第5章 进程间通信

在并发程序设计环境下，不影响其他进程也不被其他进程所影响的进程称为独立进程(Independent Process)，反之，影响其他进程或者被其他进程影响的进程则称为合作进程(Cooperating Process)。如果进程之间各自为政，老死不相往来，那么计算机世界将会完全不同。合作进程需要有一种进程之间的数据和信息交换方式。例如，在一个 shell 管道中，第一个进程的输出必须传送到第二个进程，这样沿着管道传递下去。类似这种数据和信息交换方式称为进程间通信(Inter-Process Communication，IPC)机制。

在 IPC 中，主要会涉及三类问题：第一，一个进程如何能够将信息传递给另一个进程。第二，如何确保两个或者多个进程彼此互不干扰。第三，当出现了一些依存性的时候，如何确保进程之间遵从合理的顺序执行。例如，如果进程 A 产生数据，进程 B 将这些数据打印出来，那么进程 B 必须等到进程 A 已经产生完数据之后才开始打印。

总体来说，可以将 IPC 大致划分为三类，即互斥、同步与消息传送。

第一类要保证两个或多个进程在涉及临界活动时不会彼此影响。这一类问题也称为互斥问题。

第二类涉及存在依赖关系时进行适当的排序。如果进程 A 产生数据，进程 B 打印数据，则 B 在开始打印之前必须等到 A 产生了一些数据为止。这一类问题也称为同步问题。

第三类是一个进程如何向另一个进程传送信息。这一类问题也称为消息传送。在消息传送中，又可以分为异步消息传送与同步消息传送。所谓“异步消息传送”指的是发送者将信息发送给接收者，但并不考虑接收者是否准备好接收。当发送者发送信息之后，就继续执行它的其他工作。如果接收者还没有准备好接收信息，那么便将所发送的信息放置于一个队列中供接收者以后来检索。发送者和接收者彼此异步运行，也对彼此的状态不做任何预设。所谓“同步消息传送”指的是发送者和接收者除信息交换之外，彼此还进行同步。发送者和接收者彼此同步，并彼此免于同步冲突。如果某个行为 a 需要在另一个行为 b 之后发生，那么行为 a 便会挂起，直到行为 b 完成之后。

5.1 临界区与互斥

系统中同时存在许多进程，它们共享各种资源，然而有许多资源在某一时刻只能允许一个进程使用。为了避免两个或多个进程同时访问打印机和磁带机等硬件设备以及变量和队列等数据结构这类资源，必须将它们保护起来。我们把某个时刻至多只能允许一个进程使用的资源称为临界资源(Critical Resource)。而访问临界资源的代码部分称为临

界区(Critical Region)或临界段(Critical Section)[①]。

互斥与进程并发紧密相关。正如第4章所介绍的,竞争条件是进程并发的一个典型问题,它也是互斥所要面对的典型问题之一。

【例5.1】 一个典型的竞争条件示例。

图5.1所示的程序是一个竞争条件的典型示例。程序创建了两个线程p和q,两个线程的操作是相同的,都是对一个共享变量cnt做加1操作,该操作循环1百万次。共享变量cnt的初值为0,如果正常运行,那么等待两个线程都运行结束之后,cnt的值应该为2百万。然而,运行之后,我们发现结果并非如此。

```
#include<pthread.h>
#include<semaphore.h>
#include<stdio.h>
#include<stdlib.h>
#define NITER 1000000
int cnt=0;
void * criticalSection()
{
    int tmp;
    tmp=cnt;     /*将全局变量 cnt 局部复制到 tmp 中*/
    tmp=tmp+1;   /*增加局部变量 tmp 值*/
    cnt=tmp;     /*将局部变量的值存储到全局变量 cnt 中*/
}
void * p(void * a)
{
    int i;
    for(i=0;i<NITER;i++)
        criticalSection();
}
void * q(void * a)
{
    int i;
    for(i=0;i<NITER;i++)
        criticalSection();
}
int main(int argc, char * argv[])
{
  pthread_t tid1, tid2;
  if(pthread_create(&tid1, NULL, p, NULL)){
    printf("\n ERROR creating thread 1");
    exit(1);
  }
  if(pthread_create(&tid2, NULL, q, NULL)){
    printf("\n ERROR creating thread 2");
    exit(1);
  }
```

图5.1 一个具有竞争条件的程序示例

① 临界段的概念最早由艾兹赫尔·韦伯·戴克斯特拉(Edsger Wybe Dijkstra)在《合作顺序进程》一文中提出。此后,霍尔(Hoare)将其重命名为临界区。

```
  if(pthread_join(tid1, NULL){   /* 等待第一个线程结束 */
    printf("\n ERROR joining thread");
    exit(1);
  }
  if(pthread_join(tid2, NULL)){   /* 等待第二个线程结束 */
    printf("\n ERROR joining thread");
    exit(1);
  }
  if (cnt<2 * NITER)
    printf("\n BOOM! cnt is [%d], should be %d\n", cnt, 2 * NITER);
  else
    printf("\n OK! cnt is [%d]\n", cnt);
  pthread_exit(NULL);
  return 0;
}
```

图 5.1　（续）

程序的运行结果之所以与我们所预想的完全不同，原因在于程序出现了竞争条件，可能出现的竞争条件如图 5.2 所示。

cnt 的值	线程 1	状态	线程 2	状态
50		运行		就绪
50	tmp＝cnt；	运行		就绪
50	tmp＝tmp＋1；	运行		就绪
50	中断；切换	就绪		就绪
50		就绪	tmp＝cnt；	运行
50		就绪	tmp＝tmp＋1；	运行
50		就绪	中断；切换	就绪
51	cnt＝tmp；	运行		就绪
51	中断；切换	就绪		就绪
51		就绪	cnt＝tmp；	

图 5.2　线程的跟踪：共享 cnt 产生的问题

例 5.1 中的共享变量 cnt 就是一种典型的临界资源，而使 cnt 变量进行加 1 操作的代码是典型的临界区，如图 5.1 中的 criticalSection()函数所示。

```
tmp=cnt;          /* 将全局变量 cnt 局部复制到 tmp 中 */
tmp=tmp+1;        /* 增加局部变量 tmp 值 */
cnt=tmp;          /* 将局部变量的值存储到全局变量 cnt 中 */
```

当涉及临界区访问时，合作进程需要遵从以下三个准则：[①]

① Edsger W. Dijkstra. 2002. *Cooperating sequential processes*. In the origin of concurrent programming, Per Brinch Hansen (Ed.). Springer-Verlag New York, Inc., New York, NY, USA 65-138.

(1) 互斥准则：在任意时刻，至多有一个进程在临界区中。

(2) 公平性准则：究竟哪个进程先进入临界区的决策不能够被无限期地推迟。

(3) 速度独立性准则：不同的进程各自的运行速度是彼此独立的，不能对进程的速度做任何预设。

基于合作进程的准则，临界区问题的解决方案必须满足如下三个需求：

- (RCS1)互斥(Mutual Exclusion)：如果一个进程在临界区中运行，那么其他进程就不能够进入临界区。
- (RCS2)前进(Progress)：如果没有进程在临界区中运行，而有一些进程希望能够进入临界区，那么只有那些未在非临界区运行的进程才能参与究竟哪个进程能够进入临界区的决策，同时，该选择不能够被无限期拖延。
- (RCS3)有界等待(Bounded Waiting)：在一个进程发出进入临界区的请求之后并在该请求获得授权之前，允许其他进程进入临界区的次数是有界的，或者说是有限的。

由于对临界资源的使用必须互斥进行，所以进程在进入临界区时，首先判断是否有其他进程在使用该临界资源。如果有，则该进程必须等待；如果没有，该进程才能进入临界区，执行临界区代码。同时，关闭临界区，以防其他进程进入。当进程用完临界资源时，要开放临界区，以便其他进程进入。基于临界资源和临界区的概念，进程间互斥可以描述为禁止两个或两个以上的进程同时进入并访问同一临界资源的临界区。

显然，要避免竞争条件，满足临界区问题各种需求，可以采用进程间互斥方案。一个好的进程间互斥方案应该满足如下 4 个条件(Principle of Mutual Exclusion)：

- (PME1) 任何两个进程不能同时处于临界区。
- (PME2) 不应对 CPU 的速度和数目做任何假设。
- (PME3) 临界区外的进程不得阻塞申请进入临界区的进程。
- (PME4) 不得使进程在临界区外无休止地等待。

进程间互斥方案有硬件方面的，也有软件方面的，下面列举一些主要的互斥方案。

5.1.1 禁用中断

为保证多个并发进程互斥使用临界资源，只需保证一个进程在执行临界区代码时不被中断即可，这个能力可以通过系统内核为启用和禁用中断定义的原语提供。进程可以通过下面的方法实现互斥，其示意如图 5.3 所示。

```
while(TRUE)
{
    Disable_Interrupt();          //禁用中断
    critical_region();            //临界区
    Enable Interrupt();           //启用中断
    noncritical_region();         //非临界区
}
```

图 5.3 禁用中断的互斥方案

由于进程在临界区内不能被中断，故可保证互斥。直觉上，禁用中断的互斥方案过于“强硬”，如同孙悟空念了一个“定”的咒语一般，其他进程乃至操作系统都被“定”住了，该方法代价太高，禁用中断即意味着禁用操作系统的进程调度功能，从而废除了进程的并发，降低了系统的效率。此外，禁用中断指令是特权级最高的指令，原则上一般应用进程是没有权限调用该指令的。

5.1.2　锁变量

令系统有一个单独的、共享的变量，称为锁变量，其初始值为0。当一个进程想要进入临界区时，它需要首先测试锁。如果锁变量的值为0，那么进程将它设置为1，并进入临界区。如果锁变量的值已经为1，那么进程等待，直到锁变量变为0。这样，锁变量为0表示在临界区中没有进程，锁变量为1表示在临界区中有某个进程。锁变量实现的示意代码如图5.4所示。

```
int lock=0;
if (lock==0){
    lock=1;
    critical_region();          //临界区
}
```

图5.4　锁变量的互斥方案

由于存在竞争条件，锁变量方案有先天的缺陷，并不能实现真正的互斥。假设进程P_1读取锁变量，并发现它为0。在它将锁变量置于1之前，进程切换，调度了进程P_2运行，并设置锁变量为1。当进程P_1再次运行时，它将锁变量再次设置为1，这样两个进程同时都进入临界区。进程的调度序列如图5.5所示。

进程 P_1		进程 P_2	锁变量
if (lock==0) //条件成立			0
	→	if (lock==0)//条件成立	0
		lock=1	1
		critical_region();	1
	←		1
lock=1			1
critical_region();			1

图5.5　锁变量互斥方案的进程调度序列示意

实际上，锁变量的互斥方案无异于“抱薪救焚”。互斥方案旨在解决因临界资源所引发的竞争条件问题，而锁变量自身又引发了新的竞争条件。

5.1.3　严格轮转法

要解决多进程的互斥问题是有一定难度的，现在我们将问题进行限制，只考虑两个进程的互斥问题。令系统有一个共享变量turn，用于标识与跟踪轮到哪个进程进入临界

区。假设需要互斥的进程为进程 P_1 和进程 P_2，其采用严格轮转法的互斥方案示意如图 5.6 所示。

进程 P_1	进程 P_2

while(TRUE)
{
//判定 turn 是否为 0,否则一直等待
while(turn !=0);
critical_region(); //临界区
turn=1;
noncritical_region1(); //非临界区
}
	while(TRUE)
	{
	//判定 turn 是否为 1,否则一直等待
	while(turn !=1);
	critical_region();
	//临界区
	turn=0;
	noncritical_region2(); //非临界区
	}

图 5.6 严格轮转法的互斥方案

假设 turn 初值为 0，一开始进程 P_1 检查 turn，发现它是 0，于是进入临界区。进程 P_2 同样也发现它是 0，于是执行一个等待循环，不停地检测它是否变成了 1。当进程 P_1 离开临界区时，它将 turn 置为 1，以允许进程 P_2 进入其临界区。然后进程 P_2 进入临界区，执行完毕后，进程 P_2 重新将 turn 置为 0，从而再次允许进程 P_1 进入临界区，如此反复。可以看出，如图 5.6 所示的互斥方案是一种严格的轮转，即当 turn 值为 0 时表示轮到进程 P_1 可以进入临界区，而当 turn 值为 1 时表示轮到进程 P_2 可以进入临界区。而将 turn 置 0 的权限在于进程 P_2，将 turn 置 1 的权限在于进程 1，这样进程 P_1 和进程 P_2 之间就严格地按照交替顺序进入临界区，从而实现互斥。

然而，严格轮转法的互斥方案存在如下不足：

(1) 进程的忙等待造成 CPU 资源的浪费。

当进程判断 turn 变量的值时，如果符合进入临界区的条件则退出循环，否则就一直循环，反复检测 turn 变量的值，我们将这类行为称为忙等待(Busy Waiting)。忙等待其实就是在空转，没有执行任何有价值的计算，因此浪费 CPU 时间。

(2) 当轮转的两个进程执行周期时间相差较大时，执行较快的进程将长时间等待。

在图 5.6 所示的例子中，假设进程 P_1 的非临界区执行非常快，而进程 P_2 的非临界区执行非常慢。存在某个执行序列如图 5.7 所示。当进程 P_1 执行完临界区之后，将 turn 的值置为 1，这样进程 P_2 就可以很快进入临界区执行，在离开临界区后，turn 的值被置为 0。然后进程 P_1 很快便执行完了其整个循环，它再次执行到非临界区的部分，并将 turn 置为 1。此时，进程 1 结束了其非临界区的操作并回到循环的开始，但此时由于 turn 的值仍为 1，因此它不能进入临界区，只能忙等待，而进程 P_2 还在忙于非临界区的操作。此时，进程 P_1 就必须一直等到进程 P_2 的非临界区执行完毕，并再进入一次临界区，才能将 turn 变量重新置为 0。

进程 P_1		进程 P_2	turn
while(TRUE)			0
while(turn !=0);			0
critical_region()			0
turn=1			1
	→	while(TRUE)	1
		while(turn !=1);	1
		critical_region()	1
		turn=0	0
noncritical_region1()	←		0
while(TRUE)			0
while(turn !=0);			0
critical_region()			0
turn=1			1
	→	noncritical_region2()	1
noncritical_region1()	←		1
while(TRUE)			1
while(turn !=0);			1
	→	noncritical_region2()	1
while(turn !=0);	←		1

图 5.7　严格轮转法互斥方案的示例执行序列

上述情形还违反了一个好的互斥方案(PME3)条件,即临界区外的进程 P_2 阻塞申请进入临界区的进程 P_1。

5.1.4　Dekker 算法

Dekker 算法是第一个真正解决两个并发进程互斥问题的解决方案。戴克斯特拉将该算法归功于荷兰数学家德克尔(T. J. Dekker)。

严格轮转法最大的问题在于在临界区外的进程会阻止申请进入临界区的进程。为了克服这个问题,Dekker 算法增加了一个变量来表达想进入临界区的意愿。当进程从临界区出来后,除了将轮转变量翻转之外,还将自己的意愿变量置为 FALSE。Dekker 算法的伪代码如图 5.8 所示。

```
变量定义:
    wants_to_enter: array of 2 booleans     //布尔型数组,代表进程进入临界区的意愿
    turn: integer                           //整数
    wants_to_enter[0]←FALSE
    wants_to_enter[1]←FALSE
    turn←0                                  //或者 1,代表轮到哪个进程
```

图 5.8　Dekker 算法的伪代码

```
进程 P1:
wants_to_enter[0]←TRUE
while wants_to_enter[1] {
    if turn≠0 {
        wants_to_enter[0]←FALSE
        while turn≠0 {
            //忙等待
        }
        wants_to_enter[0]←TRUE
    }
}
critical_region();//临界区
turn←1
wants_to_enter[0]←false
noncritical_region1();//非临界区
```

```
进程 P2:
wants_to_enter[1]←TRUE
while wants_to_enter[0] {
    if turn≠1 {
        wants_to_enter[1]←FALSE
        while turn≠1 {
            //忙等待
        }
        wants_to_enter[1]←TRUE
    }
}
critical_region();//临界区
turn←0
wants_to_enter[1]←false
noncritical_region2();//非临界区
```

图 5.8 （续）

在图 5.8 中，描述了两个进程的互斥方案。我们采用一个布尔数组 wants_to_enter 以及 1 个轮转变量 turn，其中前者用于来标识进程进入临界区的意愿。在进程 1 和进程 2 进入临界区之前，首先将自己意愿变量的值置为 TRUE，然后进入判定循环。判定的逻辑是首先判断一下是否对方进程希望进入临界区，如果是，便进入判定循环。在判定循环体中，首先进行条件判定，判定轮转变量是否指示轮到对方进程进入，如果是，则将自身的意愿进行调整，将意愿变量置为 FALSE，然后进入一个忙等待中直到轮转变量翻转位置。否则再次进入判定循环。

【例 5.2】 用 Dekker 算法解决例 5.1 的竞争条件问题。

例 5.1 中的程序中的竞争条件主要是因为没有对临界区访问进行互斥，我们采用 Dekker 算法实现临界区的互斥。具体代码实现如图 5.9 所示。其中函数 p 和函数 q 分别为两个线程执行体，wantp 与 wantq 代表意愿，彼此通过 Dekker 算法实现互斥。

```
#include<pthread.h>
#include<stdio.h>
#include<stdlib.h>
#define NITER 1000000
#define false 0
#define true 1
int cnt=0;
typedef int bool; //or #define bool int
pthread_t tid[2];
bool wantp=false;
bool wantq=false;
int turn=1;
void * criticalSection()
{
    int tmp;
    tmp=cnt;                /* copy the global cnt locally */
    tmp=tmp+1;              /* increment the local copy */
```

图 5.9 Dekker 算法的 C 语言实现

```
    cnt=tmp;            /* store the local value into the global cnt */
}
void * p(void * a)
{
    int i;
    for(i=0;i<NITER;i++){
        wantp=true;
        while(wantq) {
            if(turn==2) {
                wantp=false;
                pthread_yield();
                while(turn !=1) {
                }
                wantp=true;
            }
        }
        criticalSection();
        turn=2;
        wantp=false;
    }
}
void * q(void * a)
{
    int i;
    for(i=0;i<NITER;i++){
        wantq=true;
        while(wantp) {
            if(turn==1) {
                wantq=false;
                pthread_yield();
                while(turn !=2) {
                }
                wantq=true;
            }
        }
        criticalSection();
        turn=1;
        wantq=false;
    }
}
int main(int argc, char * argv[])
{
    pthread_t tid1, tid2;
    if(pthread_create(&tid1, NULL, p, NULL)){
        printf("\n ERROR creating thread 1");
        exit(1);
    }
    if(pthread_create(&tid2, NULL, q, NULL)){
        printf("\n ERROR creating thread 2");
        exit(1);
    }
```

图 5.9 （续）

```
    if(pthread_join(tid1, NULL)){     /* wait for the thread 1 to finish */
        printf("\n ERROR joining thread");
        exit(1);
    }
    if(pthread_join(tid2, NULL)){     /* wait for the thread 2 to finish */
        printf("\n ERROR joining thread");
        exit(1);
    }
    if (cnt<2 * NITER)
        printf("\n BOOM! cnt is [%d], should be %d\n", cnt, 2*NITER);
    else
        printf("\n OK! cnt is [%d]\n", cnt);
    pthread_exit(NULL);
    return 0;
}
```

图 5.9 （续）

5.1.5 Peterson 算法

1981 年，加里·彼得森(Gary L. Peterson)发现了实现两个进程间互斥的更简便方法[①]。与 Dekker 算法类似，Peterson 算法使用一个变量来表达想进入临界区的意愿，使用变量 turn 来代表轮到哪个进程进入临界区。然而，Peterson 算法与 Dekker 算法不同的是，首先 turn 变量并不是在离开临界区时翻转的，而是在每次希望进入临界区之前，进程将 turn 变量置给对方(从某种意义上而言，Peterson 算法是一种更“绅士”的算法)。Peterson 算法的伪代码如图 5.10 所示。

变量定义： wants_to_enter : array of 2 booleans　　//布尔型数组，代表进程进入临界区的意愿 turn : integer　　//整数	

进程 P_1：
wants_to_enter[0] ←TRUE
turn ← 1;
while (wants_to_enter[1]=TRUE∧turn=1){
　　//忙等待
}
critical_region();　　//临界区
wants_to_enter [0] ←FALSE;
noncritical_region1();　//非临界区
	进程 P_2：
	wants_to_enter[1] ←TRUE
	turn ← 0;
	while (wants_to_enter [0]= TRUE∧turn =0){
	　　//忙等待
	}
	critical_region();　　//临界区
	wants_to_enter [1] ←FALSE;
	noncritical_region2();　//非临界区

图 5.10　针对两个进程互斥的 Peterson 算法的伪代码

① G. L. Peterson. *Myths About the Mutual Exclusion Problem*. Information Processing Letters 12(3) 1981, 115-116.