3.5 经典同步问题

在本节，将介绍几种不同的同步问题，以作为大量并发控制问题的典型例子。这些问题也一直被用来测试各种同步方案。在这里，我们将采用信号量作为同步问题的解决方案。

3.5.1 有限缓冲区问题（bounded-buffer problem，生产者消费者问题）

有限缓冲区问题先前已讨论过，它通常用来说明同步原语的能力。这里，介绍一种该方案的通用解决结构，而不是只局限于某个特定实现。

假定缓冲池有 n 个缓冲项，每个缓冲项能存放一个数据项。二进制信号量 mutex 提供了对缓冲池访问的互斥要求，并初始化为 1。计数信号量 empty 和 full 分别用来表示空缓冲项和满缓冲项的个数。信号量 empty 初始化为 n；而信号量 full 初始化为 0。

生产者进程的代码如图 3.14 所示；消费者进程的代码如图 3.15 所示。注意生产者和消费者之间的对称性。可以这样来理解代码：生产者为消费者生产满缓冲项，而消费者为生产者生产空缓冲项。

```
do {
  // produce an item in nextp
  wait(empty);
  wait(mutex);
  // add nextp to buffer
  signal(mutex);
  signal(full);
} while (TRUE);
```

图 3.14 生产者进程结构

```
do {
  wait(full);
  wait(mutex);
  // remove an item from buffer to nextc
  signal(mutex);
  signal(empty);
  // consume the item in nextc
} while (TRUE);
```

图 3.15 消费者进程结构

3.5.2 读者一写者问题（Readers-writers problem）

一个数据库正被多个并发进程所共享。其中，有的进程可能只需要读数据库，而其他进程可能需要更新（即读和写）数据库。为了区分这两种类型的进程，将前者称为读者，而将后者称为写者。显然，如果两个读者同时访问共享数据，那么不会产生什么不利的结果。然而，如果一个写者和其他进程（既不是读者也不是写者）同时访问共享对象，很可能引起混乱。

为了确保不会产生这样的混乱，要求写者对共享数据库有排他的访问。这一同步问题称为读者一写者问题。自从它被提出后，就一直用来测试几乎所有新的同步原语。

读者一写者问题有多个变种，都与优先级有关。最为简单的，通常也被称为第一类读者一写者问题，要求除非已有一个写者已获得允许以使用共享数据库，否则没有读者需要保持等待。换句话说，没有读者会因为有一个写者在等待而会等待其他读者的完成。第二类读者一写者问题则要求，

一旦写者就绪，那么写者会尽可能快地执行其写操作。换句话说，如果一个写者等待访问数据库，那么不会有新读者开始读操作。

对这两个问题的解答都可能导致饥饿问题。对第一种情况，写者可能饥饿；对第二种情况，读者可能饥饿。由于这个原因，已经提出了问题的其他变种。

这里介绍一个对第一类读者一写者问题的解答。关于读者一写者问题的没有饥饿的解答，请参见推荐的有关文献。

对于第一读者一写者问题的解决，读者进程共享以下数据结构：

```
semaphore mutex, wrt;
int readcount;
```

信号量 mutex 和 wrt 初始化为 l；readcount 初始化为 0。信号量 wrt 为读者和写者进程所共用。信号量 mutex 用于确保在更新变量 readcount 时的互斥。变量 readcount 用来跟踪有多少进程正在读对象。信号量 wrt 供写者作为互斥信号量。它为第一个进入临界区和最后一个离开临界区的读者所使用，而不被其他读者所使用。

写者进程的代码如图 3.16 所示，读者进程的代码如图 3.17 所示。注意，如果有一个写者进程在临界区内，且 n 个读者进程处于等待，那么一个读者在 wrt 上等待，而 n-l 个读者在 mutex 上等待。而且，当一个写者执行 signal(wrt)时，可以重新启动等待读者或写者的执行。这一选择由调度程序所做。

```
do {
  wait(wrt);
  // writing is performed
  signal(wrt);
} while (TRUE);
```

图 3.16 写者进程的结构

```
do{
  wait(mutex);
  readcount++;
  if (readcount==1)
    wait(wrt);
  signal(mutex);
  // reading is performed
  wait(mutex);
  readcount--;
  if (readcount==0)
    signal(wrt);
  signal(mutex);
} while (TRUE);
```

图 3.17 读者进程的结构

读者一写者问题及其解答可以进行推广，用来对某些系统提供读写锁。在获取读写锁时，需要指定锁的模式：读访问或写访问。当一个进程只希望读共享数据时，可申请读模式的读写锁；当一个进程希望修改数据时，则必须申请写模式的读写锁。多个进程可允许并发获取读模式的读写锁；而只有一个进程可为写操作而获取读写锁。

读写锁在以下情况下最为有用：

·当可以区分哪些进程只需要读共享数据而哪些进程需要写共享数据；

·当读者进程数比写进程多时。这是因为读写锁的建立开销通常比信号量或互斥锁要大，而这一开销可以通过允许多个读者来增加并发度的方法进行弥补。

3.5.3 哲学家进餐问题（Dining-philosophers problem）

假设有 5 个哲学家，他们用一生来思考和吃饭。这些哲学家共用一个圆桌，每位都有一把椅子。在桌子中央是一碗米饭，在桌子上放着 5 只筷子（见图 3.18）。当某位哲学家思考时，他独自静静地思考，不和其他哲学家交互。时而，他会感到饥饿，并试图拿起与他相邻的两只筷子（筷子在他和他的左、右邻座之间）。一个哲学家一次只能拿起一只筷子。当一个饥饿的哲学家同时拿到两只筷子时，他就能吃了。吃完后，他会放下两只筷子，并重新进入思考。显然，一位哲学家不能从其他正在吃饭的哲学家手里抢走筷子。

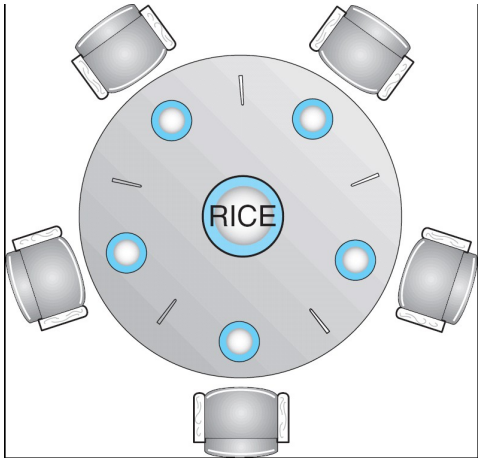

图 3.18 哲学家进餐问题的场景

哲学家进餐问题是一个典型的同步问题，这不是因为这一假想的问题本身有什么实际重要性，而是因为它是一个并发控制问题的典型例子。它需要在多个进程之间分配多个资源且不会出现死锁和饥饿的解决方案。

一种简单的解决方法是每只筷子都用一个信号量来表示。一个哲学家通过执行 wait()操作试图获取相应的筷子，他会通过执行 signal()操作以释放相应的筷子。因此，5 位哲学家的共享数据是：

```
semaphore chopstick[5];
```

其中所有 chopstick 的元素初始化为 l。哲学家 i 的进程结构如图 3.19 所示。

```
do {
  wait(chopstick[i]);
  wait(chopstick[(i+l)%5]);
  // eat
  signal(chopstick[i]);
  signal(chopstick[(i+l)%5]);
  // think
} while (TRUE);
```

图 3.19 哲学家 i 的进程结构

虽然这一解答确保没有两个哲学家同时使用同一只筷子（即确保了互斥要求），但是这一解决仍

然不能令人满意，因为它可能会导致死锁。例如，当这 5 个哲学家同时感到饥饿，且同时拿起左边的筷子时，所有筷子的信号量均变为 0；然后，当每个哲学家试图拿右边的筷子时，他就会永远等待。这样，所有的哲学家最终都会饿死。

下面列出了几个可能解决死锁问题的方法：

·最多只允许 4 个哲学家同时坐到桌子上。

·只有两只筷子都可用时才允许一个哲学家拿起它们（他必须在临界区内拿起两只筷子）。

·使用非对称解决方法，即奇数哲学家先拿起左边的筷子，接着拿起右边的筷子，而偶数哲学家先拿起右边的筷子，接着拿起左边的筷子。

最后，有关哲学家进餐问题的任何满意的解决必须确保没有一个哲学家会饿死。因为，即便找到了一个没有死锁的解决方案，并不能保证该方案消除了饿死的可能。