# 面试学习笔记

## 进程之间的通信

### 管道：

管道是最简单，效率最差的一种通信方式。

管道本质上就是内核中的一个缓存，当进程创建一个管道后，Linux会返回两个文件描述符，一个是写入端的描述符，一个是输出端的描述符，可以通过这两个描述符往管道写入或者读取数据。

如果想要实现两个进程通过管道来通信，则需要让创建管道的进程fork子进程，这样子进程们就拥有了父进程的文件描述符，这样子进程之间也就有了对同一管道的操作。

缺点：

半双工通信，一条管道只能一个进程写，一个进程读。

一个进程写完后，另一个进程才能读，反之同理。

### 消息队列：

管道的通信方式效率是低下的，不适合进程间频繁的交换数据。这个问题，消息队列的通信方式就可以解决。A进程往消息队列写入数据后就可以正常返回，B进程需要时再去读取就可以了，效率比较高。

而且，数据会被分为一个一个的数据单元，称为消息体，消息发送方和接收方约定好消息体的数据类型，不像管道是无格式的字节流类型，这样的好处是可以边发送边接收，而不需要等待完整的数据。

但是也有缺点，每个消息体有一个最大长度的限制，并且队列所包含消息体的总长度也是有上限的，这是其中一个不足之处。

另一个缺点是消息队列通信过程中存在用户态和内核态之间的数据拷贝问题。进程往消息队列写入数据时，会发送用户态拷贝数据到内核态的过程，同理读取数据时会发生从内核态到用户态拷贝数据的过程。

### 共享内存：

共享内存解决了消息队列存在的内核态和用户态之间数据拷贝的问题。

现代操作系统对于内存管理采用的是虚拟内存技术，也就是说每个进程都有自己的虚拟内存空间，虚拟内存映射到真实的物理内存。共享内存的机制就是，不同的进程拿出一块虚拟内存空间，映射到相同的物理内存空间。这样一个进程写入的东西，另一个进程马上就能够看到，不需要进行拷贝。

（这里的物理内存貌似不是内核空间的内存？）

### 信号量：

当使用共享内存的通信方式，如果有多个进程同时往共享内存写入数据，有可能先写的进程的内容被其他进程覆盖了。

因此需要一种保护机制，信号量本质上是一个整型的计数器，用于实现进程间的互斥和同步。

信号量代表着资源的数量，操作信号量的方式有两种：

P操作：这个操作会将信号量减一，相减后信号量如果小于0，则表示资源已经被占用了，进程需要阻塞等待；如果大于等于0，则说明还有资源可用，进程可以正常执行。

V操作：这个操作会将信号量加一，相加后信号量如果小于等于0，则表明当前有进程阻塞，于是会将该进程唤醒；如果大于0，则表示当前没有阻塞的进程。

（1）信号量实现互斥：

信号量初始化为1

进程 A 在访问共享内存前，先执行了 P 操作，由于信号量的初始值为 1，故在进程 A 执行 P 操作后信号量变为 0，表示共享资源可用，于是进程 A 就可以访问共享内存。

若此时，进程 B 也想访问共享内存，执行了 P 操作，结果信号量变为了 -1，这就意味着临界资源已被占用，因此进程 B 被阻塞。

直到进程 A 访问完共享内存，才会执行 V 操作，使得信号量恢复为 0，接着就会唤醒阻塞中的线程 B，使得进程 B 可以访问共享内存，最后完成共享内存的访问后，执行 V 操作，使信号量恢复到初始值 1。

（2）信号量实现同步：

由于多线程下各线程的执行顺序是无法预料的，有些时候我们希望多个线程之间能够密切合作，这时候就需要考虑线程的同步问题。

信号量初始化为0

如果进程 B 比进程 A 先执行了，那么执行到 P 操作时，由于信号量初始值为 0，故信号量会变为 -1，表示进程 A 还没生产数据，于是进程 B 就阻塞等待；

接着，当进程 A 生产完数据后，执行了 V 操作，就会使得信号量变为 0，于是就会唤醒阻塞在 P 操作的进程 B；

最后，进程 B 被唤醒后，意味着进程 A 已经生产了数据，于是进程 B 就可以正常读取数据了。

### 信号：

在Linux中，为了响应各种事件，提供了几十种信号，可以通过kill -l命令查看。

如果是运行在shell终端的进程，可以通过键盘组合键来给进程发送信号，例如使用Ctrl+C产生SIGINT信号，表示终止进程。

如果是运行在后台的进程，可以通过命令来给进程发送信号，例如使用kill -9 PID产生SIGKILL信号，表示立即结束进程。

### Socket：

前面提到的管道，消息队列，共享内存，信号量和信号都是在同一台主机上进行进程间通信，如果想要跨网络和不同主机上的进程进行通信，则需要用到socket。

实际上，Socket不仅可以跨网络和不同主机进行进程间通信，还可以在同一主机进行进程间通信。

Socket是操作系统提供给程序员操作网络的接口，根据底层不同的实现方式，通信方式也不同。

## 内核态和用户态

一、内核态、用户态概念

内核态：也叫内核空间，是内核进程/线程所在的区域。主要负责运行系统、硬件交互。

用户态：也叫用户空间，是用户进程/线程所在的区域。主要用于执行用户程序。

一般的操作系统对执行权限进行分级，分别为用用户态和内核态。

用户态相较于内核态有较低的执行权限，很多操作是不被操作系统允许的，原因简单来说就是用户态出现问题（自己写的bug），也不能让操作系统崩溃呀。

内核态相当于一个介于硬件与应用之间的层，内核有ring 0的权限，可以执行任何cpu指令，也可以引用任何内存地址，包括外围设备, 例如硬盘, 网卡，权限等级最高。

用户态则权利有限，例如在内存分配中，有一部分内存是仅为内核态使用的，用户态code则不允许访问那些内存地址，每个进程只允许访问自己申请到的内存。

而且不允许访问外围设备。另外在执行cpu指令的时候也可以被高优先级抢占。大多数时间各类程序都是执行在用户态下，毕竟内核就是基础而已。

二、内核态和用户态的区别

内核态：运行的代码不受任何限制，CPU可以执行任何指令。

用户态：运行的代码需要受到CPU的很多检查，不能直接访问内核数据和程序，也就是说不可以像内核态线程一样访问任何有效地址。

操作系统在执行用户程序时，主要工作在用户态，只有在其执行没有权限完成的任务时才会切换到内核态。

三、为什么要区分内核态和用户态

保护机制。防止用户进程误操作或者是恶意破坏系统。内核态类似于C++的私有成员，只能在类内访问，用户态类似于公有成员，可以随意访问。

四、用户态切换到内核态的方式

1、系统调用（主动）

由于用户态无法完成某些任务，用户态会请求切换到内核态，内核态通过为用户专门开放的中断完成切换。

2、异常（被动）

在执行用户程序时出现某些不可知的异常，会从用户程序切换到内核中处理该异常的程序，也就是切换到了内核态。

3、外围设备中断（被动）

外围设备发出中断信号，当中断发生后，当前运行的进程暂停运行，并由操作系统内核对中断进程处理，如果中断之前CPU执行的是用户态程序，就相当于从用户态向内核态的切换。

## 进程死锁的原因

答案：资源竞争及进程推进顺序非法

一、死锁的定义：

死锁 :是指两个或两个以上的进程在执行过程中,因争夺资源而造成的一种互相等待的现象,若无外力作用,它们都将无法推进下去。

二、原因：

（1）因为系统资源不足

（2）资源分配不当等

（3）进程运行推进顺序不合适

如果系统资源充足，进程的资源请求都能够得到满足，死锁出现的可能性就很低，否则就会因争夺有限的资源而陷入死锁。其次，进程运行推进顺序与速度不同，也可能产生死锁。

三、条件：

（1）互斥条件：一个资源每次只能被一个进程使用。

（2）不剥夺条件：进程已获得的资源，在末使用完之前，不能强行剥夺。

（3）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。

（4）循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

死锁的解除与预防：

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

其中最简单的方法就是线程都是以同样的顺序加锁和释放锁，也就是破坏了第四个条件。

四、编写死锁程序：

public class TestDeadLock {

private static Object ob1 = new Object();

private static Object ob2 = new Object();

public static void main(String[] args){

//线程0

new Thread(new Runnable(){

@Override

public void run() {

synchronized (ob1) {

System.out.println(Thread.currentThread().getName()+"获得ob1的锁");

try {

Thread.sleep(1000);

synchronized(ob2){

System.out.println(Thread.currentThread().getName()+"获得ob2的锁");

}

System.out.println(Thread.currentThread().getName()+"结束");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}).start();

//线程1

new Thread(new Runnable(){

@Override

public void run() {

synchronized (ob2) {

System.out.println(Thread.currentThread().getName()+"获得ob2的锁");

synchronized (ob1) {

System.out.println(Thread.currentThread().getName() + "获得ob1的锁");

}

System.out.println(Thread.currentThread().getName() + "结束");

}

}

}).start();

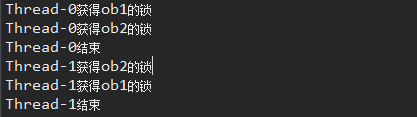
}

}

运行结果如下：

IMG_256

而不是期望的Thread-0和Thread-1能如下图一样按顺序推进执行



这是因为，Thread-0获得到ob1锁之后，sleep了1s，这个时候Thread-1已经获得了ob2的锁，而此时Thread-1想再获得ob1的锁，却发现被Thread-0占有了，同样1s后Thread-0想再获得ob2的锁，却发现被Thread-1占有了，两个线程各自占有的锁恰好是对方继续执行下去所必需的，Thread-0等待Thread-1释放ob2的锁，Thread-1等待Thread-0释放ob1的锁，相互循环，造成了死锁。

## 死锁的处理

答案：鸵鸟策略、预防策略、避免策略、检测与解除死锁

预防死锁：通过设置某些限制条件，去破坏产生死锁的四个必要条件中的一个或几个条件，来防止死锁的发生。

避免死锁：在资源的动态分配过程中，用某种方法去防止系统进入不安全状态，从而避免死锁的发生。

检测死锁：允许系统在运行过程中发生死锁，但可设置检测机构及时检测死锁的发生，并采取适当措施加以清除。

解除死锁：当检测出死锁后，便采取适当措施将进程从死锁状态中解脱出来。

6.1 预防死锁

破坏“互斥”条件:

就是在系统里取消互斥。若资源不被一个进程独占使用，那么死锁是肯定不会发生的。但一般来说在所列的四个条件中，“互斥”条件是无法破坏的。

破坏“占有并等待”条件:

破坏“占有并等待”条件，就是在系统中不允许进程在已获得某种资源的情况下，申请其他资源。即要想出一个办法，阻止进程在持有资源的同时申请其他资源。

方法一：一次申请所需的全部资源，即 “ 一次性分配”。

方法二：要求每个进程提出新的资源申请前，释放它所占有的资源。即“先释放后申请”。

破坏“不可抢占”条件：

破坏“不可抢占”条件就是允许对资源实行抢夺。

方法一：占有某些资源的同时再请求被拒绝，则该进程必须释放已占有的资源，如果有必要，可再次请求这些资源和另外的资源。

方法二：设置进程优先级，优先级高的可以抢占资源。

破坏“循环等待”条件：

将系统中的所有资源统一编号，所有进程必须按照资源编号顺序提出申请。

6.2避免死锁

加锁顺序：线程按照一定的顺序加锁。

加锁时限：线程尝试获取锁的时候加上一定的时限，超过时限则放弃对该锁的请求，并释放自己占有的锁。

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

6.3 检测死锁

一般来说，由于操作系统有并发，共享以及随机性等特点，通过预防和避免的手段达到排除死锁的目的是很困难的。这需要较大的系统开销，而且不能充分利用资源。为此，一种简便的方法是系统为进程分配资源时，不采取任何限制性措施，但是提供了检测和解脱死锁的手段：能发现死锁并从死锁状态中恢复出来。因此，在实际的操作系统中往往采用死锁的检测与恢复方法来排除死锁。

例如：

进程P1占有资源R1而申请资源R2，进程P2占有资源R2而申请资源R1，按循环等待条件，进程和资源形成了环路，所以系统是死锁状态。进程P1，P2是参与死锁的进程。

下面我们再来看一看死锁检测算法。算法使用的数据结构是如下这些：

占有矩阵A：n\*m阶，其中n表示并发进程的个数，m表示系统的各类资源的个数，这个矩阵记录了每一个进程当前占有各个资源类中资源的个数。

申请矩阵R：n\*m阶，其中n表示并发进程的个数，m表示系统的各类资源的个数，这个矩阵记录了每一个进程当前要完成工作需要申请的各个资源类中资源的个数。

空闲向量T：记录当前m个资源类中空闲资源的个数。

完成向量F：布尔型向量值为真（true）或假（false），记录当前n个并发进程能否进行完。为真即能进行完，为假则不能进行完。

临时向量W：开始时W：=T。

算法步骤：

（1）W：=T，

对于所有的i=1，2，…，n，

如果A[i]=0，则F[i]：=true；否则，F[i]：=false

（2）找满足下面条件的下标i：

F[i]：=false并且R[i]〈=W

如果不存在满足上面的条件i，则转到步骤（4）。

（3）W：=W+A[i]

F[i]：=true

转到步骤（2）

（4）如果存在i，F[i]：=false，则系统处于死锁状态，且Pi进程参与了死锁。什么时候进行死锁的检测取决于死锁发生的频率。如果死锁发生的频率高，那么死锁检测的频率也要相应提高，这样一方面可以提高系统资源的利用率，一方面可以避免更多的进程卷入死锁。如果进程申请资源不能满足就立刻进行检测，那么每当死锁形成时即能被发现，这和死锁避免的算法相近，只是系统的开销较大。为了减小死锁检测带来的系统开销，一般采取每隔一段时间进行一次死锁检测，或者在CPU的利用率降低到某一数值时，进行死锁的检测。

6.4 解除死锁

资源剥夺法：挂起某些死锁进程，并抢占它的资源，将这些资源分配给其他的死锁进程。但应防止被挂起的进程长时间得不到资源，而处于资源匮乏的状态。

撤销进程法：强制撤销部分、甚至全部死锁进程并剥夺这些进程的资源。撤销的原则可以按进程优先级和撤销进程代价的高低进行。

进程回退法：让一（多）个进程回退到足以回避死锁的地步，进程回退时自愿释放资源而不是被剥夺。要求系统保持进程的历史信息，设置还原点。