3.1 临界区问题

异步并发进程或线程之间或者是彼此独立不相关的，或者彼此之间有相互影响。这些影响可以是数据的共享、信息的传递、控制流之间的相互影响，等等。当并发进程之间相互存在关系时，它们的并发运行就需要处理一些特别的情况，这就是本章的主题。

我们先探讨最常见的若干协作进程或线程，这些进程或线程均异步执行且共享数据。最典型的具有代表性的问题有“生产者一消费者问题”。我们前面已特地采用了有限缓冲方案来处理进程共享内存的问题。

现在回到有限缓冲问题的共享内存解决方案。正如先前所指出的，该解决方案允许同时在缓冲区内最多只有 BUFFER_SIZE-1 项。假如要修政这一算法以弥补这个缺陷。一种可能方案是增加一个整数变量 counter，并初始化为 0。每当向缓冲区增加一项时，counter 就递增；每当从缓冲区移走一项时，counter 就递减。

按照这个思路，生产者进程代码可修改如下：

```
while (true) {
  /* produce an item in nextProduced */
  while (counter == BUFFER—SIZE)
    ;  /* do nothing */
  buffer[in] = nextProduced;
  in =(in+l) % BUFFER_SIZE;
  counter++;
}
```

类似地，消费者进程代码可修改如下：

```
while (true) {
  while (counter == 0)
   ; /* do nothing */
  nextConsumed = buffer[out]:
  out = (out + 1) % BUFFER_SIZE;
  counter--;
  /* consume the item in nextConsumed */
}
```

虽然生产者和消费者程序各自都是正确的，但是当并发执行时它们可能不能正确运行！

为了便于说明，假设变量 counter 的值现在为 5，且生产者进程和消费者进程正在并发执行语句“counter++”和“counter--”。显然，唯一正确的结果应是 counter == 5。但是，根据这两条语句的具体执行顺序，变量 counter 的值却可能是 4、5 或 6！为什么呢？因为，语句“counter++”在一台典型的机器上很可能是按如下方式以机器语言实现：

```
register1 = counter
register1 = register1+l
counter = register1
```

其中，register1 为 CPU 局部寄存器。类似地，语句“counter--”可按如下方式来实现：

```
register2 = counter
register2 = register2-1
```

```
counter = register2
```

其中，register2 为 CPU 局部寄存器。虽然 register1 和 register2 可以为同一寄存器（如累加器），但是要记住，中断处理程序会保存和恢复该寄存器的值。

当并发执行“counter++”和“counter--”时，相当于按任意顺序来交替执行上面所表示的机器指令级语句（每条高级语句内的顺序不能变）。假设在某一次交叉执行过程中，各指令的执行顺序如下：

步骤	执行的机器级指令	注释	结果注释
1	生产者执行 register1 = counter		register1 = 5
2	生产者执行 register1 = register1 + 1	随后进程切换	register1 = 6
3	消费者执行 register2 = counter		register2 = 5
4	消费者执行 register2 = register2 - 1		register2 = 4
5	消费者执行 counter = register2	随后进程切换	counter = 4
6	生产者执行 counter = register1		counter = 6

注意，现在得到了表示有 6 个缓冲区被占用的不正确的状态“counter==6”，而事实上有 5 个缓冲区被占用了。如果交换第 5 和第 6 两条语句，那么会得到不正确的状态“counter==4"。

之所以得到了不正确状态，是因为允许两个进程并发操作变量 counter。像这样的情况，即多个进程并发访问和操作同一数据且执行结果与访问发生的特定顺序有关，称为竞争条件(race condition)。为了避免竞争条件，需要确保一段时间内只有一个进程能操作共享变量 counter。为了实现这种保证，就要求进行一定形式的进程同步。

其实，这种情况经常出现在操作系统中。系统的不同部分往往并发操作共享的资源。显然，我们需要确保这些共享操作的正确性。正是由于同步机制的重要性，所以我们要认真分析和学习进程的同步（process synchronization）和协调（coordination）。

结论：如果生产者和消费者独立完整地执行 counter 的访问，那么会有正确的结果。如果对 counter 的访问被交织在一起，counter 的值就可能不正确。我们需要一定形式的进程同步，实现这种保证。

一般地，假设某个系统有 n 个进程{P_0，P_1，…，P_{n-1}}。每个进程有一段代码被称为临界区(critical section)，在该代码段中进程可能改变共享变量、更新一个表、写一个文件等。这种系统的重要特征是当一个进程进入临界区，没有其他进程可被允许在临界区内执行，即没有两个进程可同时在临界区内执行。这种进程之间的关系被称为互斥（mutual excluxion）。解决临界区问题(critical-section problem)就是设计一个进程之间协作的机制，以实现进程之间对各自临界区访问的互斥。