## 3.7 信号量

在前面几节我们看到，需要在几个进程之间很好地解决互斥问题，需要精巧设计的算法。即便在有诸如 TestAndSet 这样的硬件指令支持下，算法也需要良好地考虑。对于应用程序员而言，这些算法的使用比较复杂。为了解决这个困难，可以使用称为信号量（semaphore）的同步工具。

信号量 S 是个整数变量，除了初始化外，它只能通过两个标准原子操作：wait ( ) 和 signal ( ) 来访问。这些操作原来被称为 P（荷兰语 proberen，测试）和 V（荷兰语 verhogen，增加）。wait ( ) 的定义可表示为

```
wait (S)  {
  while(S<=0)
    ; //no-op
  S--;
}
```

signal 的定义可表示为

```
signal (S)  {
  S++;
}
```

在 wait()和 signal()操作中，对信号量整型值的修改必须不可分地执行，即当一个进程修改信号量值时，不能有其他进程同时修改同一信号量的值。另外，对于 wait(S)，对 S 的整型值的测试(S<=0)和对其可能的修改（S--），也必须不被中断地执行。后面我们将描述如何实现这些操作。现在先研究如何使用信号量。

### 3.7.1 信号量的用法

操作系统通常将信号量区分为计数信号量与二进制信号量。计数信号量的值域不受限制（实际上是一定范围内的自然数），而二进制信号量的值只能为 0 或 1。有的系统，将二进制信号量称为互斥锁，因为它们可以提供互斥。

可以使用二进制信号量来处理多进程的临界区问题。设 n 个进程共享一个信号量 mutex，并初始化为 1。每个进程的结构如图 3.13 所示。

```
do {
  wait(mutex);
    // critical section
  signal(mutex);
    // remainder section
}
```

图 3.13 使用信号量的互斥实现

而计数信号量则可以用来控制访问具有若干个实例的某种资源。该信号量初始化为可用资源的数量。当每个进程需要使用资源时，需要对该信号量执行 wait()操作（减少信号量的计数）。当进程释放资源时，需要对该信号量执行 signal()操作（增加信号量的计数）。当信号量的计数为 0 时，所有资源都被使用。之后，需要使用该资源的进程将会阻塞，直到其计数大于 0。

还可以使用信号量来解决其他各种同步问题。例如，有两个并发进程：Pl 有语句 Sl 而 P2 有语句 S2，假设要求只有在 Sl 执行完之后才执行 S2。使用信号量可以很容易地实现这一要求：让 Pl 和 P2 共享一个共同信号量 synch，且将其初始化为 0，然后在进程 Pl 中插入语句：

```
Sl;
signal (synch);
```
进程 P2 中插入语句:
```
wait (synch);
S2;
```
因为 synch 初始化为 0，P2 只有在 Pl 调用 signal(synch)（即 Sl）之后，才会执行 S2。

所以，我们看到，通过恰当地使用信号量并给它们设置合理的初值，可以解决进程之间的包括互斥在内的各类同步问题。

3.7.2 信号量的实现方式

前一节所定义的信号量，主要是帮助理解。在 wait 操作中，我们使用了 while 语句。这样做的主要缺点其实是要求忙等待( busy waiting)。当一个进程位于其临界区内时，任何其他试图进入其临界区的进程都必须在其进入代码中连续地循环。这种连续循环在实际多道程序系统中显然是个问题，因为这里只有一个处理器为多个进程所共享。忙等待浪费了 CPU 时钟，这本来可有效地为其他进程所使用。

这种类型的信号量也被称为自旋锁(spinlock)，这是因为进程在其等待锁时还在运行。在多处理器系统中，自旋锁有其优点，进程在等待锁时不进行上下文切换，而上下文切换可能需要花费相当长的时间。因此，如果锁的占用时间短．那么自旋锁就有用了；这样一个线程在一个处理器自旋时，另一线程可在另一处理器上在其临界区内执行。但在单处理器系统中，忙等待就是一种对 CPU 资源的浪费。

为了克服忙等待，可以修改信号量操作 wait()和 signl()的定义。当一个进程执行 wait()操作时，发现信号量值不为正，则它必须等待。然而，该进程不是忙等而是阻塞自己。阻塞操作将一个进程放入到与信号量相关的等待队列中，并将该进程的状态切换成等待状态。接着，控制转到 CPU 调度程序，以选择另一个进程来执行。

一个被阻塞在等待信号量 S 上的进程，可以在其他进程执行 signal()操作之后被重新执行。该进程的重新执行是通过 wakeup()操作来进行的，该操作将进程从等待状态切换到就绪状态。接着，该进程被放入到就绪队列中（根据 CPU 调度算法的不同，CPU 有可能会、也可能不会从正在运行的进程切换到刚刚就绪的进程）。

为了实现此种定义的信号量，将信号量定义为如下一个 C 语言结构:
```
typedef struct {
  int value;
  struct process *list;
} semaphore;
```
每个信号量都有一个整型值和一个等待进程链表。当一个进程必须等待信号量时，就加入到等待进程链表上。操作 signal()会从等待进程链表中取下一个进程以唤醒。

信号量操作 wait（）现在可按如下来定义
```
wait(semaphore *S)  {
  S->value--;
  if  (S->value < O)  {
    add this process to S->list;
    block();
   }
}
```

信号量操作 signal（）现在可按如下来定义

```
signal (semaphore *S) {
  *S->value++;
  if  (S->value <=0  )  {
    remove a process P from S->list;
    wakeup(P);
  )
}
```

操作 block()挂起调用它的进程。操作 wakeup(P)重新启动阻塞进程 P 的执行。这两个操作都是由操作系统作为基本系统调用来提供的。

注意，在具有忙等待的信号量的经典定义下，信号量的值不可能为负，但是本节新定义的实现方式可以产生负的信号量值。如果信号量的值为负，那么其绝对值就是等待该信号量的进程的个数。出现这种情况是因为 wait()操作实现中递减和测试次序的互换。

等待进程的链表可以利用进程控制块 PCB 中的一个链接域来加以轻松实现。每个信号量包括一个整型值和一个 PCB 链表的指针。确保“有限等待”要求被满足的一种方法是向链表中增加和删除进程时使用 FIFO 队列。然而，一般来说，链表可以使用任何排队策略。信号量的正确使用并不依赖于信号量链表的特定排队机制。

信号量的关键之处是它们原子地执行。必须确保没有两个进程能同时对同一信号量执行操作 wait()和 signal()。这属于临界区问题，可通过两种方法来解决。在单处理器环境下（即只有一个 CPU 存在时），可以在执行 wait()和 signal()操作时简单地禁止中断。这种方案在单处理器环境下能工作，这是因为一旦禁止中断，不同进程指令不会交织在一起。只有当前运行进程执行，直到中断重新允许和调度器能重新获得控制为止。

在多处理器环境下，必须禁止每个处理器的中断；否则，运行在不同处理器上的不同进程可能会以任意不同方式交织在一起执行。但是，禁止每个处理器的中断不仅会很困难，而且还会严重影响性能。因为，多处理器系统必须提供其他加锁技术（如自旋锁），以确保 wait()与 signal()可原子地执行。

### 3.7.3 死锁与饥饿

具有等待队列的信号量的实现可能导致这样的情况：两个或多个进程无限地等待一个事件，而该事件只能由这些等待进程之一来产生。这里的事件是 signal()操作的执行。当出现这样的状态时，这些进程就称为进入了死锁( deadlocked)。

下面是死锁的一个简单例子。考虑一个由两个进程 P0 和 Pl 组成的系统，每个都访问共享信号量 S 和 Q，这两个信号量的初值均为 1：

```
 P0              P1
wait (S);    wait (Q);
wait (Q);    wait (S);
...              ...
signal (S);  signal (Q);
signal (Q);  signal (S);
```

假设 P0 执行 wait(S)，接着 Pl 执行 wait(Q)。当 P0 执行 wait(Q)时，它必须等待，直到 Pl 执行 signal(Q).类似地，当 Pl 执行 wait(S)，它必须等待，直到 P0 执行 signal(S)。由于这两个操作都不能执行，那么 P0 和 Pl 就死锁了。

说一组进程处于死锁状态，即组内的每个进程都等待一个事件，而该事件只可能由组内的另一个进程产生。这里主要关心的事件是资源获取和释放（resource acquisition and release）。然而，如下一章所述，其他类型的事件也能导致死锁。在下一章，将讨论各种机制以处理死锁问题。

与死锁相关的另一个问题是无限期阻塞(indefinite blocking)，也被称为饥饿(starvation)，即进程在信号量内无限期等待。例如，如果对与信号量相关的链表按 LIFO 顺序来增加和移动进程，那么可能会发生无限期阻塞；或者对该链表进行优先队列式调度，那么低优先级的进程可能会遭遇无限期阻塞。