每个大类有不同的优先级，不同大类的进程的调度取决于大类的优先级，同一个大类的进程采用时间片轮询来保证公平性。

6、其他调度算法，保证调度算法保证每个进程享用的CPU时间完全一样；彩票调度算法是一种概率调度算法，通过给进程“发彩票”的多少，来赋予不同进程不同的调用时间，彩票调度算法的优点是非常灵活，如果你给短任务发更多“彩票”，那么就类似STCF调度，如果给每个进程一样多的“彩票”，那么就类似保证调度；用户公平调度算法，是按照每个用户，而不是按照每个进程来进行公平分配CPU时间，这是为了防止贪婪用户启用了过多进程导致系

7、实时系统的调度算法，实时系统需要考虑每个具体任务的响应时间必须符合要求，在截止时间前完成。

（1）EDF调度算法，就是最早截止任务优先（Earliest deadline first)算法，也就是让最早截止的任务先做。当新的任务过来时，如果它的截止时间更靠前，那么就让新任务抢占正在执行的任务。EDF算法其实是贪心算法的一种体现。如果一组任务可以被调度（也就是所有任务的截止时间在理论上都可以得到满足），那么EDF可以满足。如果一批任务不能全部满足（全部在各自的截止时间前完成），那EDF满足的任务数最多，这就是它最优的体现。EDF其实就是抢占式的STCF，只不过将程序的执行时间换成了截止时间。EDF的缺点在于需要对每个任务的截止时间做计算并动态调整优先级，并且抢占任务也需要消耗系统资源。因此它的实际效果比理论效果差一点。

（2）RMS调度算法，EDF是动态调度算法，而RMS（rate monotonic scheduling)算法是一种静态最优算法；该算法在进行调度前先计算出所有任务的优先级，然后按照计算出来的优先级进行调度，任务执行中间既不接收新任务，也不进行优先级调整或者CPU抢占。因此它的优点是系统消耗小，缺点就是不灵活了。对于RMS算法，关键点在于判断一个任务组是否能被调度，这里有一个定律，如果一个系统的所有任务的CPU利用率都低于ln2，那么这些任务的截止时间均可以得到满足，ln2约等于0.693147，也就是此时系统还剩下有30%的CPU时间。这个证明是Liu和Kayland在1973年给出的。统效率降低甚至停顿。

linux内核的三种主要调度策略：

　　1，SCHED\_OTHER 分时调度策略，

　　2，SCHED\_FIFO实时调度策略，先到先服务

　　3，SCHED\_RR实时调度策略，时间片轮转

**54.线程和进程的区别线程是否有相同的堆栈，dll是否有独立的堆栈**

线程有自己的堆栈 ，DLL的堆栈是在运行DLL的线程中的DLL本身是没有的，不过，DLL能够在进程的地址空间中创建一个堆来供自己使用！

**55读者和写者问题**

问题描述

有读者和写者两组并发进程，共享一个文件，当两个或以上的读进程同时访问共享数据时不会产生副作用，但若某个写进程和其他进程（读进程或写进程）同时访问共享数据时则可能导致数据不一致的错误。因此要求：①允许多个读者可以同时对文件执行读操作；②只允许一个写者往文件中写信息；③任一写者在完成写操作之前不允许其他读者或写者工作；④写者执行写操作前，应让已有的读者和写者全部退出。

设有一文件F，多个并发读进程和写进程都要访问，要求：

(1)读写互斥

(2)写写互斥

(3)允许多个读进程同时访问

采用记录型信号量机制解决

https://blog.csdn.net/spadgerz/article/details/53045724

**56 哲学家进餐问题**

有五个哲学家，他们的生活方式是交替地进行思考和进餐。他们共用一张圆桌，分别坐在五张椅子上。

在圆桌上有五个碗和五支筷子，平时一个哲学家进行思考，饥饿时便试图取用其左、右最靠近他的筷子，只有在他拿到两支筷子时才能进餐。进餐完毕，放下筷子又继续思考。

哲学家进餐问题的改进解法

方法一：至多只允许四位哲学家同时去拿左筷子，最终能保证至少有一位哲学家能进餐，并在用完后释放两只筷子供他人使用。

方法二：仅当哲学家的左右手筷子都拿起时才允许进餐。

方法三：规定奇数号哲学家先拿左筷子再拿右筷子，而偶数号哲学家相反。

代码部分见：https://blog.csdn.net/qq\_28602957/article/details/53538329

**57 进程状态切换图**

