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

31.4 生产者/消费者（有界缓冲区）问题

本章的下一个问题是生产者/消费者（producer/consumer）问题，有时称为有界缓冲区问题[D72]。第 30 章讲条件变量时已经详细描述了这一问题，细节请参考相应内容。

第一次尝试

第一次尝试解决该问题时，我们用两个信号量 empty 和 full 分别表示缓冲区空或者满。图 31.5 是 put()和 get()函数，图 31.6 是我们尝试解决生产者/消费者问题的代码。

```
1    int buffer[MAX];
2    int fill = 0;
3    int use  = 0;
4
5    void put(int value) {
6        buffer[fill] = value;    // line f1
7        fill = (fill + 1) % MAX; // line f2
8    }
9
10   int get() {
11       int tmp = buffer[use];    // line g1
12       use = (use + 1) % MAX;    // line g2
13       return tmp;
14   }
```

图 31.5 put()和 get()函数

```
1    sem_t empty;
2    sem_t full;
3
4    void *producer(void *arg) {
5        int i;
6        for (i = 0; i < loops; i++) {
7            sem_wait(&empty);           // line P1
8            put(i);                     // line P2
9            sem_post(&full);            // line P3
10       }
11   }
12
13   void *consumer(void *arg) {
14       int i, tmp = 0;
15       while (tmp != -1) {
16           sem_wait(&full);            // line C1
17           tmp = get();                // line C2
18           sem_post(&empty);           // line C3
19           printf("%d\n", tmp);
20       }
```

```
21  }
22
23  int main(int argc, char *argv[]) {
24      // ...
25      sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
26      sem_init(&full, 0, 0);    // ... and 0 are full
27      // ...
28  }
```

图 31.6 增加 full 和 empty 条件

本例中，生产者等待缓冲区为空，然后加入数据。类似地，消费者等待缓冲区变成有数据的状态，然后取走数据。我们先假设 MAX=1（数组中只有一个缓冲区），验证程序是否有效。

假设有两个线程，一个生产者和一个消费者。我们来看在一个 CPU 上的具体场景。消费者先运行，执行到 C1 行，调用 sem_wait(&full)。因为 full 初始值为 0，wait 调用会将 full 减为−1，导致消费者睡眠，等待另一个线程调用 sem_post(&full)，符合预期。

假设生产者然后运行。执行到 P1 行，调用 sem_wait(&empty)。不像消费者，生产者将继续执行，因为 empty 被初始化为 MAX（在这里是 1）。因此，empty 被减为 0，生产者向缓冲区中加入数据，然后执行 P3 行，调用 sem_post(&full)，把 full 从−1 变成 0，唤醒消费者（即将它从阻塞变成就续）。

在这种情况下，可能会有两种情况。如果生产者继续执行，再次循环到 P1 行，由于 empty 值为 0，它会阻塞。如果生产者被中断，而消费者开始执行，调用 sem_wait(&full)（c1 行），发现缓冲区确实满了，消费它。这两种情况都是符合预期的。

你可以用更多的线程来尝试这个例子（即多个生产者和多个消费者）。它应该仍然正常运行。

我们现在假设 MAX 大于 1（比如 MAX=10）。对于这个例子，假定有多个生产者，多个消费者。现在就有问题了：竞态条件。你能够发现是哪里产生的吗？（花点时间找一下）如果没有发现，不妨仔细观察 put()和 get()的代码。

好，我们来理解该问题。假设两个生产者（Pa 和 Pb）几乎同时调用 put()。当 Pa 先运行，在 f1 行先加入第一条数据（fill=0），假设 Pa 在将 fill 计数器更新为 1 之前被中断，Pb 开始运行，也在 f1 行给缓冲区的 0 位置加入一条数据，这意味着那里的老数据被覆盖！这可不行，我们不能让生产者的数据丢失。

解决方案：增加互斥

你可以看到，这里忘了互斥。向缓冲区加入元素和增加缓冲区的索引是临界区，需要小心保护起来。所以，我们使用二值信号量来增加锁。图 31.7 是对应的代码。

```
1   sem_t empty;
2   sem_t full;
3   sem_t mutex;
4
5   void *producer(void *arg) {
6       int i;
```

```
7       for (i = 0; i < loops; i++) {
8           sem_wait(&mutex);           // line p0 (NEW LINE)
9           sem_wait(&empty);           // line p1
10          put(i);                     // line p2
11          sem_post(&full);            // line p3
12          sem_post(&mutex);           // line p4 (NEW LINE)
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          sem_wait(&mutex);           // line c0 (NEW LINE)
20          sem_wait(&full);            // line c1
21          int tmp = get();            // line c2
22          sem_post(&empty);           // line c3
23          sem_post(&mutex);           // line c4 (NEW LINE)
24          printf("%d\n", tmp);
25      }
26  }
27
28  int main(int argc, char *argv[]) {
29      // ...
30      sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
31      sem_init(&full, 0, 0);    // ... and 0 are full
32      sem_init(&mutex, 0, 1);    // mutex=1 because it is a lock (NEW LINE)
33      // ...
34  }
```

图 31.7 增加互斥量（不正确的）

现在我们给整个 put()/get()部分都增加了锁，注释中有 NEW LINE 的几行就是。这似乎是正确的思路，但仍然有问题。为什么？死锁。为什么会发生死锁？考虑一下，尝试找出一个死锁的场景。必须以怎样的步骤执行，会导致程序死锁？

避免死锁

好，既然你想出来了，下面是答案。假设有两个线程，一个生产者和一个消费者。消费者首先运行，获得锁（c0 行），然后对 full 信号量执行 sem_wait() （c1 行）。因为还没有数据，所以消费者阻塞，让出 CPU。但是，重要的是，此时消费者仍然持有锁。

然后生产者运行。假如生产者能够运行，它就能生产数据并唤醒消费者线程。遗憾的是，它首先对二值互斥信号量调用 sem_wait()（p0 行）。锁已经被持有，因此生产者也被卡住。

这里出现了一个循环等待。消费者持有互斥量，等待在 full 信号量上。生产者可以发送 full 信号，却在等待互斥量。因此，生产者和消费者互相等待对方——典型的死锁。

最后，可行的方案

要解决这个问题，只需减少锁的作用域。图 31.8 是最终的可行方案。可以看到，我们

把获取和释放互斥量的操作调整为紧挨着临界区，把 full、empty 的唤醒和等待操作调整到锁外面。结果得到了简单而有效的有界缓冲区，多线程程序的常用模式。现在理解，将来使用。未来的岁月中，你会感谢我们的。至少在期末考试遇到这个问题时，你会感谢我们。

```
1   sem_t empty;
2   sem_t full;
3   sem_t mutex;
4
5   void *producer(void *arg) {
6       int i;
7       for (i = 0; i < loops; i++) {
8           sem_wait(&empty);           // line p1
9           sem_wait(&mutex);           // line p1.5 (MOVED MUTEX HERE...)
10          put(i);                     // line p2
11          sem_post(&mutex);           // line p2.5 (... AND HERE)
12          sem_post(&full);            // line p3
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          sem_wait(&full);            // line c1
20          sem_wait(&mutex);           // line c1.5 (MOVED MUTEX HERE...)
21          int tmp = get();            // line c2
22          sem_post(&mutex);           // line c2.5 (... AND HERE)
23          sem_post(&empty);           // line c3
24          printf("%d\n", tmp);
25      }
26  }
27
28  int main(int argc, char *argv[]) {
29      // ...
30      sem_init(&empty, 0, MAX);  // MAX buffers are empty to begin with...
31      sem_init(&full, 0, 0);     // ... and 0 are full
32      sem_init(&mutex, 0, 1);    // mutex=1 because it is a lock
33      // ...
34  }
```

图 31.8 增加互斥量（正确的）

31.5 读者—写者锁

另一个经典问题源于对更加灵活的锁定原语的渴望，它承认不同的数据结构访问可能需要不同类型的锁。例如，一个并发链表有很多插入和查找操作。插入操作会修改链表的状态（因此传统的临界区有用），而查找操作只是读取该结构，只要没有进行插入操作，我们可以并发的执行多个查找操作。读者—写者锁（reader-writer lock）就是用来完成这种操

作的[CHP71]。图 31.9 是这种锁的代码。

代码很简单。如果某个线程要更新数据结构，需要调用 rwlock_acquire_lock()获得写锁，调用 rwlock_release_writelock()释放锁。内部通过一个 writelock 的信号量保证只有一个写者能获得锁进入临界区，从而更新数据结构。

```
1   typedef struct _rwlock_t {
2     sem_t lock;      // binary semaphore (basic lock)
3     sem_t writelock; // used to allow ONE writer or MANY readers
4     int    readers;  // count of readers reading in critical section
5   } rwlock_t;
6
7   void rwlock_init(rwlock_t *rw) {
8     rw->readers = 0;
9     sem_init(&rw->lock, 0, 1);
10    sem_init(&rw->writelock, 0, 1);
11  }
12
13  void rwlock_acquire_readlock(rwlock_t *rw) {
14    sem_wait(&rw->lock);
15    rw->readers++;
16    if (rw->readers == 1)
17      sem_wait(&rw->writelock); // first reader acquires writelock
18    sem_post(&rw->lock);
19  }
20
21  void rwlock_release_readlock(rwlock_t *rw) {
22    sem_wait(&rw->lock);
23    rw->readers--;
24    if (rw->readers == 0)
25      sem_post(&rw->writelock); // last reader releases writelock
26    sem_post(&rw->lock);
27  }
28
29  void rwlock_acquire_writelock(rwlock_t *rw) {
30    sem_wait(&rw->writelock);
31  }
32
33  void rwlock_release_writelock(rwlock_t *rw) {
34    sem_post(&rw->writelock);
35  }
```

图 31.9 一个简单的读者-写者锁

读锁的获取和释放操作更加吸引人。获取读锁时，读者首先要获取 lock，然后增加 reader 变量，追踪目前有多少个读者在访问该数据结构。重要的步骤然后在 rwlock_acquire_readlock()内发生，当第一个读者获取该锁时。在这种情况下，读者也会获取写锁，即在 writelock 信号量上调用 sem_wait()，最后调用 sem_post()释放 lock。

一旦一个读者获得了读锁，其他的读者也可以获取这个读锁。但是，想要获取写锁的线程，就必须等到所有的读者都结束。最后一个退出的写者在“writelock”信号量上调用

sem_post()，从而让等待的写者能够获取该锁。

提示：简单的笨办法可能更好（Hill 定律）

我们不能小看一个概念，即简单的笨办法可能最好。某些时候简单的自旋锁反而是最有效的，因为它容易实现而且高效。虽然读者—写者锁听起来很酷，但是却很复杂，复杂可能意味着慢。因此，总是优先尝试简单的笨办法。

这种受简单吸引的思想，在多个地方都能发现。一个早期来源是 Mark Hill 的学位论文[H87]，研究如何为 CPU 设计缓存。Hill 发现简单的直接映射缓存比花哨的集合关联性设计更加有效（一个原因是在缓存中，越简单的设计，越能够更快地查找）。Hill 简洁地总结了他的工作："大而笨更好。"因此我们将这种类似的建议叫作 Hill 定律（Hill's Law）。

这一方案可行（符合预期），但有一些缺陷，尤其是公平性。读者很容易饿死写者。存在复杂一些的解决方案，也许你可以想到更好的实现？提示：有写者等待时，如何能够避免更多的读者进入并持有锁。

最后，应该指出，读者-写者锁还有一些注意点。它们通常加入了更多开锁（尤其是更复杂的实现），因此和其他一些简单快速的锁相比，读者写者锁在性能方面没有优势[CB08]。无论哪种方式，它们都再次展示了如何以有趣、有用的方式来使用信号量。

31.6 哲学家就餐问题

哲学家就餐问题（dining philosopher's problem）是一个著名的并发问题，它由 Dijkstra 提出来并解决[DHO71]。这个问题之所以出名，是因为它很有趣，引人入胜，但其实用性却不强。可是，它的名气让我们在这里必须讲。实际上，你可能会在面试中遇到这一问题，假如老师没有提过，导致你们没有通过面试，你们会责怪操作系统老师的。因此，我们这里会讨论这一问题。假如你们因为这个问题得到工作，可以向操作系统老师发感谢信，或者发一些股票期权。

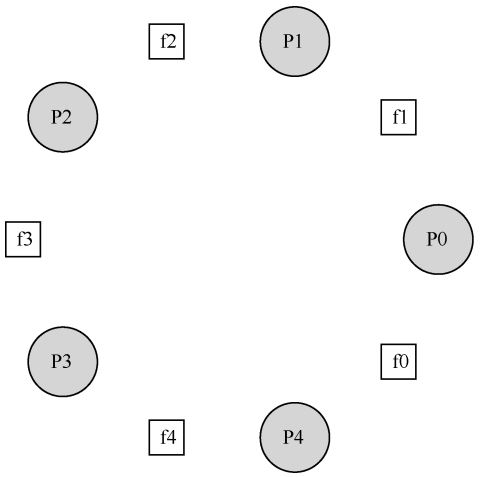

图 31.10 哲学家就餐问题

这个问题的基本情况是（见图 31.10）：假定有 5 位"哲学家"围着一个圆桌。每两位哲学家之间有一把餐叉（一共 5 把）。哲学家有时要思考一会，不需要餐叉；有时又要就餐。而一位哲学家只有同时拿到了左手边和右手边的两把餐叉，才能吃到东西。关于餐叉的竞争以及随之而来的同步问题，就是我们在并发编程中研究它的原因。

下面是每个哲学家的基本循环：

```
while (1) {
  think();
```

```
    getforks();
    eat();
    putforks();
}
```

关键的挑战就是如何实现 getforks()和 putforks()函数，保证没有死锁，没有哲学家饿死，并且并发度更高（尽可能让更多哲学家同时吃东西）。

根据 Downey 的解决方案[D08]，我们会用一些辅助函数，帮助构建解决方案。它们是：

```
int left(int p)  { return p; }
int right(int p) { return (p + 1) % 5; }
```

如果哲学家 p 希望用左手边的叉子，他们就调用 left(p)。类似地，右手边的叉子就用 right(p)。模运算解决了最后一个哲学家（p = 4）右手边叉子的编号问题，就是餐叉 0。

我们需要一些信号量来解决这个问题。假设需要 5 个，每个餐叉一个：sem_t forks[5]。

有问题的解决方案

我们开始第一次尝试。假设我们把每个信号量（在 fork 数组中）都用 1 初始化。同时假设每个哲学家知道自己的编号（p）。我们可以写出 getforks()和 putforks()函数，如图 31.11 所示。

```
1    void getforks() {
2      sem_wait(forks[left(p)]);
3      sem_wait(forks[right(p)]);
4    }
5
6    void putforks() {
7      sem_post(forks[left(p)]);
8      sem_post(forks[right(p)]);
9  }
```

图 31.11 getforks()和 putforks()函数

这个（有问题的）解决方案背后的思路如下。为了拿到餐叉，我们依次获取每把餐叉的锁——先是左手边的，然后是右手边的。结束就餐时，释放掉锁。很简单，不是吗？但是，在这个例子中，简单是有问题的。你能看到问题吗？想一想。

问题是死锁（deadlock）。假设每个哲学家都拿到了左手边的餐叉，他们每个都会阻塞住，并且一直等待另一个餐叉。具体来说，哲学家 0 拿到了餐叉 0，哲学家 1 拿到了餐叉 1，哲学家 2 拿到餐叉 2，哲学家 3 拿到餐叉 3，哲学家 4 拿到餐叉 4。所有的餐叉都被占有了，所有的哲学家都阻塞着，并且等待另一个哲学家占有的餐叉。我们在后续章节会深入学习死锁，这里只要知道这个方案行不通就可以了。

一种方案：破除依赖

解决上述问题最简单的方法，就是修改某个或者某些哲学家的取餐叉顺序。事实上，Dijkstra 自己也是这样解决的。具体来说，假定哲学家 4（编写最大的一个）取餐叉的顺序

不同。相应的代码如下：

```
1    void getforks() {
2      if (p == 4) {
3        sem_wait(forks[right(p)]);
4        sem_wait(forks[left(p)]);
5      } else {
6        sem_wait(forks[left(p)]);
7        sem_wait(forks[right(p)]);
8      }
9    }
```

因为最后一个哲学家会尝试先拿右手边的餐叉，然后拿左手边，所以不会出现每个哲学家都拿着一个餐叉，卡住等待另一个的情况，等待循环被打破了。想想这个方案的后果，让你自己相信它有效。

还有其他一些类似的“著名”问题，比如吸烟者问题（cigarette smoker’s problem），理发师问题（sleeping barber problem）。大多数问题只是让我们去理解并发，某些问题的名字很吸引人。感兴趣的读者可以去查阅相关资料，或者通过一些更实际的思考去理解并发行为[D08]。

31.7 如何实现信号量

最后，我们用底层的同步原语（锁和条件变量），来实现自己的信号量，名字叫作Zemaphore。这个任务相当简单，如图 31.12 所示。

```
1    typedef struct   _Zem_t {
2        int value;
3        pthread_cond_t cond;
4        pthread_mutex_t lock;
5    } Zem_t;
6
7    // only one thread can call this
8    void Zem_init(Zem_t *s, int value) {
9        s->value = value;
10       Cond_init(&s->cond);
11       Mutex_init(&s->lock);
12   }
13
14   void Zem_wait(Zem_t *s) {
15       Mutex_lock(&s->lock);
16       while (s->value <= 0)
17           Cond_wait(&s->cond, &s->lock);
18       s->value--;
19       Mutex_unlock(&s->lock);
20   }
21
22   void Zem_post(Zem_t *s) {
```

```
23        Mutex_lock(&s->lock);
24        s->value++;
25        Cond_signal(&s->cond);
26        Mutex_unlock(&s->lock);
27    }
```

图 31.12 用锁和条件变量实现 Zemahpore

可以看到，我们只用了一把锁、一个条件变量和一个状态的变量来记录信号量的值。请自己研究这些代码，直到真正理解它。去做吧！

我们实现的 Zemaphore 和 Dijkstra 定义的信号量有一点细微区别，就是我们没有保持当信号量的值为负数时，让它反映出等待的线程数。事实上，该值永远不会小于 0。这一行为更容易实现，并符合现有的 Linux 实现。

提示：小心泛化

在系统设计中，泛化的抽象技术是很有用处的。一个好的想法稍微扩展之后，就可以解决更大一类问题。然而，泛化时要小心，正如 Lampson 提醒我们的“不要泛化。泛化通常都是错的。”[L83]

我们可以把信号量当作锁和条件变量的泛化。但这种泛化有必要吗？考虑基于信号量去实现条件变量的难度，可能这种泛化并没有你想的那么通用。

很奇怪，利用信号量来实现锁和条件变量，是棘手得多的问题。某些富有经验的并发程序员曾经在 Windows 环境下尝试过，随之而来的是很多缺陷[B04]。你自己试一下，看看是否能明白为什么使用信号量实现条件变量比看起来更困难。

31.8 小结

信号量是编写并发程序的强大而灵活的原语。有程序员会因为简单实用，只用信号量，不用锁和条件变量。

本章展示了几个经典问题和解决方案。如果你有兴趣了解更多，有许多资料可以参考。Allen Downey 关于并发和使用信号量编程的书[D08]就很好（免费的参考资料）。该书包括了许多谜题，你可以研究它们，从而深入理解具体的信号量和一般的并发。成为一个并发专家需要多年的努力，学习本课程之外的内容，无疑是掌握这个领域的关键。

参考资料

[B04]“Implementing Condition Variables with Semaphores”Andrew Birrell

December 2004

一本关于在信号量上实现条件变量有多困难，以及作者和同事在此过程中犯的错误的有趣读物。因为该小组进行了大量的并发编程，所以讲述特别中肯。例如，Birrell 以编写各种线程编程指南而闻名。

[CB08]“Real-world Concurrency”Bryan Cantrill and Jeff Bonwick
ACM Queue. Volume 6, No. 5. September 2008
一篇很好的文章，来自一家以前名为 Sun 的公司的一些内核黑客，讨论了并发代码中面临的实际问题。

[CHP71]“Concurrent Control with Readers and Writers”
P.J. Courtois, F. Heymans, D.L. Parnas Communications of the ACM, 14:10, October 1971
读者—写者问题的介绍以及一个简单的解决方案。后来的工作引入了更复杂的解决方案，这里跳过了，因为它们非常复杂。

[D59]“A Note on Two Problems in Connexion with Graphs”
E. W. Dijkstra
Numerische Mathematik 1, 269271, 1959
你能相信人们在 1959 年从事算法工作吗？我们很难相信。即使在计算机用起来有趣之前，这些人都感觉到他们会改变世界……

[D68a]“Go-to Statement Considered Harmful”
E.W. Dijkstra
Communications of the ACM, volume 11(3): pages 147148, March 1968
有时被认为是软件工程领域的开始。

[D68b]“The Structure of the THE Multiprogramming System”
E.W. Dijkstra
Communications of the ACM, volume 11(5), pages 341346, 1968
最早的论文之一，指出计算机科学中的系统工作是一项引人入胜的智力活动，也为分层系统式的模块化进行了强烈辩护。

[D72]“Information Streams Sharing a Finite Buffer”
E.W. Dijkstra
Information Processing Letters 1: 179180, 1972
Dijkstra 创造了一切吗？不，但可能差不多。他当然是第一位明确写下并发代码中的问题的人。然而，操作系统设计的从业者确实知道 Dijkstra 所描述的许多问题，所以将太多东西归功于他也许是对历史的误传。

[D08]“The Little Book of Semaphores”
A.B. Downey
一本关于信号量的好书（而且免费!）。如果你喜欢这样的事情，有很多有趣的问题等待解决。

[DHO71]“Hierarchical ordering of sequential processes”
E.W. Dijkstra
介绍了许多并发问题，包括哲学家就餐问题。关于这个问题，维基百科也给出了很丰富的内容。

[GR92]“Transaction Processing: Concepts and Techniques”Jim Gray and Andreas Reuter
Morgan Kaufmann, September 1992

我们发现特别幽默的引用就在第 485 页，第 8.8 节开始处：“第一个多处理器，大约在 1960 年，就有测试并设置指令……大概是 OS 的实现者想出了正确的算法，尽管通常认为 Dijkstra 在多年后发明信号量。”

[H87]“Aspects of Cache Memory and Instruction Buffer Performance”Mark D. Hill
Ph.D. Dissertation, U.C. Berkeley, 1987
Hill 的学位论文工作，给那些痴迷于早期系统缓存的人。量化论文的一个很好的例子。

[L83]“Hints for Computer Systems Design”Butler Lampson
ACM Operating Systems Review, 15:5, October 1983
著名系统研究员 Lampson 喜欢在设计计算机系统时使用暗示。暗示经常是正确的，但可能是错误的。在这种用法中，signal()告诉等待线程它改变了等待的条件，但不要相信当等待线程唤醒时条件将处于期望的状态。在这篇关于系统设计的暗示的文章中， Lampson 的一般暗示是你应该使用暗示。这并不像听上去那么令人困惑。