然而，他的许多基础性工作早在多年前就在埃因霍温理工大学（Technische Hochshule of Eindhoven，THE）进行，其中包括这篇著名的关于“Cooperating Sequential Processes”的论文，该论文基本上概述了编写多线程程序必须考虑的所有问题。Dijkstra 是在以他的学校命名的操作系统“THE”上工作时做出这些研究发现的。“THE”读作 THE，而不是“the”。

[GR92]“Transaction Processing: Concepts and Techniques”Jim Gray and Andreas Reuter
Morgan Kaufmann, September 1992
这本书是交易处理的宝典，由该领域的传奇人物之一 Jim Gray 撰写。出于这个原因，它也被认为是 Jim Gray 的“大脑转储”，其中写下了他所知道的关于数据库管理系统如何工作的一切。难过的是，Gray 在几年前不幸去世了，我们中的许多人（包括这本书的合著者）失去了一位朋友和伟大的导师。我们在研究生学习期间有幸与 Gray 交流过。

[L+93]“Atomic Transactions”
Nancy Lynch, Michael Merritt, William Weihl, Alan Fekete Morgan Kaufmann, August 1993
这是一本关于分布式系统原子事务的一些理论和实践的不错教材。对于一些人来说，也许有点正式，但在这里可以找到很多很好的材料。

[SR05]“Advanced Programming in the UNIX Environment”
我们说过很多次，购买这本书，然后一点一点阅读，建议在睡前阅读。这样，你实际上会更快地入睡。更重要的是，可以多学一点如何成为一名称职的 UNIX 程序员。

作业

x86.py 这个程序让你看到不同的线程交替如何导致或避免竞态条件。请参阅 README 文件，了解程序如何工作及其基本输入的详细信息，然后回答以下问题。

问题

1．开始，我们来看一个简单的程序，“loop.s”。首先，阅读这个程序，看看你是否能理解它：cat loop.s。然后，用这些参数运行它：

```
./x86.py -p loop.s -t 1 -i 100 -R dx
```

这指定了一个单线程，每 100 条指令产生一个中断，并且追踪寄存器%dx。你能弄清楚%dx 在运行过程中的价值吗？你有答案之后，运行上面的代码并使用-c 标志来检查你的答案。注意答案的左边显示了右侧指令运行后寄存器的值（或内存的值）。

2．现在运行相同的代码，但使用这些标志：

```
./x86.py -p loop.s -t 2 -i 100 -a dx=3,dx=3 -R dx
```

这指定了两个线程，并将每个%dx 寄存器初始化为 3。%dx 会看到什么值？使用-c 标志

运行以查看答案。多个线程的存在是否会影响计算？这段代码有竞态条件吗？

3．现在运行以下命令：

```
./x86.py -p loop.s -t 2 -i 3 -r -a dx=3,dx=3 -R dx
```

这使得中断间隔非常小且随机。使用不同的种子和-s 来查看不同的交替。中断频率是否会改变这个程序的行为？

4．接下来我们将研究一个不同的程序（looping-race-nolock.s）。

该程序访问位于内存地址 2000 的共享变量。简单起见，我们称这个变量为 *x*。使用单线程运行它，并确保你了解它的功能，如下所示：

```
./x86.py -p looping-race-nolock.s -t 1 -M 2000
```

在整个运行过程中，*x*（即内存地址为 2000）的值是多少？使用-c 来检查你的答案。

5．现在运行多个迭代和线程：

```
./x86.py -p looping-race-nolock.s -t 2 -a bx=3 -M 2000
```

你明白为什么每个线程中的代码循环 3 次吗？*x* 的最终值是什么？

6．现在以随机中断间隔运行：

```
./x86.py -p looping-race-nolock.s -t 2 -M 2000 -i 4 -r -s 0
```

然后改变随机种子，设置-s 1，然后-s 2 等。只看线程交替，你能说出 *x* 的最终值是什么吗？中断的确切位置是否重要？在哪里发生是安全的？中断在哪里会引起麻烦？换句话说，临界区究竟在哪里？

7．现在使用固定的中断间隔来进一步探索程序。

运行：

```
./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 1
```

看看你能否猜测共享变量 *x* 的最终值是什么。当你改用-i 2，-i 3 等标志呢？对于哪个中断间隔，程序会给出“正确的”最终答案？

8．现在为更多循环运行相同的代码（例如 set -a bx = 100）。使用-i 标志设置哪些中断间隔会导致“正确”结果？哪些间隔会导致令人惊讶的结果？

9．我们来看本作业中最后一个程序（wait-for-me.s）。

像这样运行代码：

```
./x86.py -p wait-for-me.s -a ax=1,ax=0 -R ax -M 2000
```

这将线程 0 的%ax 寄存器设置为 1，并将线程 1 的值设置为 0，在整个运行过程中观察%ax 和内存位置 2000 的值。代码的行为应该如何？线程使用的 2000 位置的值如何？它的最终值是什么？

10．现在改变输入：

```
./x86.py -p wait-for-me.s -a ax=0,ax=1 -R ax -M 2000
```

线程行为如何？线程 0 在做什么？改变中断间隔（例如，-i 1000，或者可能使用随机间隔）会如何改变追踪结果？程序是否高效地使用了 CPU？

第 27 章　插叙：线程 API

本章介绍了主要的线程 API。后续章节也会进一步介绍如何使用 API。更多的细节可以参考其他书籍和在线资源[B89，B97，B+96，K+96]。随后的章节会慢慢介绍锁和条件变量的概念，因此本章可以作为参考。

关键问题：如何创建和控制线程？

操作系统应该提供哪些创建和控制线程的接口？这些接口如何设计得易用和实用？

27.1　线程创建

编写多线程程序的第一步就是创建新线程，因此必须存在某种线程创建接口。在 POSIX 中，很简单：

```
#include <pthread.h>
int
pthread_create(      pthread_t *         thread,
               const pthread_attr_t *    attr,
                     void *              (*start_routine)(void*),
                     void *              arg);
```

这个函数声明可能看起来有一点复杂（尤其是如果你没在 C 中用过函数指针），但实际上它并不差。该函数有 4 个参数：thread、attr、start_routine 和 arg。第一个参数 thread 是指向 pthread_t 结构类型的指针，我们将利用这个结构与该线程交互，因此需要将它传入 pthread_create()，以便将它初始化。

第二个参数 attr 用于指定该线程可能具有的任何属性。一些例子包括设置栈大小，或关于该线程调度优先级的信息。一个属性通过单独调用 pthread_attr_init()来初始化。有关详细信息，请参阅手册。但是，在大多数情况下，默认值就行。在这个例子中，我们只需传入 NULL。

第三个参数最复杂，但它实际上只是问：这个线程应该在哪个函数中运行？在 C 中，我们把它称为一个函数指针（function pointer），这个指针告诉我们需要以下内容：一个函数名称（start_routine），它被传入一个类型为 void *的参数（start_routine 后面的括号表明了这一点），并且它返回一个 void *类型的值（即一个 void 指针）。

如果这个函数需要一个整数参数，而不是一个 void 指针，那么声明看起来像这样：

```
int pthread_create(..., // first two args are the same
                   void *    (*start_routine)(int),
                   int       arg);
```

如果函数接受 void 指针作为参数，但返回一个整数，函数声明会变成：

```
int pthread_create(..., // first two args are the same
                   int       (*start_routine)(void *),
                   void *     arg);
```

最后，第四个参数 arg 就是要传递给线程开始执行的函数的参数。你可能会问：为什么我们需要这些 void 指针？好吧，答案很简单：将 void 指针作为函数的参数 start_routine，允许我们传入任何类型的参数，将它作为返回值，允许线程返回任何类型的结果。

下面来看图 27.1 中的例子。这里我们只是创建了一个线程，传入两个参数，它们被打包成一个我们自己定义的类型（myarg_t）。该线程一旦创建，可以简单地将其参数转换为它所期望的类型，从而根据需要将参数解包。

```
int pthread_join(pthread_t thread, void **value_ptr);
```

```
1    #include <pthread.h>
2
3    typedef struct   myarg_t {
4        int a;
5        int b;
6    } myarg_t;
7
8    void *mythread(void *arg) {
9        myarg_t *m = (myarg_t *) arg;
10       printf("%d %d\n", m->a, m->b);
11       return NULL;
12   }
13
14   int
15   main(int argc, char *argv[]) {
16       pthread_t p;
17       int rc;
18
19       myarg_t args;
20       args.a = 10;
21       args.b = 20;
22       rc = pthread_create(&p, NULL, mythread, &args);
23       ...
24   }
```

图 27.1 创建线程

它就在那里！一旦你创建了一个线程，你确实拥有了另一个活着的执行实体，它有自己的调用栈，与程序中所有当前存在的线程在相同的地址空间内运行。好玩的事开始了！

27.2 线程完成

上面的例子展示了如何创建一个线程。但是，如果你想等待线程完成，会发生什么情

况？你需要做一些特别的事情来等待完成。具体来说，你必须调用函数 pthread_join()。

该函数有两个参数。第一个是 pthread_t 类型，用于指定要等待的线程。这个变量是由线程创建函数初始化的（当你将一个指针作为参数传递给 pthread_create()时）。如果你保留了它，就可以用它来等待该线程终止。

第二个参数是一个指针，指向你希望得到的返回值。因为函数可以返回任何东西，所以它被定义为返回一个指向 void 的指针。因为 pthread_join()函数改变了传入参数的值，所以你需要传入一个指向该值的指针，而不只是该值本身。

我们来看另一个例子（见图 27.2）。在代码中，再次创建单个线程，并通过 myarg_t 结构传递一些参数。对于返回值，使用 myret_t 型。当线程完成运行时，主线程已经在 pthread_join()函数内等待了[①]。然后会返回，我们可以访问线程返回的值，即在 myret_t 中的内容。

有几点需要说明。首先，我们常常不需要这样痛苦地打包、解包参数。如果我们不需要参数，创建线程时传入 NULL 即可。类似的，如果不需要返回值，那么 pthread_join()调用也可以传入 NULL。

```
1   #include <stdio.h>
2   #include <pthread.h>
3   #include <assert.h>
4   #include <stdlib.h>
5
6   typedef struct   myarg_t {
7       int a;
8       int b;
9   } myarg_t;
10
11  typedef struct   myret_t {
12      int x;
13      int y;
14  } myret_t;
15
16  void *mythread(void *arg) {
17      myarg_t *m = (myarg_t *) arg;
18      printf("%d %d\n", m->a, m->b);
19      myret_t *r = Malloc(sizeof(myret_t));
20      r->x = 1;
21      r->y = 2;
22      return (void *) r;
23  }
24
25  int
26  main(int argc, char *argv[]) {
27      int rc;
28      pthread_t p;
29      myret_t *m;
```

① 注意我们在这里使用了包装的函数。具体来说，我们调用了 Malloc()、Pthread_join()和 Pthread_create()，它们只是调用了与它们命名相似的小写版本，并确保函数不会返回任何意外。

```
30
31        myarg_t args;
32        args.a = 10;
33        args.b = 20;
34        Pthread_create(&p, NULL, mythread, &args);
35        Pthread_join(p, (void **) &m);
36        printf("returned %d %d\n", m->x, m->y);
37        return 0;
38    }
```

图 27.2 等待线程完成

其次，如果我们只传入一个值（例如，一个 int），也不必将它打包为一个参数。图 27.3 展示了一个例子。在这种情况下，更简单一些，因为我们不必在结构中打包参数和返回值。

```
void *mythread(void *arg) {
    int m = (int) arg;
    printf("%d\n", m);
    return (void *) (arg + 1);
}

int main(int argc, char *argv[]) {
    pthread_t p;
    int rc, m;
    Pthread_create(&p, NULL, mythread, (void *) 100);
    Pthread_join(p, (void **) &m);
    printf("returned %d\n", m);
    return 0;
}
```

图 27.3 较简单的向线程传递参数示例

再次，我们应该注意，必须非常小心如何从线程返回值。特别是，永远不要返回一个指针，并让它指向线程调用栈上分配的东西。如果这样做，你认为会发生什么？（想一想！）下面是一段危险的代码示例，对图 27.2 中的示例做了修改。

```
1    void *mythread(void *arg) {
2        myarg_t *m = (myarg_t *) arg;
3        printf("%d %d\n", m->a, m->b);
4        myret_t r; // ALLOCATED ON STACK: BAD!
5        r.x = 1;
6        r.y = 2;
7        return (void *) &r;
8    }
```

在这个例子中，变量 *r* 被分配在 mythread 的栈上。但是，当它返回时，该值会自动释放（这就是栈使用起来很简单的原因！），因此，将指针传回现在已释放的变量将导致各种不好的结果。当然，当你打印出你以为的返回值时，你可能会感到惊讶（但不一定！）。试试看，自己找出真相[①]！

① 幸运的是，编译器 gcc 在编译这样的代码时可能会报警，这是注意编译器警告的又一个原因。

最后，你可能会注意到，使用 pthread_create()创建线程，然后立即调用 pthread_join()，这是创建线程的一种非常奇怪的方式。事实上，有一个更简单的方法来完成这个任务，它被称为过程调用（procedure call）。显然，我们通常会创建不止一个线程并等待它完成，否则根本没有太多的用途。

我们应该注意，并非所有多线程代码都使用 join 函数。例如，多线程 Web 服务器可能会创建大量工作线程，然后使用主线程接受请求，并将其无限期地传递给工作线程。因此这样的长期程序可能不需要 join。然而，创建线程来（并行）执行特定任务的并行程序，很可能会使用 join 来确保在退出或进入下一阶段计算之前完成所有这些工作。

27.3 锁

除了线程创建和 join 之外，POSIX 线程库提供的最有用的函数集，可能是通过锁（lock）来提供互斥进入临界区的那些函数。这方面最基本的一对函数是：

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

函数应该易于理解和使用。如果你意识到有一段代码是一个临界区，就需要通过锁来保护，以便像需要的那样运行。你大概可以想象代码的样子：

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

这段代码的意思是：如果在调用 pthread_mutex_lock()时没有其他线程持有锁，线程将获取该锁并进入临界区。如果另一个线程确实持有该锁，那么尝试获取该锁的线程将不会从该调用返回，直到获得该锁（意味着持有该锁的线程通过解锁调用释放该锁）。当然，在给定的时间内，许多线程可能会卡住，在获取锁的函数内部等待。然而，只有获得锁的线程才应该调用解锁。

遗憾的是，这段代码有两个重要的问题。第一个问题是缺乏正确的初始化（lack of proper initialization）。所有锁必须正确初始化，以确保它们具有正确的值，并在锁和解锁被调用时按照需要工作。

对于 POSIX 线程，有两种方法来初始化锁。一种方法是使用 PTHREAD_MUTEX_INITIALIZER，如下所示：

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

这样做会将锁设置为默认值，从而使锁可用。初始化的动态方法（即在运行时）是调用 pthread_mutex_init()，如下所示：

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

此函数的第一个参数是锁本身的地址，而第二个参数是一组可选属性。请你自己去详

细了解这些属性。传入 NULL 就是使用默认值。无论哪种方式都有效，但我们通常使用动态（后者）方法。请注意，当你用完锁时，还应该相应地调用 pthread_mutex_destroy()，所有细节请参阅手册。

上述代码的第二个问题是在调用获取锁和释放锁时没有检查错误代码。就像 UNIX 系统中调用的任何库函数一样，这些函数也可能会失败！如果你的代码没有正确地检查错误代码，失败将会静静地发生，在这种情况下，可能会允许多个线程进入临界区。至少要使用包装的函数，它对函数成功加上断言（见图 27.4）。更复杂的（非玩具）程序，在出现问题时不能简单地退出，应该检查失败并在获取锁或释放锁未成功时执行适当的操作。

```
// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
  int rc = pthread_mutex_lock(mutex);
  assert(rc == 0);
}
```

图 27.4 包装函数示例

获取锁和释放锁函数不是 pthread 与锁进行交互的仅有的函数。特别是，这里有两个你可能感兴趣的函数：

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            struct timespec *abs_timeout);
```

这两个调用用于获取锁。如果锁已被占用，则 trylock 版本将失败。获取锁的 timedlock 定版本会在超时或获取锁后返回，以先发生者为准。因此，具有零超时的 timedlock 退化为 trylock 的情况。通常应避免使用这两种版本，但有些情况下，避免卡在（可能无限期的）获取锁的函数中会很有用，我们将在以后的章节中看到（例如，当我们研究死锁时）。

27.4 条件变量

所有线程库还有一个主要组件（当然 POSIX 线程也是如此），就是存在一个条件变量（condition variable）。当线程之间必须发生某种信号时，如果一个线程在等待另一个线程继续执行某些操作，条件变量就很有用。希望以这种方式进行交互的程序使用两个主要函数：

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

要使用条件变量，必须另外有一个与此条件相关的锁。在调用上述任何一个函数时，应该持有这个锁。

第一个函数 pthread_cond_wait()使调用线程进入休眠状态，因此等待其他线程发出信号，通常当程序中的某些内容发生变化时，现在正在休眠的线程可能会关心它。典型的用法如下所示：

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond = PTHREAD_COND_INITIALIZER;
```

```
Pthread_mutex_lock(&lock);
while (ready == 0)
    Pthread_cond_wait(&cond, &lock);
Pthread_mutex_unlock(&lock);
```

在这段代码中，在初始化相关的锁和条件之后[①]，一个线程检查变量 ready 是否已经被设置为零以外的值。如果没有，那么线程只是简单地调用等待函数以便休眠，直到其他线程唤醒它。

唤醒线程的代码运行在另外某个线程中，像下面这样：

```
Pthread_mutex_lock(&lock);
ready = 1;
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);
```

关于这段代码有一些注意事项。首先，在发出信号时（以及修改全局变量 ready 时），我们始终确保持有锁。这确保我们不会在代码中意外引入竞态条件。

其次，你可能会注意到等待调用将锁作为其第二个参数，而信号调用仅需要一个条件。造成这种差异的原因在于，等待调用除了使调用线程进入睡眠状态外，还会让调用者睡眠时释放锁。想象一下，如果不是这样：其他线程如何获得锁并将其唤醒？但是，在被唤醒之后返回之前，pthread_cond_wait()会重新获取该锁，从而确保等待线程在等待序列开始时获取锁与结束时释放锁之间运行的任何时间，它持有锁。

最后一点需要注意：等待线程在 while 循环中重新检查条件，而不是简单的 if 语句。在后续章节中研究条件变量时，我们会详细讨论这个问题，但是通常使用 while 循环是一件简单而安全的事情。虽然它重新检查了这种情况（可能会增加一点开销），但有一些 pthread 实现可能会错误地唤醒等待的线程。在这种情况下，没有重新检查，等待的线程会继续认为条件已经改变。因此，将唤醒视为某种事物可能已经发生变化的暗示，而不是绝对的事实，这样更安全。

请注意，有时候线程之间不用条件变量和锁，用一个标记变量会看起来很简单，很吸引人。例如，我们可以重写上面的等待代码，像这样：

```
while (ready == 0)
    ; // spin
```

相关的发信号代码看起来像这样：

```
ready = 1;
```

千万不要这么做。首先，多数情况下性能差（长时间的自旋浪费 CPU）。其次，容易出错。最近的研究[X+10]显示，线程之间通过标志同步（像上面那样），出错的可能性让人吃惊。在那项研究中，这些不正规的同步方法半数以上都是有问题的。不要偷懒，就算你想到可以不用条件变量，还是用吧。

如果条件变量听起来让人迷惑，也不要太担心。后面的章节会详细介绍。在此之前，只要知道它们存在，并对为什么要使用它们有一些概念即可。

---

① 请注意，可以使用 pthread_cond_init()（和对应的 pthread_cond_destroy()调用），而不是使用静态初始化程序 PTHREAD_COND_INITIALIZER。听起来像是工作更多了？是的。

27.5 编译和运行

本章所有代码很容易运行。代码需要包括头文件 pthread.h 才能编译。链接时需要 pthread 库，增加-pthread 标记。

例如，要编译一个简单的多线程程序，只需像下面这样做：

```
prompt> gcc -o main main.c -Wall -pthread
```

只要 main.c 包含 pthreads 头文件，你就已经成功地编译了一个并发程序。像往常一样，它是否能工作完全是另一回事。

27.6 小结

我们介绍了基本的 pthread 库，包括线程创建，通过锁创建互斥执行，通过条件变量的信号和等待。要想写出健壮高效的多线程代码，只需要耐心和万分小心！

本章结尾我们给出编写一些多线程代码的建议（参见补充内容）。API 的其他方面也很有趣。如果需要更多信息，请在 Linux 系统上输入 man -k pthread，查看构成整个接口的超过一百个 API。但是，这里讨论的基础知识应该让你能够构建复杂的（并且希望是正确的和高性能的）多线程程序。线程难的部分不是 API，而是如何构建并发程序的棘手逻辑。请继续阅读以了解更多信息。

补充：线程 API 指导

当你使用 POSIX 线程库（或者实际上，任何线程库）来构建多线程程序时，需要记住一些小而重要的事情：

- **保持简洁**。最重要的一点，线程之间的锁和信号的代码应该尽可能简洁。复杂的线程交互容易产生缺陷。
- **让线程交互减到最少**。尽量减少线程之间的交互。每次交互都应该想清楚，并用验证过的、正确的方法来实现（很多方法会在后续章节中学习）。
- **初始化锁和条件变量**。未初始化的代码有时工作正常，有时失败，会产生奇怪的结果。
- **检查返回值**。当然，任何 C 和 UNIX 的程序，都应该检查返回值，这里也是一样。否则会导致古怪而难以理解的行为，让你尖叫，或者痛苦地揪自己的头发。
- **注意传给线程的参数和返回值**。具体来说，如果传递在栈上分配的变量的引用，可能就是在犯错误。
- **每个线程都有自己的栈**。类似于上一条，记住每一个线程都有自己的栈。因此，线程局部变量应该是线程私有的，其他线程不应该访问。线程之间共享数据，值要在堆（heap）或者其他全局可访问的位置。
- **线程之间总是通过条件变量发送信号**。切记不要用标记变量来同步。
- **多查手册**。尤其是 Linux 的 pthread 手册，有更多的细节、更丰富的内容。请仔细阅读！

参考资料

[B89]“An Introduction to Programming with Threads”Andrew D. Birrell
DEC Technical Report,January, 1989
它是线程编程的经典，但内容较陈旧。不过，仍然值得一读，而且是免费的。

[B97]“Programming with POSIX Threads”David R. Butenhof
Addison-Wesley, May 1997
又是一本关于编程的书。

[B+96]“PThreads Programming: A POSIX Standard for Better Multiprocessing”
Dick Buttlar, Jacqueline Farrell, Bradford Nichols O’Reilly, September 1996
O’Reilly 出版的一本不错的书。我们的书架当然包含了这家公司的大量书籍，其中包括一些关于 Perl、Python 和 JavaScript 的优秀产品（特别是 Crockford 的《JavaScript: The Good Parts”）。

[K+96]“Programming With Threads”
Steve Kleiman, Devang Shah, Bart Smaalders Prentice Hall, January 1996
这可能是这个领域较好的书籍之一。从当地图书馆借阅，或从老一辈程序员那里“偷”来读。认真地说，只要向老一辈程序员借的话，他们会借给你的，不用担心。

[X+10]“Ad Hoc Synchronization Considered Harmful”
Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, Zhiqiang Ma OSDI 2010, Vancouver, Canada
本文展示了看似简单的同步代码是如何导致大量错误的。使用条件变量并正确地发送信号！

第 28 章　锁

通过对并发的介绍，我们看到了并发编程的一个最基本问题：我们希望原子式执行一系列指令，但由于单处理器上的中断（或者多个线程在多处理器上并发执行），我们做不到。本章介绍了锁（lock），直接解决这一问题。程序员在源代码中加锁，放在临界区周围，保证临界区能够像单条原子指令一样执行。

28.1　锁的基本思想

举个例子，假设临界区像这样，典型的更新共享变量：

```
balance = balance + 1;
```

当然，其他临界区也是可能的，比如为链表增加一个元素，或对共享结构的复杂更新操作。为了使用锁，我们给临界区增加了这样一些代码：

```
1    lock_t mutex; // some globally-allocated lock 'mutex'
2    ...
3    lock(&mutex);
4    balance = balance + 1;
5    unlock(&mutex);
```

锁就是一个变量，因此我们需要声明一个某种类型的锁变量（lock variable，如上面的 mutex），才能使用。这个锁变量（简称锁）保存了锁在某一时刻的状态。它要么是可用的（available，或 unlocked，或 free），表示没有线程持有锁，要么是被占用的（acquired，或 locked，或 held），表示有一个线程持有锁，正处于临界区。我们也可以保存其他的信息，比如持有锁的线程，或请求获取锁的线程队列，但这些信息会隐藏起来，锁的使用者不会发现。

lock()和 unlock()函数的语义很简单。调用 lock()尝试获取锁，如果没有其他线程持有锁（即它是可用的），该线程会获得锁，进入临界区。这个线程有时被称为锁的持有者（owner）。如果另外一个线程对相同的锁变量（本例中的 mutex）调用 lock()，因为锁被另一线程持有，该调用不会返回。这样，当持有锁的线程在临界区时，其他线程就无法进入临界区。

锁的持有者一旦调用 unlock()，锁就变成可用了。如果没有其他等待线程（即没有其他线程调用过 lock()并卡在那里），锁的状态就变成可用了。如果有等待线程（卡在 lock()里），其中一个会（最终）注意到（或收到通知）锁状态的变化，获取该锁，进入临界区。

锁为程序员提供了最小程度的调度控制。我们把线程视为程序员创建的实体，但是被操作系统调度，具体方式由操作系统选择。锁让程序员获得一些控制权。通过给临界区加锁，可以保证临界区内只有一个线程活跃。锁将原本由操作系统调度的混乱状态变得更为可控。

28.2 Pthread 锁

POSIX 库将锁称为互斥量（mutex），因为它被用来提供线程之间的互斥。即当一个线程在临界区，它能够阻止其他线程进入直到本线程离开临界区。因此，如果你看到下面的 POSIX 线程代码，应该理解它和上面的代码段执行相同的任务（我们再次使用了包装函数来检查获取锁和释放锁时的错误）。

```
1    pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2
3    Pthread_mutex_lock(&lock);    // wrapper for pthread_mutex_lock()
4    balance = balance + 1;
5    Pthread_mutex_unlock(&lock);
```

你可能还会注意到，POSIX 的 lock 和 unlock 函数会传入一个变量，因为我们可能用不同的锁来保护不同的变量。这样可以增加并发：不同于任何临界区都使用同一个大锁（粗粒度的锁策略），通常大家会用不同的锁保护不同的数据和结构，从而允许更多的线程进入临界区（细粒度的方案）。

28.3 实现一个锁

我们已经从程序员的角度，对锁如何工作有了一定的理解。那如何实现一个锁呢？我们需要什么硬件支持？需要什么操作系统的支持？本章下面的内容将解答这些问题。

关键问题：怎样实现一个锁

如何构建一个高效的锁？高效的锁能够以低成本提供互斥，同时能够实现一些特性，我们下面会讨论。需要什么硬件支持？什么操作系统支持？

我们需要硬件和操作系统的帮助来实现一个可用的锁。近些年来，各种计算机体系结构的指令集都增加了一些不同的硬件原语，我们不研究这些指令是如何实现的（毕竟，这是计算机体系结构课程的主题），只研究如何使用它们来实现像锁这样的互斥原语。我们也会研究操作系统如何发展完善，支持实现成熟复杂的锁库。

28.4 评价锁

在实现锁之前，我们应该首先明确目标，因此我们要问，如何评价一种锁实现的效果。为了评价锁是否能工作（并工作得好），我们应该先设立一些标准。第一是锁是否能完成它的基本任务，即提供互斥（mutual exclusion）。最基本的，锁是否有效，能够阻止多个线程进入临界区？

第二是公平性（fairness）。当锁可用时，是否每一个竞争线程有公平的机会抢到锁？用

另一个方式来看这个问题是检查更极端的情况：是否有竞争锁的线程会饿死（starve），一直无法获得锁？

最后是性能（performance），具体来说，是使用锁之后增加的时间开销。有几种场景需要考虑。一种是没有竞争的情况，即只有一个线程抢锁、释放锁的开支如何？另外一种是一个 CPU 上多个线程竞争，性能如何？最后一种是多个 CPU、多个线程竞争时的性能。通过比较不同的场景，我们能够更好地理解不同的锁技术对性能的影响，下面会进行介绍。

28.5 控制中断

最早提供的互斥解决方案之一，就是在临界区关闭中断。这个解决方案是为单处理器系统开发的。代码如下：

```
1    void lock() {
2        DisableInterrupts();
3    }
4    void unlock() {
5        EnableInterrupts();
6    }
```

假设我们运行在这样一个单处理器系统上。通过在进入临界区之前关闭中断（使用特殊的硬件指令），可以保证临界区的代码不会被中断，从而原子地执行。结束之后，我们重新打开中断（同样通过硬件指令），程序正常运行。

这个方法的主要优点就是简单。显然不需要费力思考就能弄清楚它为什么能工作。没有中断，线程可以确信它的代码会继续执行下去，不会被其他线程干扰。

遗憾的是，缺点很多。首先，这种方法要求我们允许所有调用线程执行特权操作（打开关闭中断），即信任这种机制不会被滥用。众所周知，如果我们必须信任任意一个程序，可能就有麻烦了。这里，麻烦表现为多种形式：第一，一个贪婪的程序可能在它开始时就调用 lock()，从而独占处理器。更糟的情况是，恶意程序调用 lock()后，一直死循环。后一种情况，系统无法重新获得控制，只能重启系统。关闭中断对应用要求太多，不太适合作为通用的同步解决方案。

第二，这种方案不支持多处理器。如果多个线程运行在不同的 CPU 上，每个线程都试图进入同一个临界区，关闭中断也没有作用。线程可以运行在其他处理器上，因此能够进入临界区。多处理器已经很普遍了，我们的通用解决方案需要更好一些。

第三，关闭中断导致中断丢失，可能会导致严重的系统问题。假如磁盘设备完成了读取请求，但 CPU 错失了这一事实，那么，操作系统如何知道去唤醒等待读取的进程？

最后一个不太重要的原因就是效率低。与正常指令执行相比，现代 CPU 对于关闭和打开中断的代码执行得较慢。

基于以上原因，只在很有限的情况下用关闭中断来实现互斥原语。例如，在某些情况下操作系统本身会采用屏蔽中断的方式，保证访问自己数据结构的原子性，或至少避免某些复杂的中断处理情况。这种用法是可行的，因为在操作系统内部不存在信任问题，它总

是信任自己可以执行特权操作。

补充：DEKKER 算法和 PETERSON 算法

20 世纪 60 年代，Dijkstra 向他的朋友们提出了并发问题，他的数学家朋友 Theodorus Jozef Dekker 想出了一个解决方法。不同于我们讨论的需要硬件指令和操作系统支持的方法，Dekker 的算法（Dekker's algorithm）只使用了 load 和 store（早期的硬件上，它们是原子的）。

Peterson 后来改进了 Dekker 的方法[P81]。同样只使用 load 和 store，保证不会有两个线程同时进入临界区。以下是 Peterson 算法（Peterson's algorithm，针对两个线程），读者可以尝试理解这些代码吗？flag 和 turn 变量是用来做什么的？

```
int flag[2];
int turn;

void init() {
    flag[0] = flag[1] = 0;      // 1->thread wants to grab lock
    turn = 0;                   // whose turn? (thread 0 or 1?)
}
void lock() {
    flag[self] = 1;             // self: thread ID of caller
    turn = 1 - self;            // make it other thread's turn
    while ((flag[1-self] == 1) && (turn == 1 - self))
        ; // spin-wait
}
void unlock() {
    flag[self] = 0;             // simply undo your intent
}
```

一段时间以来，出于某种原因，大家都热衷于研究不依赖硬件支持的锁机制。后来这些工作都没有太多意义，因为只需要很少的硬件支持，实现锁就会容易很多（实际在多处理器的早期，就有这些硬件支持）。而且上面提到的方法无法运行在现代硬件（应为松散内存一致性模型），导致它们更加没有用处。更多的相关研究也湮没在历史中……

28.6 测试并设置指令（原子交换）

因为关闭中断的方法无法工作在多处理器上，所以系统设计者开始让硬件支持锁。最早的多处理器系统，像 20 世纪 60 年代早期的 Burroughs B5000[M82]，已经有这些支持。今天所有的系统都支持，甚至包括单 CPU 的系统。

最简单的硬件支持是测试并设置指令（test-and-set instruction），也叫作原子交换（atomic exchange）。为了理解 test-and-set 如何工作，我们首先实现一个不依赖它的锁，用一个变量标记锁是否被持有。

在第一次尝试中（见图 28.1），想法很简单：用一个变量来标志锁是否被某些线程占用。第一个线程进入临界区，调用 lock()，检查标志是否为 1（这里不是 1），然后设置标志为 1，表明线程持有该锁。结束临界区时，线程调用 unlock()，清除标志，表示锁未被持有。