彎曲評論

科技 · 人物 · 潮流

《孙钟秀 . 操作系统教程》注释 (稿)

(第三章--同步, 通信与死锁)

校注: 陈怀临

2014年10月8日

同步是为了几个具备一些依赖性的任务一起协调工作。

通信是为了同步的手段，可以是基于共享内存的，或者消息传递的方式。
死锁是一种导致整个系统无妨运行的算法状态。互相牵制了。

第三章

饿死（Starving）

同步、通信与死锁

并发进程

可以加一个“饿死”。意思是有的任务总是有资源；有的一直无法得到。

3.1.1 顺序程序设计

早期，人们引入程序的概念，编写程序并在计算机上运行程序以完成所需要的功能。程序是实现算法的操作(指令)序列，程序执行的顺序性是指其在处理器上的执行是严格有序的，即只有在前一个操作结束后，才能开始后继操作，这称为程序内部的顺序性；如果完成一个任务需要若干不同的程序，则这些程序也按照调用次序严格有序执行，这称为程序外部的顺序性。传统的程序设计方法是顺序程序设计，把程序设计成顺序执行的指令序列，不同程序也按顺序执行。顺序程序设计具有以下一些特性。

(1) 执行的顺序性

一个程序在处理器上是严格按序执行的，每个操作必须在下一个操作开始之前结束。

(2) 环境的封闭性

运行程序独占全机资源，资源的状态只能由此程序本身决定和改变，不受外界因素的影响。

(3) 结果的确定性

程序在执行过程中允许出现中断，但这种中断不会对程序的最终结果产生影响，也就是说程序的执行结果与它的执行速度无关。

(4) 过程的可再现性

程序针对同一个数据集合的执行过程在下一次执行时会重现，即重复执行程序会获得相同的执行过程和计算结果。

顺序程序设计的顺序性、封闭性、确定性和可再现性表明程序及其执行(计算)是一一对应的，为程序的编制和调试带来很大的方便，其缺点是计算机系统效率不高。

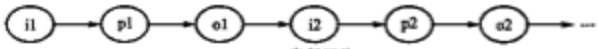
a3和b3没有约束关系。可以a3，b3的顺序，或者b3，a3. 我们说a3和b3是并发的（Concurrent Event）。

3.1.2 进程的并发性

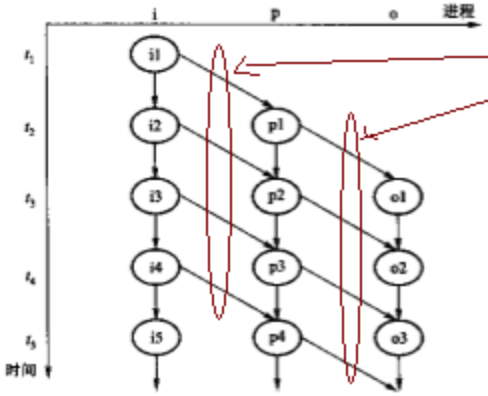
1. 并发程序设计

操作系统中引入并发程序设计技术之后，程序的执行不再是顺序的，一个程序未执行完前另一个程序便已开始执行，程序外部的顺序特性消失，程序与计算不再一一对应。所以，引入进程的概念来描述这种变化。进程的并发性是指一组进程的执行在时间上是重叠的，所谓时间重叠是指一个进程执行第一条指令是在另一个进程执行完最后一条指令之前开始的。例如，两个进程 A 和 B 分别执行操作 a_1、a_2、a_3 和 b_1、b_2、b_3。在单处理器上，顺序执行操作序列分别为 a_1，a_2，a_3 和 b_1，b_2，b_3，然而，若允许进程交叉执行，如执行操作序列为 a_1，b_1，a_2，b_2，a_3，b_3 或 a_1，b_1，a_2，b_2，b_3，a_3 等，则称进程 A 和 B 的执行是并发的。从宏观上来看，并发性反映一个时间段内有几个进程都处于运行态但运行尚未结束的状态；从微观上来看，任一时刻仅有一个进程的一个操作在处理器上执行。反过来看，并发的实质是处理器在几个进程之间的多路复用，并发是对有限物理资源强制行使多用户共享，消除计算机部件之间的互等现象，提高系统资源的利用率。程序的并发执行产生资源共享的需求，从而使程序失去封闭性、顺序性、确定性和可再现性。

现代计算机硬部件能同时进行工作，但怎样才能充分发挥不同硬部件的并行工作能力呢？很重要的一点是取决于程序编制。在图 3.1(a)中，由于程序是按照 while(TRUE){input, process, output}串行地输入－处理－输出来编制的，所以此程序只能顺序执行，这时系统的效率相当低。如果把求解这个问题的程序分成 3 个部分：



(a) 串行工作



同步机制。

(b) 并行工作

图 3.1 串行工作和并行工作

通过send, recv等通信机制, 实现有依赖性的并发任务的同步。


```
while(TRUE) {input,send}
while(TRUE) {receive,process,send}
while(TRUE) {receive,output}
```

每一部分是一个小程序，它们可以并发执行，并会产生制约关系，其中 send 和 receive 操作用于小程序之间通过通信机制解决制约关系，以便协调一致地工作。3 个小程序的功能分别是：

小程序 1：循环执行，读入字符，将读入字符送缓冲区 1；

小程序 2：循环执行，处理缓冲区 1 中的字符，把计算结果送缓冲区 2；

小程序 3：循环执行，取出缓冲区 2 中的计算结果并写到磁带上。

从图 3.1(b)可以看出，3 个小程序能并发执行，使得在 t_3 时刻输入 i3、处理 p2 与输出 o1 可并行工作，在 t_4、t_5 等时刻同样可以让 CPU、输入设备和输出设备并行工作。显然，这种程序设计方法能发挥 CPU 和设备的并行工作能力，从而提高计算机系统的效率。一个程序被分成若干可同时执行的小程序的程序设计方法称为并发程序设计(concurrent programming)，每个小程序及其执行时所处理的数据就组成一个进程。

2. 并发进程的特性

并发进程可能是无关的，也可能是交互的。无关的并发进程是指它们分别在不同的变量集合上操作，一个进程的执行与其他并发进程的进展无关，即一个进程不会改变另一个与其并发执行的进程的变量。然而，交互的并发进程共享某些变量，一个进程的执行可能会影响其他进程的执行结果，交互的并发进程之间具有制约关系。因此，进程的交互必须是有控制的，否则会出现不正确的计算结果。

并发进程的无关性是进程的执行与时间无关的一个充分条件，这一条件在 1966 年首先由 Bernstein 提出，假设

$R(P_i)=\{a_1,a_2,\cdots,a_n\}$，　程序 P_i 在执行期间所引用的变量集

$W(P_i)=\{b_1,b_2,\cdots,b_m\}$，　程序 P_i 在执行期间所改变的变量集

若两个进程的程序 P_1 和 P_2 能满足 Bernstein 条件，即引用变量集与改变变量集的交集之并为空集：

$$R(P_1)\cap W(P_2)\cup R(P_2)\cap W(P_1)\cup W(P_1)\cap W(P_2)=\varnothing$$

则并发进程的执行与时间无关。上述条件中 $R(P_1)\cap W(P_2)\cup R(P_2)\cap W(P_1)=\varnothing$ 表明一个程序在两次读操作之间，存储单元的数据不会被改变；$W(P_1)\cap W(P_2)=\varnothing$ 表明程序的写操作结果不会丢失。因此只要满足 Bernstein 条件，并发执行的程序就可以保持封闭性和可再现性。例如，有如下 4 条语句：

S1：a = x + y；　S2：b = z + 1；　S3：c = a − b；　S4：w = c + 1；

于是有：$R(S1)=\{x,y\}$，$R(S2)=\{z\}$，$R(S3)=\{a,b\}$，$R(S4)=\{c\}$；$W(S1)=\{a\}$，$W(S2)=\{b\}$，$W(S3)=\{c\}$，$W(S4)=\{w\}$。可见 S1 和 S2 可以并发执行，因为满足 Bernstein 条件，其他语句之间因变量交集之并非空，并发执行可能会产生与时间有关的错误。

采用并发程序设计的好处是：

读和写不相关。

写和写不覆盖

(1) 若为单处理器系统，可以有效地利用资源，让处理器和设备、设备和设备同时工作，充分发挥硬部件的并行工作能力；

(2) 若为多处理器系统，可让进程在不同的处理器上物理地并行工作，加快计算速度；

(3) 简化程序设计任务，一般来说，编制并发执行的小程序进度快，容易保证正确性。

并发程序设计是在多道程序设计的基础上发展起来的，采用并发程序设计的目的是：充分发挥硬件的并行性，消除处理器和设备的互等现象，提高系统效率。计算机硬部件能并行工作仅具备提高效率的可能性，而并行工作的实现还需要软件技术来发挥，这种软件技术就是并发程序设计。

3. 与时间有关的错误

两个交互的并发进程，其中一个进程对另一个进程的影响往往不可预期，甚至无法再现，这是因为它们执行时的相对速度不可预测，交互进程的速率不仅受到操作系统处理中断的方式及处理器调度策略的影响，而且还受到与其交互的并发进程的影响，甚至受到与其无关的其他进程的影响。由于一个进程的执行速度通常无法为另一个进程所知，对于共享公共变量(资源)的并发进程来说，计算结果往往取决于这一组并发进程执行的相对速度。可能出现各种与时间有关的错误，与时间有关的错误有两种表现形式：一种形式是结果不唯一；另一种形式是永远等待。为了说明与时间有关的错误，现在观察下面的例子。

例 1 (结果不唯一)飞机票售票问题。假设飞机票预订系统有两个终端，售票进程分别运行函数 T1()和 T2()。此系统的公共数据区中的一些单元 $A_j(j=1,2,3,\cdots)$分别存放某月某日某次航班的余票数，而 X1 和 X2 表示进程 T1 和 T2 执行时所用的工作单元。

```
//飞机票售票问题
void T1() {                          void T2() {
    {按旅客订票要求找到 Aj};             {按旅客订票要求找到 Aj};
    int X1 = Aj;                         int X2 = Aj;
    if(X1 >= 1) {                        if(X2 >= 1) {
        X1 -- ;                              X2 -- ;
        Aj = X1;                             Aj = X2;
        {输出一张票};                        {输出一张票};
    }                                    }
    else                                 else
        {输出信息"票已售完"};                {输出信息"票已售完"};
}                                    }
```

存取共享变量

写操作。

由于售票进程可并发执行，它们运行在同一个计算机系统中，共享同一批票源数据，因此，可能出现如下交叉情况：

```
T1:X1 = Aj;                    //X1 = m    T2:X2 = Aj;                    //X2 = m
T2:X2 -- ;Aj = X2;{输出一张票}; //Aj = m-1  T1:X1 -- ;Aj = X1;{输出一张票}; //Aj = m-1
```

显然，此时出现把同一张票卖给两位旅客的情况，两位旅客可能各自都买到一张同天同次航班的

正确的剩余的票应该是m－2.

系统因为没有保护共享变量Aj，结果出现了错误。

在while的判断和进入sleep之间，进程是可以并发的；中断也是可以进来的。

机票，可是，Aj 的值实际上只减去 1，造成余票数不正确。特别是，当某次航班只有一张余票时，可能把一张票同时出售给两位旅客。

例 2 （永远等待）主存管理问题。假定有两个可并发执行的程序 borrow 和 return 分别负责申请和归还主存资源，进程在申请和归还主存资源时调用它们。在算法描述中，X 表示现有空闲主存总量，B 表示申请或归还的主存量。

导致borrow会永远无法被唤醒。

```
//申请和归还主存资源问题
int X = memory;        //memory 为初始主存容量
void borrow(int B) {                          void return(int B) {
    while(B>X)                                    X = X + B;
        {进程进入等待主存资源队列};               {修改主存分配表};
    X = X - B;                                    {释放等待主存资源的进程};
    {修改主存分配表，进程获得主存资源};          }
}
```

由于 borrow 和 return 共享代表主存物理资源的临界变量 X，对并发执行不加限制将会导致错误。例如，一个进程调用 borrow 申请主存资源，在执行比较 B 和 X 的大小的指令后，发现 B>X，但在执行{进程进入等待主存资源队列}之前，另一个进程调用 return 抢先执行，归还所借全部主存资源；这时，由于前一个进程尚未成为等待者，return 中的{释放等待主存资源的进程}相当于空操作，以后当调用 borrow 的应用进程被置成{等待主存资源}时，可能已经没有其他进程再来归还主存，从而，申请资源的进程处于永远等待状态。

3.1.3 进程的交互：协作和竞争

导致borrow会永远无法被唤醒。

在多道程序设计系统中，同一时刻可能存在许多进程，它们可以并发或并行地执行。这些进程之间存在两种基本关系：竞争和协作。在引入线程的系统中，多线程之间的制约关系是类似的。

1. 竞争关系

批处理系统中建立多个批处理进程，分时系统中建立多个交互式进程，它们共享一套计算机系统资源，使得原本不存在逻辑关系的诸进程因共享资源而产生交互和制约关系，这是间接制约关系，又称互斥关系，操作系统必须协调进程对共享资源的争用。

由于竞争资源的独立进程之间并不交换信息，一个进程的执行可能影响到竞争资源的其他进程，如果它们要访问同一独占型资源，那么，一个进程通过操作系统分配获得这一资源，另一个进程将不得不等待。在极端的情况下，被阻塞进程永远得不到访问权，从而不能成功地终止。资源竞争会引发两个控制问题：一是死锁(deadlock)，一组进程如果都获得部分资源，还想要得到其他进程所占有的资源，最终所有进程将陷入永远等待的状态；二是饥饿(starvation)，一个可运行进程由于其他进程总是优先于它，而被调度程序无限期地拖延而不能被执行。在并发系统中，任何时刻都会产生申请资源的事件，需要规定一些策略来决定在什么时候、哪个进程应该获得资源，有些策略看似合理，却可能使某些进程总是得不到服务，但死锁却并未发生。例如，为了打印

防止不公平，或者饿死低优先级别的进程，是调度系统必须考虑的问题。

同步机制： 1. 基于共享内存 。 2. 基于消息传递

尽量多的文件，系统规定一种打印机分配策略：总是把打印机分配给所打印文件最短的进程（假设每个打印文件的长度已知），在一个繁忙的系统中，如果有进程生成一个大文件需要打印，每当打印机空闲时，系统会检查所有打印文件的长度，并把打印机分配给所打印文件最短的进程。如果不断地有新进程加入系统，且它们都只打印小文件，那么要打印大文件的进程总是得不到打印机，尽管这里并未产生死锁，但却出现饥饿，此进程被无限期推迟执行。解决饥饿问题的最简单策略是 FCFS 资源分配策略，等待最久的进程会是下一个被调度的进程，随着时间的流失，每个进程都有机会变成最"老"的进程，因而能获得资源并完成任务。由于操作系统负责资源分配，对资源竞争的控制需由系统来解决，应提供各种支持设施。例如，向并发进程提供锁机制，在使用共享资源之前表达互斥要求，系统需要保证诸进程能互斥地访问共享资源，既要解决死锁问题，又要解决饥饿问题。

进程互斥（mutual exclusion）是指若干进程因相互争夺独占型资源而产生的竞争制约关系。

2. 协作关系

一个用户作业可涉及一组并发进程，它们为了完成共同的任务需要分工协作。由于每个进程都独立地以不可预知的速度推进，在执行的先后次序上就要有约束，需要相互协作的进程在某些关键点上协调各自的工作。当其中的一个进程到达关键点后，在尚未得到其伙伴进程发来的消息或信号之前应阻塞自己，等待协作者发来信号或消息后方被唤醒并继续执行。这种协作进程之间需要排定执行先后次序的协调关系是直接制约关系，称为进程同步。回想 input、process 和 output 这 3 个进程分工协作完成读入数据、加工处理和打印输出任务的例子，是典型的进程同步问题，未输入完一块数据之前不能加工处理，未加工处理完一块数据之前不能打印输出，每个进程都要接收到协作进程完成一次处理的消息后，才能进行下一步工作。进程间的协作可以是双方不知道对方名字的间接协作，例如，通过访问共享资源进行松散式协作；也可以是双方知道对方名字，通过通信机制进行的紧密协作，允许进程协同工作有利于共享信息、加快计算速度、实现模块化程序设计。

进程同步（synchronization）是指为完成共同任务的并发进程基于某个条件来协调其活动，因为需要在某些位置上排定执行的先后次序而等待、传递信号或消息所产生的协作制约关系。

不难看出，进程互斥关系是一种特殊的进程同步关系，即逐次使用互斥共享资源，也是对进程使用资源的次序的一种协调。在操作系统中，信号量、锁、管程、消息及其他软硬件机制都可被看做由进程和线程请求与释放的资源，多数同步机制主要用于内核层，有一些同步机制也可用于应用层实现程序。

通过事件先后，自然的达到一种序（Order）。不存在先后的事件，
发事件。意思是：无所谓先后。不影响结果的。

3.2 临界区管理

例如，操作系统Kernel中的全局数据结构
就是被大家共享的。 Critical Area Management

3.2.1 互斥和临界区

例 1 中的飞机票售票系统之所以会产生错误，原因在于多个售票进程交叉访问共享变量

A。并发进程中与共享变量有关的程序段称为"临界区"(critical section)。共享变量所代表的资源称为"临界资源"(critical resource),如独占型硬件是临界资源,被共享的数据结构和文件也是临界资源。当多个并发进程访问临界资源时,结果依赖于它们执行的相对速度,便称出现了竞争条件(race condition)。显而易见,临界区必须以一种相对于其他进程而言互相排斥的方式执行。例1中的临界区表述如下:

```
//进程T1的临界区                    //进程T2的临界区
X1=Aj;                              X2=Aj;
if(X1>=1){                          if(X2>=1){
    X1--;                               X2--;
    Aj=X1;                              Aj=X2;
}                                   }
```

与同一共享变量有关的临界区分散在各个并发进程的程序段中,而进程的执行速度不可预知,如果能够保证一个进程在临界区执行时,不让另一个进程进入相同的临界区,即各进程对共享变量的访问是互斥的,那么,就不会引发与时间有关的错误。

临界区的概念由 Dijkstra 于 1965 年首先提出。为了正确而有效地使用临界资源,共享变量的并发进程应遵守临界区调度的三个原则:

(1) 一次至多有一个进程进入临界区内执行; 独占性。

(2) 如果已有进程在临界区中,试图进入此临界区的其他进程应等待; 排他性。

(3) 进入临界区内的进程应在有限时间内退出,以便让等待队列中的一个进程进入。

可把临界区的调度原则总结成 3 句话:互斥使用,有空让进;忙则等待,有限等待;择一而入,算法可行。其中,"算法可行"是指:不能因为所选的调度策略造成进程饥饿甚至死锁。

3.2.2 临界区管理的尝试

考虑并发进程在单处理器或共享主存的多处理器计算机系统上执行,先讨论采用软件方法实现临界区管理,通常假设具有存储器访问级的基本互斥性,对主存中的同一个单元同时访问时,必定由存储器进行仲裁使其串行化,此外,硬件、操作系统或语言未提供任何支持。临界区的管理可采用标志方式,即用标志来表示哪个进程可进入临界区。然而,如何使用标志,仍是值得探讨的问题。下列程序是第一种尝试,对进程 P1 和 P2 分别用标志 inside1 和 inside2 与其相对应,当进程在它的临界区内时其值为"真"(true),进程不在临界区内时其值为"假"(false);P1(P2)进入它的临界区前先测试 inside2(inside1),确保当前无进程在临界区中,然后把 inside1(inside2)置成 true,封锁 P2(P1)进入临界区,直至 P1(P2)执行完毕退出临界区,再将相应的标志置成 false。

```
//临界区管理尝试1
bool inside1=false;          //P1不在其临界区内
bool inside2=false;          //P2不在其临界区内
```

如果两个task的执行顺序是如图标注的1，2，3，4.会出现混乱
都进入临界区。


```
cobegin                          //cobegin 和 coend 表示括号中的进程是一组并发进程
process P1() {                   process P2() {
(1) while(inside2);              //等待  (2) while(inside1);        //等待
(3) inside1 = true;                    (4) inside2 = true;
    {临界区};                              {临界区};
    inside1 = false;                       inside2 = false;
}                                }
coend
```

但是，这种管理是不正确的，原因是在进程 P1(P2)测试 inside2(inside1)及随后置 inside1(inside2)为 true 之间，P2(P1)可能发现 inside1(inside2)的值为 false，于是它将置 inside2(inside1)为 true，并且与进程 P1(P2)同时进入临界区。

第二种尝试对第一种方案作一定的修正：延迟进程 P1(P2)对 inside2(inside1)的测试，先置 inside1(inside2)为 true，用以封锁 P2(P1)，修正后的程序如下所示。遗憾的是，它也是无效的，有可能每个进程都把自己的标志置成 true，从而出现死循环，这时没有进程能在有限的时间内进入临界区，造成永远等待的现象发生。

```
//临界区管理尝试 2
bool inside1 = false;        //P1 不在其临界区内
bool inside2 = false;        //P2 不在其临界区内
cobegin
process P1() {                   process P2() {
(1) inside1 = true;                    (2) inside2 = true;
(3) while(inside2);          //等待  (4) while(inside1);        //等待
    {临界区};                              {临界区};
    inside1 = false;                       inside2 = false;
}                                }
coend
```

导致两个进程都在等待－死锁（Deadlock）

3.2.3 实现临界区管理的软件算法

1981 年，G.L.Perterson 提出一个简单且巧妙的算法来解决进程互斥进入临界区的问题，此方法为每个进程设置标志，当标志值为 true 时表示此进程要求进入临界区，另外再设置一个指示器 turn 以指示可以由哪个进程进入临界区，当 turn = i 时则可由进程 P_i 进入临界区。

```
//Peterson 算法
bool inside[2];
inside[0] = false;inside[1] = false;
enum {0,1} turn;
```

[注释]： 同步是为了协同工作。通信分为：共享内存（memory）方式 和基于消息（message）方式。通信可以来实现同步。例如，send / recv。死锁和饿死都是调度要避免的。

3.2.3 实现临界区管理的软件算法

1981 年，G.L.Perterson 提出一个简单且巧妙的算法来解决进程互斥进入临界区的问题，此方法为每个进程设置标志，当标志值为 false 时表示此进程要求进入临界区，另外再设置一个指示器 turn 以指示可以由哪个进程进入临界区，当 turn = i 时则可由进程 Pi 进入临界区。

```
//Peterson 算法
bool inside[2];
inside[0] = false;inside[1] = false;
enum {0,1} turn;
```

假设两个任务都在临界区，条件是：
inside[0] = true; inside[[1]=true； turn=0;
turn =1; Turn变量同时为0和1是不可能的。
因此，Peterson算法是正确的。

P0: 在临界区时，需要满足：
inside [0] =True，
并且
inside [1] =False，或者 Turn =0

P1: 在临界区时，需要满足：
inside [1] =True，
并且
inside [0] =False，或者 Turn =1


```
cobegin
process P0() {
    inside[0] = true;
    turn = 1;
    while(inside[1] && turn == 1);
    {临界区};
    inside[0] = false;
}
coend
```

我想进来
我把票给你先。
别人在里面了

```
process P1() {
    inside[1] = true;
    turn = 0;
    while(inside[0] && turn == 0);
    {临界区};
    inside[1] = false;
}
```

我想进来
我把票给你先。
别人在里面了

在上面的程序中，利用对 turn 的赋值和 while 语句来限制每次最多只有一个进程进入临界区，当有进程在临界区执行时，不会有另一个进程闯入；进程执行完临界区的程序之后，修改 inside[i] 的状态而使等待进入临界区的进程可以在有限时间内进入。所以，Peterson 算法满足对临界区管理的三个原则。由于 while 语句中的判别条件是"inside[i] && turn = 0(或 1)"，因此，任意进程进入临界区的条件是对方不在临界区或对方不想进入临界区，于是，任何进程均可多次进入临界区。

例如，时钟中断会定期的发生。如果时钟中断处理程序与用户态进程共同读写一个共享的资源。就需要临界区管理。否则数据可能乱。

时钟中断被关之后，正常的进程调度也会被暂停。所以，不会有别的进程能切换。确保了进程之间不会有临界区的冲突。

3.2.4 实现临界区管理的硬件设施

在单处理器计算机系统中，并发进程不会同时执行，只会交替地执行。为了保证互斥性，仅需保证进程不被中断。下面是一些硬件设施，可用来实现对临界区的管理。

1. 关中断

实现互斥的最简单方法是在进程进入临界区时关中断，在进程退出临界区时开中断。中断被关掉后，时钟中断也被屏蔽，因为进程上下文切换都是由中断事件引起的，这样进程的执行再也不会被打断，因而，采用关中断、开中断的方法就可确保并发进程互斥地进入临界区。关中断的方法简单、有效，对操作系统自身很有用，可在更新共享变量或列表的几条指令期间禁止中断，但不适合作为通用的互斥机制，关中断的时间过长会影响性能和系统效率；不适用于多处理器计算机系统，因为一个处理器关中断，并不能防止进程在其他处理器上执行相同的临界段代码；若将这项权力赋予用户也存在危险，如果用户未开中断，则系统可能因此而终止。

2. 测试并建立指令

另一种方法是使用硬件所提供的"测试并建立"机器指令 TS(Test and Set)，可把这条指令看做函数，它有布尔型参数 x 和返回条件码，当 TS(&x)测到 x 值为 true 时则置 x 为 false，且根据所测试到的 x 值形成条件码。下面给出 TS 指令的处理过程。

```
bool TS(bool &x) {
  if(x) {
    x = false;
    return true;
```

通过硬件指令，确保原子性，不会被中断所打断。

```
    }
    else
        return false;
}
```

用 TS 指令管理临界区时，可把一个临界区与一个布尔型变量 s 相关联，由于变量 s 代表临界资源的状态，把它看成一把锁，s 的初值置为 true，表示没有进程在临界区内，资源可用，系统利用 TS 指令实现临界区的上锁和开锁原语操作。在进入临界区之前，首先用 TS 指令测试 s，如果没有进程在临界区内，则可以进入，否则必须循环测试直至 s 值为 true；当进程退出临界区时，把 s 值置为 true。由于 TS 指令是不可分割指令，在测试和形成条件码之间不可能有其他进程测试变量值，从而保证临界区管理的正确性。

```
//TS 指令实现进程互斥
bool s = true;
cobegin
process Pi() {                  //i = 1,2,…,n
    while(! TS(s));             //上锁
    {临界区};
    s = true;                   //开锁
}
coend
```

spinlock。 自旋锁。

3. 对换指令

对换指令的功能是交换两个字的内容，其处理过程描述如下：

```
void SWAP(bool &a,bool &b) {
    bool temp = a;
    a = b;
    b = temp;
}
```

在 Intel x86 中，对换指令称为 XCHG，使用它可以简单而有效地实现互斥，方法是为每个临界区设置布尔型锁变量，如 lock，当其值为 false 时表示无进程在临界区内。

```
//对换指令实现进程互斥
bool lock = false;
cobegin
process Pi() {                  //i = 1,2,…,n
    bool keyi = true;
    do {
        SWAP(keyi,lock);
```

都是实现0-1的变换，而且确保是原子性，在赋值过程中不会被中断进入。

通过do－while实现一个自旋锁。一旦lock可以用（available），迅速抢占。


```
    {while(keyi);              //上锁
    {临界区};
    SWAP(keyi,lock);           //开锁
}
coend
```

3.3 信号量与PV操作

3.3.1 同步和同步机制

著名的生产者－消费者问题(producer-consumer problem)是计算机操作系统中并发进程内在关系的一种抽象，是典型的进程同步问题。在操作系统中，生产者进程可以是计算进程、发送进程；而消费者进程可以是打印进程、接收进程等，解决好生产者－消费者问题就解决了一类并发进程的同步问题。

生产者－消费者问题表述如下：有 n 个生产者和 m 个消费者，连接在具有 k 个单位缓冲区的有界环状缓冲上，故又称有界缓冲问题。其中，p_i 和 c_j 都是并发进程，只要缓冲区未满，生产者进程 p_i 所生产的产品就可投入缓冲区；类似地，只要缓冲区非空，消费者进程 c_j 就可从缓冲区取走并消耗产品。

```
//生产者－消费者问题的程序描述
    int k;
    typedef anyitem item;                    //item 类型
    item buffer[k];
    int in=0,out=0,counter=0;
    process producer(void) {
        while(true)  {                       //无限循环
            {produce an item in nextp};      //生产一个产品
            if(counter==k)                   //缓冲区满时，生产者睡眠
                sleep(producer);
            buffer[in]=nextp;                //将一个产品放入缓冲区
            in=(in+1)%k;                     //指针推进
            counter++;                       //缓冲内产品数加 1
            if(counter==1)                   //缓冲区空了，加进一件产品
                wakeup(consumer);            //并唤醒消费者
        }
    }
```

被消费者之后唤醒。例如，缓冲区有空地方了。

k+1%K =1，从第一个开始。

消费者者缓冲区为0的时候，有一个sleep的相应动作。

1. 在生产者counter＋＋为9没有执行完成之前；消费者把8读走了。然后，在生产者counter＋＋完成后，消费者counter－－把7覆盖了counter的值。


```
process consumer(void) {
    while(true) {                          //无限循环
        if(counter == 0)                   //缓冲区为空，消费者睡眠
            sleep(consumer);
        nextc = buffer[out];               //取一个产品到 nextc
        out = (out + 1) % k;               //指针推进
        counter --;                        //取走一个产品，计数器减 1
        if(counter == k - 1)               //缓冲区满，取走一件产品
            wakeup(producer);              //并唤醒生产者
        {consume the item in nextc};       //消耗产品
    }
}
```

一般的高级语言都有 sleep()和 wakeup()函数，从上面的程序代码可以看出，算法是正确的，两组进程顺序执行的结果也正确。但若进程并发执行，就会出现错误的计算结果，出错的根源在于进程之间共享变量 counter，对 counter 的访问未加限制。

生产者进程和消费者进程对 counter 的交替操作会使其结果不唯一，例如，counter 的当前值为 8，如果生产者生产一件产品并投入缓冲区，拟执行 counter 加 1 操作；同时消费者获取一个产品消费，拟执行 counter 减 1 操作；假如两者交替执行加或减 1 操作，取决于其执行速度，counter 的值可能是 9，也可能是 7，但其正确值应为 8。

更为严重的是，生产者进程和消费者进程的交替执行会导致进程永远等待，造成系统死锁。假定消费者读取 counter 值时发现它为 0，此时调度程序暂停消费者进程让生产者进程运行，生产者加入一个产品，将 counter 值加 1，它想当然地推想由于 counter 值刚才为 0，所以，此时消费者一定是在睡眠，于是生产者调用 wakeup 函数来唤醒消费者；遗憾的是，消费者尚未睡眠时，唤醒信号被丢失，当消费者下次运行时，因已测到 counter 值为 0，于是去睡眠，这样生产者迟早会填满缓冲区，然后去睡眠，造成所有进程都永远处于睡眠状态。

出现不正确的结果并不是因为并发进程共享缓冲区，而是因为它们访问缓冲区的速度不匹配，或者说 p_i，c_j 的相对速度不协调，需要调整并发进程的推进速度，并发进程间的这种制约关系称为进程同步，交互的并发进程之间通过交换信号或消息来达到调整相互速度、保证进程协调运行的目的。

操作系统实现进程同步的机制称为同步机制，它通常由同步原语组成。不同的同步机制采用不同的同步方法，迄今已设计出多种同步机制，本书将介绍最常用的同步机制：信号量及 PV 操作、管程和消息传递。

(1) (2) (3)

这些机制之间都是等价的。

3.3.2 信号量与 PV 操作

前面介绍各种方法解决进程互斥，软件算法太过复杂，效率低下；硬件方法虽然简单、有效，

if（counter ＝＝ 0）
／／这个地方，sleep之前可能被调度
sleep（消费者）

spin lock, test and set的while自旋。

但采用忙式等待，白白浪费CPU时间；将测试能否进入临界区的责任推卸给各个竞争的进程，会削弱系统的可靠性，加重用户的编程负担，而且这些方案只能解决进程竞争却不能解决进程协作问题。

1965年，荷兰计算机科学家Edsger W. Dijkstra提出新的同步工具—信号量和P、V操作，他将交通管制中多种颜色的信号灯管理方法引入操作系统，让两个或多个进程通过特殊变量展开交互。一个进程在某一关键点上被迫停止执行直至接收到对应的特殊变量值，通过这一措施，任何复杂的进程交互要求均可得到满足，这种特殊变量就是信号量(semaphore)。为了通过信号量传送信号，进程可通过P、V两个特殊操作来发送和接收信号，如果协作进程的相应信号仍未送到，则进程被挂起直至信号到达为止。

在操作系统中，用信号量表示物理资源的实体，是一个与队列有关的整型变量。具体实现时，信号量是一种变量类型，用一个结构型数据结构表示，有两个分量：一个是信号量的值，另一个是信号量队列的指针。信号量在操作系统中主要用于封锁临界区、进程同步及维护资源计数。

除了赋初值之外，信号量仅能由同步原语P和V对其进行操作，不存在其他方法可以检查和操作信号量，PV操作的不可分割性确保执行时的原子性及信号量值的完整性。Dijkstra发明信号量操作原语：P和V操作(荷兰语中"检测"(Proberen)和"增量"(Verhogen)的首字母)，常用的符号还有up和down等。本书采用Dijkstra最早论文中使用的符号P和V。利用信号量和P、V操作既可解决并发进程的竞争问题，又可解决并发进程的协作问题。

信号量按其用途可分为两种：(1)公用信号量，联系一组并发进程，相关的进程均可在此信号量上执行P、V操作，初值往往为1，用于实现进程互斥；(2)私有信号量，联系一组并发进程，仅允许此信号量所拥有的进程执行P操作，而其他相关进程可在其上施行V操作，初值往往为0或正整数，多用于并发进程同步。信号量按其取值可分为两种：(1)二元信号量，仅允许取值为0或1，主要用于解决进程互斥问题；(2)一般信号量，又称计数信号量，允许取大于1的整型值，主要用于解决进程同步问题。

1. 一般信号量

设s为一个结构型数据结构，其中value为整型变量，系统初始化时为其赋值；list是等待使用此类资源的进程队列的头指针，初始状态为空队列，P和V原语操作的描述如下：

(1) P(s)：将信号量value值减1，若结果小于0，则执行P操作的进程被阻塞，排入与s信号量有关的list所指队列中；若结果大于等于0，则执行P操作的进程继续执行。

(2) V(s)：将信号量value值加1，若结果不大于0，则执行V操作的进程从信号量s有关的list所指队列中释放一个进程，使其转换为就绪态，自己则继续执行；若结果大于0，则执行V操作的进程继续执行。

结构型信号量和P、V操作定义为如下的数据结构和不可中断过程。

```
typedef struct semaphore {
    int value;                  //信号量值
    struct pcb * list;          //信号量队列指针
```

当信号量是1的时候，信号量就可以完美的实现一个"lock"锁机制。有时也称mutex。

该信号量被消耗完毕，需要等了

有Task在等待队列中。

```
};
void P(semaphore &s) {
    s.value--;                      //信号量值减 1
    if(s.value<0)                   //若信号量值小于 0,执行 P 操作的进程
        W(s.list);                  //调用 W(s.list)阻塞自己,被置成等待信号量 s 状态
}                                   //并移入 s 信号量队列,然后转向进程调度
void V(semaphore &s) {
    s.value++;                      //信号量值加 1
    if(s.value<=0)                  //若信号量值小于等于 0,则调用 R(s.list)
        R(s.list);                  //从信号量 s 队列中释放一个等待信号量 s 的进程
}                                   //并转换成就绪态,自己则继续执行
```

其中 W(s.list)和 R(s.list)是操作系统的基本系统调用,执行前者表示把调用过程的进程置成等待信号量 s 的状态,并移入 s 信号量队列,同时释放 CPU,转向进程调度;执行后者表示释放一个等待信号量 s 的进程,从 s 信号量队列中移出一个进程,转换成就绪态并移入就绪队列,执行 V 操作的进程继续执行,或转向进程调度。需要注意的是,当有多个进程等待同一个信号量时,它们被排成等待队列,由信号量指针指向队首进程的 PCB,而队列中的其他进程则通过其 PCB 中的指针项进行链接,队尾进程 PCB 的链接指针为 0。

进程从队列中移出的次序应遵循的公平策略是 FCFS 算法,被阻塞时间最长久的进程最先从队列中被释放,此策略确保进程不会饥饿,信号量 s 的初值可定义为 0、1 或其他整型数,在系统初始化时确定。从信号量和 P、V 操作的定义可获得如下推论。

推论 1 若信号量 s.value 为正值,此值等于在封锁进程之前对信号量 s 可施行的 P 操作数,亦即 s 所代表的实际可用的物理资源数。

推论 2 若信号量 s.value 为负值,其绝对值等于登记排列在 s 信号量队列之中等待的进程个数,即恰好等于对信号量 s 实施 P 操作而被封锁并进入信号量 s 等待队列的进程数。

推论 3 P 操作通常意味着请求一个资源,V 操作意味着释放一个资源,在一定的条件下,P 操作代表挂起进程的操作,而 V 操作代表唤醒被挂起进程的操作。

2. 二元信号量

设 s 为一个结构型数据结构,其中分量 value 仅能取值 0 或 1,分量 list 为等待信号量进程队列的头指针,若把二元信号量的 P、V 操作记为 BP 和 BV,其定义如下。

```
typedef struct binary_semaphore {
    int value;                      //value 取值 0 或 1
    struct pcb *list;
};
void BP(binary_semaphore &s) {
    if(s.value==1)
```

二元锁

```
        s.value=0;
    else
        W(s.list);
}
void BV(binary_semaphore &s)
    if(s.list is empty())
        s.value=1;
    else
        R(s.list);
}
```

信号量用来对资源不是特别急的场景；TS指令，或者说spinlock自旋锁常用来处理立刻要处理的。信号量涉及队列操作，存在延时，不确定性等问题。

虽然二元信号量仅能取值0或1，但可以证明它有着与其他结构型信号量一样的表达能力。

3.3.3 信号量实现互斥

信号量和P、V操作可用来解决进程互斥问题，与TS指令相比较，P、V操作也用测试信号量的方法来决定是否进入临界区，但不同的是，P、V操作只对信号量测试一次，而TS指令则必须反复测试。用信号量和P、V操作管理并发进程互斥进入临界区的一般形式为：

```
semaphore mutex;
mutex=1;
cobegin
process Pi() { //i=1,2,…,n
  P(mutex);
  {临界区};
  V(mutex);
}
coend
```

当有进程在临界区中时，mutex的值为0或负值；否则mutex值为1，因为只有一个进程可用P操作把mutex减至0，故可保证互斥操作，这时试图进入临界区的其他进程会因执行P(mutex)而被迫等待。mutex的取值范围是1～－(n－1)，表明有一个进程在临界区内执行，最多有n－1个进程在信号量队列中等待。

3.3.4 信号量解决5位哲学家吃通心面问题

下面讨论使用信号量和P、V操作解决操作系统中经典的5位哲学家吃通心面问题(Dijkstra，1968年)。如图3.2所示，有5位哲学家围坐在一张圆桌旁，桌子中央放有一盘通心面，每人面前有一只空盘子，每两人之间放一把叉子；每位哲学家思考、饥饿，然后吃通心面；为了吃面，哲学家必须获得两把叉子，且每人只能直接从紧邻自己的左边或右边去取叉子。

[注释]:学习同步和并发控制时，要注意几个要点：1. 同步原语都是等价的。 2. 为什么需要原子性？因为，一个简单的value＋＋在指令级别时需要3条指令才能完成：从内存中Load；寄存器操作－加一。 Save到内存中。在3个指令之间，中断可以任意发生。形成并发。

每一个叉子通过一个Semphore来互斥保护

在这道题目中，每把叉子都必须互斥使用，因此，应为每把叉子设置互斥信号量 fork[i](i=0,1,2,3,4)，其初值均为 1，当一位哲学家吃通心面之前必须执行两个 P 操作，获得自己左边和右边的两把叉子；在吃完通心面后必须执行两个 V 操作，放下两把叉子。

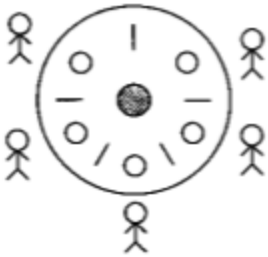


图 3.2 5 位哲学家吃通心面问题

```
//信号量解决哲学家吃通心面问题
semaphore fork[5];
for(int i=0;i<5;i++)
    fork[i]=1;
cobegin
process philosopher_i() {      //i= 0,1,2,3,4
    while(true) {
        think();
        P(fork[i]);
        P(fork[(i+1)%5]);
        eat();
        V(fork[i]);
        V(fork[(i+1)%5]);
    }
}
coend
```

试图连续获得左右两把叉子。

释放叉子。

在上述解法中，如果 5 位哲学家同时拿起右边(或左边)的叉子，当每位哲学家企图再拿起其左边(或右边)的叉子时，将出现死锁。有若干种方法可以避免这类死锁。

(1) 至多允许 4 位哲学家同时吃面；

(2) 奇数号哲学家先取左边的叉子，然后再取右边的叉子；偶数号哲学家先取右边的叉子，然后再取左边的叉子；

(3) 每位哲学家取到手边的两把叉子才开始吃面，否则，连一把叉子也不取。

3.3.5 信号量解决生产者－消费者问题

信号量和 P、V 操作不仅可以解决进程互斥问题，而且是实现进程同步的有力工具。在协作进程之间，一个进程的执行依赖于协作进程的信号或消息，在尚未得到来自协作进程的信号或消息时则等待，直至信号或消息到达时才被唤醒。

生产者－消费者问题是典型的进程同步问题，这些进程必须按照一定的生产率和消费率来访问共享缓冲区，用 P、V 操作解决生产者和消费者共享单缓冲区的问题，可设置两个信号量 empty 和 full，其初值分别为 1 和 0，empty 指示能否向缓冲区放入产品，full 指示能否从缓冲区取出产品。

互斥 本身也可理解是一种同步。为了整体不出现混乱，需要互斥同步。互斥需要通过等待来实现，是一种同步。

如果是空的，获得进入；否则，睡眠；等待被消费者认为队列为空的时候，唤醒生产者。


```
//单缓冲区生产者－消费者问题
    int B;
    semaphore empty;                    //可用的空缓冲区数
    semaphore full;                     //缓冲区内可用的产品数
    empty=1;                            //缓冲区内允许放入一件产品
    full=0;                             //缓冲区内没有产品
    cobegin
    process producer(){                     process consumer(){
        while(true) {                           while(true) {
            produce();                              P(full);
            P(empty);                               take() from B;
            append() to B;                          V(empty);
            V(full);                                consume();
        }                                       }
    }                                       }
    coend
```

如果有产品，消费；否则睡眠等待被唤醒。

唤醒　唤醒

需要注意的是，如果 P、V 操作使用不当，仍会出现与时间有关的错误。例如，有 m 个生产者和 n 个消费者，它们共享可存放 k 件产品的缓冲区，为了使其协调工作，必须使用一个信号量 mutex（初值为 1），以限制生产者和消费者互斥地对缓冲区进行存取，另用两个信号量 empty（初值为 k）和 full（初值为 0），以保证生产者不向已满的缓冲区中放入产品，消费者不从空缓冲区中取产品。

```
//m 个生产者和 n 个消费者共享 k 件产品缓冲区的问题
    item B[k];
    semaphore empty; empty=k;           //可用的空缓冲区数
    semaphore full;full=0;              //缓冲区内可用的产品数
    semaphore mutex; mutex=1;           //互斥信号量
    int in=0;                           //放入缓冲区指针
    int out=0;                          //取出缓冲区指针
    cobegin
    process producer_i() {                  process consumer_j() {
        while(true) {                           while(true) {
            produce();                              P(full);
            P(empty);                               P(mutex);
            P(mutex);                               take() from B[out];
            append to B[in];                        out=(out+1)%k;
```

mutex用来保护数组的存取。

P操作会导致进程进入睡眠状态。


```
        in=(in+1)%k;                    V(mutex);
        V(mutex);                       V(empty);
        V(full);                        consume();
    }                               }
}                               }
coend
```

程序中的 P(mutex)和 V(mutex)必须成对出现，夹在两者之间的代码段是进程的临界区；施加于信号量 empty 和 full 上的 P、V 操作也必须成对出现，但分别位于不同的程序中。在这个问题中，P 操作的次序是很重要的，如果把生产者进程中的两个 P 操作交换次序，那么，当缓冲区中存满 k 件产品(empty=0，mutex=1，full=k)时，生产者又生产一件产品，在它欲向缓冲区存放时，将在 P(empty)上等待(注意，现在 empty=−1)，由于此时 mutex=0，它已经占有缓冲区，这时消费者欲取产品时将停留在 P(mutex)上而得不到使用缓冲区的权力，这就导致生产者等待消费者取走产品，而消费者却在等待生产者释放缓冲区的占有权，这种相互之间的等待永远不可能结束。所以，在使用信号量和 P、V 操作实现进程同步时，特别要当心 P 操作的次序，而 V 操作的次序无关紧要。一般来说，用于互斥的信号量上的 P 操作总是在后面执行。

3.3.6 信号量解决读者－写者问题

读者－写者问题(reader writer problem)(Courtois，Heymans，Parnas，1971 年)也是一个经典的并发程序设计问题。有两组并发进程：读者和写者，共享文件 F，要求：

(1) 允许多个读者同时对文件执行读操作；

(2) 只允许一个写者对文件执行写操作；

(3) 任何写者在完成写操作前不允许其他读者或写者工作；

(4) 写者在执行写操作前，应让已有的写者和读者全部退出。

单纯使用信号量不能解决此问题，必须引入计数器 readcount 对读进程计数，mutex 是用于对计数器 readcount 操作的互斥信号量，writeblock 表示是否允许写的信号量。

```
//信号量解决读者－写者问题
int readcount=0;                    //读进程计数
semaphore writeblock,mutex;
writeblock=1;mutex=1;
cobegin
process reader_i() {                process writer_j() {
    P(mutex);                           P(writeblock);
    readcount++;                        {写文件};
    if(readcount==1)                    V(writeblock);
        P(writeblock);              }
```

当拥有一个锁 A后，进程再sleep在另外一个锁B上，需要非常小心。可能导致死锁。例如，另外一个进程拥有锁B，但在等待锁A的释放。互相等待了。

```
        V(mutex);
        {读文件};
        P(mutex);
        readcount--;
        if(readcount==0)
            V(writeblock);
        V(mutex);
    }
    coend
```

对read count临界区的mutex保护

如果有其他的读，不放writeblock，禁止写操作进入。

此解法中，读者是优先的，当存在读者时，写者将被延迟，且只要有一个读者活跃，随后而来的读者都将被允许访问文件，从而导致写者长时间等待，并有可能出现写者饥饿的现象。增加信号量并修改此程序可得到写者具有优先权的解决方案，确保当一个写进程声明想写时，不允许新的读者访问共享文件。

为了有效地解决读者－写者问题，有的操作系统专门引进读者－写者锁，允许多名读者同时以只读方式存取有锁保护的对象；或一位写者以写方式存取有锁保护的对象。当一名或多名读者已经上锁后，此时形成读锁，写者将不能访问有读锁保护的对象；当锁被请求者用于写操作时，形成写锁，其他进程的读写操作必须等待。

rwlock

3.3.7 信号量解决理发师问题

另一个经典的进程同步问题是 1968 年由 Dijkstra 提出的睡眠理发师问题(sleepy barber problem)。理发店里有一位理发师、一把理发椅和 n 把供等候理发的顾客休憩的椅子；如果没有顾客，理发师便在理发椅上睡觉，当有顾客到来时，他唤醒理发师；如果理发师正在理发时又有新的顾客来到，那么如果还有空椅子，顾客就坐下来等待，否则就离开理发店。

给出的解法中引入 3 个信号量和一个控制变量：控制变量 waiting 记录等候理发的顾客坐的椅子数，初值为 0；信号量 customers 记录等候理发的顾客数，并用于阻塞理发师进程，其初值为 0；信号量 barbers 记录正在等候顾客的理发师数，并用于阻塞顾客进程，其初值为 0；信号量 mutex 用于互斥，其初值为 1。

```
//信号量解决理发师问题
int waiting=0;                  //等候理发的顾客坐的椅子数
int CHAIRS=N;                   //为顾客准备的椅子数
semaphore customers,barbers,mutex;
customers=0;barbers=0;mutex=1;
cobegin
process barber() {
    while(true) {
```

```
            P(customers);          //判断是否有顾客,若无顾客,理发师睡眠
            P(mutex);              //若有顾客,进入临界区
            waiting--;             //等候顾客数减 1
            V(barbers);            //理发师准备为顾客理发
            V(mutex);              //退出临界区
            cut_hair();            //理发师正在理发(非临界区)
        }
    }
    process customer_i() {
        P(mutex);                  //进入临界区
        if(waiting<CHAIRS) {       //判断是否有空椅子
            waiting++;             //等候顾客数加 1
            V(customers);          //唤醒理发师
            V(mutex);              //退出临界区
            P(barbers);            //理发师忙,顾客坐着等待
            get_haircut();         //否则,顾客可以理发
        }
        else
            V(mutex);              //人满了,顾客离开
    }
    coend
```

Java中thread同步广泛使用了monitor（管程）的机制。

图灵奖获得者。CSP的发明者。并发程序语言和逻辑的大师。

管 程 Monitor

3.4.1 管程和条件变量

使用信号量和 P、V 操作实现进程同步时,对共享资源的管理分散于各个进程中,进程能够直接对共享变量进行处理,不利于系统对临界资源的管理,难以防止进程有意或无意的违法同步操作,容易造成程序设计错误。在进程共享主存的前提下,如果能集中和封装针对一个共享资源的所有访问,包括所需的同步操作,即把相关的共享变量及其操作集中在一起统一控制和管理,就可以方便地管理和使用共享资源,使并发进程之间的相互作用更为清晰,更易于编写正确的并发程序。

在 1974 年和 1977 年,Hoare 和 Brinch Hansen 根据抽象数据类型的原理提出新的同步机制——管程(monitor)。其基本思路是:把分散在各进程中的临界区集中起来管理,并把共享资源用数据结构抽象地表示出来,由于临界区是访问共享资源的代码段,建立一个"秘书"程序管理

管程与信号量PV操作是等价的。是不同的表达方式，但可以实现相同的同步功能。

到来的访问。"秘书"每次只让一个进程来访，这样既便于对共享资源进行管理，又能实现互斥访问。在后来的实现中，"秘书"程序更名为管程。采用这种方法，对共享资源的管理可借助数据结构及其上所实施操作的若干进程来进行；对共享资源的申请和释放通过进程在数据结构上的操作来实现。代表共享资源的数据结构及并发进程在其上执行的一组过程就构成管程。管程被请求和释放资源的进程所调用，管程实质上是把临界区集中到抽象数据类型模板中，可作为程序设计语言的一种结构成分。对于同步问题的解决，管程和信号量具有同等的表达能力。

管程有以下属性，因而，调用管程的过程时要有一定的限制。

(1) 共享性

管程中的移出过程可被所有要调用管程的过程的进程所共享。

(2) 安全性

管程的局部变量只能由此管程的过程访问，不允许进程或其他管程来直接访问，一个管程的过程也不应访问任何非局部于它的变量。

(3) 互斥性

在任一时刻，共享资源的进程可以访问管程中的管理此资源的过程，但最多只有一个调用者能够真正地进入管程，其他调用者必须等待直至管程可用。

可见，管程是由局部于自己的若干公共变量及其说明和所有访问这些公共变量的过程所组成的软件模块，它提供一种互斥机制，进程可互斥地调用这些过程。管程把分散在各个进程中互斥地访问公共变量的那些临界区集中在一起，提供对它们的保护。由于共享变量每次只能被一个进程所访问，把代表共享资源状态的共享变量放置在管程中，那么，管程就可以控制共享资源的使用。管程可作为程序设计语言的一个成分，采用管程作为同步机制便于用高级语言来编写程序，也便于程序正确性验证。管程的概念和机制已在 Pascal_plus、Modula-2、Modula-3 和 Java 等语言中实现。

每个管程都有一个名字以供标识，一般的形式表示为：

```
type 管程名 = monitor {
    局部变量说明；
    条件变量说明；
    初始化语句；
    define 管程内定义的、管程外可调用的过程或函数名列表；
    use 管程外定义的、管程内将调用的过程或函数名列表；
    过程名/函数名(形式参数表) {
        <过程/函数体>；
}
...
    过程名/函数名(形式参数表) {
        <过程/函数体>；
```

管程 就是 通过 面向对象技术，对共享资源，相关访问代码的一个封装。并发控制，例如，锁机制，都在内部解决。不需要程序员再去关心。

通过调用 Monitor的method（方法），管程会负责"互斥"并保证对共享资源一致性语义的保证。至于如何实现的，可以通过sem，或者mutex。

管程的结构如图 3.3 所示。

管程把 semaphore, conditional varlable合并封装在一起, 并提供一个API接口给调用方。

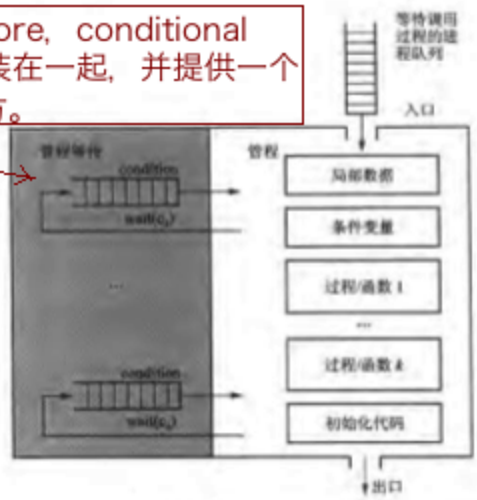


图 3.3 管程结构示意图

管程通过防止对一个资源的并发访问，达到实现临界区的效果，提供一种实现互斥的简单途径，但是并未提供进程与其他进程通信或同步的手段。为了进行并发处理，需要引入同步工具使得进程在资源不能满足而无法继续运行时被阻塞，同时还需开放管程。解决的方法是，使用称作条件变量(conditional variable)的同步机制，让等待的进程临时放弃管程控制权，然后，在适当的时刻，再尝试检测管程内状态的变化。

(1) 条件变量

条件变量是出现在管程内的一种数据结构，且只有在管程中才能被访问，它对管程内的所有过程是全局的，只能通过两个原语操作来控制它。

(2) wait()

挂起调用进程并释放管程，直至另一个进程在条件变量上执行 signal()。

(3) signal()

如果有其他进程因对条件变量执行 wait()而被挂起，便释放之；如果没有进程在等待，那么，信号不被保存。

不建议这样比较。Monitor和Process是不同的东西。Monitor是一个同步，互斥机制。Process是程序的执行的生命体。

当进程调用管程的过程发现无法继续执行，例如，发现没有可用资源，它在某些条件变量 c 上执行 c.wait()，此操作将引起调用进程阻塞并移入与条件变量 c 相关的队列中，这时应开放管程，允许被阻塞在管程之外的一个进程进入管程；另一个进程可通过对其伙伴在等待的同一个条件变量 c 上执行 c.signal()来唤醒等待进程并将其移出条件变量 c 队列，然后进程会在 wait 操作之后继续执行。值得注意的是，虽然条件变量也是一种信号量，但它并不是 P、V 操作中纯粹的计数信号量，没有与条件变量关联的值，不能像信号量那样积累供以后使用，仅仅起到维护等待进程队列的作用，当在一个条件变量上不存在等待的进程时，signal 操作所发出的信号被丢弃，等于空操作。wait 操作一般应在 signal 操作之前发出，这一规则能大大简化管程的实现。

管程有时要对一个不能得到共享资源的进程进行延迟，当其他进程释放它所需要的资源时，再恢复其执行，wait 原语可延迟进程的执行，signal 原语使得被延迟进程中的某一个进程恢复执行。现在的问题是，当某进程执行 signal 操作之后，另一个被延迟的进程可恢复执行，于是可能有两个进程同时调用一个管程中的两个过程，造成管程中有两个进程同时执行，根据管程的互斥性，这是绝对不允许的。采用两种方法来防止这种现象的出现。

假定进程 P 执行 signal 操作，条件变量队列中有一个进程 Q 在等待，当进程 P 执行 signal 操作后，进程 Q 被释放，对它们可作如下处理：

(1) 进程 P 等待直至进程 Q 退出管程，或者进程 Q 等待另一个条件。

(2) 进程 Q 等待直至进程 P 退出管程，或者进程 P 等待另一个条件。

Hoare 采用第一种方法，而 Hansen 选择两者的折中，他规定管程中的过程所执行的 signal 操作是过程体的最后一个操作，于是，进程 P 执行 signal 操作后立即退出管程，因而，进程 Q 马上恢复执行。可见 Hansen 所实现的管程功能较弱，本章介绍 Hoare 管程实现方法，Hansen 管程实现方法作为练习由读者自己完成。

在此进一步把管程和进程作比较，它们存在以下区别。

(1) 管程所定义的是公用数据结构，而进程所定义的是私有数据结构。

(2) 管程把共享变量上的同步操作集中起来统一管理，而临界区却分散在每个进程中。

(3) 管程是为解决进程共享资源的互斥而建立的，而进程是为占有系统资源和实现系统并发性而引入的。

(4) 管程被欲使用共享资源的所有进程所调用，管程和调用它的进程不能并行工作；而进程之间能够并行工作，并发性是其固有特性。

(5) 管程可作为语言或操作系统成分，不必创建或撤销；而进程有生命周期，由创建而产生至撤销便消亡。

3.4.2 管程的实现

Hoare 方法使用 PV 操作原语来实现对管程中过程的互斥调用功能，并实现对共享资源互斥使用的管理。每当有进程等待资源时，此方法将让执行 signal 操作的进程挂起，直到被它释放的进程退出管程或产生其他的等待条件为止，而且 wait 和 signal 操作可被设计成两个可中断的过程。

等价性。可以通过彼此来实现。

[注释]:同步机制可以通过Sem（信号量）。但语义较低。需要在共享内存模式下（例如，多线程），写比较难的控制代码。容易出错。管程（Monitor）是一等价的同步机制。把共享数据和访问的代码封装为对象。提供一个一致的同步API接口。语义友善。Java等编程语言对Monitor机制的支持

已经很完善并被广泛使用,例如”Synchronized”的关键字的使用。

OO的思想。使用Monitor，都是通过调用对外的Method。Method本身通过Mutex（锁）来保护，从而达到Thread－Safe，互斥使用。

1. 互斥信号量 mutex

对于每个管程,使用一个用于管程中过程互斥调用的信号量 mutex(初值为 1)。任何进程调用管程中的任何过程时,应执行 P(mutex);进程退出管程时应执行 V(mutex)开放管程,以便让其他调用者进入。为了使进程在等待资源期间,其他进程能够进入管程,故在 wait 操作中也必须执行 V(mutex),否则,会妨碍其他进程进入管程,导致无法释放资源。

2. 挂起发出 signal 操作的进程的信号量 next 和计数器 next_count

对于每个管程,还必须引入信号量 next(初值为 0),凡发出 signal 操作的进程用 P(next)挂起自己,直到被释放进程退出管程或产生其他等待条件。每个进程在退出管程的过程之前,必须检查是否有其他进程在信号量 next 上等待,若有,则用 V(next)唤醒它。因此,在引入信号量 next 的同时,提供 next_count(初值为 0),记录在信号量 next 上等待的进程个数。

3. 挂起等待资源的进程的信号量 x_sem 和计数器 x_count

为了使申请资源者在资源被占用时能将其封锁起来,引入信号量 x_sem(初值为 0),当申请资源得不到满足时,执行 P(x_sem)挂起自己。由于在释放资源时,需要知道是否有其他进程正在等待资源,因而,要用一个计数器 x_count(初值为 0)记录等待资源的进程数。在执行 signal 操作时,应让等待资源的诸进程中的某个进程立即恢复运行,而不让其他进程抢先进入管程,这可以用 V(x_sem)来实现。

```
//Hoare 方法数据结构和 enter、wait、signal 及 leave 操作
typedef struct InterfaceModule {          //InterfaceModule 是结构体名
    semaphore mutex;                      //进程调用管程的过程之前所使用的互斥信号量
    semaphore next;                       //发出 signal 操作的进程挂起自己的信号量
    int next_count;                       //在 next 上等待的进程数
};
    mutex=1;next=0;next_count=0;          //初始化语句
void enter(InterfaceModule &IM) {
    P(IM.mutex);                          //互斥进入管程
}
void leave(InterfaceModule &IM) {
    if(IM.next_count>0)                   //判断有否发出 signal 操作的进程
        V(IM.next);                       //若有就释放一个发出 signal 的进程
    else
        V(IM.mutex);                      //否则开放管程
}
void wait(semaphore &x_sem,int &x_count,InterfaceModule &IM) {
    x_count++;                            //等待资源的进程数加 1,x_count 初始化为 0
    if(IM.next_count>0)                   //判断有否发出 signal 操作的进程
```

这个语句需要隐含包括放掉mutex的动作。否则，mutex没有释放，导致其他进程不能使用该Monitor。

在conditional等待的时候，需要放锁。很重要。在POSIX下，不同的实现，语义会略微不一样。编程时要注意手册。

被释放的task会进入READY的调度队列


```
        V(IM.next);                    //若有就释放一个
    else
        V(IM.mutex);                   //否则开放管程
    P(x_sem);                          //等待资源的进程阻塞自己,x_sem 初始化为 0
    x_count--;                         //等待资源的进程数减 1
}
void signal(semaphore &x_sem,int &x_count,InterfaceModule &IM) {
    if(x_count>0) {                    //判断是否有等待资源的进程
        IM.next_count++;               //发出 signal 操作的进程数加 1
        V(x_sem);                      //释放一个等待资源的进程
        P(IM.next);                    //发出 signal 操作的进程阻塞自己
        IM.next_count--;               //发出 signal 操作的进程数减 1
    }
}
```

需要释放mutex

需要释放mutex。 被唤醒后，要迅速重新拿锁。

对应的操作。

需要释放mutex
被唤醒后，要迅速重新拿锁。

3.4.3 使用管程解决进程同步问题

1. 管程解决 5 位哲学家吃通心面问题

首先引入枚举类型 enum {thinking,hungry,eating} state[5]表示哲学家的状态,哲学家 i 建立状态 state[i] = eating 仅当他的两个邻座不在吃面的时候,即 state[(i-1)%5]! = eating 及 state[(i+1)%5]! = eating;还要引入条件变量(信号量)semaphore self[5],当哲学家 i 饥饿但又不能获得两把叉子时,进入其信号量等待队列。

```
//Hoare 方法解决哲学家吃通心面问题
type dining_philosophers = monitor
    enum {thinking,hungry,eating} state[5];
    semaphore self[5];
    int self_count[5];
    InterfaceModule IM;
    for(int i=0;i<5;i++)                    //初始化,i 为进程号
        state[i] = thinking;
    define pickup,putdown;
    use enter,leave,wait,signal;
void pickup(int i) {                        //i=0,1,…,4
    enter(IM);
    state[i] = hungry;
    test(i);
```

Monitor定义：
数据类型部分

管程Montior机制维护的临界区数据结构。

管程Method

Monitor定义：
函数部分

管程的Method。具备互斥性。通过enter／leave pair来实现。

```
        if(state[i]!=eating)
            wait(self[i],self_count[i],IM);
        leave(IM);
    }
    void putdown(int i){                                    //i=0,1,…,4
        enter(IM);
        state[i]=thinking;
        test((i-1)%5);
        test((i+1)%5);
        leave(IM);
    }
    void test(int k){                                       //k=0,1,…,4
        if((state[(k-1)%5]!=eating)&&(state[k]==hungry)
            &&(state[(k+1)%5]!=eating)){
            state[k]=eating;
            signal(self[k],self_count[k],IM);
        }
    }
```

通过对Mutex的使用，确保任何时刻只有一个thread能在这个区域。

任何一位哲学家想吃通心面时调用过程 pickup，吃完通心面之后调用过程 putdown。

```
cobegin
process philosopher_i(){ //i=0,1,…,4
    while(true){
        thinking();
        dining_philosophers.pickup(i);
        eating();
        dining_philosophers.putdown(i);
    }
}
coend
```

调用Monitor的Method

调用Monitor的Method

2. 管程解决生产者－消费者问题

```
//管程解决生产者－消费者问题
type producer_consumer=monitor
    item B[k];                                              //缓冲区个数
    int in,out;                                             //存取指针
    int count;                                              //缓冲区中产品数
```

```
    semaphore notfull,notempty;                    //条件变量
    int notfull_count,notempty_count;
    InterfaceModule IM;
    define append,take;
    use enter,leave,wait,signal;
    void append(item & x) {
        enter(IM);
        if(count==k)                               //缓冲区已满
            wait(notfull,notfull_count,IM);
        B[in]=x;
        in=(in+1)%k;
        count++;                                   //增加一个产品
        signal(notempty,notempty_count,IM);        //唤醒等待的消费者
        leave(IM);
    }
    void take(item &x) {
        enter(IM);
        if(count==0)
            wait(notempty,notempty_count,IM);      //缓冲区已空
        x=B[out];
        out=(out+1)%k;
        count--;                                   //减少一个产品
        signal(notfull,notfull_count,IM);          //唤醒等待的生产者
    leave(IM);
    }
cobegin
process producer_i() {                             //i=1,2,…,n
    item x;
    produce(x);
    producer_consumer.append(x);
}
process consumer_j() {                             //j=1,2,…,m
    item x;
    producer_consumer.take(x);
    consume(x);
```

互斥

互斥

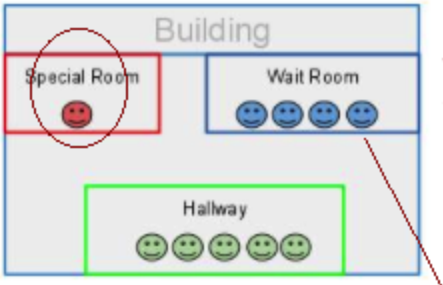
使用Monitor（管程）

使用Monitor（管程）

wait／signal存在释放mutex和重新拿到mutex的行为。

1. What is a Monitor?

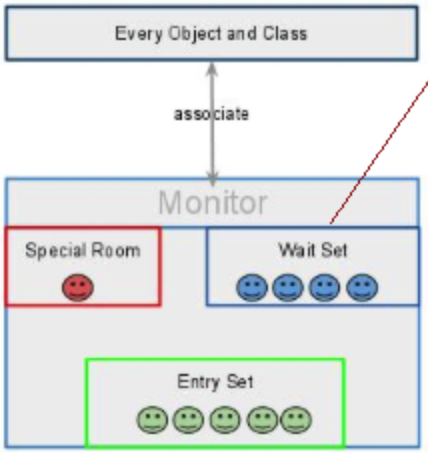
A monitor can be considered as a building which contains a special room. The special room can be occupied by only one customer(thread) at a time. The room usually contains some data and code.



从而避免程序员自己手工写各种同步，互斥代码。

互斥性。通过一个mutex来实现

If a customer wants to occupy the special room, he has to enter the Hallway(Entry Set) to wait first. Scheduler will pick one based on some criteria(e.g. FIFO). If he is suspended for some reason, he will be sent to the wait room, and be scheduled to reenter the special room later. As it is shown in the diagram above, there are 3 rooms in this building.



被唤醒的时候，操作系统根据一个算法把锁给一个Task。从而，获得进入Special Room。

In brief, a monitor is a facility which monitors the threads' access to the special room. It ensures that only one thread can access the protected data or code.

How to make a method thread-safe in Java?

By X Wang

Interview Question:

Is the following method thread-safe? How to make it thread-safe?

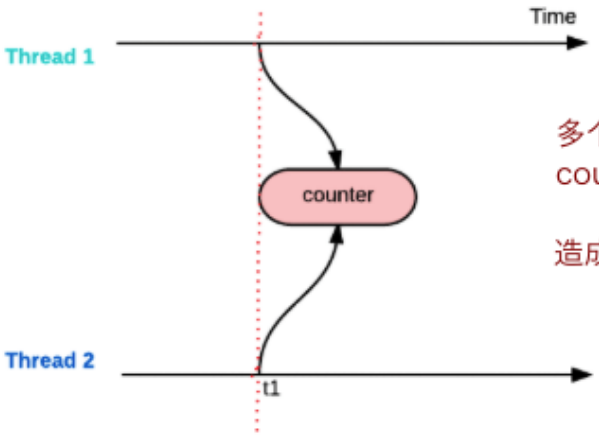
```
class MyCounter {
        private static int counter = 0;

        public static int getCount() {
                return counter++;
        }
}
```

一个Class的Static变量。

This post explains a general interview question that has been asked by Google and a lot of companies. It's low-level and not about how to design concurrent program.

First of all, the answer is NO. The method is not thread-safe, because the *counter++* operation is not atomic, which means it consists more than one atomic operations. In this case, one is accessing value and the other is increasing the value by one.



多个该Class的Object实例可以并发存取counter变量。

造成结果可能混乱。

Make a method thread-safe - Method 1

Adding synchronized to this method will makes it thread-safe. When *synchronized* is added to a static method, the *Class* object is the object which is locked.

Is marking it synchronized enough? The answer is YES.

```
class MyCounter {
        private static int counter = 0;

        public static synchronized int getCount() {
                return counter++;
        }
}
```

Java语言库会给这个Object配套一个相应的Monitor（管程）。从而保证互斥。

If the method is not static, then adding *synchronized* keyword willsynchronize the instance of the class, not the *Class* object.

2. How is it implemented in Java?

In the Java virtual machine, every object and class is logically associated with a monitor. To implement the mutual exclusion capability of monitors, a lock (sometimes called a mutex) is associated with each object and class. This is called a semaphore in operating systems books, mutex is a binary semaphore.

If one thread owns a lock on some data, then no others can obtain that lock until the thread that owns the lock releases it. It would be not convenient if we need to write a semaphore all the time when we do multi-threading programming. Luckily, we don't need to since JVM does that for us automatically.

To claim a monitor region which means data not accessible by more than one thread, Java provide synchronized statements and synchronized methods. Once the code is embedded with synchronized keyword, it is a monitor region. The locks are implemented in the background automatically by JVM.

3. In Java synchronization code, which part is monitor?

We know that each object/class is associated with a Monitor. I think it is good to say that each object has a monitor, since each object could have its own critical section, and capable of monitoring the thread sequence.

To enable collaboration of different threads, Java provide wait() and notify() to suspend a thread and to wake up another thread that are waiting on the object respectively. In addition, there are 3 other versions:

当使用wait和notify的时候，是已经拿到了Mutex，进入了critical area。

```
wait(long timeout, int nanos)
wait(long timeout) notified by other threads or notified by timeout.
notify(all)
```

Those methods can only be invoked within a synchronized statement or synchronized method. The reason is that if a method does not require mutual exclusion, there is no need to monitor or collaborate between threads, every thread can access that method freely.

Here are some synchronization code examples.

[注释]:Monitor是一个很有用的机制，特别是在面向对象的程序语言中，都提供了相应的机制。其特点是：通过sem，mutex的封装，使得程序员不需要过多的关心低级同步原语（raw primitives）的使用，从而可以在比较高的语义上把握同步，并专注其应用问题本身。

数据交换（Data Exchange）

}

coend

控制信号（Control）

3.5 进程通信

并发进程之间的交互必须满足两个基本要求：同步和通信。进程同步本质上是一种仅传送信号的进程通信，通过修改信号量，进程之间可以建立联系，相互协调运行和协同工作，但它缺乏传递数据的能力。在多任务环境中，可由多个进程分工协作完成同一任务，于是，它们需要共享一些数据和相互交换信息，某些情况下所交换的信息量很少，但在很多场合需要交换大批数据，可以通过通信机制来完成，进程之间互相交换信息的工作称为进程通信（InterProcess Communication，IPC）。通信方式有很多，包括：

(1) 信号(signal)通信机制

(2) 管道(pipeline)通信机制

(3) 消息传递(message passing)通信机制

(4) 信号量(semaphore)通信机制

(5) 共享主存(shared memory)通信机制

4， 5是主要的方式。

其中，前两种是 UNIX 最早的版本就提供的进程通信机制，信号通信机制只能发送单个信号而不能传送数据；管道通信机制虽然能传送数据，却只能在进程家族内使用，应用范围有限。此后，UNIX System V 版研制和开发后三种进程通信机制，这就是著名的 System V IPC，其共性是，在同一机器上的任何进程都可以使用这些机制通信，且相互通信的进程并不需要有家族关系。而 BSD UNIX 则实现了套接字(socket)网络进程通信机制，第八章将对此作简单介绍。

3.5.1 信号通信机制

信号(signal)是一种软中断，是传递短消息的简单通信机制，通过发送指定信号来通知进程某个异步事件发生，以迫使进程执行信号处理程序。信号处理完毕后，被中断进程将恢复执行。同中断和异常相比，多数信号对用户态进程是可见的，可以被应用进程捕获。信号可通过异常(硬中断)直接产生，如进程执行非法指令或段违例等，内核会发送给它 SIGILL 或 SIGSEGV；进程也可发出信号，以实现进程的同步或终止，如 SIGKILL 将强迫另一进程终止。一般地，分或操作系统标准信号和应用进程定义信号，这种机制模拟硬中断，但不分优先级，简单且有效，但不能传送数据，故能力较弱。

有关软中断与硬中断的相似点和差别已在中断机制中介绍过，举例来说，当用户正在运行一个耗时的应用程序时，如果已发现有错误，且断定此程序要失败，为了节省时间，可杀死当前运行进程。系统中所发生的相应事件链如下：

步骤 1：用户按中断组合键 Ctrl+C。

步骤 2：终端驱动程序收到输入字符，并调用信号系统；

多用来发异步信号。不太作为数据交换手段。比较适合发控制信息

1. SIGUSR1 - User defined signal 1
2. SIGUSR2 - User defined signal 2

这两个SIGNAL是用户可定义的类型。在用户态处理。

步骤 3:信号系统发送 SIGINT 信号给 shell,shell 再把它发送给进程;

步骤 4:进程收到 SIGINT 信号;

步骤 5:进程撤销。

用户、内核和进程都能生成信号请求:

(1) 用户

用户能通过按键 Ctrl+C,或终端驱动程序分配给信号控制字符的其他键来请求内核产生信号。

(2) 内核 通常不可以捕获。进程会在最快的被杀掉。

当进程执行出错时,内核检测到事件并向进程发送信号,例如,非法段存取、浮点数溢出或非法操作码,内核也利用信号通知进程有特定事件发生。

(3) 进程

进程可通过系统调用 kill 给另一个进程发送信号,一个进程可通过信号与另一个进程通信。

由进程执行指令而产生的信号称为同步信号,如被 0 除;像击键之类的进程以外的事件所引起的信号称为异步信号,接收进程对信号所做出的响应为:执行默认操作(如 SIGINT 的默认处理是进程撤销)、执行预置的信号处理程序或忽略此信号。标准版 UNIX 提供 19 个信号,Linux 中定义 64 个信号,采用与标准版兼容且增加新信号的方法保证信号通信的一致性和兼容性。信号可分成以下几类、

(1) 与终止进程相关的信号:SIGCHLD(子进程暂停或终止)、SIGHUP(进程挂起)、SIGKILL(强行杀死进程)、SIGABRT(进程异常终止)、SIGSTOP(被调试器阻塞,进程暂停)、SIGTSTP(来自终端的暂停信号)等;

(2) 与例外事件相关的信号:SIGBUS(总线超时)、SIGSEGV(段违例)、SIGPWR(电源故障)、SIGFPE(浮点溢出)、SIGSTKFLT(协处理器栈出错)、SIGURG(套接字紧急情况)、SIGXCPU(CPU 限时超出)、SIGXFSZ(文件限制超出)、SIGWINCH(窗口大小变化)等;

(3) 与执行系统调用相关的信号:SIGPIPE(管道有写者无读者)、SIGILL(非法指令)、SIGIO(与 I/O 操作有关)等;

(4) 与终端交互相关的信号:SIGINT(键盘中断)、SIGQUIT(输入退出命令)、SIGTTIN(后台进程读终端)SIGTTOU(后台进程写终端)等;

(5) 与用户进程相关的信号:SIGTERM(软件终止)、SIGALRM(定时报警)、SIGUSR1(用户定义信号 1)、SIGUSR2(用户定义信号 2)等;

(6) 与跟踪进程执行相关的信号:SIGTRAP(跟踪陷阱)等。

信号有一个产生、传送、捕获和释放的过程,Linux 信号实现机制包括:数据结构、信号的产生和发送、信号的检测和处理。

(1) 每个进程 task_struct 结构中的 signal 域专门保存接收到的信号,内核根据所发生的事件产生相应的信号并发送给接收进程。进程接收到信号时,对应位设置为 1,相当于"中断请求寄存器"的某位置位,由于所发送的信号被置于接收进程的进程控制块中,所以,无论进程是否在主存,总可以立即被记录。一个进程可能收到多个不同的信号,系统仅允许进程每次处理编号最小

有些Signal是用户进程可以捕获, override的。
有些是用户进程不能控制的。

的一个，其余信号只有在此进程下次调度运行时才被处理。

(2) 进程 task_struct 结构中的 blocked 是信号屏蔽标记，相当于"中断屏蔽寄存器"。进程需要忽略某信号时就把对应的位设置为 1。sigpending 标志指示在 sigqueue 队列中是否有挂起的信号，如果信号发送之后目标进程在睡眠就唤醒它，让它接收和处理此信号，任何时候一种类型的信号至多只有一个信号挂起。

(3) 信号处理程序的入口存放在进程 task_struct 的 sigaction[]数组中，数组共有 64 个元素，每个元素占 32 bit，信号的编号对应于数组下标，数组元素的值是信号处理程序的入口地址。有很多方法可设置信号处理方式，除了 signal 之外，如执行函数 fork、exec、exit 等都会改变数组的内容。

(4) 函数 sigaction(signo,act,oldact)为指定信号预置处理程序，signo 指出接收信号的类型，act 是收到信号后希望调用的信号处理函数地址，oldact 存放最近一次为信号 signo 定义的函数地址。为保持兼容性，仍保留 signal 函数，但功能上已被 sigaction 替代。

(5) 函数 kill(pid,sig)用来向指定的进程发送指定信号。当信号被发送时，内核的工作是更新进程控制块的信号域，并对与信号相关的数据结构进行操作，如置位、屏蔽和清除等。pid 确定信号 sig 所发往的进程，sig 是信号类型，由于这种信号要知道所发往进程的标识号，所以，信号发送通常在关系密切的进程之间进行。

(6) 信号的检测和响应总发生在系统空间，在中断或异常处理程序末尾，在进程从核心态返回用户态之前，或在处理运行进程所遇到的时钟中断结束之前，或进程以 interruptible 状态进入等待队列之前(若它已收到信号，就不睡眠)，系统会调用内核函数 do_signal()检查此进程是否已收到信号，若是则执行 handle_signal()让它返回用户态并转入信号处理程序执行。当信号处理结束后，通过执行系统调用 sigreturn()陷入内核，内核做好善后工作后返回用户态，回到应用程序的断点执行。如图 3.4 所示是运行进程收到一个信号后的检测和处理流程。

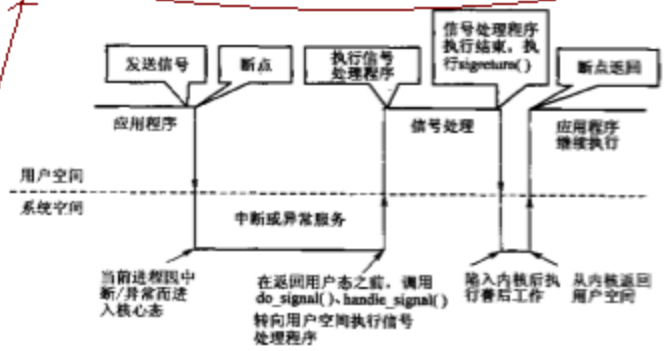


图 3.4 信号的检测和处理流程

Signal是一种异步机制。进程在几个调度点的时候，（才）做检查。看是否有Signal Pending。

Pipeline的本质是通过一个文件（File）来做载体。

Windows 操作系统的信号通信机制统一在分派器对象(dispatcher object)的公共框架中，每个分派器对象可处于两种状态之一：已发信号(signaled)和未发信号(unsignaled)。进程可以使用函数 WaitForSingleObject 把自己阻塞，以等待一个未发信号的对象，当某个进程把此对象的状态改为已发信号时，此进程就会恢复运行。进程也可使用函数 WaitForMultipleObjects 来在多个对象上阻塞自己，当所有对象均被发送信号后，它才会恢复执行。对象的类型包括进程、线程、文件、事件、信号量、定时器、互斥量和队列。例如，当线程终止时，就会向线程对象发送信号，所有等待线程都被唤醒；当线程释放互斥锁之后，就会向互斥量对象发送信号，一个等待线程被唤醒；当定时器到时的时候，就会向定时器对象发信号，可以唤醒所有等待线程或仅唤醒一个等待线程；当某文件的 I/O 操作终止时，就会向此文件对象发送信号，所有等待线程将被唤醒。

3.5.2 管道通信机制

管道(pipeline)是 UNIX 的传统进程通信方式，也是 UNIX 发展最有意义的贡献之一。它是连接读写进程的一个特殊文件，允许按照 FCFS 方式传送数据，也能使进程同步执行。管道是单向的，发送进程视管道文件为输出文件，以字符流的形式把大量数据送入管道；接收进程视管道文件为输入文件，从管道中接收数据，所以也称为管道通信。由于管道通信机制方便有效，能够在进程间进行大信息量的通信，目前已被引入许多操作系统中。

管道的实质是一个共享文件，即利用辅助存储器来进行数据通信。因此，管道通信基本上可借助于文件系统的机制来实现，包括管道文件的创建、打开、关闭和读写。但是，写进程和读进程之间的相互协调单靠文件系统机制是解决不了的，读写进程相互同步，必须做到以下几点。

(1) 进程正在使用管道写入或读取数据时，另一个进程必须等待，这一点是进程在读写管道之前，通过测试文件 inode 节点的特征位——读写互斥标志来保证的；若已锁住，进程便等待，否则，把 inode 上锁，然后，进行读写操作。操作结束后再行解锁并唤醒因节点上锁而等待的进程。

(2) 发送者和接收者双方必须知道对方是否存在，如果对方已经不存在，就没有必要再发送或接收信息，这时系统会发出 SIGPIPE 信号通知进程。

(3) 发送信息和接收信息之间要实现正确的同步关系，这是由于管道文件只使用 inode 节点中的直接地址项，长度限于 10 个盘块，管道的长度限制对于进程的 write 和 read 操作会产生影响。如果进程执行一次写操作，且管道有足够的空间，那么，write 操作把数据写入管道后唤醒因此管道空而等待的进程；如果此次操作会引起管道溢出，则本次 write 操作必须暂停，直到其他进程从管道中读取数据，使管道有空余空间为止，这叫做 write 阻塞。解决这个问题的方法是：把数据进行切分，每次均小于管道限制的字节数，写完后此进程睡眠，直到读进程把管道中的数据取走，并判别有进程等待时唤醒对方，以便继续写下一批数据。反之，当读进程读空管道时，要出现 read 阻塞，读进程应睡眠，直到写进程唤醒它。

应用程序用 pipe()创建管道，pipe 分配文件磁盘 inode 和活动 inode 节点，然后，为读写进程各分配一个文件描述符，同时，系统打开文件表中应建立两个 file 结构，分别控制管道的写操作和读操作，即定义 pipe 文件的写入端和读出端。两个 file 结构分别与文件描述符相连，并指向同

每个进程独有的数据结构

一个活动 inode 节点，活动 inode 节点的 i_count 值为 2，因为有两个 file 项指向它。内核在执行完 pipe()系统调用创建无名管道后，返回文件句柄 files[0]和 files[1]，接收者进程通过 files[0]从管道中用 read()读数据，发送者进程则通过 files[1]用 write()向管道写入数据，结束之后用 close()关闭管道。pipe 的数据结构如图 3.5 所示。

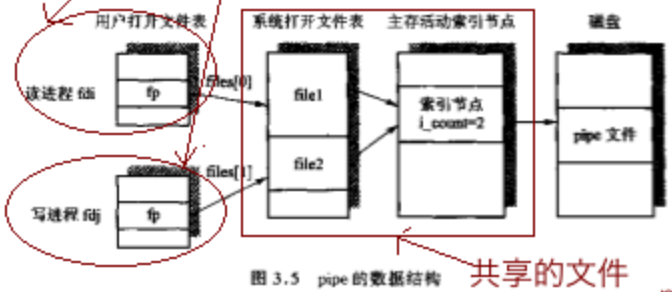


图 3.5 pipe 的数据结构

共享的文件

named pipe

管道是一种功能很强的通信机制，但仅用于连接具有共同祖先的进程，使用时需要临时建立，难以提供全局服务。为了克服这些缺点，UNIX 推出管道的一个变种，称为有名管道或 FIFO 通信机制，用来在不同的地址空间之间进行通信，特别为服务器通过网络与多个客户进行交互而设计。这是一种永久性通信机制，具有文件名、目录项、访问权限，能像一般文件一样被操作，但在读或写时，其性能与管道相同。可以通过系统调用 mknod 而非 pipe 来创建命名管道，然后，接收者进程像使用其他文件一样通过系统调用 open 来打开管道以便取走信息；发送者进程则通过系统调用 open 来打开管道以写入数据。对管道的读写操作通过系统调用 read 和 write 来实现。

3.5.3 共享主存通信机制

Shared Memory

共享主存是指在主存中开辟一个公用存储区，要通信的进程把自己的虚地址空间映射到共享主存区，如图 3.6 所示。当发送进程将信息写入共享主存区的某个位置时，接收进程可从此位置读取信息，由于共享主存区同时出现在进程 1 和进程 2 的虚存空间中，从而能实现进程之间的通信。这也是进程通信中最快捷和最有效的方法，此机制最早是在 UNIX System V 中作为进程通信的一部分而设计的。通过共享主存 API，允许进程动态地定义共享主存区，由于进程的虚拟地址空间相当大，所定义的共享主存区应对应于一段未使用的虚地址区，以免与进程映像区发生冲突。共享主存的页面在每个共享进程的页表中都有页表项引用，但无须在所有进程的虚存段都有相同的地址。因为不

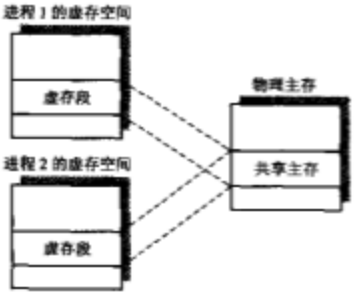


图 3.6 共享主存通信机制

这个System V机制用的越来越少。许多程序都是多线程模型了。所以，都直接共享全局变量了。不需要这样的IPC了。

止一个进程可将共享主存映射到自己的虚地址空间中去，读写共享主存区的代码段通常被认为是临界区。

3.5.4 消息传递通信机制

Message Passing

1. 消息传递的概念

交互式并发进程通过信号量及 PV 操作可实现进程的互斥和同步。生产者－消费者问题是一组相互协作的进程，它们通过信号量来协同工作，另外引入有界缓冲区来存取产品，这种低级通信方式既不方便且局限性也大。当进程之间需要交换更多的信息时，例如，要求把数据从一个进程传送给另一个进程，甚至不同机器上的进程之间要传送数据，需要引入高级通信方式——消息传递(message passing)机制(UNIX 和 Linux 中称其为消息队列)来实现。由于操作系统所提供的这类机制隐蔽实现细节，通过消息传递机制完成通信，就能简化程序编制的复杂性，方便易用。

消息传递的复杂性在于：地址空间的隔离，发送进程无法将消息直接复制到接收进程的地址空间中，这项工作仅能由操作系统来完成。为此，消息传递机制至少需要提供两条原语 send 和 receive，前者向一个给定的目标进程发送一条消息，后者则从一个给定的源进程接收一条消息。如果没有消息可用，则接收者可能阻塞直至一条消息到达，或者也可立即返回，并带回一个错误码。采用消息传递机制后，进程之间用消息来交换信息，一个运行进程可在任何时刻向另一个运行进程发送一条消息；一个运行进程也可在任何时刻向另一个运行进程请求一条消息，如果进程在某一时刻的执行依赖于另一进程的消息或等待其他进程对所发消息的应答，那么，消息传递机制将紧密地与进程的阻塞和释放相联系，使得进程通信机制不但具有进程通信的能力，还提供进程同步的能力。

mailbox

为了实现异步通信，必须采用间接的通信方式，进程之间发送或接收消息通过一个共享的数据结构——信箱进行，消息可被理解成信件，每个信箱都有唯一标识符。当两个以上的进程拥有共享信箱时，它们就能进行通信。间接通信解除了发送进程和接收进程之间的直接联系，在消息的使用上加大了灵活性。一个进程可以分别与多个进程共享信箱，于是，一个进程可同时和多个进程进行通信，一对一关系允许在两个进程间建立不受干扰的专用通信链接；多对一关系对客户－服务器间的交互非常有用；一个进程为其他进程提供服务，这时的信箱又称作端口(port)，端口通常划归接收进程所有并由接收进程创建，服务进程被撤销时，其端口也随之毁灭；一对多关系适用于一个发送者和多个接收者，可在一组进程间发送广播消息；多对多关系允许建立公用信箱，多个进程既可向信箱发送信件，也可从中取出所属信件。间接通信方式中的"发送"和"接收"原语的形式如下：

(1) send(A，消息)：把一条消息(信件)传送到信箱 A。

(2) receive(A，消息)：从信箱 A 接收一条消息(信件)。

信箱是存放信件的存储区域，每个信箱可分成信箱头和信箱体，信箱头指出信箱容量、信件格式、存放信件的位置的指针等；信箱体用来存放信件，信箱体分成若干区，每个区可容纳一条消

息。"发送"和"接收"两条原语的功能为：

(1) 发送信件

如果指定的信箱未满，则将信件送入信箱中由指针所指示的位置，并释放等待此信箱中信件的等待接收者；否则，发送信件者被置成等待信箱状态。

(2) 接收信件

如果指定的信箱中有信，则取出一封信件，并释放等待信箱的等待发送者，否则，接收信件者被置成等待信箱信件的状态。

2. 信箱的设置

信箱可以在用户空间或系统空间开辟。若信箱设置在用户空间，那么，此信箱属于进程私有，需要区分信箱的所有者及其使用者，创建信箱的进程就是信箱的所有者，它有权从中收信，其他进程都可以向此进程的信箱发信，当拥有信箱的进程撤销时，其信箱也随之消失，这时必须把这一情况及时通知信箱的所有使用者。

一种替代方法是在系统空间设置每个进程的私有信箱，并推迟消息的复制操作，直到接收进程发出接收系统调用。由于进程不能直接访问消息，能够避免信箱和消息被破坏的危险，但需要系统为所有进程的信箱分配存储空间，分配多大空间较难决定，给定时间内等待传送的消息数目也会受到限制。一种改进的方法是：在系统空间统一设置公用信箱，供系统中的所有进程共享，B.Hansen 的消息缓冲机制就是一个著名的例子。

3. 通信进程的同步

两个进程间的消息通信就隐含着某种程度的同步，当发送进程执行 send()发出消息后，本身的执行可分为两种情况，一种是同步的(阻塞型)，等待接收进程回答消息后才继续进行；另一种是异步的(非阻塞型)，将消息传送到接收进程的信箱中，允许继续运行，直到某个时刻需要接收进程送来回答消息时，才查询和处理。对于接收进程而言，执行 receive 后也可以是阻塞型(同步的)和非阻塞型(异步的)，前者是指直到消息交付完成它都处于等待消息的状态；后者则不要求接收进程等待，当它需要消息时，再接收并处理消息。

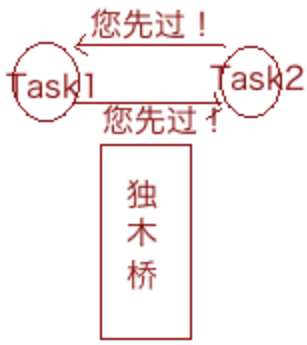
无论同步或异步 send 操作，都会有失败的情形。如果发送进程试图向一个并不存在的进程发送消息，操作系统将无法识别用哪个信箱来缓存消息，那么，下一步如何处理呢？同步 send 时，会返回一个错误码至发送方，发送方依赖错误码来工作；异步 send 时，发送方可继续发送消息，而不期望有任何返回码，系统可以使用像 UNIX 中的信号机制来告诉发送方消息的发送失败。

receive 操作可以是阻塞型或非阻塞型的。阻塞型接收操作的行为是：如果信箱中没有消息，接收进程会被挂起，直到有消息投入信箱；如果信箱中有消息，则立即获得一条消息并返回。因而，阻塞型 receive 操作能够同步发送进程和接收进程的通信；非阻塞型 receive 操作在查询信箱之后，立即向调用进程返还控制权，如果信箱中有消息，就返回消息，否则返回标志码，表明无消息可用。这种方法允许接收进程轮询信箱，如果信箱中没有待处理的消息，它可以去做其他工作。

send／recv的配合，本身就蕴含了两个进程之间的同步关系。

[注释]:System V IPC通信机制是实现多进程之间通信和同步的重要手段。例如，message queue，shared memory，pipe等。其重要前提是：不同地址空间上的多进程之间的通信和数据交换。System V的IPC用的比较少了。重要原因是：多线程的同一进程编程模型的流行。

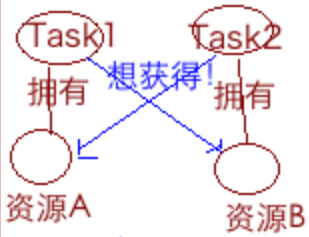
LiveLock是相对应的一个概念。其实与DeadLock差不多。但进程没有死，而是中那里来回变化状态，但也是没法往下执行。经典的例子是小走廊（Corridor）问题。两个人中走廊相遇。互相侧身让，左让，右让，循环往复，大家都过不去。



LiveLock的例子：
大家都忙死了。
但什么进展也没有。

DeadLock:所有相关的Task都在Sleep状态。
LiveLock：所有相关的Task都在原地震荡（状态变化），但无法前行。
Task Starvation：严重不公平现象发生了

DeadLock是并发程序设计中的一个重要的研究课题。但在实际工作中，其实不太常见。但需要了解其相关原理。



死锁。都不愿放弃已经拥有的；同时还想获取另外的。

3.6 死　锁

3.6.1 死锁产生

计算机系统中有一些资源，在任一时刻都只能被一个进程所使用，如磁带机、绘图仪等独占型外部设备，或共享数据结构、等待链表等软件资源。两个进程同时向一台打印机输出数据将导致一片混乱，两个进程同时进入临界区将导致数据错误乃至系统崩溃，因此，所有操作系统都具

例如，占有一个锁A，进入临界区，然后又想own另外一个锁B。

有向一个进程授权独立访问某个资源的能力，进程使用独占型资源必须按照以下的次序进行：申请资源、使用资源、归还资源。若资源不可用，则申请进程进入等待态。对于不同类型的独占型资源，进程的等待方式是有差异的，如申请打印机资源、临界区资源时，申请失败将意味着阻塞申请进程；而申请打开文件资源时，申请失败将返回一个错误代码，由申请进程等待一段时间后再次尝试。

在许多应用中，进程需要以独占方式访问多个资源，而当操作系统允许多个进程并发执行时，可能出现进程永远被阻塞的现象。例如，两个进程分别等待对方所占有的一个资源，于是两者都不能执行而处于永远等待状态，此现象称为"死锁"。产生死锁的原因有很多，例如，进程推进顺序不当、P或V操作使用不妥、同类资源分配不均或对某些资源的使用未加限制，可见，产生死锁的因素不仅与系统拥有的资源数量有关，而且与资源分配策略、进程对资源的使用要求以及并发进程的推进顺序有关。

死锁通常被定义为：如果一个进程集合中的每个进程都在等待只能由此集合中的其他进程才能引发的事件，而无限期陷入僵持的局面称为死锁。

本章基于进程来讨论死锁，这些概念也同样适用于线程之间产生死锁的情况。死锁会让系统造成很大的损失，必须付出额外的代价来解决死锁问题，主要的解决方法是：死锁防止(deadlock prevention)、死锁避免(deadlock avoidance)、死锁检测和恢复(deadlock detection and recovery)。

动态检测和恢复不太常采用。

3.6.2 死锁防止

1. 死锁产生的条件

1971年，Coffman总结系统产生死锁必定同时保持4个必要条件如下。

(1) 互斥条件(mutual exclusion)：系统中存在临界资源，进程应互斥地使用这些资源。

(2) 占有和等待条件(hold and wait)：进程在请求资源得不到满足而等待时，不释放已占有的资源。

(3) 不剥夺条件(no preemption)：已被占用的资源只能由属主释放，不允许被其他进程剥夺。

(4) 循环等待条件(circular wait)：存在循环等待链，其中，每个进程都在链中等待下一个进程所持有的资源，造成这组进程处于永远等待状态。

前3个条件是死锁存在的必要条件，但不是充分条件，第4个条件是前3个条件同时存在时所产生的结果，故条件并不完全独立。但是，单独考虑每个条件是有用的，只要能破坏4个必要条件之一，就可以防止死锁。

2. 死锁防止策略

(1) 破坏条件1(互斥条件)

有些优化工作，例如，支持同时多个READ读操作；rwlock。

使资源可同时访问而非互斥使用，也就没有进程会阻塞在资源上，从而不发生死锁。可重入程序、只读数据文件、时钟、磁盘等软硬件资源均可采用这种办法管理，但有许多资源如可写文

1,2,3不太采纳。通常还是采用避免循环等待条件。

传统操作系统的大锁（Big Lock）就是上来把所有的资源覆盖。坏处是：锁的粒度太大，导致效率低。

件、磁带机等由于其特殊性质决定只能互斥地占有，而不能被同时访问，所以这种做法在许多场合行不通。

(2) 破坏条件 2(占有和等待条件)

静态分配是指进程必须在执行之前就申请所需要的全部资源，且直至所要的资源都得到满足后才开始执行，无疑所有并发执行的进程所要求的资源总和不超过系统拥有的资源数。采用静态分配策略后，进程在执行过程中不再申请资源，就不会出现占有某些资源再等待另一些资源的情况。这种策略实现简单，被许多操作系统所采用，但会严重地降低资源利用率，因为在每个进程所占有的资源中，有些资源在运行后期使用，甚至有些资源在例外情况下才被使用，可能会造成进程占有一些几乎用不到的资源，而使其他想使用这些资源的进程等待。

(3) 破坏条件 3(不剥夺条件)

剥夺调度能够防止死锁，但只适用于主存和处理器资源。方法一是占有资源的进程若要申请新的资源，必须主动释放已占用资源(剥夺式)，若仍需要占用此资源，应该向系统重新提出申请，从而破坏了不剥夺条件，但会造成进程重复地申请和释放资源；方法二是资源管理程序为进程分配新资源时，若有则分配之，否则将剥夺此进程所占有的全部资源，并让进程进入等待资源的状态，资源充足后再唤醒它重新申请所有所需资源。

(4) 破坏条件 4(循环等待条件)

采用层次分配策略，将系统中的所有资源排列到不同的层次中，一个进程得到某层的一个资源后，只能再申请较高一层的资源；当进程释放某层的一个资源时，必须先释放所占用的较高层资源；当进程获得某层的一个资源后，如果想申请同层的另一个资源，必须先释放此层中的已占用资源。

层次分配策略的一个变种是按序分配策略。把系统的所有资源按顺序编号，规定进程请求所需资源的顺序必须按照资源的编号依次进行。例如，若系统中共有 n 个进程，共有 m 个资源，用 r_i 表示第 i 个资源，于是 m 个资源是 r_1、r_2、…、r_m，规定进程不得在占用资源 $r_i(1\leqslant i\leqslant m)$ 后再申请 $r_j(j<i)$，就可防止系统发生死锁，因为采用有序使用资源的方法能够有效地破坏循环等待条件。

假设存在死锁：Task A和B。A有 e个资源，请求第（e＋k）个资源。B有 f个资源，请求第（f＋p）个资源。不失去一般性，假设e＜f，显然，Task B要求的第（f＋p）＞e. 所要的资源不可能锁A拥有。因此，A和B不可能死锁。

3.6.3 死锁避免

1. 银行家算法

各种死锁防止方法能够防止发生死锁，但必然会降低系统的并发性并导致低效的资源利用率。死锁避免却与此相反，允许系统中同时存在前 3 个必要条件，通过合适的资源分配算法确保不会出现进程循环等待链，从而避免死锁。死锁避免方法能支持更多的进程并发执行，它不是对进程随意强加规则，而是动态地确定是否分配资源给提出请求的进程。如果一个进程当前请求的资源会导致死锁，系统将拒绝启动此进程；如果一个资源的分配会导致下一步死锁，系统便拒绝本次分配。

Dijkstra 于 1965 年提出能够避免死锁的调度方法，称为银行家算法。此算法描述如下，假定

有的系统实现cond var（条件变量）时是这么做的。做sleep在条件变量上的时候，要把之前占有的mutex释放掉。

小城镇银行家拥有资金，数量为Σ，被 N 个客户共享，银行家对客户提出下列约束条件：

(1) 每个客户必须预先说明所要的最大资金量；

(2) 每个客户每次提出部分资金量的申请并获得分配；

(3) 如果银行满足客户对资金的最大需求量，那么客户在资金运作后，应在有限的时间内全部归还银行。

只要客户遵守上述约束条件，银行家将保证做到：若一个客户所要的最大资金量不超过Σ，银行一定会接纳此客户，并满足其资金需求；银行在收到一个客户的资金申请时，可能会因资金不足而让客户等待，但保证在有限的时间内让客户获得资金。在银行家算法中，客户可看做进程，资金可看做资源，银行家可看做操作系统。银行家算法虽然能避免死锁，但实现时受到种种限制，要求所涉及的进程不相交，即不能有同步要求，事先要知道进程的总数和每个进程请求的最大资源数，这些都很难办到。

这个算法要求不太现实。

2. 银行家算法数据结构

考虑一个系统有 n 个进程和 m 种不同类型的资源，定义以下向量和矩阵数据结构。

(1) 系统中每类资源总数向量

$$\text{Resource} = (R_1, R_2, \cdots, R_m)$$

(2) 系统中当前每类资源可用数向量

$$\text{Available} = (V_1, V_2, \cdots, V_m)$$

(3) 进程对各类资源的最大需求矩阵（C_{ij} 表示进程 P_i 需要 R_j 类资源的最大数目）

$$\text{Claim} = \begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ & & \cdots & \\ C_{n1} & C_{n2} & \cdots & C_{nm} \end{bmatrix}$$

(4) 系统中当前资源的已分配情况矩阵（A_{ij} 表示进程 P_i 已分到 R_j 类资源的数目）

$$\text{Allocation} = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ & & \cdots & \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{bmatrix}$$

于是下列关系式成立：

$R_i = V_i + \sum_{k=1}^{n} A_{ki}$，　对 $i = 1,2,\cdots,m$ 表示所有资源要么已被分配，要么尚可分配

$C_{ki} \leqslant R_i$，　对 $i = 1,2,\cdots,m$；$k = 1,2,\cdots,n$ 表示进程申请资源数不能超过系统拥有这类资源的总数

$A_{ki} \leqslant C_{ki}$，　对 $i = 1,2,\cdots,m$，$k = 1,2,\cdots,n$ 表示进程申请任何类型资源数不能超过声明的最大资源需求数

类似银行避免挤兑的机制。系统里要事先有足够的储备现金。

说白了，就是想要的总和不能大于上限。

系统若要启动一个新进程工作，其对资源 R_i 的需求应满足不等式：

$$\boxed{R_i \geqslant C_{(n+1)i} + \sum_{k=1}^{n} C_{ki}, \quad i = 1,2,\cdots,m; k = 1,2,\cdots,n}$$

也就是说，应满足当前系统中所有进程对资源 R_i 的最大需求数，加上启动的新进程的资源最大需求数，不超过系统所拥有的最大资源数时，才能启动此进程，这种死锁避免方法也称作"进程启动拒绝法"。这个算法考虑最坏的情况，即所有进程同时要使用它们声明的资源最大需求数，因而很难是最优的。

现在给出系统安全性的定义：在时刻 T_0 系统是安全的，仅当存在一个进程序列 $P_1, P_2, \cdots, P_n$，对进程 $P_k(k=1,2,\cdots,n)$ 满足公式：

空缺

$$\boxed{C_{ki} - A_{ki}} \leqslant \boxed{V_i + \sum_{j=1}^{k-1} A_{ji}, \quad k = 1,2,\cdots,n; i = 1,2,\cdots,m}$$

此序列称为安全序列。其中，公式左边表示进程 P_k 尚缺少的各类资源，V_i 是 T_0 时刻系统尚可用于分配且为 P_k 所想要的那类资源数，$\sum_{j=1}^{k-1} A_{ji}$ 表示排在进程 P_k 之前的所有进程所占用的 P_k 所需要的资源总数。显然，进程 P_k 所需资源若不能立即被满足，那么，在所有 $P_j(j=1,2,\cdots,k-1)$ 运行完成后可以满足，然后，P_k 也能获得资源以完成任务；当 P_k 释放全部资源后，P_{k+1} 也能获得资源以完成任务；如此下去，直到最后一个进程完成任务，从 T_0 时刻起按照这个进程序列运行的系统是安全的，绝对不会产生死锁。

3. 银行家算法描述

银行家算法又称"资源分配拒绝"法，其基本思想是：系统中的所有进程放入进程集合，在安全状态下系统收到进程的资源请求后，先把资源试探性地分配给它。现在，系统将剩下的可用资源和进程集合中其他进程还需要的资源数作比较，找出剩余资源能满足最大需求量的进程，从而保证进程运行完毕并归还全部资源。这时，把这个进程从进程集合中删除，归还其所占用的所有资源，系统的剩余资源则更多，反复执行上述步骤。最后，检查进程集合，若为空则表明本次申请可行，系统处于安全状态，可以真正实施本次分配；否则，只要进程集合非空，系统便处于不安全状态，本次资源分配暂不实施，让申请资源的进程等待。

```
//避免死锁的银行家算法
    typedef struct state{                          //全局数据结构
        int resource[m];
        int available[m];
        int claim[n][m];
        int allocation[n][m];
    };
    void resource_allocation() {                   //资源分配算法
        if(allocation[i, * ] + request[ * ]>claim[i, * ])
```

系统还剩下的 + 目前被其他进程占据但会逐步释放回来的。

3.6.4 死锁检测和解除

1. 资源分配图和死锁定理

对资源的分配加以适当限制可防止和避免死锁的发生，但不利于进程对系统资源的充分共享。解决死锁的另一条途径是死锁检测和解除，这种方法对资源的分配不施加任何限制，也不采取死锁避免措施，系统定时地运行"死锁检测"程序，判断系统内是否已经出现死锁，如果检测到系统已死锁，再采取措施解除它。这种方法的难点在于：要确定何时运行死锁检测算法，如果这一算法执行得很频繁，将会浪费处理器时间；如果这一算法执行得太稀疏，则死锁进程和系统资源会一直被锁定。

设某个计算机系统中有多种资源和多个进程，每个资源类用一个方框表示，方框中的黑圆点表示此资源类中的各个资源，每个进程用一个圆圈来表示，用有向边表示进程申请资源和资源被

操作系统或者一个编程环境自动检测死锁比较学术化。在工程中不太常见。多数锁下图勾画勾画。不出现典型回路。就可以了。

分配的情况。约定 $P_i \rightarrow R_j$ 为请求边，表示进程 P_i 申请资源类 R_j 中的一个资源得不到满足而当前处于等待 R_j 类资源的状态，这条有向边从进程开始指向资源类方框的边缘。反之 $R_j \rightarrow P_i$ 为分配边，表示 R_j 类中的一个资源已被进程 P_i 占用，故此有向边从方框内的某个黑圆点出发指向进程。如图 3.7 所示是进程－资源分配图的例子，其中共有 3 个资源类，每个进程的资源占有和申请情况已在图中表示出来。在此例中，由于存在占有和等待资源的环路，导致一组进程永远处于等待资源的状态，因而发生了死锁。

在进程－资源分配图中存在环路并不一定必发生死锁，因为循环等待资源仅是死锁发生的必要条件，而不是充分条件，如图 3.8 所示便是一个有环路而无死锁的例子。虽然进程 P_1 和进程 P_3 分别占有资源 R_1 和 R_2，并且等待另一个资源 R_2 和另一个资源 R_1 而形成环路，但进程 P_2 和进程 P_4 分别占有资源 R_1 和 R_2 各一个，它们所申请的资源已得到全部满足，因而，能在有限时间内归还已占有的资源。于是进程 P_1 和进程 P_3 分别获得另一个所需资源，这时进程－资源分配图中已减少两条请求边，环路不再存在，系统中就不存在死锁。

R1和R2不是唯一的。

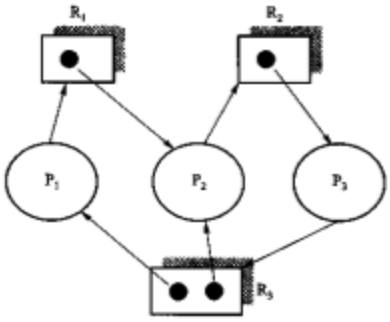


图 3.7 进程－资源分配图的例子

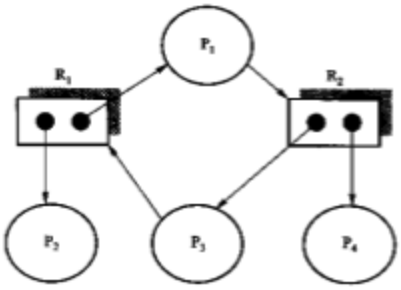


图 3.8 有环路而无死锁的例子

[注释]:DeadLock， LiveLock，Starvation都是在并发编程中需要注意的事情。 DeadLock需要避免进程之间互相要锁然后都进入了等待睡眠的状态；LiveLock要避免虽然各个进程不都在睡眠状态，但都在原地抖动,停止不前；例如，吃饭时互相谦让谁先动筷子, 结果谁也没有吃,在无解的谦让；

Starvation饿死的场景很直接，避免资源的不公平利用,例如, 不能某些任务总是获得资源, 有些任务即使长期在等待下, 却没有被分配到资源。

能释放出足够的资源来解除死锁进程的死锁。

尽管检测死锁是否出现和发现死锁后实现恢复的代价大于防止和避免死锁所花费的代价，但由于死锁不是经常出现的，因而这样做还是值得的。检测策略的代价依赖于死锁出现的频率，而恢复的代价是指处理器时间的损失。

3.7 Linux 同步机制和通信机制

中断，多核，多CPU系统带来的kernel本身存在并发。

3.7.1 Linux 内核同步机制

操作系统内核在执行过程中，由于以下原因会造成并发执行。(1) 中断及异步信号：随时可能打断正在执行的内核代码；(2) 可抢占：如果内核具有抢占性，运行的内核任务会被另一个任务所抢占；(3) 对称式多处理器系统：两个或多个 CPU 同时执行内核代码，访问同一共享数据结构。这就要求有同步机制来保证不出现竞争条件，Linux 提供多种内核同步机制。

1. 原子操作

原子操作在执行过程中保证不被打断，以防止简单的竞争条件的发生，确保操作结果的正确性，复杂的锁机制能够在原子操作的基础上构建。Linux 内核定义以下两类原子操作。

(1) 原子整数操作

针对整型变量，定义特殊数据类型 atomic_t 和专门的原子整型操作函数，典型的用途是实现计数器。以下是部分原子整型操作：atomic_int(int i)(初始化原子变量为 i)、atomic_read(v)(读整数值 v)、atomic_set(v,i)(把 v 设置成 i)、atomic_add(i,v)(把 v 增加量值 i)、atomic_sub(i,v)(把 v 减去量值 i)、atomic_sub_and_test(i,v)(从 v 中减去 i，当结果为 0 时，则返回 1，否则返回 0)、atomic_add_negative(i,v)(将 v 中增加 i，当结果为 0 时，则返回 1，否则返回 0)、atomic_inc(v)(将 v 加 1)和 atomic_dec(v)(将 v 减 1)等。

(2) 原子位图操作

针对指针变量所指定的任意一块主存区域的位序列进行操作。以下是部分原子位图操作：set_bit(int nr,void * addr)(设置位图地址 addr 的 nr 位)、clear_bit(int nr,void * addr)(清除位图地址 addr 的 nr 位)、change_bit(int nr,void * addr)(反转位图地址 addr 的 nr 位)和 test_bit(int nr,void * addr)(返回位图地址 addr 的 nr 位的值)等。

2. 内核信号量

在 Linux 系统中，也使用等待队列来实现内核使用的信号量机制。内核信号量 semaphore 定义如下：

```
struct semaphore {
    atomic_t count;
    int sleepers;
    wait_queue_head_t wait;
```

Atomic原子操作许多是通过对应的CPU提供的特殊指令来实现。可以参阅相关代码。

提供一个等待／睡眠／唤醒 的机制。 适合非实时的资源等待。

Down: 在等待睡眠的时候，signal不能唤醒
Down－Interruptible：可以唤醒。放弃等待资源，加入被调度的READY序列。

};

(1) 资源计数器 count 表示可用的某种资源数，若为正整数则表示这些资源尚可用；若为0或负整数则表示资源已用完，且因申请资源而等待的进程数是 count 的绝对值。sema_init 宏用于初始化 count 为任意值，可以是二元信号量，也可以是一般信号量。在 DOWN 操作中，count 减 1 后的值若小于 0，进程便进入等待队列；在 UP 操作中，count 加 1 后的值若大于 0，立即唤醒等待队列中的所有进程。

(2) sleepers 对等待当前临界资源的进程个数进行辅助计数。

(3) wait 用于存放等待队列链表的地址，此链表包含当前正在等待信号量(资源)而被阻塞的所有进程。

内核信号量上所定义的函数有 DOWN、DOWN_interruptible、DOWN_trylock 和 UP，分别对应信号量的 P、V 操作。此外，还提供读者－写者信号量，相应的操作有 down_read 读者 down 操作、up_read 读者 up 操作、down_write 写者 down 操作及 up_write 写者 up 操作。

3. 等待队列

Block的方式

Linux 内核信号量采用非忙式等待来实现，当进程执行 DOWN 操作而等待时，将被放入等待信号量队列；事实上，并发进程同步时，只要等待条件不满足，就必须挂起，放入相应的等待队列。所以，等待队列是支持进程同步的重要数据结构，定义如下：

```
struct wait_queue {
    unsigned int compiler_warning;
    struct task_struct  * task;          //指向等待进程的 PCB
    struct list_head task_list;          //等待队列链表
};
```

等待队列也是临界资源，对其进行修改时应加锁，以避免竞争条件。内核为每个事件都设立一个等待队列，当进程等待一个指定事件发生时，必须调用 add_wait_queue()把自己加入此事件的等待队列，并通过 schedule()交出 CPU。被等待的事件发生后，内核调用 remove_wait_queue()将一个进程从等待队列中移出。

4. 关中断

一个正在对内核数据结构进行操作的进程可以被 I/O 设备中断，如果设备中断处理程序恰好也要访问这一数据结构，就会引起系统的不一致状态。解决这类问题的有效方法是关中断。关中断是把内核态执行的程序段作为一个临界区来保护的一种手段，主要保护中断处理程序也要访问的数据结构，如磁盘中断就需要被屏蔽。此外，关中断还能禁止内核抢占。为了防止死锁，内核在关中断期间不能执行阻塞操作。在对称式多处理器环境中，关中断只能防止来自本机其他中断处理程序的并发访问，需要引入自旋锁在禁止本地中断的同时，防止来自它机的并发访问。

5. 自旋锁

原子操作能够保证对变量操作的原子性，比如，原子加操作把读取和增加变量的操作包含在一个单步中执行，从而防止发生竞争。可是临界区却不像增加变量这样简单，有时它可以跨越多

关中断只影响当前执行下的CPU，不影响其他CPU。所以，如果要彻底去除并发，还应该加一个全局的spinlock，排除其他CPU的竞争。

自选锁通常用在需要快速的场景；
信号量用在等待时间不确定的场景。

个函数。例如，先从一个数据结构中读取数据，对其进行格式转换和解析，再把它写到另一个数据结构中。整个执行过程必须是原子的，在数据更新完毕前，不能有其他代码读取这些数据。显然，简单的原子操作对此无能为力，这就需要使用更为复杂的同步方法——锁来提供保护。

Linux 内核中最常见的锁是自旋锁，自旋锁最多只能被一个可执行线程持有。如果一个执行线程试图获得一个已被锁住的自旋锁，那么此线程就会一直进行忙式等待(旋转)，等待锁重新可用，期间这个 CPU 不能处理其他工作，同时等待其他 CPU 上运行的进程执行解锁操作。如果锁未被征用，请求锁的执行线程便可立刻锁住它，继续执行。在任意时刻，自旋锁都可以防止多于一个的执行线程同时进入临界区。

自旋锁是最简单的一种锁原语，锁的取值为 0 表示资源可用，锁的取值为 1 表示资源加锁。自旋锁很像二元信号量，但在具体实现上还是有区别的。若一个资源得到自旋锁的保护，另一个试图取得此资源的内核例程将保持忙式等待，直至资源被解锁。而在实现二元信号量时，不必循环等待信号量，而是进入等待队列，暂时放弃对资源的请求。

在 Linux 中，自旋锁变量 lock 被定义成 spinlock_t 类型，lock 成员只有最低位被使用，如果锁可用，lock 的最低位为 0；如果上锁，lock 的最低位为 1。lock 成员在初始化时被设为 0，即未上锁。自旋锁定义为：

```
typedef struct {
    volatile unsigned int lock;
}spinlock_t;
void spin_lock(spinlock_t *plock) {
    int flag;
    do{
        flag = plock->lock&1;          //取出 lock 的第 0 位，若为 1，表明已上锁
        plock->lock = 1;               //上锁，将 lock 的第 0 位设置为 1
    }while(flag! = 0);                 //若已上锁，则循环测试，直到成功
}
void spin_unlock(spinlock_t *plock) {
    plock->lock& = ~1;                 //开锁，将 lock 的第 0 位设置为 0
}
```

在对称式多处理器系统中，自旋锁最重要的特点是内核例程在等待锁被释放时一直占据着某个 CPU，用来保护那些只需简短访问数据结构的操作，如从双向队列链表中增加或删除一个元素。因此，在内核需要进程同步的位置，自旋锁使用得比较频繁。

自旋锁的上锁或解锁通过以下宏来实现：spin_lock(lock)/spin_unlock(lock)(获得/释放自旋锁)、spin_lock_irq(lock)/spin_unlock_irq(lock)(获得自旋锁并关闭中断/释放自旋锁并开放中断)、spin_lock_bh(lock)/spin_unlock_bh(lock)(获得自旋锁并关闭 bh 的执行/释放自旋锁并开放 bh 的执行)和 spin_lock_init(lock)(初始化自旋锁)等。

各种优化和组合，例如，考虑具备CPU的中断的情况

rwSpinlock。可以支持多个读者。通过一个counter来支持多个读者进入。

读者－写者自旋锁机制允许在内核中实现比基本自旋锁更大的并发度。每个读者－写者自旋锁包含一个24位的读者计数器和一个解锁标记，有三种状态：当计数器＝0、标记＝1时，自旋锁释放，可以使用；当计数器＝0、标记＝0时，自旋锁被一个写线程所获得；当计数器＞0、标记＝0时，自旋锁已被若干读线程所获得。

因为自旋锁在同一时刻至多被一个执行线程持有，所以一个时刻只能有一个线程位于临界区内，这就为多处理器系统提供防止并发访问所需的保护机制。在单处理器系统中，编译操作系统时并不会加入自旋锁，它仅仅被当做一个设置内核抢占机制是否被启用的开关。

3.7.2 System Ⅴ IPC 机制

1. Linux 的 System Ⅴ IPC 系统调用

在 UNIX System Ⅴ 系统中，把信号量、消息队列和共享主存资源统称为 IPC（InterProcess Communication，进程间通信）资源，用于进程之间的通信。Linux 提供给用户的 IPC 资源是通过系统调用来实现的，它们为应用进程提供三种服务：

（1）用信号量对进程所要访问的临界资源进行保护；

（2）用消息队列在进程之间以异步方式发送和接收消息；

（3）预留共享主存段以供进程之间交换和共享数据。

IPC 和管道之间的区别是，它允许同一机器上的许多进程互相通信，而不是仅限于两个进程，且管道有一个限制，两个通信进程必须是相关的，它们都有共同的祖先进程。System Ⅴ IPC 没有这方面的限制，只需要一个经过协商的协议。

每个 IPC 资源都有一个 32 位的 IPC 关键字（IPC key）和一个 32 位的 IPC 标识符（IPC identifier），IPC 标识符由内核分配给 IPC 资源，在系统内部是唯一的，而 IPC 关键字可由程序员自由选择。当进程间需要通过 IPC 资源进行通信时，就要引用它的 IPC 标识符。

IPC 数据结构是在进程请求 IPC 资源（指信号量、消息队列或共享主存）时动态地创建的，任何进程都可以使用 IPC 资源，即使是祖先进程创建 IPC 资源，但某些进程并不由这个祖先进程派生而来，这样的进程也可以使用 IPC 资源。每个 IPC 资源都有一个 ipc_perm 数据结构记录 IPC 关键字、创建者和所有者的 UID 和 GID，及每个创建者、创建者群组和其他用户的读写许可权限，还有 IPC 的关键字。

IPC 资源中信号量、消息队列和共享主存的创建分别通过系统调用 semget()、msgget()、shmget()函数来完成，这 3 个函数都以 IPC 关键字作为第一个参数，创建成功后返回 IPC 标识符，以后进程就可以引用这个 IPC 标识符对资源进行访问。用完 IPC 资源后必须使用 ipcrm 命令显式地释放 IPC 资源，否则 IPC 资源将永远驻留在主存中。

在创建 IPC 资源之后，可通过 xxxctl()函数对 IPC 资源进行控制，这些函数为用户提供命令来获得和设置资源的状态信息。当一个 IPC 资源被创建后，除了可对其进行控制和处理之外，还可通过专用函数对其进行操作，如获得和释放一个 IPC 信号量、发送和接收一个 IPC 消息以及将共享主存段附加到进程的地址空间或把共享主存段从进程的地址空间剥离。

用户态下的进程之间的通信机制。对于同一个进程内部的多线程通信，可以参阅PTHREAD编程模型。

2. Linux 的 System V IPC 实现概要

(1) IPC 信号量

内核所提供的信号量资源是一个指针数组，semid_ds * semary[SEMMNI]数组中的每一个元素都是 struct semid_ds 结构类型的指针，数组中的每一个元素描述一个 IPC 信号量资源，系统最多提供 128 个信号量资源。内核在 IPC 信号量机制中，sys_semop()函数的作用是增加或减少一个信号量集合中的信号量的值。当用户调用 semop()函数时，相应的内核函数将执行以下操作：

① 在访问临界资源之前，等待信号量集合的信号量可用。这个操作将导致信号量集合中相应的信号量的计数值减 1。

② 在访问临界资源之后，通知信号量集合释放相应的信号量。这个操作将导致信号量集合中相应的信号量的计数值加 1。

当进程进入临界区后，如果因为某种原因而无法退出，此时挂起在信号量等待队列上的等待进程将永远得不到机会运行，为此，Linux 维护一个信号量数组 struct sem_undo * undo 结构的调正链表来解决这个问题。

例如，发生了死锁现象。

(2) IPC 消息队列

应用程序可用 msgget()函数创建一个消息队列，然后用 msgsnd()将消息插入消息队列的末尾，用 msgrcv()将消息移出消息队列，用 msgctl()来释放或改变消息队列的许可权。IPC 资源的一条消息由两部分组成：一部分是消息的首部，由 struct msg 结构描述；另一部分是消息的正文，正文区的长度由相应的结构成员给出。内核中代表 IPC 消息的数据结构是 struct msgid_ds。

因为消息的发送是异步的，因此允许把一条消息写入消息队列旋即离开。但是，消息队列有最大容量的限制，当消息队列达到规定容量时，如果还有向此队列发送消息的进程则必须等待，直至队列中有消息被读取而腾出空间，或者在这种情况下发送消息的尝试被立刻拒绝。当消息队列中没有要读取的消息时，要么读消息的尝试被立即拒绝，要么读进程进入相应的等待队列。

(3) IPC 共享主存

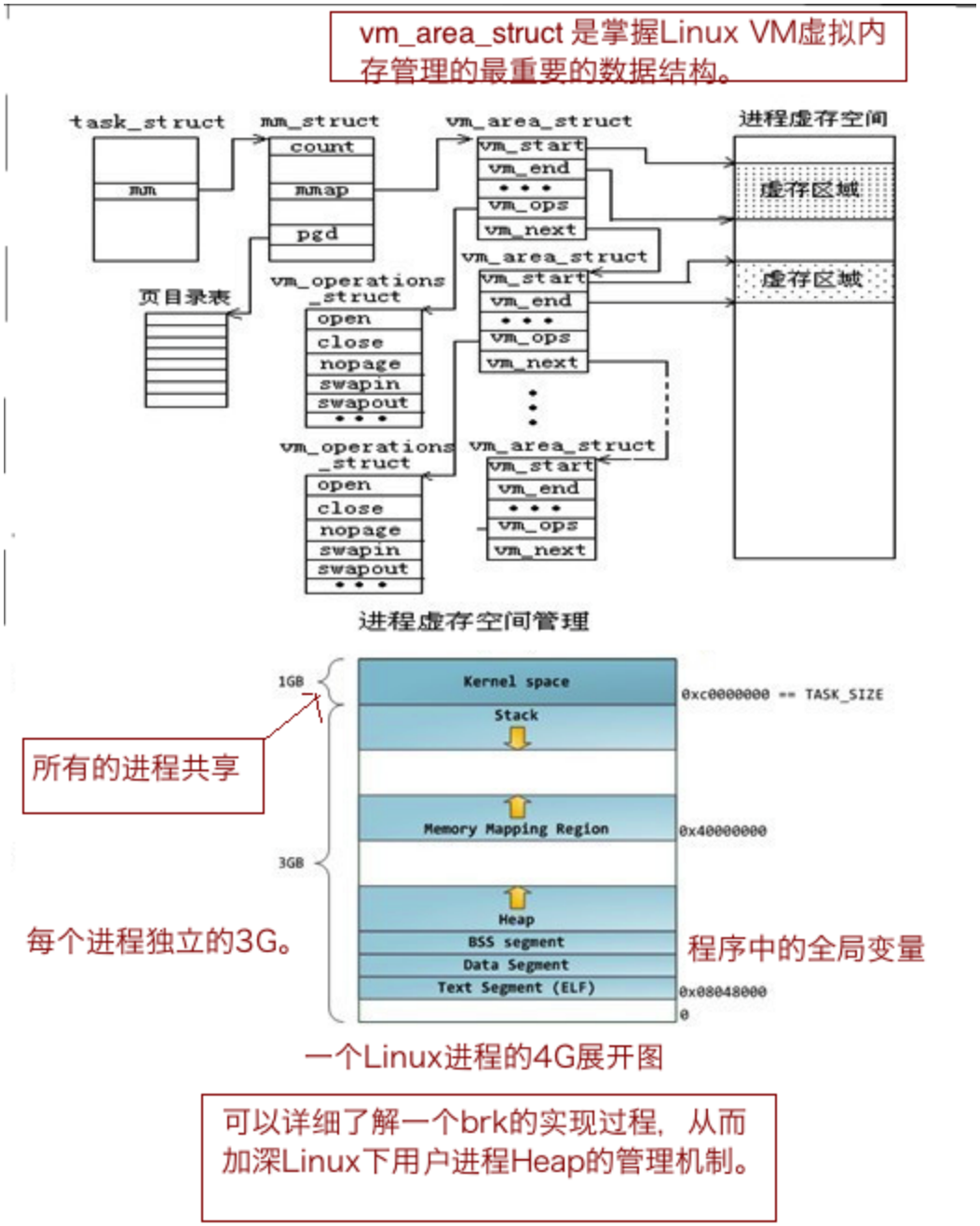
多个通信进程访问共享主存段时，都要将其映射到自己地址空间中的一个虚拟地址范围内，对于每个进程而言，这个地址段可以不同；在映射之后，进程访问这个地址段就和访问本地空间一样，无须使用读写系统调用。

① IPC 共享主存的数据结构

采用 IPC 共享主存进行进程间通信时，要发出系统调用 shmget()，由其调用内核函数获得一个共享主存段的 IPC 标识符 shmid。如果此共享主存段尚未存在，则要创建它。用来描述 IPC 共享主存段的数据结构是 struct shmid_kernel。

当进程第一次访问所创建的这块共享主存段之前，必须通过系统调用 shmat()申请连接，它要申请一个 vm_area_struct 结构专门用于映射共享主存段，同时重新填写结构成员 vm_ops 的操

每个进程都是自己的，对立的4G虚拟空间。其中3-4G是Linux的Kernel空间。被所有的进程所共享。



[注释]:1. Sem和Spinlock的用法区别在于：如果等待资源时间短和可预期，可以用自旋锁；否则用Sem，通过睡眠/唤醒来处理。Sem由于涉及队列操作，系统存在不确定的延时效率问题。2. 在多核多CPU情况下，关中断只能排除

当前CPU的并发，只有通过另外一个全局自旋锁的介入，
才能封住其他CPU的竞争。