缓冲区；消费者从缓冲区取走数据项，以某种方式消费。

很多实际的系统中都会有这种场景。例如，在多线程的网络服务器中，一个生产者将 HTTP 请求放入工作队列（即有界缓冲区），消费线程从队列中取走请求并处理。

我们在使用管道连接不同程序的输出和输入时，也会使用有界缓冲区，例如 grep foo file.txt | wc -l。这个例子并发执行了两个进程，grep 进程从 file.txt 中查找包括“foo”的行，写到标准输出；UNIX shell 把输出重定向到管道（通过 pipe 系统调用创建）。管道的另一端是 wc 进程的标准输入，wc 统计完行数后打印出结果。因此，grep 进程是生产者，wc 是进程是消费者，它们之间是内核中的有界缓冲区，而你在这个例子里只是一个开心的用户。

因为有界缓冲区是共享资源，所以我们必须通过同步机制来访问它，以免[①]产生竞态条件。为了更好地理解这个问题，我们来看一些实际的代码。

首先需要一个共享缓冲区，让生产者放入数据，消费者取出数据。简单起见，我们就拿一个整数来做缓冲区（你当然可以想到用一个指向数据结构的指针来代替），两个内部函数将值放入缓冲区，从缓冲区取值。图 30.4 为相关代码。

```
1    int buffer;
2    int count = 0; // initially, empty
3
4    void put(int value) {
5        assert(count == 0);
6        count = 1;
7        buffer = value;
8    }
9
10   int get() {
11       assert(count == 1);
12       count = 0;
13       return buffer;
14   }
```

图 30.4 put 和 get 函数（第 1 版）

很简单，不是吗？put()函数会假设缓冲区是空的，把一个值存在缓冲区，然后把 count 设置为 1 表示缓冲区满了。get()函数刚好相反，把缓冲区清空后（即将 count 设置为 0），并返回该值。不用担心这个共享缓冲区只能存储一条数据，稍后我们会一般化，用队列保存更多数据项，这会比听起来更有趣。

现在我们需要编写一些函数，知道何时可以访问缓冲区，以便将数据放入缓冲区或从缓冲区取出数据。条件是显而易见的：仅在 count 为 0 时（即缓冲器为空时），才将数据放入缓冲器中。仅在计数为 1 时（即缓冲器已满时），才从缓冲器获得数据。如果我们编写同步代码，让生产者将数据放入已满的缓冲区，或消费者从空的数据获取数据，就做错了（在这段代码中，断言将触发）。

这项工作将由两种类型的线程完成，其中一类我们称之为生产者（producer）线程，另一类我们称之为消费者（consumer）线程。图 30.5 展示了一个生产者的代码，它将一个整数放入共享缓冲区 loops 次，以及一个消费者，它从该共享缓冲区中获取数据（永远不停），

① 这里我们用了某种严肃的古英语和虚拟语气形式。

每次打印出从共享缓冲区中提取的数据项。

```
1   void *producer(void *arg) {
2       int i;
3       int loops = (int) arg;
4       for (i = 0; i < loops; i++) {
5           put(i);
6       }
7   }
8
9   void *consumer(void *arg) {
10      int i;
11      while (1) {
12          int tmp = get();
13          printf("%d\n", tmp);
14      }
15  }
```

图 30.5 生产者/消费者线程（第 1 版）

有问题的方案

假设只有一个生产者和一个消费者。显然，put()和 get()函数之中会有临界区，因为 put()更新缓冲区，get()读取缓冲区。但是，给代码加锁没有用，我们还需别的东西。不奇怪，别的东西就是某些条件变量。在这个（有问题的）首次尝试中（见图 30.6），我们用了条件变量 cond 和相关的锁 mutex。

```
1   cond_t cond;
2   mutex_t mutex;
3
4   void *producer(void *arg) {
5       int i;
6       for (i = 0; i < loops; i++) {
7           Pthread_mutex_lock(&mutex);            // p1
8           if (count == 1)                        // p2
9               Pthread_cond_wait(&cond, &mutex);  // p3
10          put(i);                                // p4
11          Pthread_cond_signal(&cond);            // p5
12          Pthread_mutex_unlock(&mutex);          // p6
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          Pthread_mutex_lock(&mutex);            // c1
20          if (count == 0)                        // c2
21              Pthread_cond_wait(&cond, &mutex);  // c3
22          int tmp = get();                       // c4
23          Pthread_cond_signal(&cond);            // c5
24          Pthread_mutex_unlock(&mutex);          // c6
```

```
25            printf("%d\n", tmp);
26        }
27    }
```

图 30.6 生产者/消费者：一个条件变量和 if 语句

来看看生产者和消费者之间的信号逻辑。当生产者想要填充缓冲区时，它等待缓冲区变空（p1～p3）。消费者具有完全相同的逻辑，但等待不同的条件——变满（c1～c3）。

当只有一个生产者和一个消费者时，图 30.6 中的代码能够正常运行。但如果有超过一个线程（例如两个消费者），这个方案会有两个严重的问题。哪两个问题？

……（暂停思考一下）……

我们来理解第一个问题，它与等待之前的 if 语句有关。假设有两个消费者（T_{c1} 和 T_{c2}），一个生产者（T_p）。首先，一个消费者（T_{c1}）先开始执行，它获得锁（c1），检查缓冲区是否可以消费（c2），然后等待（c3）（这会释放锁）。

接着生产者（T_p）运行。它获取锁（p1），检查缓冲区是否满（p2），发现没满就给缓冲区加入一个数字（p4）。然后生产者发出信号，说缓冲区已满（p5）。关键的是，这让第一个消费者（T_{c1}）不再睡在条件变量上，进入就绪队列。T_{c1} 现在可以运行（但还未运行）。生产者继续执行，直到发现缓冲区满后睡眠（p6,p1-p3）。

这时问题发生了：另一个消费者（T_{c2}）抢先执行，消费了缓冲区中的值（c1,c2,c4,c5,c6，跳过了 c3 的等待，因为缓冲区是满的）。现在假设 T_{c1} 运行，在从 wait 返回之前，它获取了锁，然后返回。然后它调用了 get() (p4)，但缓冲区已无法消费！断言触发，代码不能像预期那样工作。显然，我们应该设法阻止 T_{c1} 去消费，因为 T_{c2} 插进来，消费了缓冲区中之前生产的一个值。表 30.1 展示了每个线程的动作，以及它的调度程序状态（就绪、运行、睡眠）随时间的变化。

表 30.1 追踪线程：有问题的方案（第 1 版）

T_{c1}	状态	T_{c2}	状态	T_p	状态	count	注释
c1	运行		就绪		就绪	0	
c2	运行		就绪		就绪	0	
c3	睡眠		就绪		就绪	0	没数据可取
	睡眠		就绪	p1	运行	0	
	睡眠		就绪	p2	运行	0	
	睡眠		就绪	p4	运行	1	缓冲区现在满了
	就绪		就绪	p5	运行	1	T_{c1} 唤醒
	就绪		就绪	p6	运行	1	
	就绪		就绪	p1	运行	1	
	就绪		就绪	p2	运行	1	
	就绪		就绪	p3	睡眠	1	缓冲区满了，睡眠
	就绪	c1	运行		睡眠	1	T_{c2} 插入……
	就绪	c2	运行		睡眠	1	
	就绪	c4	运行		睡眠	0	……抓取了数据

续表

T_{c1}	状态	T_{c2}	状态	T_p	状态	count	注释
	就绪	c5	运行		就绪	0	T_p 唤醒
	就绪	c6	运行		就绪	0	
c4	运行		就绪		就绪	0	啊！没数据

问题产生的原因很简单：在 T_{c1} 被生产者唤醒后，但在它运行之前，缓冲区的状态改变了（由于 T_{c2}）。发信号给线程只是唤醒它们，暗示状态发生了变化（在这个例子中，就是值已被放入缓冲区），但并不会保证在它运行之前状态一直是期望的情况。信号的这种释义常称为 Mesa 语义（Mesa semantic），为了纪念以这种方式建立条件变量的首次研究[LR80]。另一种释义是 Hoare 语义（Hoare semantic），虽然实现难度大，但是会保证被唤醒线程立刻执行[H74]。实际上，几乎所有系统都采用了 Mesa 语义。

较好但仍有问题的方案：使用 While 语句替代 If

幸运的是，修复这个问题很简单（见图 30.7）：把 if 语句改为 while。当消费者 T_{c1} 被唤醒后，立刻再次检查共享变量（c2）。如果缓冲区此时为空，消费者就会回去继续睡眠（c3）。生产者中相应的 if 也改为 while（p2）。

```
1    cond_t cond;
2    mutex_t mutex;
3
4    void *producer(void *arg) {
5        int i;
6        for (i = 0; i < loops; i++) {
7            Pthread_mutex_lock(&mutex);              // p1
8            while (count == 1)                       // p2
9                Pthread_cond_wait(&cond, &mutex);    // p3
10           put(i);                                  // p4
11           Pthread_cond_signal(&cond);              // p5
12           Pthread_mutex_unlock(&mutex);            // p6
13       }
14   }
15
16   void *consumer(void *arg) {
17       int i;
18       for (i = 0; i < loops; i++) {
19           Pthread_mutex_lock(&mutex);              // c1
20           while (count == 0)                       // c2
21               Pthread_cond_wait(&cond, &mutex);    // c3
22           int tmp = get();                         // c4
23           Pthread_cond_signal(&cond);              // c5
24           Pthread_mutex_unlock(&mutex);            // c6
25           printf("%d\n", tmp);
26       }
27   }
```

图 30.7 生产者/消费者：一个条件变量和 while 语句

由于 Mesa 语义，我们要记住一条关于条件变量的简单规则：总是使用 while 循环（always use while loop）。虽然有时候不需要重新检查条件，但这样做总是安全的，做了就开心了。

但是，这段代码仍然有一个问题，也是上文提到的两个问题之一。你能想到吗？它和我们只用了一个条件变量有关。尝试弄清楚这个问题是什么，再继续阅读。想一下！

……（暂停想一想，或者闭一下眼）……

我们来确认一下你想得对不对。假设两个消费者（T_{c1} 和 T_{c2}）先运行，都睡眠了（c3）。生产者开始运行，在缓冲区放入一个值，唤醒了一个消费者（假定是 T_{c1}），并开始睡眠。现在是一个消费者马上要运行（T_{c1}），两个线程（T_{c2} 和 T_p）都等待在同一个条件变量上。问题马上就要出现了：让人感到兴奋！

消费者 T_{c1} 醒过来并从 wait()调用返回（c3），重新检查条件（c2），发现缓冲区是满的，消费了这个值（c4）。这个消费者然后在该条件上发信号（c5），唤醒一个在睡眠的线程。但是，应该唤醒哪个线程呢？

因为消费者已经清空了缓冲区，很显然，应该唤醒生产者。但是，如果它唤醒了 T_{c2}（这绝对是可能的，取决于等待队列是如何管理的），问题就出现了。具体来说，消费者 T_{c2} 会醒过来，发现队列为空（c2），又继续回去睡眠（c3）。生产者 T_p 刚才在缓冲区中放了一个值，现在在睡眠。另一个消费者线程 T_{c1} 也回去睡眠了。3 个线程都在睡眠，显然是一个缺陷。由表 30.2 可以看到这个可怕灾难的步骤。

表 30.2　　追踪线程：有问题的方案（第 2 版）

T_{c1}	状态	T_{c2}	状态	T_p	状态	count	注释
c1	运行		就绪		就绪	0	
c2	运行		就绪		就绪	0	
c3	睡眠		就绪		就绪	0	没数据可取
	睡眠	c1	运行		就绪	0	
	睡眠	c2	运行		就绪	0	
	睡眠	c3	睡眠		就绪	0	没数据可取
	睡眠		睡眠	p1	运行	0	
	睡眠		睡眠	p2	运行	0	
	睡眠		睡眠	p4	运行	1	缓冲区现在满了
	就绪		睡眠	p5	运行	1	T_{c1} 唤醒
	就绪		睡眠	p6	运行	1	
	就绪		睡眠	p1	运行	1	
	就绪		睡眠	p2	运行	1	
	就绪		睡眠	p3	睡眠	1	必须睡（满了）
c2	运行		睡眠		睡眠	1	重新检查条件
c4	运行		睡眠		睡眠	0	T_{c1} 抓取数据
c5	运行		就绪		睡眠	0	啊！唤醒 T_{c2}
c6	运行		就绪		睡眠	0	

续表

T_{c1}	状态	T_{c2}	状态	T_p	状态	count	注释
c1	运行		就绪		睡眠	0	
c2	运行		就绪		睡眠	0	
c3	睡眠		就绪		睡眠	0	没数据可取
	睡眠	c2	运行		睡眠	0	
	睡眠	c3	睡眠		睡眠	0	大家都睡了……

信号显然需要，但必须更有指向性。消费者不应该唤醒消费者，而应该只唤醒生产者，反之亦然。

单值缓冲区的生产者/消费者方案

解决方案也很简单：使用两个条件变量，而不是一个，以便正确地发出信号，在系统状态改变时，哪类线程应该唤醒。图 30.8 展示了最终的代码。

```
1    cond_t  empty, fill;
2    mutex_t mutex;
3
4    void *producer(void *arg) {
5        int i;
6        for (i = 0; i < loops; i++) {
7            Pthread_mutex_lock(&mutex);
8            while (count == 1)
9                Pthread_cond_wait(&empty,  &mutex);
10           put(i);
11           Pthread_cond_signal(&fill);
12           Pthread_mutex_unlock(&mutex);
13       }
14   }
15
16   void *consumer(void *arg) {
17       int i;
18       for (i = 0; i < loops; i++) {
19           Pthread_mutex_lock(&mutex);
20           while (count == 0)
21               Pthread_cond_wait(&fill, &mutex);
22           int tmp = get();
23           Pthread_cond_signal(&empty);
24           Pthread_mutex_unlock(&mutex);
25           printf("%d\n", tmp);
26       }
27   }
```

图 30.8 生产者/消费者：两个条件变量和 while 语句

在上述代码中，生产者线程等待条件变量 empty，发信号给变量 fill。相应地，消费者线程等待 fill，发信号给 empty。这样做，从设计上避免了上述第二个问题：消费者再也不会唤醒消费者，生产者也不会唤醒生产者。

最终的生产者/消费者方案

我们现在有了可用的生产者/消费者方案，但不太通用。我们最后的修改是提高并发和效率。具体来说，增加更多缓冲区槽位，这样在睡眠之前，可以生产多个值。同样，睡眠之前可以消费多个值。单个生产者和消费者时，这种方案因为上下文切换少，提高了效率。多个生产者和消费者时，它甚至支持并发生产和消费，从而提高了并发。幸运的是，和现有方案相比，改动也很小。

第一处修改是缓冲区结构本身，以及对应的 put()和 get()方法（见图 30.9）。我们还稍稍修改了生产者和消费者的检查条件，以便决定是否要睡眠。图 30.10 展示了最终的等待和信号逻辑。生产者只有在缓冲区满了的时候才会睡眠（p2），消费者也只有在队列为空的时候睡眠（c2）。至此，我们解决了生产者/消费者问题。

```
1    int buffer[MAX];
2    int fill  = 0;
3    int use   = 0;
4    int count = 0;
5
6    void put(int value) {
7        buffer[fill] = value;
8        fill = (fill + 1) % MAX;
9        count++;
10   }
11
12   int get() {
13       int tmp = buffer[use];
14       use = (use + 1) % MAX;
15       count--;
16       return tmp;
17   }
```

图 30.9 最终的 put()和 get()方法

```
1    cond_t empty, fill;
2    mutex_t mutex;
3
4    void *producer(void *arg) {
5        int i;
6        for (i = 0; i < loops; i++) {
7            Pthread_mutex_lock(&mutex);              // p1
8            while (count == MAX)                     // p2
9                Pthread_cond_wait(&empty, &mutex);   // p3
10           put(i);                                  // p4
11           Pthread_cond_signal(&fill);              // p5
12           Pthread_mutex_unlock(&mutex);            // p6
13       }
14   }
15
16   void *consumer(void *arg) {
17       int i;
18       for (i = 0; i < loops; i++) {
```

```
19          Pthread_mutex_lock(&mutex);                // c1
20          while (count == 0)                         // c2
21              Pthread_cond_wait(&fill, &mutex);      // c3
22          int tmp = get();                           // c4
23          Pthread_cond_signal(&empty);               // c5
24          Pthread_mutex_unlock(&mutex);              // c6
25          printf("%d\n", tmp);
26      }
27  }
```

图 30.10 最终有效方案

提示：对条件变量使用 while（不是 if）

多线程程序在检查条件变量时，使用 while 循环总是对的。if 语句可能会对，这取决于发信号的语义。因此，总是使用 while，代码就会符合预期。

对条件变量使用 while 循环，这也解决了假唤醒（spurious wakeup）的情况。某些线程库中，由于实现的细节，有可能出现一个信号唤醒两个线程的情况[L11]。再次检查线程的等待条件，假唤醒是另一个原因。

30.3 覆盖条件

现在再来看条件变量的一个例子。这段代码摘自 Lampson 和 Redell 关于飞行员的论文[LR80]，同一个小组首次提出了上述的 Mesa 语义（Mesa semantic，他们使用的语言是 Mesa，因此而得名）。

他们遇到的问题通过一个简单的例子就能说明，在这个例子中，是一个简单的多线程内存分配库。图 30.11 是展示这一问题的代码片段。

```
1   // how many bytes of the heap are free?
2   int bytesLeft = MAX_HEAP_SIZE;
3
4   // need lock and condition too
5   cond_t  c;
6   mutex_t m;
7
8   void *
9   allocate(int size) {
10      Pthread_mutex_lock(&m);
11      while (bytesLeft < size)
12          Pthread_cond_wait(&c, &m);
13      void *ptr = ...; // get mem from heap
14      bytesLeft -= size;
15      Pthread_mutex_unlock(&m);
16      return ptr;
17  }
18
```

```
19   void free(void *ptr, int size) {
20       Pthread_mutex_lock(&m);
21       bytesLeft += size;
22       Pthread_cond_signal(&c); // whom to signal??
23       Pthread_mutex_unlock(&m);
24   }
```

图 30.11　覆盖条件的例子

从代码中可以看出，当线程调用进入内存分配代码时，它可能会因为内存不足而等待。相应的，线程释放内存时，会发信号说有更多内存空闲。但是，代码中有一个问题：应该唤醒哪个等待线程（可能有多个线程）？

考虑以下场景。假设目前没有空闲内存，线程 T_a 调用 allocate(100)，接着线程 T_b 请求较少的内存，调用 allocate(10)。T_a 和 T_b 都等待在条件上并睡眠，没有足够的空闲内存来满足它们的请求。

这时，假定第三个线程 T_c 调用了 free(50)。遗憾的是，当它发信号唤醒等待线程时，可能不会唤醒申请 10 字节的 T_b 线程。而 T_a 线程由于内存不够，仍然等待。因为不知道唤醒哪个（或哪些）线程，所以图中代码无法正常工作。

Lampson 和 Redell 的解决方案也很直接：用 pthread_cond_broadcast()代替上述代码中的 pthread_cond_signal()，唤醒所有的等待线程。这样做，确保了所有应该唤醒的线程都被唤醒。当然，不利的一面是可能会影响性能，因为不必要地唤醒了其他许多等待的线程，它们本来（还）不应该被唤醒。这些线程被唤醒后，重新检查条件，马上再次睡眠。

Lampson 和 Redell 把这种条件变量叫作覆盖条件（covering condition），因为它能覆盖所有需要唤醒线程的场景（保守策略）。成本如上所述，就是太多线程被唤醒。聪明的读者可能发现，在单个条件变量的生产者/消费者问题中，也可以使用这种方法。但是，在这个例子中，我们有更好的方法，因此用了它。一般来说，如果你发现程序只有改成广播信号时才能工作（但你认为不需要），可能是程序有缺陷，修复它！但在上述内存分配的例子中，广播可能是最直接有效的方案。

30.4　小结

我们看到了引入锁之外的另一个重要同步原语：条件变量。当某些程序状态不符合要求时，通过允许线程进入休眠状态，条件变量使我们能够漂亮地解决许多重要的同步问题，包括著名的（仍然重要的）生产者/消费者问题，以及覆盖条件。

参考资料

[D72]“Information Streams Sharing a Finite Buffer”
E.W. Dijkstra
Information Processing Letters 1: 179180, 1972

这是一篇介绍生产者/消费者问题的著名文章。

[D01]“My recollections of operating system design”
E.W. Dijkstra April, 2001
如果你对这一领域的先驱们如何提出一些非常基本的概念（诸如“中断”和“栈”等概念）感兴趣，那么它是一本很好的读物！

[H74]“Monitors: An Operating System Structuring Concept”
C.A.R. Hoare
Communications of the ACM, 17:10, pages 549–557, October 1974
Hoare 在并发方面做了大量的理论工作。不过，他最出名的工作可能还是快速排序算法，那是世上最酷的排序算法，至少本书的作者这样认为。

[L11]“Pthread cond signal Man Page”
Linux 手册页展示了一个很好的简单例子，以说明为什么线程可能会发生假唤醒——因为信号/唤醒代码中的竞态条件。

[LR80]“Experience with Processes and Monitors in Mesa”
B.W. Lampson, D.R. Redell
Communications of the ACM. 23:2, pages 105-117, February 1980
一篇关于如何在真实系统中实际实现信号和条件变量的极好论文，导致了术语“Mesa”语义，说明唤醒意味着什么。较早的语义由 Tony Hoare [H74]提出，于是被称为“Hoare”语义。

第 31 章　信号量

我们现在知道，需要锁和条件变量来解决各种相关的、有趣的并发问题。多年前，首先认识到这一点的人之中，有一个就是 Edsger Dijkstra（虽然很难知道确切的历史[GR92]）。他出名是因为图论中著名的“最短路径”算法[D59]，因为早期关于结构化编程的论战“Goto 语句是有害的”[D68a]（这是一个极好的标题！），还因为他引入了名为信号量[D68b，D72]的同步原语，正是这里我们要学习的。事实上，Dijkstra 及其同事发明了信号量，作为与同步有关的所有工作的唯一原语。你会看到，可以使用信号量作为锁和条件变量。

关键问题：如何使用信号量？

如何使用信号量代替锁和条件变量？什么是信号量？什么是二值信号量？用锁和条件变量来实现信号量是否简单？不用锁和条件变量，如何实现信号量？

31.1　信号量的定义

信号量是有一个整数值的对象，可以用两个函数来操作它。在 POSIX 标准中，是 sem_wait()和 sem_post()[①]。因为信号量的初始值能够决定其行为，所以首先要初始化信号量，才能调用其他函数与之交互，如图 31.1 所示。

```
1    #include <semaphore.h>
2    sem_t s;
3    sem_init(&s, 0, 1);
```

图 31.1　初始化信号量

其中申明了一个信号量 s，通过第三个参数，将它的值初始化为 1。sem_init()的第二个参数，在我们看到的所有例子中都设置为 0，表示信号量是在同一进程的多个线程共享的。读者可以参考手册，了解信号量的其他用法（即如何用于跨不同进程的同步访问），这要求第二个参数用不同的值。

信号量初始化之后，我们可以调用 sem_wait()或 sem_post()与之交互。图 31.2 展示了这两个函数的不同行为。

我们暂时不关注这两个函数的实现，这显然是需要注意的。多个线程会调用 sem_wait()和 sem_post()，显然需要管理这些临界区。我们首先关注如何使用这些原语，稍后再讨论如何实现。

① 历史上，sem_wait()开始被 Dijkstra 称为 P()（代指荷兰语单词“to probe”），而 sem_post()被称为 V()（代指荷兰语单词“to test”）。有时候，人们也会称它们为下（down）和上（up）。使用荷兰语版本，给你的朋友留下深刻印象。

```
1    int sem_wait(sem_t *s) {
2        decrement the value of semaphore s by one
3        wait if value of semaphore s is negative
4    }
5
6    int sem_post(sem_t *s) {
7        increment the value of semaphore s by one
8        if there are one or more threads waiting, wake one
9    }
```

图 31.2 信号量：Wait 和 Post 的定义

我们应该讨论这些接口的几个突出方面。首先，sem_wait()要么立刻返回（调用 sem_wait()时，信号量的值大于等于 1），要么会让调用线程挂起，直到之后的一个 post 操作。当然，也可能多个调用线程都调用 sem_wait()，因此都在队列中等待被唤醒。

其次，sem_post()并没有等待某些条件满足。它直接增加信号量的值，如果有等待线程，唤醒其中一个。

最后，当信号量的值为负数时，这个值就是等待线程的个数[D68b]。虽然这个值通常不会暴露给信号量的使用者，但这个恒定的关系值得了解，可能有助于记住信号量的工作原理。

先（暂时）不用考虑信号量内的竞争条件，假设这些操作都是原子的。我们很快就会用锁和条件变量来实现。

31.2 二值信号量（锁）

现在我们要使用信号量了。信号量的第一种用法是我们已经熟悉的：用信号量作为锁。在图 31.3 所示的代码片段里，我们直接把临界区用一对 sem_wait()/sem_post()环绕。但是，为了使这段代码正常工作，信号量 m 的初始值（图中初始化为 *X*）是至关重要的。*X* 应该是多少呢？

```
1    sem_t m;
2    sem_init(&m, 0, X); // initialize semaphore to X; what should X be?
3
4    sem_wait(&m);
5    // critical section here
6    sem_post(&m);
```

图 31.3 二值信号量（就是锁）

……（读者先思考一下再继续学习）……

回顾 sem_wait()和 sem_post()函数的定义，我们发现初值应该是 1。

为了说明清楚，我们假设有两个线程的场景。第一个线程（线程 0）调用了 sem_wait()，它把信号量的值减为 0。然后，它只会在值小于 0 时等待。因为值是 0，调用线程从函数返回并继续，线程 0 现在可以自由进入临界区。线程 0 在临界区中，如果没有其他线程尝试获取锁，当它调用 sem_post()时，会将信号量重置为 1（因为没有等待线程，不会唤醒其他线程）。表 31.1 追踪了这一场景。

表 31.1 追踪线程：单线程使用一个信号量

信号量的值	线程 0	线程 1
1		
1	调用 sem_wait()	
0	sem_wait()返回	
0	（临界区）	
0	调用 sem_post()	
1	sem_post()返回	

如果线程 0 持有锁（即调用了 sem_wait()之后，调用 sem_post()之前），另一个线程（线程 1）调用 sem_wait()尝试进入临界区，那么更有趣的情况就发生了。这种情况下，线程 1 把信号量减为−1，然后等待（自己睡眠，放弃处理器）。线程 0 再次运行，它最终调用 sem_post()，将信号量的值增加到 0，唤醒等待的线程（线程 1），然后线程 1 就可以获取锁。线程 1 执行结束时，再次增加信号量的值，将它恢复为 1。

表 31.2 追踪了这个例子。除了线程的动作，表中还显示了每一个线程的调度程序状态（scheduler state）：运行、就绪（即可运行但没有运行）和睡眠。特别要注意，当线程 1 尝试获取已经被持有的锁时，陷入睡眠。只有线程 0 再次运行之后，线程 1 才可能会唤醒并继续运行。

表 31.2 追踪线程：两个线程使用一个信号量

值	线程 0	状态	线程 1	状态
1		运行		就绪
1	调用 sem_wait()	运行		就绪
0	sem_wait()返回	运行		就绪
0	（临界区：开始）	运行		就绪
0	中断；切换到→T1	就绪		运行
0		就绪	调用 sem_wait()	运行
−1		就绪	sem 减 1	运行
−1		就绪	(sem<0) →睡眠	睡眠
−1		运行	切换到→T0	睡眠
−1	（临界区：结束）	运行		睡眠
−1	调用 sem_post()	运行		睡眠
0	增加 sem	运行		睡眠
0	唤醒 T1	运行		就绪
0	sem_post()返回	运行		就绪
0	中断；切换到→T1	就绪		运行
0		就绪	sem_wait()返回	运行
0		就绪	（临界区）	运行
0		就绪	调用 sem_post()	运行
1		就绪	sem_post()返回	运行

如果你想追踪自己的例子，那么请尝试一个场景，多个线程排队等待锁。在这样的追踪中，信号量的值会是什么？

我们可以用信号量来实现锁了。因为锁只有两个状态（持有和没持有），所以这种用法有时也叫作二值信号量（binary semaphore）。事实上这种信号量也有一些更简单的实现，我们这里使用了更为通用的信号量作为锁。

31.3 信号量用作条件变量

信号量也可以用在一个线程暂停执行，等待某一条件成立的场景。例如，一个线程要等待一个链表非空，然后才能删除一个元素。在这种场景下，通常一个线程等待条件成立，另外一个线程修改条件并发信号给等待线程，从而唤醒等待线程。因为等待线程在等待某些条件（condition）发生变化，所以我们将信号量作为条件变量（condition variable）。

下面是一个简单例子。假设一个线程创建另外一线程，并且等待它结束（见图 31.4）。

```
1    sem_t s;
2
3    void *
4    child(void *arg) {
5        printf("child\n");
6        sem_post(&s); // signal here: child is done
7        return NULL;
8    }
9
10   int
11   main(int argc, char *argv[]) {
12       sem_init(&s, 0, X); // what should X be?
13       printf("parent: begin\n");
14       pthread_t c;
15       Pthread_create(c, NULL, child, NULL);
16       sem_wait(&s); // wait here for child
17       printf("parent: end\n");
18       return 0;
19   }
```

图 31.4 父线程等待子线程

该程序运行时，我们希望能看到这样的输出：

```
parent: begin
child
parent: end
```

然后问题就是如何用信号量来实现这种效果。结果表明，答案也很容易理解。从代码中可知，父线程调用 sem_wait()，子线程调用 sem_post()，父线程等待子线程执行完成。但是，问题来了：信号量的初始值应该是多少？

（再想一下，然后继续阅读）

当然，答案是信号量初始值应该是 0。有两种情况需要考虑。第一种，父线程创建了子线程，但是子线程并没有运行。这种情况下（见表 31.3），父线程调用 sem_wait()会先于子线程调用 sem_post()。我们希望父线程等待子线程运行。为此，唯一的办法是让信号量的值不大于 0。因此，0 为初值。父线程运行，将信号量减为−1，然后睡眠等待；子线程运行的时候，调用 sem_post()，信号量增加为 0，唤醒父线程，父线程然后从 sem_wait()返回，完成该程序。

表 31.3　　追踪线程：父线程等待子线程（场景 1）

值	父线程	状态	子线程	状态
0	create(子线程）	运行	(子线程产生)	就绪
0	调用 sem_wait()	运行		就绪
−1	sem 减 1	运行		就绪
−1	(sem<0)→ 睡眠	睡眠		就绪
−1	切换到→子线程	睡眠	子线程运行	运行
−1		睡眠	调用 sem_post()	运行
0		睡眠	sem 增 1	运行
0		就绪	wake(父线程)	运行
0		就绪	sem_post()返回	运行
0		就绪	中断；切换到→父线程	就绪
0	sem_wait()返回	运行		就绪

第二种情况是子线程在父线程调用 sem_wait()之前就运行结束（见表 31.4）。在这种情况下，子线程会先调用 sem_post()，将信号量从 0 增加到 1。然后当父线程有机会运行时，会调用 sem_wait()，发现信号量的值为 1。于是父线程将信号量从 1 减为 0，没有等待，直接从 sem_wait()返回，也达到了预期效果。

表 31.4　　追踪线程：父线程等待子线程（场景 2）

值	父线程	状态	子线程	状态
0	create（子线程）	运行	(子线程产生)	就绪
0	中断；切换到→子线程	就绪	子线程运行	运行
0		就绪	调用 sem_post()	运行
1		睡眠	sem 增 1	运行
1		就绪	wake(没有线程)	运行
1		就绪	sem_post()返回	运行
1	父线程运行	运行	中断；切换到→父线程	就绪
1	调用 sem_wait()	运行		就绪
0	sem 减 1	运行		就绪
0	(sem>=0)→不用睡眠	运行		就绪
0	sem_wait()返回	运行		就绪