简单但无法扩展

你可以看到，没有同步机制的计数器很简单，只需要很少代码就能实现。现在我们的下一个挑战是：如何让这段代码线程安全（thread safe）？图 29.2 展示了我们的做法。

```
1    typedef struct   counter_t {
2        int            value;
3        pthread_mutex_t lock;
4    } counter_t;
5
6    void init(counter_t *c) {
7        c->value = 0;
8        Pthread_mutex_init(&c->lock,  NULL);
9    }
10
11   void increment(counter_t *c) {
12       Pthread_mutex_lock(&c->lock);
13       c->value++;
14       Pthread_mutex_unlock(&c->lock);
15   }
16
17   void decrement(counter_t *c) {
18       Pthread_mutex_lock(&c->lock);
19       c->value--;
20       Pthread_mutex_unlock(&c->lock);
21   }
22
23   int get(counter_t *c) {
24       Pthread_mutex_lock(&c->lock);
25       int rc = c->value;
26       Pthread_mutex_unlock(&c->lock);
27       return rc;
28   }
```

图 29.2 有锁的计数器

这个并发计数器简单、正确。实际上，它遵循了最简单、最基本的并发数据结构中常见的数据模式：它只是加了一把锁，在调用函数操作该数据结构时获取锁，从调用返回时释放锁。这种方式类似基于观察者（monitor）[BH73]的数据结构，在调用、退出对象方法时，会自动获取锁、释放锁。

现在，有了一个并发数据结构，问题可能就是性能了。如果这个结构导致运行速度太慢，那么除了简单加锁，还需要进行优化。如果需要这种优化，那么本章的余下部分将进行探讨。请注意，如果数据结构导致的运行速度不是太慢，那就没事！如果简单的方案就能工作，就不需要精巧的设计。

为了理解简单方法的性能成本，我们运行一个基准测试，每个线程更新同一个共享计数器固定次数，然后我们改变线程数。图 29.3 给出了运行 1 个线程到 4 个线程的总耗时，其中每个线程更新 100 万次计数器。本实验是在 4 核 Intel 2.7GHz i5 CPU 的 iMac 上运行。通过增加 CPU，我们希望单位时间能够完成更多的任务。

从图 29.3 上方的曲线（标为“精确”）可以看出，同步的计数器扩展性不好。单线程完成 100 万次更新只需要很短的时间（大约 0.03s），而两个线程并发执行，每个更新 100 万次，性能下降很多（超过 5s！）。线程更多时，性能更差。

理想情况下，你会看到多处理上运行的多线程就像单线程一样快。达到这种状态称为完美扩展（perfect scaling）。虽然总工作量增多，但是并行执行后，完成任务的时间并没有增加。

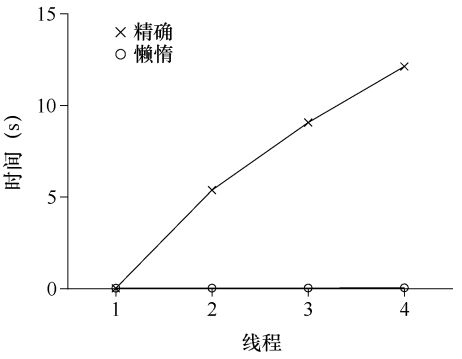

图 29.3 传统计数器与懒惰计数器

可扩展的计数

令人吃惊的是，关于如何实现可扩展的计数器，研究人员已经研究了多年[MS04]。更令人吃惊的是，最近的操作系统性能分析研究[B+10]表明，可扩展的计数器很重要。没有可扩展的计数，一些运行在 Linux 上的工作在多核机器上将遇到严重的扩展性问题。

尽管人们已经开发了多种技术来解决这一问题，我们将介绍一种特定的方法。这个方法是最近的研究提出的，称为懒惰计数器（sloppy counter）[B+10]。

懒惰计数器通过多个局部计数器和一个全局计数器来实现一个逻辑计数器，其中每个 CPU 核心有一个局部计数器。具体来说，在 4 个 CPU 的机器上，有 4 个局部计数器和 1 个全局计数器。除了这些计数器，还有锁：每个局部计数器有一个锁，全局计数器有一个。

懒惰计数器的基本思想是这样的。如果一个核心上的线程想增加计数器，那就增加它的局部计数器，访问这个局部计数器是通过对应的局部锁同步的。因为每个 CPU 有自己的局部计数器，不同 CPU 上的线程不会竞争，所以计数器的更新操作可扩展性好。

但是，为了保持全局计数器更新（以防某个线程要读取该值），局部值会定期转移给全局计数器，方法是获取全局锁，让全局计数器加上局部计数器的值，然后将局部计数器置零。

这种局部转全局的频度，取决于一个阈值，这里称为 S（表示 sloppiness）。S 越小，懒惰计数器则越趋近于非扩展的计数器。S 越大，扩展性越强，但是全局计数器与实际计数的偏差越大。我们可以抢占所有的局部锁和全局锁（以特定的顺序，避免死锁），以获得精确值，但这种方法没有扩展性。

为了弄清楚这一点，来看一个例子（见表 29.1）。在这个例子中，阈值 S 设置为 5，4 个 CPU 上分别有一个线程更新局部计数器 $L_1,\cdots, L_4$。随着时间增加，全局计数器 G 的值也会记录下来。每一段时间，局部计数器可能会增加。如果局部计数值增加到阈值 S，就把局部值转移到全局计数器，局部计数器清零。

表 29.1 追踪懒惰计数器

时间	L_1	L_2	L_3	L_4	G
0	0	0	0	0	0
1	0	0	1	1	0
2	1	0	2	1	0
3	2	0	3	1	0

续表

时间	L_1	L_2	L_3	L_4	G
4	3	0	3	2	0
5	4	1	3	3	0
6	5→0	1	3	4	5（来自 L_1）
7	0	2	4	5→0	10（来自 L_4）

图 29.3 中下方的线，是阈值 S 为 1024 时懒惰计数器的性能。性能很高，4 个处理器更新 400 万次的时间和一个处理器更新 100 万次的几乎一样。

图 29.4 展示了阈值 S 的重要性，在 4 个 CPU 上的 4 个线程，分别增加计数器 100 万次。如果 S 小，性能很差（但是全局计数器精确度高）。如果 S 大，性能很好，但是全局计数器会有延时。懒惰计数器就是在准确性和性能之间折中。

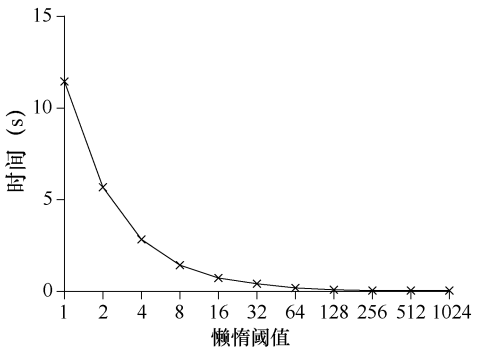

图 29.4 扩展懒惰计数器

图 29.5 是这种懒惰计数器的基本实现。阅读它，或者运行它，尝试一些例子，以便更好地理解它的原理。

```
1    typedef struct   counter_t {
2        int             global;             // global count
3        pthread_mutex_t glock;              // global lock
4        int             local[NUMCPUS];     // local count (per cpu)
5        pthread_mutex_t llock[NUMCPUS];     // ... and locks
6        int             threshold;          // update frequency
7    } counter_t;
8
9    // init: record threshold, init locks, init values
10   //       of all local counts and global count
11   void init(counter_t *c, int threshold) {
12       c->threshold = threshold;
13
14       c->global = 0;
15       pthread_mutex_init(&c->glock,  NULL);
16
17       int i;
18       for (i = 0; i < NUMCPUS; i++) {
19           c->local[i] = 0;
```

```
20          pthread_mutex_init(&c->llock[i],  NULL);
21      }
22  }
23
24  // update: usually, just grab local lock and update local amount
25  //         once local count has risen by 'threshold', grab global
26  //         lock and transfer local values to it
27  void update(counter_t *c, int threadID, int amt) {
28      pthread_mutex_lock(&c->llock[threadID]);
29      c->local[threadID] += amt;                // assumes amt > 0
30      if (c->local[threadID] >= c->threshold) { // transfer to global
31          pthread_mutex_lock(&c->glock);
32          c->global += c->local[threadID];
33          pthread_mutex_unlock(&c->glock);
34          c->local[threadID] = 0;
35      }
36      pthread_mutex_unlock(&c->llock[threadID]);
37  }
38
39  // get: just return global amount (which may not be perfect)
40  int get(counter_t *c) {
41      pthread_mutex_lock(&c->glock);
42      int val = c->global;
43      pthread_mutex_unlock(&c->glock);
44      return val; // only approximate!
45  }
```

图 29.5 懒惰计数器的实现

29.2 并发链表

接下来看一个更复杂的数据结构，链表。同样，我们从一个基础实现开始。简单起见，我们只关注链表的插入操作，其他操作比如查找、删除等就交给读者了。图 29.6 展示了这个基本数据结构的代码。

```
1   // basic node structure
2   typedef struct   node_t {
3       int                key;
4       struct   node_t        *next;
5   } node_t;
6
7   // basic list structure (one used per list)
8   typedef struct   list_t {
9       node_t                *head;
10      pthread_mutex_t    lock;
11  } list_t;
12
13  void List_Init(list_t *L) {
14      L->head = NULL;
```

```
15      pthread_mutex_init(&L->lock,  NULL);
16  }
17
18  int List_Insert(list_t *L, int key) {
19      pthread_mutex_lock(&L->lock);
20      node_t *new = malloc(sizeof(node_t));
21      if (new == NULL) {
22          perror("malloc");
23          pthread_mutex_unlock(&L->lock);
24          return -1; // fail
25      }
26      new->key  = key;
27      new->next = L->head;
28      L->head = new;
29      pthread_mutex_unlock(&L->lock);
30      return 0; // success
31  }
32
33  int List_Lookup(list_t *L, int key) {
34      pthread_mutex_lock(&L->lock);
35      node_t *curr = L->head;
36      while (curr) {
37          if (curr->key == key) {
38              pthread_mutex_unlock(&L->lock);
39              return 0; // success
40          }
41          curr = curr->next;
42      }
43      pthread_mutex_unlock(&L->lock);
44      return -1; // failure
45  }
```

图 29.6 并发链表

从代码中可以看出，代码插入函数入口处获取锁，结束时释放锁。如果 malloc 失败（在极少的时候），会有一点小问题，在这种情况下，代码在插入失败之前，必须释放锁。

事实表明，这种异常控制流容易产生错误。最近一个 Linux 内核补丁的研究表明，有 40%都是这种很少发生的代码路径（实际上，这个发现启发了我们自己的一些研究，我们从 Linux 文件系统中移除了所有内存失败的路径，得到了更健壮的系统[S+11]）。

因此，挑战来了：我们能够重写插入和查找函数，保持并发插入正确，但避免在失败情况下也需要调用释放锁吗？

在这个例子中，答案是可以。具体来说，我们调整代码，让获取锁和释放锁只环绕插入代码的真正临界区。前面的方法有效是因为部分工作实际上不需要锁，假定 malloc()是线程安全的，每个线程都可以调用它，不需要担心竞争条件和其他并发缺陷。只有在更新共享列表时需要持有锁。图 29.7 展示了这些修改的细节。

对于查找函数，进行了简单的代码调整，跳出主查找循环，到单一的返回路径。这样做减少了代码中需要获取锁、释放锁的地方，降低了代码中不小心引入缺陷（诸如在返回前忘记释放锁）的可能性。

```
1   void List_Init(list_t *L) {
2       L->head = NULL;
3       pthread_mutex_init(&L->lock,  NULL);
4   }
5
6   void List_Insert(list_t *L, int key) {
7       // synchronization not needed
8       node_t *new = malloc(sizeof(node_t));
9       if (new == NULL) {
10          perror("malloc");
11          return;
12      }
13      new->key = key;
14
15      // just lock critical section
16      pthread_mutex_lock(&L->lock);
17      new->next = L->head;
18      L->head = new;
19      pthread_mutex_unlock(&L->lock);
20  }
21
22  int List_Lookup(list_t *L, int key) {
23      int rv = -1;
24      pthread_mutex_lock(&L->lock);
25      node_t *curr = L->head;
26      while (curr) {
27          if (curr->key == key) {
28              rv = 0;
29              break;
30          }
31          curr = curr->next;
32      }
33      pthread_mutex_unlock(&L->lock);
34      return rv; // now both success and failure
35  }
```

图 29.7 重写并发链表

扩展链表

尽管我们有了基本的并发链表，但又遇到了这个链表扩展性不好的问题。研究人员发现的增加链表并发的技术中，有一种叫作过手锁（hand-over-hand locking，也叫作锁耦合，lock coupling）[MS04]。

原理也很简单。每个节点都有一个锁，替代之前整个链表一个锁。遍历链表的时候，首先抢占下一个节点的锁，然后释放当前节点的锁。

从概念上说，过手锁链表有点道理，它增加了链表操作的并发程度。但是实际上，在遍历的时候，每个节点获取锁、释放锁的开销巨大，很难比单锁的方法快。即使有大量的线程和很大的链表，这种并发的方案也不一定会比单锁的方案快。也许某种杂合的方案（一

定数量的节点用一个锁）值得去研究。

提示：更多并发不一定更快

如果方案带来了大量的开销（例如，频繁地获取锁、释放锁），那么高并发就没有什么意义。如果简单的方案很少用到高开销的调用，通常会很有效。增加更多的锁和复杂性可能会适得其反。话虽如此，有一种办法可以获得真知：实现两种方案（简单但少一点并发，复杂但多一点并发），测试它们的表现。毕竟，你不能在性能上作弊。结果要么更快，要么不快。

提示：当心锁和控制流

有一个通用建议，对并发代码和其他代码都有用，即注意控制流的变化导致函数返回和退出，或其他错误情况导致函数停止执行。因为很多函数开始就会获得锁，分配内存，或者进行其他一些改变状态的操作，如果错误发生，代码需要在返回前恢复各种状态，这容易出错。因此，最好组织好代码，减少这种模式。

29.3 并发队列

你现在知道了，总有一个标准的方法来创建一个并发数据结构：添加一把大锁。对于一个队列，我们将跳过这种方法，假定你能弄明白。

我们来看看 Michael 和 Scott [MS98]设计的、更并发的队列。图 29.8 展示了用于该队列的数据结构和代码。

```
1   typedef struct __node_t {
2       int                 value;
3       struct __node_t     *next;
4   } node_t;
5
6   typedef struct   queue_t {
7       node_t            *head;
8       node_t            *tail;
9       pthread_mutex_t    headLock;
10      pthread_mutex_t    tailLock;
11  } queue_t;
12
13  void Queue_Init(queue_t *q) {
14      node_t *tmp = malloc(sizeof(node_t));
15      tmp->next = NULL;
16      q->head = q->tail = tmp;
17      pthread_mutex_init(&q->headLock,  NULL);
18      pthread_mutex_init(&q->tailLock,  NULL);
19  }
20
21  void Queue_Enqueue(queue_t *q, int value) {
22      node_t *tmp = malloc(sizeof(node_t));
```

```
23      assert(tmp != NULL);
24      tmp->value = value;
25      tmp->next  = NULL;
26
27      pthread_mutex_lock(&q->tailLock);
28      q->tail->next = tmp;
29      q->tail = tmp;
30      pthread_mutex_unlock(&q->tailLock);
31  }
32
33  int Queue_Dequeue(queue_t *q, int *value) {
34      pthread_mutex_lock(&q->headLock);
35      node_t *tmp = q->head;
36      node_t *newHead = tmp->next;
37      if (newHead == NULL) {
38          pthread_mutex_unlock(&q->headLock);
39          return -1; // queue was empty
40      }
41      *value = newHead->value;
42      q->head = newHead;
43      pthread_mutex_unlock(&q->headLock);
44      free(tmp);
45      return 0;
46  }
```

图 29.8 Michael 和 Scott 的并发队列

仔细研究这段代码，你会发现有两个锁，一个负责队列头，另一个负责队列尾。这两个锁使得入队列操作和出队列操作可以并发执行，因为入队列只访问 tail 锁，而出队列只访问 head 锁。

Michael 和 Scott 使用了一个技巧，添加了一个假节点（在队列初始化的代码里分配的）。该假节点分开了头和尾操作。研究这段代码，或者输入、运行、测试它，以便更深入地理解它。

队列在多线程程序里广泛使用。然而，这里的队列（只是加了锁）通常不能完全满足这种程序的需求。更完善的有界队列，在队列空或者满时，能让线程等待。这是下一章探讨条件变量时集中研究的主题。读者需要看仔细了！

29.4 并发散列表

我们讨论最后一个应用广泛的并发数据结构，散列表（见图 29.9）。我们只关注不需要调整大小的简单散列表。支持调整大小还需要一些工作，留给读者作为练习。

```
1   #define BUCKETS (101)
2
3   typedef struct __hash_t {
4       list_t lists[BUCKETS];
5   } hash_t;
```

```
6
7   void Hash_Init(hash_t *H) {
8       int i;
9       for (i = 0; i < BUCKETS; i++) {
10          List_Init(&H->lists[i]);
11      }
12  }
13
14  int Hash_Insert(hash_t *H, int key) {
15      int bucket = key % BUCKETS;
16      return List_Insert(&H->lists[bucket], key);
17  }
18
19  int Hash_Lookup(hash_t *H, int key) {
20      int bucket = key % BUCKETS;
21      return List_Lookup(&H->lists[bucket], key);
22  }
```

图 29.9 并发散列表

本例的散列表使用我们之前实现的并发链表，性能特别好。每个散列桶（每个桶都是一个链表）都有一个锁，而不是整个散列表只有一个锁，从而支持许多并发操作。

图 29.10 展示了并发更新下的散列表的性能（同样在 4 CPU 的 iMac，4 个线程，每个线程分别执行 1 万～5 万次并发更新）。同时，作为比较，我们也展示了单锁链表的性能。可以看出，这个简单的并发散列表扩展性极好，而链表则相反。

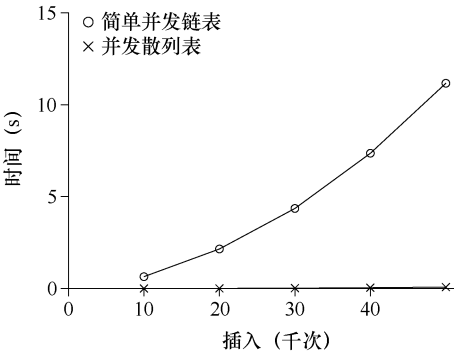

图 29.10 扩展散列表

建议：避免不成熟的优化（Knuth 定律）

实现并发数据结构时，先从最简单的方案开始，也就是加一把大锁来同步。这样做，你很可能构建了正确的锁。如果发现性能问题，那么就改进方法，只要优化到满足需要即可。正如 Knuth 的著名说法“不成熟的优化是所有坏事的根源。”

许多操作系统，在最初过渡到多处理器时都是用一把大锁，包括 Sun 和 Linux。在 Linux 中，这个锁甚至有个名字，叫作 BKL（大内核锁，big kernel lock）。这个方案在很多年里都很有效，直到多 CPU 系统普及，内核只允许一个线程活动成为性能瓶颈。终于到了为这些系统优化并发性能的时候了。Linux 采用了简单的方案，把一个锁换成多个。Sun 则更为激进，实现了一个最开始就能并发的新系统，Solaris。读者可以通过 Linux 和 Solaris 的内核资料了解更多信息[BC05，MM00]。

29.5 小结

我们已经介绍了一些并发数据结构，从计数器到链表队列，最后到大量使用的散列表。

同时，我们也学习到：控制流变化时注意获取锁和释放锁；增加并发不一定能提高性能；有性能问题的时候再做优化。关于最后一点，避免不成熟的优化（premature optimization），对于所有关心性能的开发者都有用。我们让整个应用的某一小部分变快，却没有提高整体性能，其实没有价值。

当然，我们只触及了高性能数据结构的皮毛。Moir 和 Shavit 的调查提供了更多信息，包括指向其他来源的链接[MS04]。特别是，你可能会对其他结构感兴趣（比如 B 树），那么数据库课程会是一个不错的选择。你也可能对根本不用传统锁的技术感兴趣。这种非阻塞数据结构是有意义的，在常见并发问题的章节中，我们会稍稍涉及。但老实说这是一个广泛领域的知识，远非本书所能覆盖。感兴趣的读者可以自行研究。

参考资料

[B+10]“An Analysis of Linux Scalability to Many Cores”
Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, Nickolai Zeldovich
OSDI ’10, Vancouver, Canada, October 2010
关于 Linux 在多核机器上的表现以及对一些简单的解决方案的很好的研究。

[BH73]“Operating System Principles”Per Brinch Hansen, Prentice-Hall, 1973
最早的操作系统图书之一。当然领先于它的时代。将观察者作为并发原语引入。

[BC05]“Understanding the Linux Kernel (Third Edition)”Daniel P. Bovet and Marco Cesati
O’Reilly Media, November 2005
关于 Linux 内核的经典书籍。你应该阅读它。

[L+13]“A Study of Linux File System Evolution”
Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Shan Lu FAST ’13, San Jose, CA, February 2013
我们的论文研究了近十年来 Linux 文件系统的每个补丁。论文中有很多有趣的发现，读读看！这项工作很痛苦，这位研究生 Lanyue Lu 不得不亲自查看每一个补丁，以了解它们做了什么。

[MS98]“Nonblocking Algorithms and Preemption-safe Locking on Multiprogrammed Sharedmemory Multiprocessors”
M. Michael and M. Scott
Journal of Parallel and Distributed Computing, Vol. 51, No. 1, 1998
Scott 教授和他的学生多年来一直处于并发算法和数据结构的前沿。浏览他的网页，并阅读他的大量的论文和书籍，可以了解更多信息。

[MS04]“Concurrent Data Structures”Mark Moir and Nir Shavit
In Handbook of Data Structures and Applications
(Editors D. Metha and S.Sahni) Chapman and Hall/CRC Press, 2004

关于并发数据结构的简短但相对全面的参考。虽然它缺少该领域的一些最新作品（由于它的时间），但仍然是一个令人难以置信的有用的参考。

[MM00]“Solaris Internals: Core Kernel Architecture”Jim Mauro and Richard McDougall
Prentice Hall, October 2000
Solaris 之书。如果你想详细了解 Linux 之外的其他内容，就应该阅读本书。

[S+11]“Making the Common Case the Only Case with Anticipatory Memory Allocation”Swaminathan Sundararaman, Yupu Zhang, Sriram Subramanian,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau FAST’11, San Jose, CA, February 2011
我们关于从内核代码路径中删除可能失败的 malloc 调用的工作。其主要想法是在做任何工作之前分配所有可能需要的内存，从而避免存储栈内部发生故障。

第 30 章　条件变量

到目前为止，我们已经形成了锁的概念，看到了如何通过硬件和操作系统支持的正确组合来实现锁。然而，锁并不是并发程序设计所需的唯一原语。

具体来说，在很多情况下，线程需要检查某一条件（condition）满足之后，才会继续运行。例如，父线程需要检查子线程是否执行完毕 [这常被称为 join()]。这种等待如何实现呢？我们来看如图 30.1 所示的代码。

```
1    void *child(void *arg) {
2        printf("child\n");
3        // XXX how to indicate we are done?
4        return NULL;
5    }
6
7    int main(int argc, char *argv[]) {
8        printf("parent: begin\n");
9        pthread_t c;
10       Pthread_create(&c, NULL, child, NULL); // create child
11       // XXX how to wait for child?
12       printf("parent: end\n");
13       return 0;
14   }
```

图 30.1　父线程等待子线程

我们期望能看到这样的输出：

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

我们可以尝试用一个共享变量，如图 30.2 所示。这种解决方案一般能工作，但是效率低下，因为主线程会自旋检查，浪费 CPU 时间。我们希望有某种方式让父线程休眠，直到等待的条件满足（即子线程完成执行）。

```
1    volatile int done = 0;
2
3    void *child(void *arg) {
4        printf("child\n");
5        done = 1;
6        return NULL;
7    }
8
9    int main(int argc, char *argv[]) {
10       printf("parent: begin\n");
```

```
11      pthread_t c;
12      Pthread_create(&c, NULL, child, NULL); // create child
13      while (done == 0)
14          ; // spin
15      printf("parent: end\n");
16      return 0;
17  }
```

图 30.2 父线程等待子线程：基于自旋的方案

关键问题：如何等待一个条件？

多线程程序中，一个线程等待某些条件是很常见的。简单的方案是自旋直到条件满足，这是极其低效的，某些情况下甚至是错误的。那么，线程应该如何等待一个条件？

30.1 定义和程序

线程可以使用条件变量（condition variable），来等待一个条件变成真。条件变量是一个显式队列，当某些执行状态（即条件，condition）不满足时，线程可以把自己加入队列，等待（waiting）该条件。另外某个线程，当它改变了上述状态时，就可以唤醒一个或者多个等待线程（通过在该条件上发信号），让它们继续执行。Dijkstra 最早在“私有信号量”[D01]中提出这种思想。Hoare 后来在关于观察者的工作中，将类似的思想称为条件变量[H74]。

要声明这样的条件变量，只要像这样写：pthread_cond_t c;，这里声明 c 是一个条件变量（注意：还需要适当的初始化）。条件变量有两种相关操作：wait()和 signal()。线程要睡眠的时候，调用 wait()。当线程想唤醒等待在某个条件变量上的睡眠线程时，调用 signal()。具体来说，POSIX 调用如图 30.3 所示。

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t  *c);
```

```
1   int done  = 0;
2   pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3   pthread_cond_t c  = PTHREAD_COND_INITIALIZER;
4
5   void thr_exit() {
6       Pthread_mutex_lock(&m);
7       done = 1;
8       Pthread_cond_signal(&c);
9       Pthread_mutex_unlock(&m);
10   }
11
12   void *child(void *arg) {
13       printf("child\n");
14       thr_exit();
15       return NULL;
16  }
```

```
17
18  void thr_join() {
19      Pthread_mutex_lock(&m);
20      while (done == 0)
21          Pthread_cond_wait(&c, &m);
22      Pthread_mutex_unlock(&m);
23  }
24
25  int main(int argc, char *argv[]) {
26      printf("parent: begin\n");
27      pthread_t p;
28      Pthread_create(&p, NULL, child, NULL);
29      thr_join();
30      printf("parent: end\n");
31      return 0;
32  }
```

图 30.3 父线程等待子线程：使用条件变量

我们常简称为 wait()和 signal()。你可能注意到一点，wait()调用有一个参数，它是互斥量。它假定在 wait()调用时，这个互斥量是已上锁状态。wait()的职责是释放锁，并让调用线程休眠（原子地）。当线程被唤醒时（在另外某个线程发信号给它后），它必须重新获取锁，再返回调用者。这样复杂的步骤也是为了避免在线程陷入休眠时，产生一些竞态条件。我们观察一下图 30.3 中 join 问题的解决方法，以加深理解。

有两种情况需要考虑。第一种情况是父线程创建出子线程，但自己继续运行（假设只有一个处理器），然后马上调用 thr_join()等待子线程。在这种情况下，它会先获取锁，检查子进程是否完成（还没有完成），然后调用 wait()，让自己休眠。子线程最终得以运行，打印出“child”，并调用 thr_exit()函数唤醒父进程，这段代码会在获得锁后设置状态变量 done，然后向父线程发信号唤醒它。最后，父线程会运行（从 wait()调用返回并持有锁），释放锁，打印出“parent:end”。

第二种情况是，子线程在创建后，立刻运行，设置变量 done 为 1，调用 signal 函数唤醒其他线程（这里没有其他线程），然后结束。父线程运行后，调用 thr_join()时，发现 done 已经是 1 了，就直接返回。

最后一点说明：你可能看到父线程使用了一个 while 循环，而不是 if 语句来判断是否需要等待。虽然从逻辑上来说没有必要使用循环语句，但这样做总是好的（后面我们会加以说明）。

为了确保理解 thr_exit()和 thr_join()中每个部分的重要性，我们来看一些其他的实现。首先，你可能会怀疑状态变量 done 是否需要。代码像下面这样如何？正确吗？

```
1   void thr_exit() {
2       Pthread_mutex_lock(&m);
3       Pthread_cond_signal(&c);
4       Pthread_mutex_unlock(&m);
5   }
6
7   void thr_join() {
8       Pthread_mutex_lock(&m);
9       Pthread_cond_wait(&c, &m);
```

```
10        Pthread_mutex_unlock(&m);
11    }
```

这段代码是有问题的。假设子线程立刻运行，并且调用 thr_exit()。在这种情况下，子线程发送信号，但此时却没有在条件变量上睡眠等待的线程。父线程运行时，就会调用 wait 并卡在那里，没有其他线程会唤醒它。通过这个例子，你应该认识到变量 done 的重要性，它记录了线程有兴趣知道的值。睡眠、唤醒和锁都离不开它。

下面是另一个糟糕的实现。在这个例子中，我们假设线程在发信号和等待时都不加锁。会发生什么问题？想想看！

```
1     void thr_exit() {
2         done = 1;
3         Pthread_cond_signal(&c);
4     }
5
6     void thr_join() {
7         if (done == 0)
8             Pthread_cond_wait(&c);
9     }
```

这里的问题是一个微妙的竞态条件。具体来说，如果父进程调用 thr_join()，然后检查完 done 的值为 0，然后试图睡眠。但在调用 wait 进入睡眠之前，父进程被中断。子线程修改变量 done 为 1，发出信号，同样没有等待线程。父线程再次运行时，就会长眠不醒，这就惨了。

提示：发信号时总是持有锁

尽管并不是所有情况下都严格需要，但有效且简单的做法，还是在使用条件变量发送信号时持有锁。虽然上面的例子是必须加锁的情况，但也有一些情况可以不加锁，而这可能是你应该避免的。因此，为了简单，请在调用 signal 时持有锁（hold the lock when calling signal）。

这个提示的反面，即调用 wait 时持有锁，不只是建议，而是 wait 的语义强制要求的。因为 wait 调用总是假设你调用它时已经持有锁、调用者睡眠之前会释放锁以及返回前重新持有锁。因此，这个提示的一般化形式是正确的：调用 signal 和 wait 时要持有锁（hold the lock when calling signal or wait），你会保持身心健康的。

希望通过这个简单的 join 示例，你可以看到使用条件变量的一些基本要求。为了确保你能理解，我们现在来看一个更复杂的例子：生产者/消费者（producer/consumer）或有界缓冲区（bounded-buffer）问题。

30.2 生产者/消费者（有界缓冲区）问题

本章要面对的下一个问题，是生产者/消费者（producer/consumer）问题，也叫作有界缓冲区（bounded buffer）问题。这一问题最早由 Dijkstra 提出[D72]。实际上也正是通过研究这一问题，Dijkstra 和他的同事发明了通用的信号量（它可用作锁或条件变量）[D01]。

假设有一个或多个生产者线程和一个或多个消费者线程。生产者把生成的数据项放入