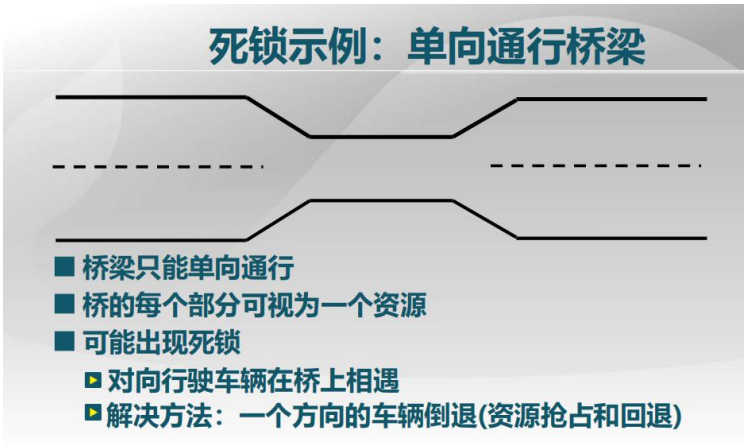
第一节 死锁概念

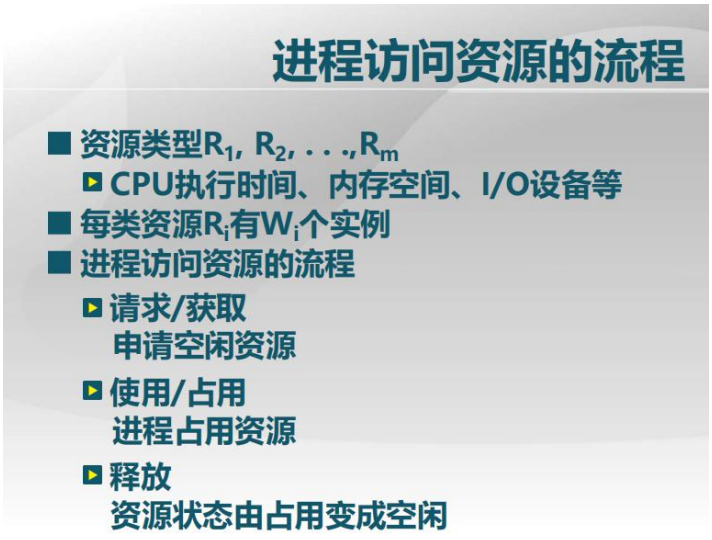
死锁是进程之间由于共享资源所导致的一种无限期等待的情况。

本讲还将介绍进程间通讯方法：信号、管道、消息队列和共享内存。

一个具体的例子(中间窄的那段是桥梁，两边宽的是马路可以同时双向通行)：

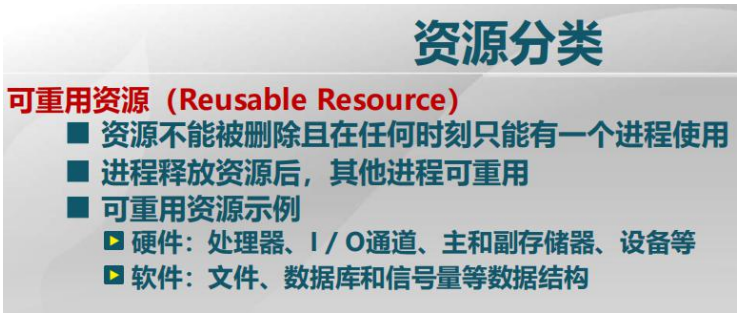


先看进程访问资源的流程：



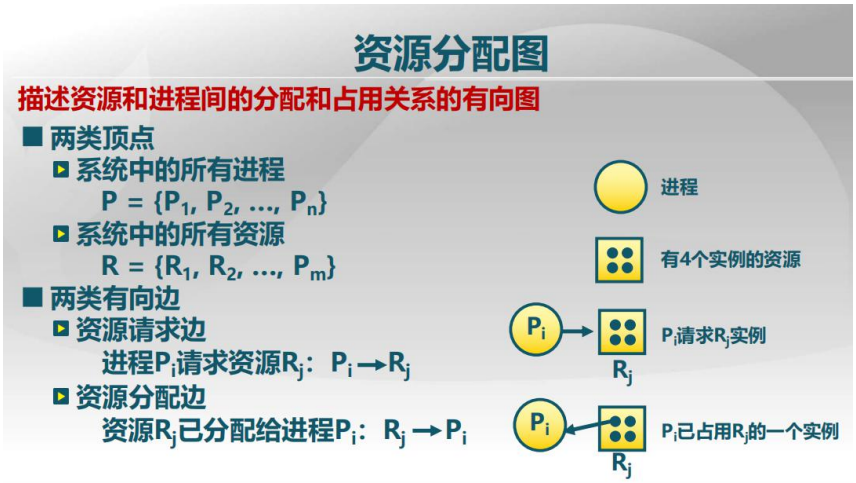
系统中存在各种类型的资源，每一类资源可能有多个实例。

资源的特征：



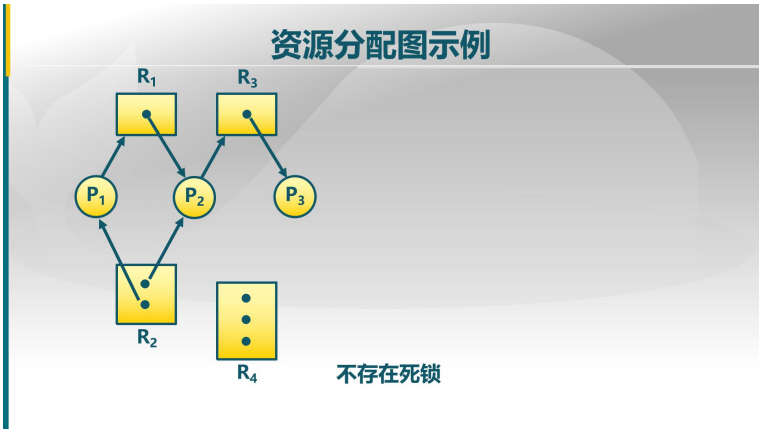
处理器即 CPU。文件(可以不删除的类型)，数据库，信号量也是可重用资源。

对资源和进程访问资源进行描述后，进程和资源之间的关系可描述如下：

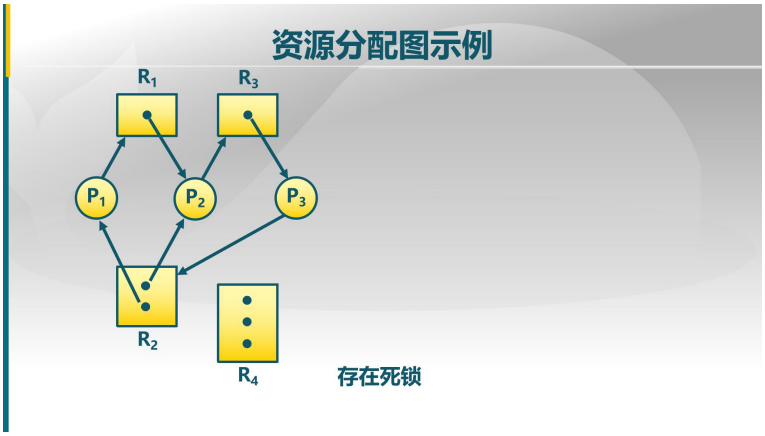


用上图就可表示进程和资源之间的申请和占用关系。

下面是实例：

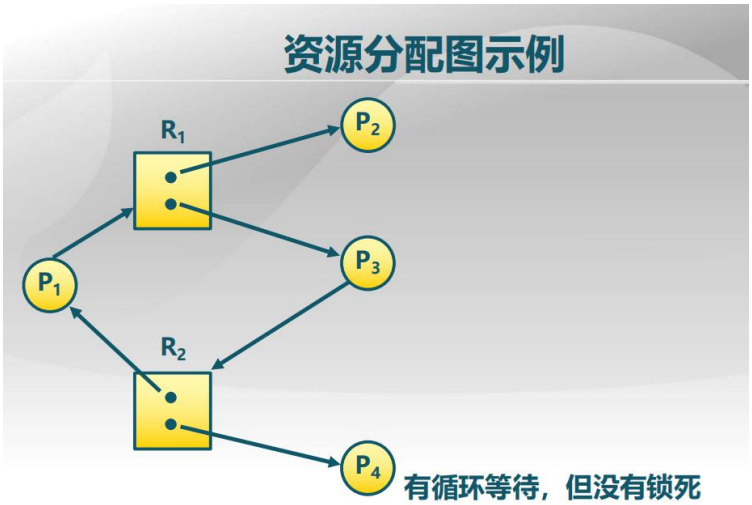


P3 占用 R3，P3 用完 R3 后可释放 R3，这时 P2 可运行，P2 运行结束释放资源后 P1 可运行。



上图出现了循环等待：P2 请求 R3，R3 却分配给了 P3，P3 请求 R2，R2 却分配给了 P2，出现死锁。对于这种简单的情况，可以直接从图上看出死锁，但实际系统中有几百个进程，有几十种或者更多资源，这时再判断就会有时间开销，也比较困难。

再看一例：



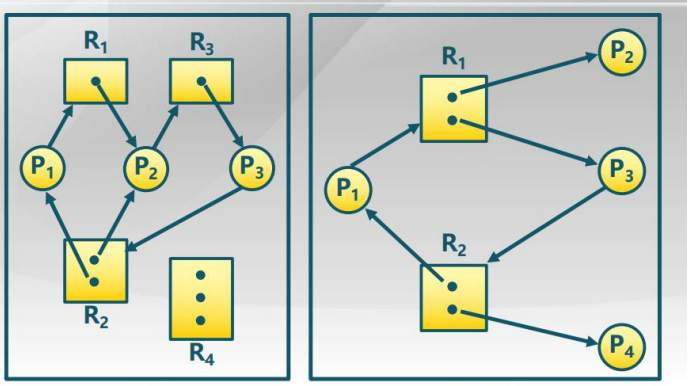
与上例相反，尽管本例也有循环等待，但不会出现死锁。因此不能仅仅依靠循环来判断是否出现死锁。

下面是出现死锁的必要条件：

出现死锁的必要条件

- 互斥
  - 任何时刻只能有一个进程使用一个资源实例
- 持有并等待
  - 进程保持至少一个资源，并正在等待获取其他进程持有的资源
- 非抢占
  - 资源只能在进程使用后自愿释放
- 循环等待
  - 存在等待进程集合{P_0，P_1，…，P_N}，
    P_0正在等待P_1所占用的资源，
    P_1 正在等待P_2占用的资源，…，
    P_{N-1}在等待P_N所占用资源，
    P_N正在等待P_0所占用的资源

出现死锁的必要条件



死锁　　没有死锁

根据以上 4 个条件可判断是否有死锁：右图中 P2 和 P4 不满足持有并等待。

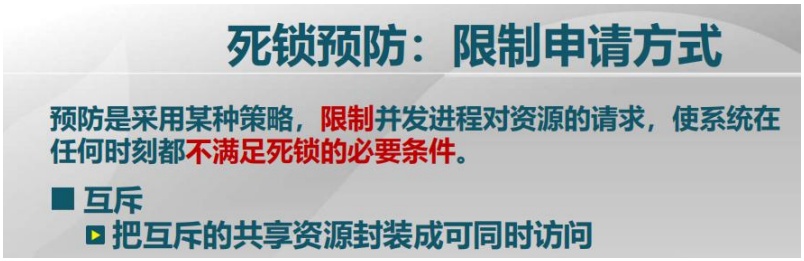
第二节 死锁处理方法

死锁处理方法

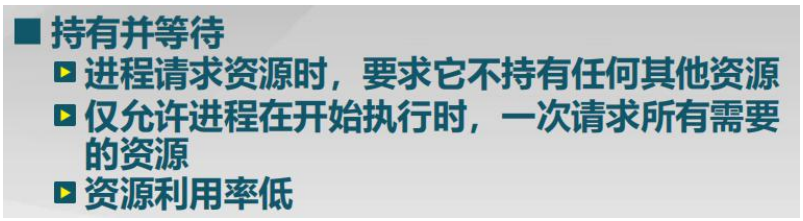
- 死锁预防(Deadlock Prevention)
  - 确保系统永远不会进入死锁状态
- 死锁避免(Deadlock Avoidance)
  - 在使用前进行判断，只允许不会出现死锁的进程请求资源
- 死锁检测和恢复(Deadlock Detection & Recovery)
  - 在检测到运行系统进入死锁状态后，进行恢复
- 由应用进程处理死锁
  - 通常操作系统忽略死锁
    - 大多数操作系统（包括UNIX）的做法

死锁预防：去掉形成死锁的必要条件，就不会进入死锁了，但利用这种方法系统的资源利用效率比较低。

下面看具体处理方法：



例子：打印机中添加缓冲，当有两个打印作业，把其中一个放在缓冲区中，待处理完一个再处理缓冲区中的。(我的理解：外面看是同时访问，但内部是在对进程获取资源的顺序进行排队等待)

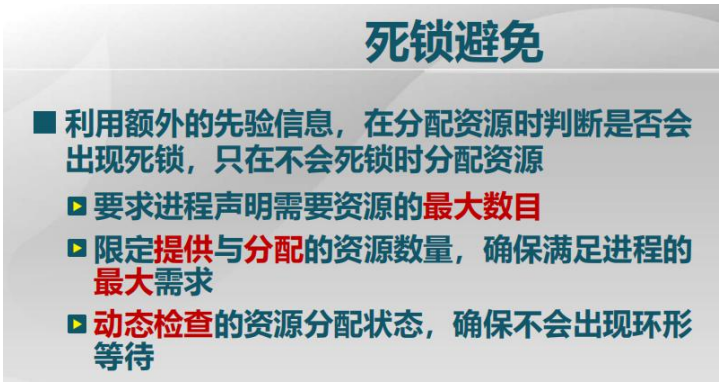


进程请求资源时，要求它不持有任何其他资源，即仅允许进程在开始执行时，一次请求所有需要的资源，其资源利用率低。

与持有并等待的做法有相似之处。

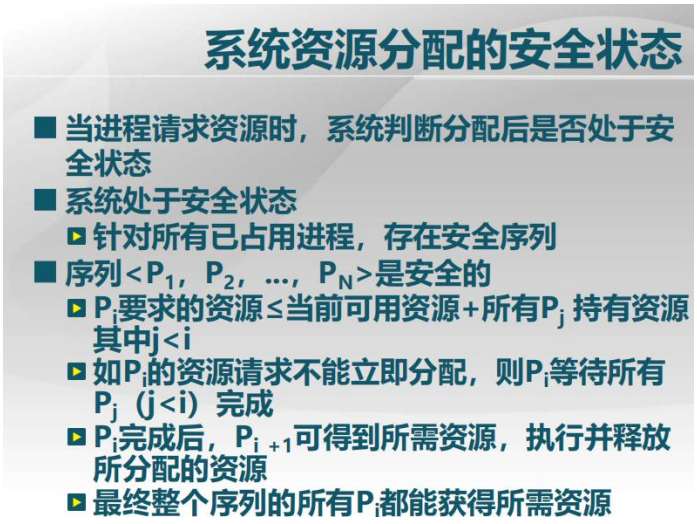
按照固定顺序申请资源，效率低。

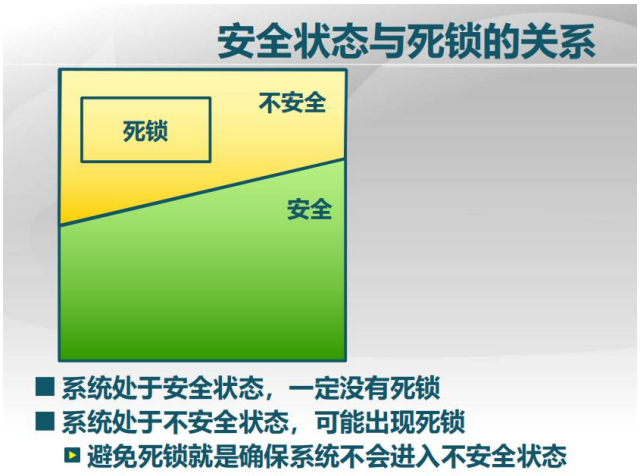
死锁避免方法：



死锁预防资源利用效率低，提高效率的办法是给进程分配资源时手续变复杂。死锁避免有风险，要求先验信息准确。

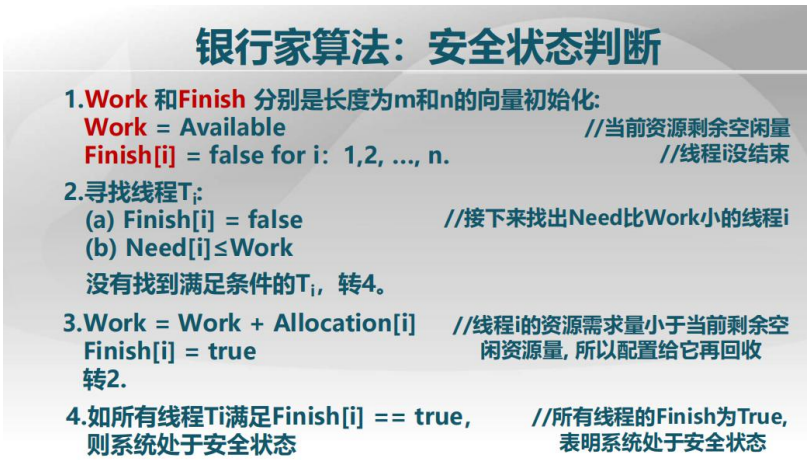
如何进行上面所说的动态检查？如下图：针对所有占用资源的进程，应存在安全序列。





第三节 银行家算法

银行家算法是一种死锁避免方法。



2 处是循环，找到一个 Need 比 Work 小的进程就转 3，或者说没找到转 4，或者所有 Finish[i]==true 转 4.

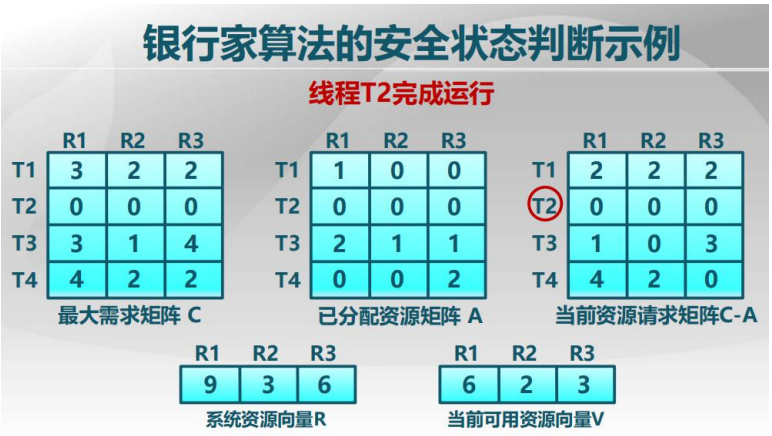
具体地，银行家算法如下：

循环第一步：检查进程请求的量是否超过他声称的未来需要量，如果超过是不被允许的。

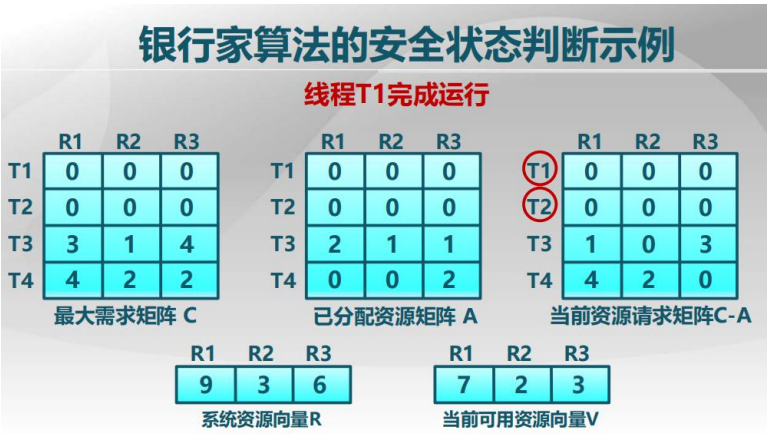
注意第三步循环需要调用前面讲到的安全状态判断。

下面是一个具体的例子：

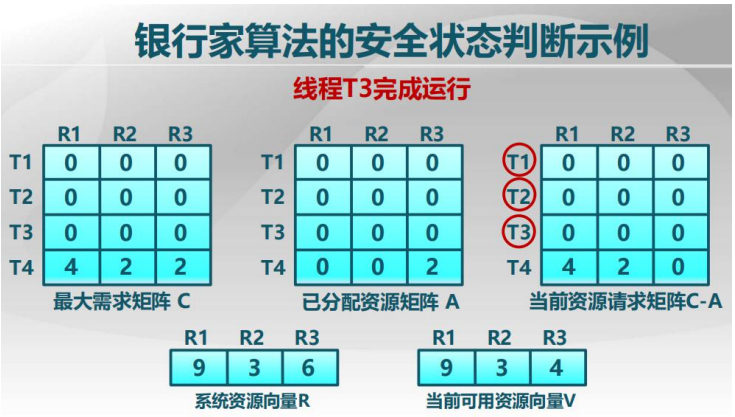
对照当前可用资源向量和当前资源请求矩阵，知 T2 可以运行，运行完后，将资源释放，变为下图：



然后满足 T1:



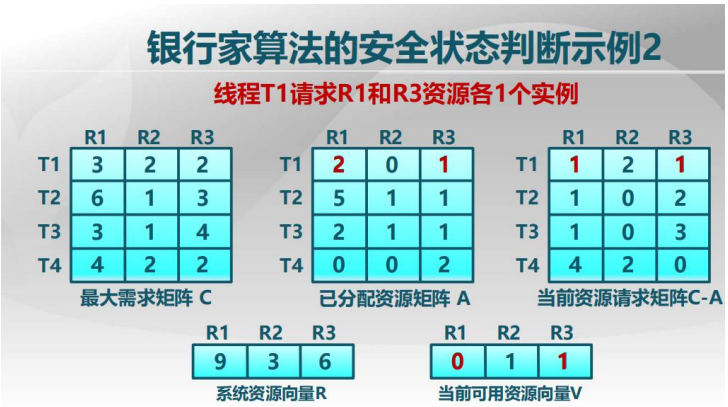
然后满足 T3:



最后 T4 完成，可见该状态是安全的。

另一个例子：

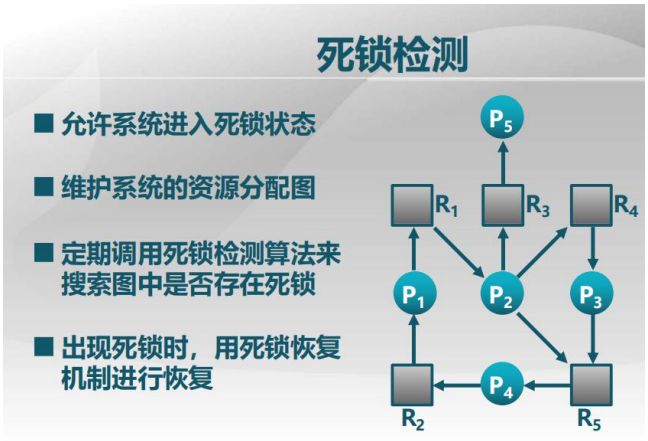
然后请求更改状态：



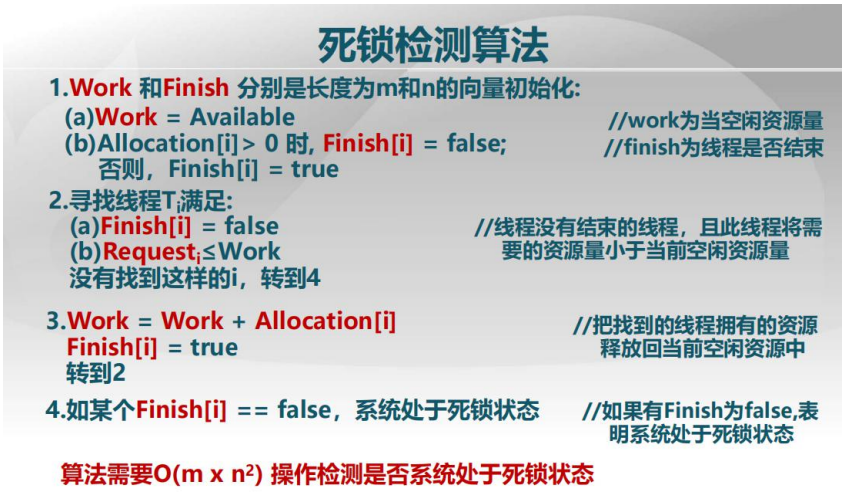
可见此时状态不再安全，因为当前剩余资源已经没有办法满足这四个线程中的任何一个未来需要(C-A)了。也就是说线程 T1 请求 R1 和 R3 资源各 1 个实例是不能同意的，同意了系统就是不安全的了。

第四节 死锁检测

死锁检测和银行家算法不同的地方在于：不管分配时的状态，只是在系统运行不下去时去检测，查看系统里的资源分配状态，查看是否已经出现死锁。



死锁检测和银行家算法比没有最大资源请求量的判断，其它它们是很相似的。



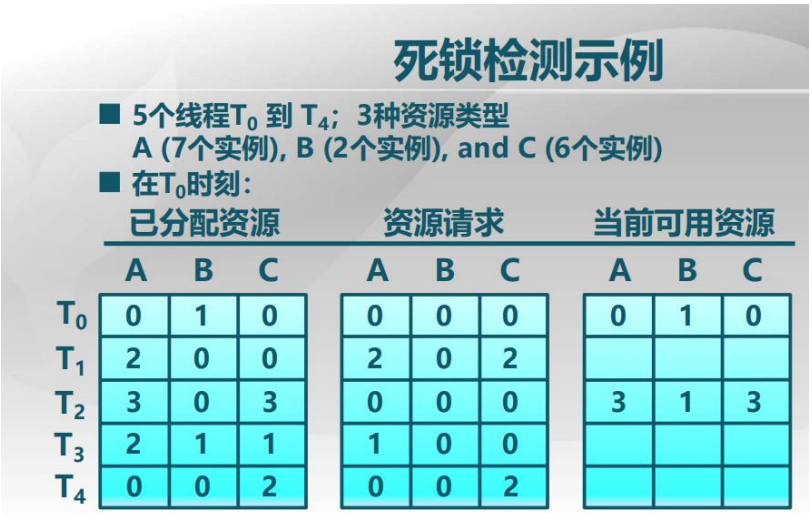
算法需要O(m x n²) 操作检测是否系统处于死锁状态

2 步即找到一个未完成的线程，并且其请求资源可以获得，如果存在这样的线程，则进入第三步，完成这个线程的执行，并且将其资源归还给系统。否则的话，说明可能有线程未完成，但系统已经无法为其分配资源，或者说所有线程均已执行完毕，这时转入第四步判断即可。O(m*n^2)的复杂度来自于：对每个线程进行资源检测时，都需要 m 次判断，来判断是不是 m 个资源都是充足的，每次第 2 步的循环，可能都要找到最后一个才能找到资源可以满足的线程，这样 2 的每次循环都要 n 次后才进入 3，然后 3 再返回 2 继续找资源可以满足的线程，每次有都要找 n 个线程才可以找到，这样就构成了 O(m*n^2)的复杂度。

所以说如果进程数目和资源量很大的话，死锁检测的开销就很大。另外一个问题就是多长时间检测一次，综合以上考虑，实际上开销是很大的。因此操作系统内部通常不管死锁的事情，也就是由于这个原因。

下面看一个实例：

此时 A,B,C 三种资源都被分配完，一般情况下，也就是没有可用资源时，才进行死锁的检测。首先，两个没有请求的线程 T0,T2 肯定可以执行结束，让他们执行结束得到：

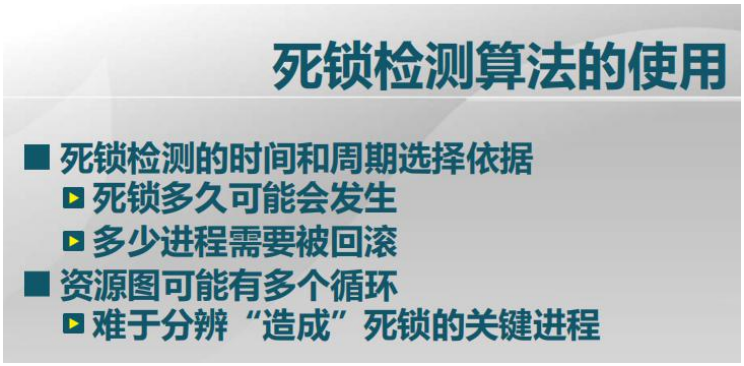


此时的可用资源可满足所有有资源请求的线程，所以可以正常结束，没有死锁：

可找到一个安全序列如上图所示，安全序列也就是各线程的执行顺序。

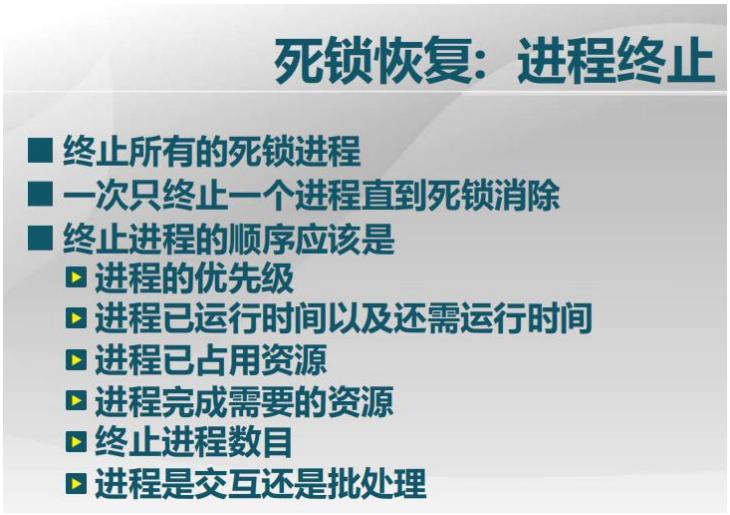
现在看另一个例子，仅在上面那个例子上做了很微小的改动：

上图中出现了死锁。那么出现死锁后，死锁如何处理？在这之前，先看死锁检测算法怎么使用：



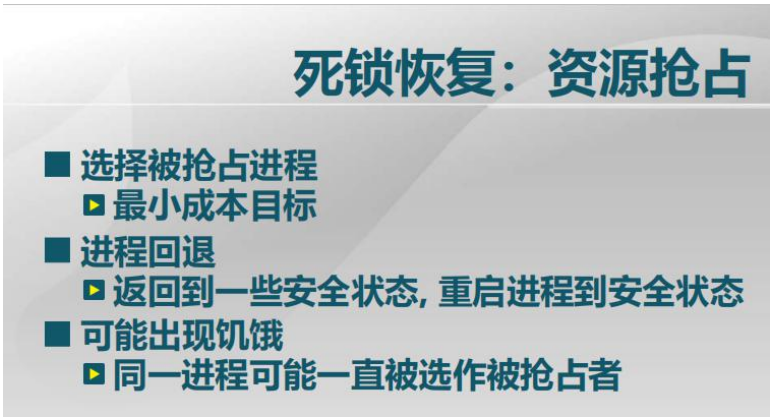
不能等系统完全搞不动了才来做检测，这时回滚的量是会很大的。
找出造成死锁的关键进程，是为了减少终止进程的数目。

假定可以找到是否有死锁，然后进行进程终止以处理死锁：



进程终止可一次性终止所有死锁进程，或者一次只终止一个进程直到死锁消除。一般用后者。优先级最低的先终止；执行时间短的先终止，因为它还未占用系统资源算很长时间；终止进程数目越小越好；优先终止后台批处理进程，因为我们希望用户能交互下去。

那么如何终止进程呢？如下图所示：

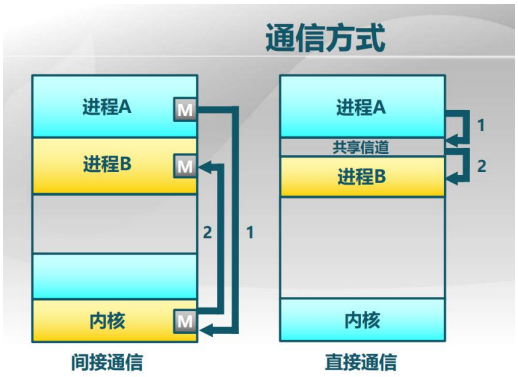


可以抢占成本最小的进程，或者回退到以前的某个状态。上面资源抢占终止进程的方法可能出现饥饿。

第五节 进程通信概念

进程通信流程：若有第三方参与，通讯的三方之间建立相应的通讯链路，然后通过 send/receive 交换消息。不同通讯进程间的链路特征是不同的：物理链路和逻辑链路。

下面是进程间通讯方式：

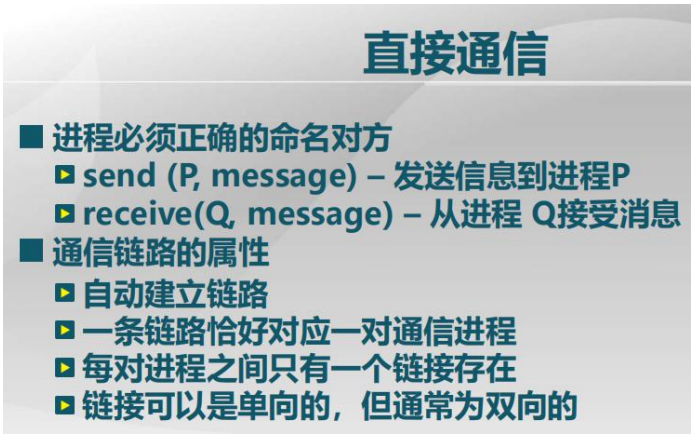


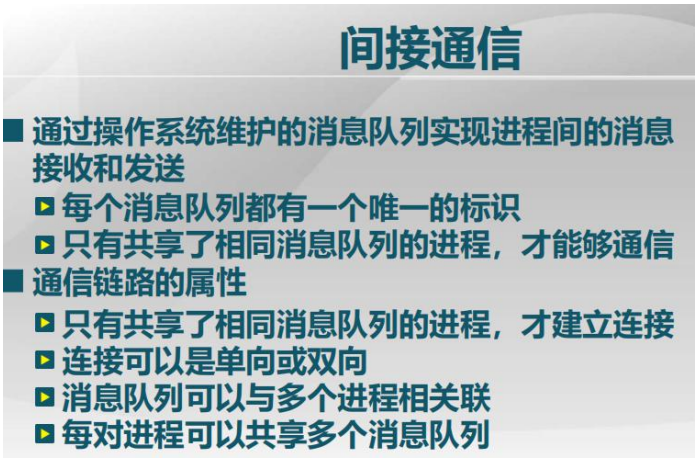
间接通讯：依赖于操作系统内核完成的进程间通讯。首先通讯进程都和内核建立相应的机构，以支持这种通讯，比如，建立消息队列，然后，一个进程 A 可以把信息发送到内核

的消息队列上，另一个进程 B 从消息队列中读出，从而实现进程 A，B 之间的通讯，这个通讯过程，A,B 的生命周期可以不同，比如 A 发信息时 B 还没有创建，而 B 接收数据时 A 可能已经关闭了。

直接通讯：在两个进程之间建立一个通讯信道，即共享信道，这时两个进程必须同时存在才能进行通讯：发方进程 A 向共享信道里发送数据，收方进程 B 从共享信道中读取数据。

下面对这两种通讯方式进行具体的讨论：





间接通讯可理解为进程和内核之间的一种通讯机制。只有共享了相同消息队列的进程，才能够通信，因此可以通过操作系统内核的安全机制，实现两个进程之间通讯的保密性。

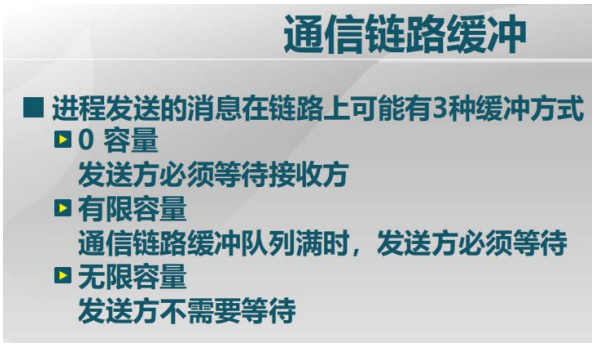
最后信息收发完毕后，消息队列需要撤销，否则在内核当中可能会有多个没有进程再去用的消息队列。与直接通信不同，发方不关心收方是谁，它关心是消息队列是谁：send(A, message)，接收也是一样的。

通讯的过程当中，还有一个属性是我们关心的：

阻塞通讯，每次都要保证通讯成功，这是阻塞通讯的基本特征。

与阻塞通讯相对应的是非阻塞通讯，非阻塞接收不会等着一定要接收到一条消息。

下面是通讯链路的缓冲特征：



在收发消息时，针对是否可以在链路上可对信息进行缓存：
①0 容量：发出的数据必须有接收方接受，否则就进行不下去，即没有接收方时就等待。
②有限容量：通信链路缓冲队列满时，发送方必须等待。
③无限容量：发送方不需等待。

关于进程通信原理的阐述正确的是 ABCD
A.进程通信是进程进行通信和同步的机制
B.进程通信可划分为阻塞（同步）或非阻塞（异步）
C.进程通信可实现为直接通信和间接通信
D.进程通信的缓冲区是有限的
注意 D 项是对的

第六节 信号和管道

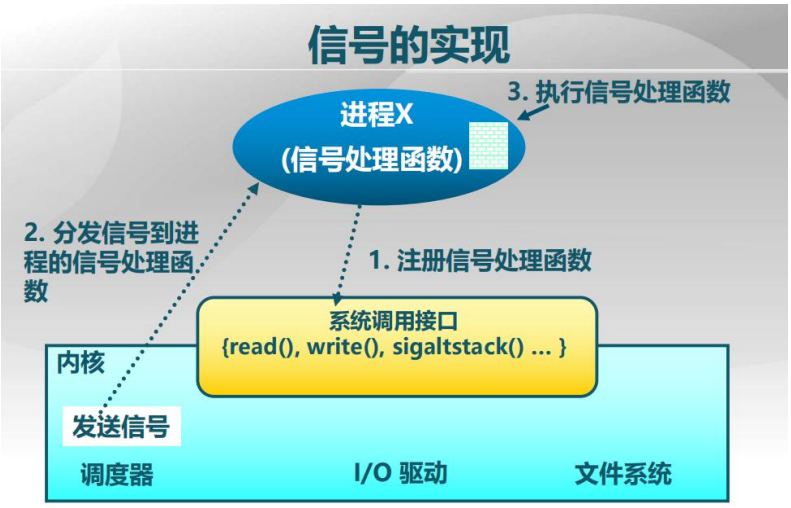
本节开始讨论进程通讯机制的具体实现，首先讨论信号和管道，信号和管道是操作系统提供的两种简单的通讯机制。首先是信号：

前面讲过的中断，实际上是 CPU 在执行指令时候的一种异常处理机制，这种机制借鉴到进程中来，就是信号。进程在执行过程中，有正常的执行流程，如果有意外的事情发生，可用信号处理。我们在实际系统中见到的一个例子就是：Ctrl+C，我们在一个进程执行过程当中，我们按 Ctrl+C 可以把这个进程停下来，但在实际代码当中，并没有一段代码是完成对 Ctrl+C 的处理，这个处理是利用信号实现的。操作系统在编译应用程序时，会缺省的加上对这些信号的处理例程，如果用户想指定自己的信号处理例程，就需要在应用程序中给出相应

的实现，比如：SIGKILL, SIGSTOP, SIGCONT 就是几个常见的信号。

屏蔽实例：登录程序，按 Ctrl+C 无法停止，因为这时把相应处理屏蔽掉了。

信号仅仅用做一种快速的响应机制，它比别的通讯机制要快。



信号的实现：1.注册：进程 X 在启动时，注册相应的信号处理例程给操作系统内核，以便于操作系统内核在有相应信号送过来时，能够知道去执行哪一个处理函数。

2.有其他进程或者设备发出该信号时，操作系统内核负责把这个信号送给指定的进程，并且

3.启动其中的信号处理函数，执行信号处理函数时，完成相应的处理。比如将当前正在执行的进程掐掉或者暂停、忽视，都是由信号处理函数完成的。

下面是一个实例：

信号使用示例

```
#include <stdio.h>
#include <signal.h>
void sigproc()
{
signal(SIGINT, sigproc);   /* NOTE some versions of UNIX will reset
                            * signal to default after each call. So for
                            * portability reset signal each time */

    printf("you have pressed ctrl-c - disabled \n");
}

void quitproc()
{
    printf("ctrl-\\ pressed to quit\n");   /* this is "ctrl" & "\" */
    exit(0); /* normal exit status */
}

main()
{
signal(SIGINT, sigproc);    /* DEFAULT ACTION: term */
signal(SIGQUIT, quitproc);  /* DEFAULT ACTION: term */
    printf("ctrl-c disabled use ctrl-\\ to quit\n");

    for(;;);
}
```

main 主程序中 signal 是注册信号处理例程的系统调用，主程序中注册了两个信号 SIGINT,SIGQUIT，signal(SIGINT,sigproc)是 SIGINT 的实现，这里实现中用 SIGINT 重新注册是为了兼容性，这里 SIGINT 只实现为打印一个字符串。SIGQUIT 的实现 quitproc()实现的是退出的功能，这是原来按 Ctrl+C 的功能。

注意，SIGINT 指的就是用户按下 Ctrl+C,SIGQUIT 指用户按下 Ctrl+\，上面程序重新注册了这两个信号对应的处理例程，从而实现了和缺省不同的效果：

该程序执行后，首先输出：

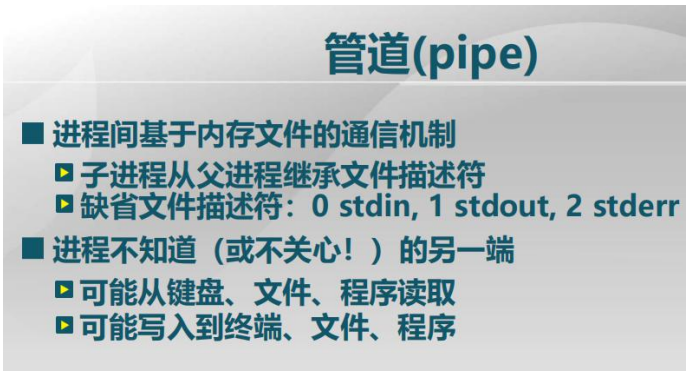
ctrl-c disabled use ctrl-\\ to quit

如果用户按 Ctrl+C，则继续输出：

you have pressed ctrl-c - disabled，然后继续执行死循环。

而如果按 Ctrl+\，程序就会退出。

下面看管道：



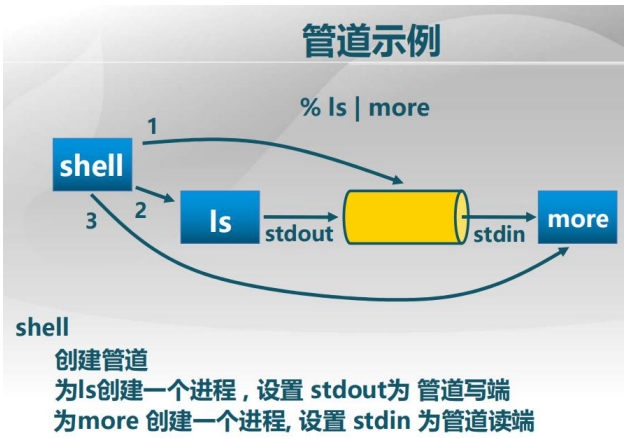
管道：两个进程想通信，中间的数据放在哪里？在内存中建了一个临时文件，将数据放在里面，这就是管道。它利用父进程创建子进程的过程当中，继承文件描述符，这几个缺省的文件描述符在子进程里都是有继承的。0 stdin 标准输入, 1 stdout 标准输出, 2 stderr 标准错误输出。　　创建一个管道时，只关心通讯的管道是谁，不关心另一头是谁往里放的数据，也就是一种间接通讯机制。

读管道：创建的管道有一个 FD 文件描述符，管道创建完之后，就可以利用 FD 进行读了。库函数 scanf()就是基于管道的读实现的。

写管道：和通常的文件读写的系统调用是完全一样的。库函数 printf()就是基于管道的写实现的。

创建管道：创建管道的结果会生成一个文件描述符数组 rgfd，其有两个成员，我们就是利用继承的关系，在不同的进程当中，使用不同的文件描述符，一头读，一头写，就实现两者之间的通讯了。

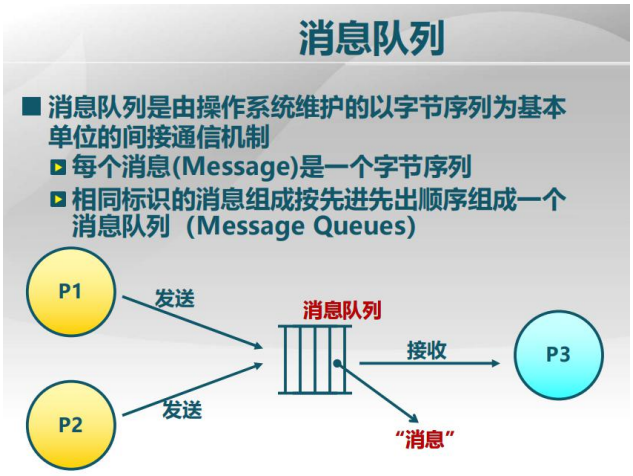
例子：



ls 和 more 是两个系统命令，在这两个命令间加一个竖线就表示 shell 给他们俩之间建立一条管道。1.执行这条命令时，shell 解释这条命令，首先会解释管道，建立相应的管道(橙色的是管道)　　2.为 ls 创建一个进程，执行 ls，并把它的标准输出(out)接到管道上，作为输入的这一端。　3.创建另外一个进程，执行 more 命令，more 的输入(in)来源于管道。这样就把两个命令，一个的输出和另一个的输入接到了一起，这在 Linux 和 Unix 中是一种常见的做法。

第七节 消息队列和共享内存

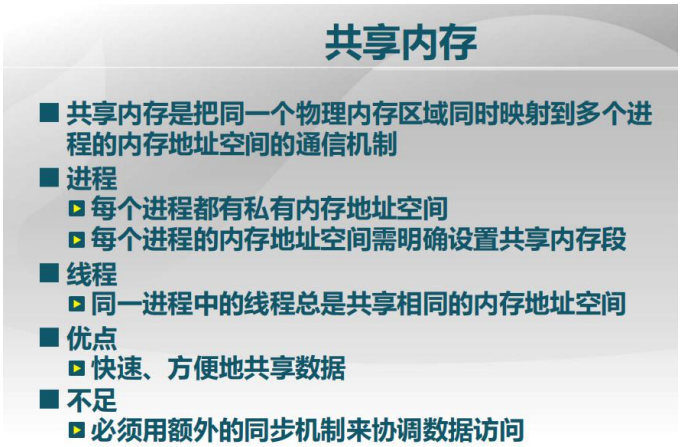
本节看消息队列和共享内存这两种进程通讯机制。首先是消息队列：



若干个进程可以向消息队列中发送信息，另外的进程可以从消息队列中读出消息。消息是一个最基本的字节序列。

消息队列控制指：进程结束时，它所占用的资源都会被释放掉，但消息队列是独立于创建它的进程的，所以一个进程可以创建一个消息队列，这个进程结束了，这个队列却可以继续存在下去，从而后续的创建的进程可以去读取消息队列的内容，实现两个生命周期不同的进程之间的通讯，所以在这里需要有专门的系统调用来完成对消息队列的创建和删除，即msgctl。

下面是共享内存，某种角度说它是一种内存的共享机制，但这里做通讯用：

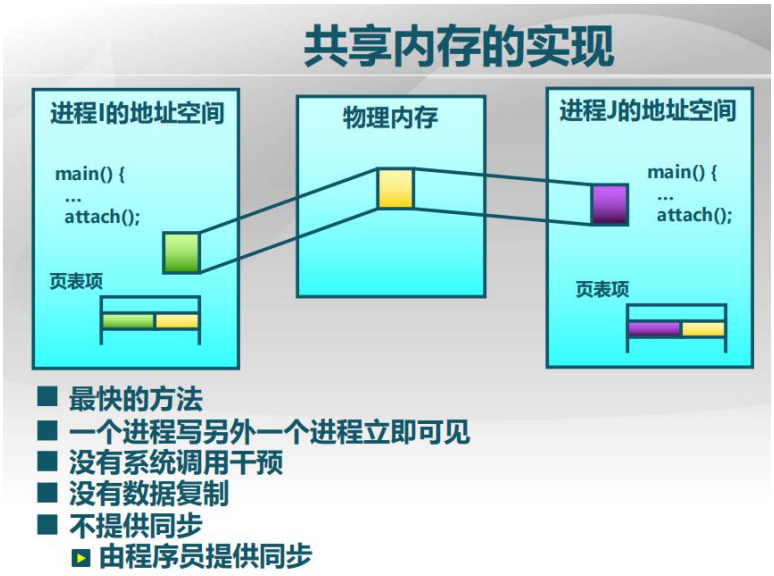


进程需要显式设置共享内存段，而同一进程中的线程由于其天然属性，不需显式设置共享内存段。

其优点：快速、方便，一个进程写进去，另一个进程马上就能看见。没有系统调用进行用户和内核之间的切换。

不足：仅靠共享内存无法实现完整的通讯，需要加同步机制，以避免一个进程在写的过程当中，还未写完之前，另一个进程从中读等情况。

共享内存的实现机制如下图：



两个进程的地址空间各自不同，中间是物理内存，将一块物理内存区域(黄色)映射到两个进程(靠两边的页表项实现，即两个进程各有一个逻辑地址对应的是相同的物理地址，即黄色的那个地址)，这时在一个进程里写，在另一个进程里就可以读，从而实现通讯。

与共享内存设置相关的系统调用有：

这几个系统调用，主要完成共享关系的建立，而正常的共享数据访问，只需要读写指令，不需要专门系统调用。

关于消息队列和共享内存的进程通信机制的阐述正确的是 ABCD

A.消息队列是由操作系统维护的以字节序列为基本单位的间接通信机制

B.共享内存是把同一个物理内存区域同时映射到多个进程的内存地址空间的通信机制

C.消息队列机制可用于进程间的同步操作

D.共享内存机制可用于进程间的数据共享

注意 C 是对的，消息队列与管程中的等待队列有异曲同工之妙。