POSIX 线程详解

一种支持内存共享的简捷工具

级别：初级

Daniel Robbins, 总裁/CEO, Gentoo Technologies, Inc.

2000 年 7 月 01 日

POSIX（可移植操作系统接口）线程是提高代码响应和性能的有力手段。在本系列中，Daniel Robbins 向您精确地展示在编程中如何使用线程。其中还涉及大量幕后细节，读完本系列文章，您完全可以运用 POSIX 线程创建多线程程序。

线程是有趣的

了解如何正确运用线程是每一个优秀程序员必备的素质。线程类似于进程。如同进程，线程由内核按时间分片进行管理。在单处理器系统中，内核使用时间分片来模拟线程的并发执行，这种方式和进程的相同。而在多处理器系统中，如同多个进程，线程实际上一样可以并发执行。

那么为什么对于大多数合作性任务，多线程比多个独立的进程更优越呢？这是因为，线程共享相同的内存空间。不同的线程可以存取内存中的同一个变量。所以，程序中的所有线程都可以读或写声明过的全局变量。如果曾用 fork() 编写过重要代码，就会认识到这个工具的重要性。为什么呢？虽然 fork() 允许创建多个进程，但它还会带来以下通信问题：如何让多个进程相互通信，这里每个进程都有各自独立的内存空间。对这个问题没有一个简单的答案。虽然有许多不同种类的本地 IPC (进程间通信），但它们都遇到两个重要障碍：

- 强加了某种形式的额外内核开销，从而降低性能。
- 对于大多数情形，IPC 不是对于代码的“自然”扩展。通常极大地增加了程序的复杂性。

双重坏事：开销和复杂性都非好事。如果曾经为了支持 IPC 而对程序大动干戈过，那么您就会真正欣赏线程提供的简单共享内存机制。由于所有的线程都驻留在同一内存空间，POSIX 线程无需进行开销大而复杂的长距离调用。只要利用简单的同步机制，程序中所有的线程都可以读取和修改已有的数据结构。而无需将数据经由文件描述符转储或挤入紧窄的共享内存空间。仅此一个原因，就足以让您考虑应该采用单进程/多线程模式而非多进程/单线程模式。

线程是快捷的

不仅如此。线程同样还是非常快捷的。与标准 fork() 相比，线程带来的开销很小。内核无需单独复制进程的内存空间或文件描述符等等。这就节省了大量的 CPU 时间，使得线程创建比新进程创建快上十到一百倍。因为这一点，可以大量使用线程而无需太过于担心带来的 CPU 或内存不足。使用 fork() 时导致的大量 CPU 占用也不复存在。这表示只要在程序中有意义，通常就可以创建线程。

当然，和进程一样，线程将利用多 CPU。如果软件是针对多处理器系统设计的，这就真的是一大特性（如果软件是开放源码，则最终可能在不少平台上运行）。特定类型线程程序（尤其是 CPU 密集型程序）的性能将随系统中处理器的数目几乎线性地提高。如果正在编写 CPU 非常密集型的程序，则绝对想设法在代码中使用多线程。一旦掌握了线程编码，无需使用繁琐的 IPC 和其它复杂的通信机制，就能够以全新和创造性的方法解决编码难题。所有这些特性配合在一起使得多线程编程更有趣、快速和灵活。

线程是可移植的

如果熟悉 Linux 编程，就有可能知道 __clone() 系统调用。__clone() 类似于 fork()，同时也有许多线程的特性。例如，使用 __clone()，新的子进程可以有选择地共享父进程的执行环境（内存空间，文件描述符等）。这是好的一面。但 __clone() 也有不足之处。正如__clone() 在线帮助指出：

“__clone 调用是特定于 Linux 平台的，不适用于实现可移植的程序。欲编写线程化应用程序（多线程控制同一内存空间），最好使用实现 POSIX 1003.1c 线程 API 的库，例如 Linux-Threads 库。参阅 pthread_create(3thr)。”

虽然 __clone() 有线程的许多特性，但它是不可移植的。当然这并不意味着代码中不能使用它。但在软件中考虑使用 __clone() 时应当权衡这一事实。值得庆幸的是，正如 __clone() 在线帮助指出，有一种更好的替代方案：POSIX 线程。如果想编写 **可移植的** 多线程代码，代码可运行于 Solaris、FreeBSD、Linux 和其它平台，POSIX 线程是一种当然之选。

第一个线程

下面是一个 POSIX 线程的简单示例程序：

thread1.c

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

 void *thread_function(void *arg) {
  int i;
  for ( i=0; i<20; i++) {
    printf("Thread says hi!\n");
    sleep(1);
  }
  return NULL;
}

int main(void) {

  pthread_t mythread;

  if ( pthread_create( &mythread, NULL, thread_function, NULL) ) {
    printf("error creating thread.");
    abort();
  }

  if ( pthread_join ( mythread, NULL ) ) {
    printf("error joining thread.");
    abort();
  }

  exit(0);

}
```

要编译这个程序，只需先将程序存为 thread1.c，然后输入：

```
$ gcc thread1.c -o thread1 -lpthread
```

运行则输入：

```
$ ./thread1
```

理解 thread1.c

thread1.c 是一个非常简单的线程程序。虽然它没有实现什么有用的功能，但可以帮助理解线程的运行机制。下面，我们一步一步地了解这个程序是干什么的。main() 中声明了变量 mythread，类型是 pthread_t。pthread_t 类型在 pthread.h 中定义，通常称为“线程 id”（缩写为 "tid"）。可以认为它是一种线程句柄。

mythread 声明后（记住 mythread 只是一个 "tid"，或是将要创建的线程的句柄），调用 pthread_create 函数创建一个真实活动的线程。不要因为 pthread_create() 在 "if" 语句内而受其迷惑。由于 pthread_create() 执行成功时返回零而失败时则返回非零值，将 pthread_create() 函数调用放在 if() 语句中只是为了方便地检测失败的调用。让我们查看一下 pthread_create 参数。第一个参数 &mythread 是指向 mythread 的指针。第二个参数当前为 NULL，可用来定义线程的某些属性。由于缺省的线程属性是适用的，只需将该参数设为 NULL。

第三个参数是新线程启动时调用的函数名。本例中，函数名为 thread_function()。当 thread_function() 返回时，新线程将终止。本例中，线程函数没有实现大的功能。它仅将 "Thread says hi!" 输出 20 次然后退出。注意 thread_function() 接受 void * 作为参数，同时返回值的类型也是 void *。这表明可以用 void * 向新线程传递任意类型的数据，新线程完成时也可返回任意类型的数据。那如何向线程传递一个任意参数？很简单。只要利用 pthread_create() 中的第四个参数。本例中，因为没有必要将任何数据传给微不足道的 thread_function()，所以将第四个参数设为 NULL。

您也许已推测到，在 pthread_create() 成功返回之后，程序将包含两个线程。等一等， **两个** 线程？我们不是只创建了一个线程吗？不错，我们只创建了一个进程。但是主程序同样也是一个线程。可以这样理解：如果编写的程序根本没有使用 POSIX 线程，则该程序是单线程的（这个单线程称为“主”线程）。创建一个新线程之后程序总共就有两个线程了。

我想此时您至少有两个重要问题。第一个问题，新线程创建之后主线程如何运行。答案，主线程按顺序继续执行下一行程序（本例中执行 "if (pthread_join(...))"）。第二个问题，新线程结束时如何处理。答案，新线程先停止，然后作为其清理过程的一部分，等待与另一个线程合并或“连接”。

现在，来看一下 pthread_join()。正如 pthread_create() 将一个线程拆分为两个， pthread_join() 将两个线程合并为一个线程。pthread_join() 的第一个参数是 tid mythread。第二个参数是指向 void 指针的指针。如果 void 指针不为 NULL，pthread_join 将线程的 void * 返回值放置在指定的位置上。由于我们不必理会 thread_function() 的返回值，所以将其设为 NULL.

您会注意到 thread_function() 花了 20 秒才完成。在 thread_function() 结束很久之前，主线程就已经调用了 pthread_join()。如果发生这种情况，主线程将中断（转向睡眠）然后等待 thread_function() 完成。当 thread_function() 完成后, pthread_join() 将返回。这时程序又只有一个主线程。当程序退出时，所有新线程已经使用 pthread_join() 合并了。这就是应该如何处理在程序中创建的每个新线程的过程。如果没有合并一个新线程，则它仍然对系统的最大线程数限制不利。这意味着如果未对线程做正确的清理，最终会导致 pthread_create() 调用失败。

无父，无子

如果使用过 fork() 系统调用，可能熟悉父进程和子进程的概念。当用 fork() 创建另一个新进程时，新进程是子进程，原始进程是父进程。这创建了可能非常有用的层次关系，尤其是等待子进程终止时。例如，waitpid() 函数让当前进程等待所有子进程终止。waitpid() 用来在父进程中实现简单的清理过程。

而 POSIX 线程就更有意思。您可能已经注意到我一直有意避免使用“父线程”和“子线程”的说法。这是因为 POSIX 线程中不存在这种层次关系。虽然主线程可以创建一个新线程，新线程可以创建另一个新线程，POSIX 线程标准将它们视为等同的层次。所以等待子线程退出的概念在这里没有意义。POSIX 线程标准不记录任何“家族”信息。缺少家族信息有一个主要含意：如果要等待一个线程终止，就必须将线程的 tid 传递给 pthread_join()。线程库无法为您断定 tid。

对大多数开发者来说这不是个好消息，因为这会使有多个线程的程序复杂化。不过不要为此担忧。POSIX 线程标准提供了有效地管理多个线程所需要的所有工具。实际上，没有父/子关系这一事实却为在程序中使用线程开辟了更创造性的方法。例如，如果有一个线程称为线程 1，线程 1 创建了称为线程 2 的线程，则线程 1 自己没有必要调用 pthread_join() 来合并线程 2，

程序中其它任一线程都可以做到。当编写大量使用线程的代码时，这就可能允许发生有趣的事情。例如，可以创建一个包含所有已停止线程的全局"死线程列表"，然后让一个专门的清理线程专等停止的线程加到列表中。这个清理线程调用 pthread_join() 将刚停止的线程与自己合并。现在，仅用一个线程就巧妙和有效地处理了全部清理。

同步漫游

现在我们来看一些代码，这些代码做了一些意想不到的事情。thread2.c 的代码如下：

thread2.c

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int myglobal;

 void *thread_function(void *arg) {
  int i,j;
  for ( i=0; i<20; i++) {
    j=myglobal;
    j=j+1;
    printf(".");
    fflush(stdout);
    sleep(1);
    myglobal=j;
  }
  return NULL;
}

int main(void) {

  pthread_t mythread;
  int i;

  if ( pthread_create( &mythread, NULL, thread_function, NULL) ) {
    printf("error creating thread.");
    abort();
  }

  for ( i=0; i<20; i++) {
     myglobal=myglobal+1;
    printf("o");
    fflush(stdout);
    sleep(1);
  }
```

```
  if ( pthread_join ( mythread, NULL ) ) {
    printf("error joining thread.");
    abort();
  }

  printf("\nmyglobal equals %d\n",myglobal);

  exit(0);

}
```

理解 thread2.c

如同第一个程序，这个程序创建一个新线程。主线程和新线程都将全局变量 myglobal 加一 20 次。但是程序本身产生了某些意想不到的结果。编译代码请输入：

```
$ gcc thread2.c -o thread2 -lpthread
```

运行请输入：

```
$ ./thread2
```

输出：

```
$ ./thread2
..o.o.o.o.oo.o.o.o.o.o.o.o.o.o..o.o.o.o.o
myglobal equals 21
```

非常意外吧！因为 myglobal 从零开始，主线程和新线程各自对其进行了 20 次加一，程序结束时 myglobal 值应当等于 40。由于 myglobal 输出结果为 21，这其中肯定有问题。但是究竟是什么呢？

放弃吗？好，让我来解释是怎么一回事。首先查看函数 thread_function()。注意如何将 myglobal 复制到局部变量 "j" 了吗? 接着将 j 加一，再睡眠一秒，然后到这时才将新的 j 值复制到 myglobal？这就是关键所在。设想一下，如果主线程就在新线程将 myglobal 值复制给 j **后** 立即将 myglobal 加一，会发生什么？当 thread_function() 将 j 的值写回 myglobal 时，就覆盖了主线程所做的修改。

当编写线程程序时，应避免产生这种无用的副作用，否则只会浪费时间（当然，除了编写关于 POSIX 线程的文章时有用）。

那么，如何才能排除这种问题呢？

由于是将 myglobal 复制给 j 并且等了一秒之后才写回时产生问题，可以尝试避免使用临时局部变量并直接将 myglobal 加一。虽然这种解决方案对这个特定例子适用，但它还是不正确。如果我们对 myglobal 进行相对复杂的数学运算，而不是简单的加一，这种方法就会失效。但是为什么呢？

要理解这个问题，必须记住线程是并发运行的。即使在单处理器系统上运行（内核利用时间分片模拟多任务）也是可以的，从程序员的角度，想像两个线程是同时执行的。thread2.c 出现问题是因为 thread_function() 依赖以下论据：在 myglobal 加一之前的大约一秒钟期间不会修改 myglobal。需要有些途径让一个线程在对 myglobal 做更改时通知其它线程“不要靠近”。我将在下一篇文章中讲解如何做到这一点。到时候见。

通用线程：POSIX 线程详解，第 2 部分

称作互斥对象的小玩意

POSIX 线程是提高代码响应和性能的有力手段。在此三部分系列文章的第二篇中，Daniel Robbins 将说明，如何使用被称为互斥对象的灵巧小玩意，来保护线程代码中共享数据结构的完整性。

互斥我吧！

在 前一篇文章中 ，谈到了会导致异常结果的线程代码。两个线程分别对同一个全局变量进行了二十次加一。变量的值最后应该是 40，但最终值却是 21。这是怎么回事呢？因为一个线程不停地"取消"了另一个线程执行的加一操作，所以产生这个问题。现在让我们来查看改正后的代码，它使用 **互斥对象**(mutex)来解决该问题：

thread3.c

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int myglobal;
pthread_mutex_t mymutex=PTHREAD_MUTEX_INITIALIZER;

 void *thread_function(void *arg) {
  int i,j;
  for ( i=0; i<20; i++) {
    pthread_mutex_lock(&mymutex);
    j=myglobal;
    j=j+1;
    printf(".");
    fflush(stdout);
    sleep(1);
    myglobal=j;
    pthread_mutex_unlock(&mymutex);
  }
  return NULL;
}

int main(void) {

  pthread_t mythread;
  int i;

  if ( pthread_create( &mythread, NULL, thread_function, NULL) ) {
    printf("error creating thread.");
    abort();
  }
```

```
  for ( i=0; i<20; i++) {
    pthread_mutex_lock(&mymutex);
    myglobal=myglobal+1;
    pthread_mutex_unlock(&mymutex);
    printf("o");
    fflush(stdout);
    sleep(1);
  }

  if ( pthread_join ( mythread, NULL ) ) {
    printf("error joining thread.");
    abort();
  }

  printf("\nmyglobal equals %d\n",myglobal);

  exit(0);

}
```

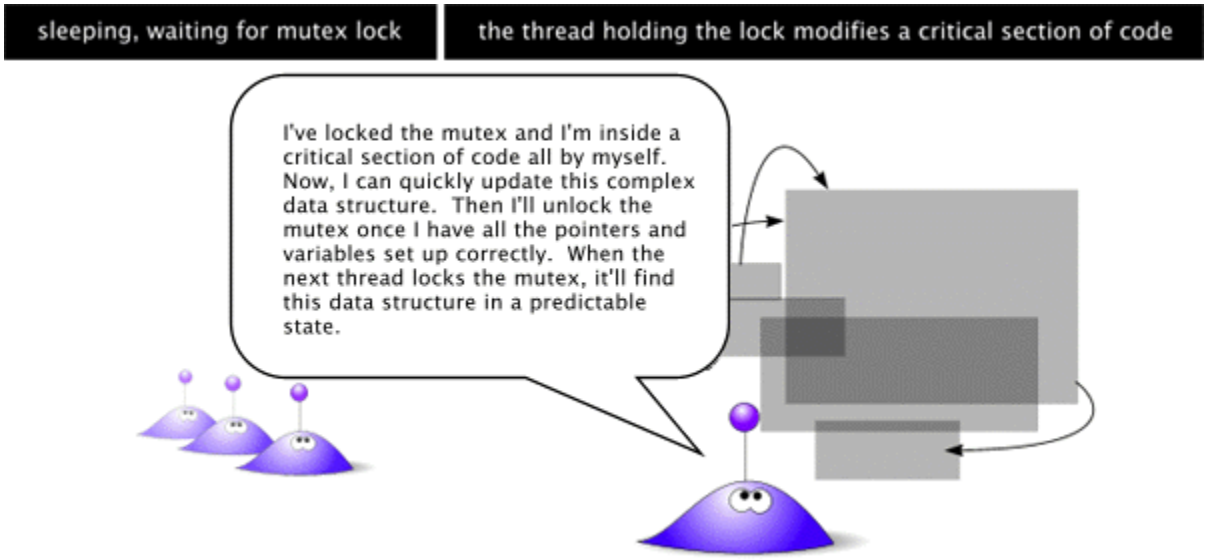
解读一下

如果将这段代码与 前一篇文章 中给出的版本作一个比较，就会注意到增加了 pthread_mutex_lock() 和 pthread_mutex_unlock() 函数调用。在线程程序中这些调用执行了不可或缺的功能。他们提供了一种 *相互排斥*的方法（互斥对象即由此得名）。两个线程不能同时对同一个互斥对象加锁。

互斥对象是这样工作的。如果线程 a 试图锁定一个互斥对象，而此时线程 b 已锁定了同一个互斥对象时，线程 a 就将进入睡眠状态。一旦线程 b 释放了互斥对象（通过 pthread_mutex_unlock() 调用），线程 a 就能够锁定这个互斥对象（换句话说，线程 a 就将从 pthread_mutex_lock() 函数调用中返回，同时互斥对象被锁定）。同样地，当线程 a 正锁定互斥对象时，如果线程 c 试图锁定互斥对象的话，线程 c 也将临时进入睡眠状态。对已锁定的互斥对象上调用 pthread_mutex_lock() 的所有线程都将进入睡眠状态，这些睡眠的线程将“排队”访问这个互斥对象。

通常使用 pthread_mutex_lock() 和 pthread_mutex_unlock() 来保护数据结构。这就是说，通过线程的锁定和解锁，对于某一数据结构，确保某一时刻只能有一个线程能够访问它。可以推测到，当线程试图锁定一个未加锁的互斥对象时，POSIX 线程库将同意锁定，而不会使线程进入睡眠状态。

请看这幅轻松的漫画，四个小精灵重现了最近一次 pthread_mutex_lock() 调用的一个场面。



图中，锁定了互斥对象的线程能够存取复杂的数据结构，而不必担心同时会有其它线程干扰。那个数据结构实际上是"冻结"了，直到互斥对象被解锁为止。pthread_mutex_lock() 和 pthread_mutex_unlock() 函数调用，如同"在施工中"标志一样，将正在修改和读取的某一特定共享数据包围起来。这两个函数调用的作用就是警告其它线程，要它们继续睡眠并等待轮到它们对互斥对象加锁。当然，除非在 *每个* 对特定数据结构进行读写操作的语句前后，都分别放上 pthread_mutex_lock() 和 pthread_mutext_unlock() 调用，才会出现这种情况。

为什么要用互斥对象？

听上去很有趣，但究竟为什么要让线程睡眠呢？要知道，线程的主要优点不就是其具有独立工作、更多的时候是同时工作的能力吗？是的，确实是这样。然而，每个重要的线程程序都需要使用某些互斥对象。让我们再看一下示例程序以便理解原因所在。请看 thread_function()，循环中一开始就锁定了互斥对象，最后才将它解锁。在这个示例程序中，mymutex 用来保护 myglobal 的值。仔细查看 thread_function()，加一代码把 myglobal 复制到一个局部变量，对局部变量加一，睡眠一秒钟，在这之后才把局部变量的值传回给 myglobal。不使用互斥对象时，即使主线程在 thread_function() 线程睡眠一秒钟期间内对 myglobal 加一，thread_function() 苏醒后也会覆盖主线程所加的值。使用互斥对象能够保证这种情形不会发生。（您也许会想到，我增加了一秒钟延迟以触发不正确的结果。把局部变量的值赋给 myglobal 之前，实际上没有什么真正理由要求 thread_function() 睡眠一秒钟。）使用互斥对象的新程序产生了期望的结果：

```
$ ./thread3
o..o..o.o..o..o.o.o.o.o..o..o..o.ooooooo
myglobal equals 40
```

为了进一步探索这个极为重要的概念，让我们看一看程序中进行加一操作的代码：

```
thread_function() 加一代码：
   j=myglobal;
   j=j+1;
   printf(".");
   fflush(stdout);
   sleep(1);
```

```
    myglobal=j;

主线程加一代码：
    myglobal=myglobal+1;
```

如果代码是位于单线程程序中，可以预期 thread_function() 代码将完整执行。接下来才会执行主线程代码（或者是以相反的顺序执行）。在不使用互斥对象的线程程序中，代码可能（几乎是，由于调用了 sleep() 的缘故）以如下的顺序执行：

```
    thread_function() 线程        主线程

    j=myglobal;
    j=j+1;
    printf(".");
    fflush(stdout);
    sleep(1);                  myglobal=myglobal+1;
    myglobal=j;
```

当代码以此特定顺序执行时，将覆盖主线程对 myglobal 的修改。程序结束后，就将得到不正确的值。如果是在操纵指针的话，就可能产生段错误。注意到 thread_function() 线程按顺序执行了它的所有指令。看来不象是 thread_function() 有什么次序颠倒。问题是，同一时间内，另一个线程对同一数据结构进行了另一个修改。

线程内幕 1

在解释如何确定在何处使用互斥对象之前，先来深入了解一下线程的内部工作机制。请看第一个例子：

假设主线程将创建三个新线程：线程 a、线程 b 和线程 c。假定首先创建线程 a，然后是线程 b，最后创建线程 c。

```
    pthread_create( &thread_a, NULL, thread_function, NULL);
    pthread_create( &thread_b, NULL, thread_function, NULL);
    pthread_create( &thread_c, NULL, thread_function, NULL);
```

在第一个 pthread_create() 调用完成后，可以假定线程 a 不是已存在就是已结束并停止。第二个 pthread_create() 调用后，主线程和线程 b 都可以假定线程 a 存在（或已停止）。

然而，就在第二个 create() 调用返回后，主线程无法假定是哪一个线程（a 或 b）会首先开始运行。虽然两个线程都已存在，线程 CPU 时间片的分配取决于内核和线程库。至于谁将首先运行，并没有严格的规则。尽管线程 a 更有可能在线程 b 之前开始执行，但这并无保证。对于多处理器系统，情况更是如此。如果编写的代码假定在线程 b 开始执行之前实际上执行线程 a 的代码，那么，程序最终正确运行的概率是 99%。或者更糟糕，程序在您的机器上 100% 地正确运行，而在您客户的四处理器服务器上正确运行的概率却是零。

从这个例子还可以得知，线程库保留了每个单独线程的代码执行顺序。换句话说，实际上那三个 pthread_create() 调用将按它们出现的顺序执行。从主线程上来看，所有代码都是依次执行的。有时，可以利用这一点来优化部分线程程序。例如，在上例中，线程 c 就可以假定线程 a 和线程 b 不是正在运行就是已经终止。它不必担心存在还没有创建线程 a 和线程 b 的可能性。可以使用这一逻辑来优化线程程序。

线程内幕 2

现在来看另一个假想的例子。假设有许多线程，他们都正在执行下列代码：

```
    myglobal=myglobal+1;
```

那么，是否需要在加一操作语句前后分别锁定和解锁互斥对象呢？也许有人会说“不”。编译器极有可能把上述赋值语句编译成一条机器指令。大家都知道，不可能"半途"中断一条机器指令。即使是硬件中断也不会破坏机器指令的完整性。基于以上考虑，很可能倾向于完全省略 pthread_mutex_lock() 和 pthread_mutex_unlock() 调用。不要这样做。

我在说废话吗？不完全是这样。首先，不应该假定上述赋值语句一定会被编译成一条机器指令，除非亲自验证了机器代码。即使插入某些内嵌汇编语句以确保加一操作的完整执行——甚至，即使是自己动手写编译器！-- 仍然可能有问题。

答案在这里。使用单条内嵌汇编操作码在单处理器系统上可能不会有什么问题。每个加一操作都将完整地进行，并且多半会得到期望的结果。但是多处理器系统则截然不同。在多 CPU 机器上，两个单独的处理器可能会在几乎同一时刻（或者，就在同一时刻）执行上述赋值语句。不要忘了，这时对内存的修改需要先从 L1 写入 L2 高速缓存、然后才写入主存。（SMP 机器并不只是增加了处理器而已；它还有用来仲裁对 RAM 存取的特殊硬件。）最终，根本无法搞清在写入主存的竞争中，哪个 CPU 将会"胜出"。要产生可预测的代码，应使用互斥对象。互斥对象将插入一道"内存关卡"，由它来确保对主存的写入按照线程锁定互斥对象的顺序进行。

考虑一种以 32 位块为单位更新主存的 SMP 体系结构。如果未使用互斥对象就对一个 64 位整数进行加一操作，整数的最高 4 位字节可能来自一个 CPU，而其它 4 个字节却来自另一 CPU。糟糕吧！最糟糕的是，使用差劲的技术，您的程序在重要客户的系统上有可能不是很长时间才崩溃一次，就是早上三点钟就崩溃。David R. Butenhof 在他的《POSIX 线程编程》（请参阅本文末尾的 参考资料部分）一书中，讨论了由于未使用互斥对象而将产生的种种情况。

许多互斥对象

如果放置了过多的互斥对象，代码就没有什么并发性可言，运行起来也比单线程解决方案慢。如果放置了过少的互斥对象，代码将出现奇怪和令人尴尬的错误。幸运的是，有一个中间立场。首先，互斥对象是用于串行化存取*共享数据*。不要对非共享数据使用互斥对象，并且，如果程序逻辑确保任何时候都只有一个线程能存取特定数据结构，那么也不要使用互斥对象。

其次，如果要使用共享数据，那么在读、写共享数据时都应使用互斥对象。用 pthread_mutex_lock() 和 pthread_mutex_unlock() 把读写部分保护起来，或者在程序中不固定的地方随机使用它们。学会从一个线程的角度来审视代码，并确保程序中每一个线程对内存的观点都是一致和合适的。为了熟悉互斥对象的用法，最初可能要花好几个小时来编写代码，但是很快就会习惯并且*也*不必多想就能够正确使用它们。

使用调用：初始化

现在该来看看使用互斥对象的各种不同方法了。让我们从初始化开始。在 thread3.c 示例 中，我们使用了静态初始化方法。这需要声明一个 pthread_mutex_t 变量，并赋给它常数 PTHREAD_MUTEX_INITIALIZER：

```
pthread_mutex_t mymutex=PTHREAD_MUTEX_INITIALIZER;
```

很简单吧。但是还可以动态地创建互斥对象。当代码使用 malloc() 分配一个新的互斥对象时，使用这种动态方法。此时，静态初始化方法是行不通的，并且应当使用例程 pthread_mutex_init()：

```
int pthread_mutex_init( pthread_mutex_t *mymutex, const pthread_mutexattr_t
*attr)
```

正如所示，pthread_mutex_init 接受一个指针作为参数以初始化为互斥对象，该指针指向一块已分配好的内存区。第二个参数，可以接受一个可选的 pthread_mutexattr_t 指针。这个结构可用来设置各种互斥对象属性。但是通常并不需要这些属性，所以正常做法是指定 NULL。

一旦使用 pthread_mutex_init() 初始化了互斥对象，就应使用 pthread_mutex_destroy() 消除它。pthread_mutex_destroy() 接受一个指向 pthread_mutext_t 的指针作为参数，并释放创建互斥对象时分配给它的任何资源。请注意， pthread_mutex_destroy() **不会** 释放用来存储 pthread_mutex_t 的内存。释放自己的内存完全取决于您。还必须注意一点，pthread_mutex_init() 和 pthread_mutex_destroy() 成功时都返回零。

使用调用：锁定

```
pthread_mutex_lock(pthread_mutex_t *mutex)
```

pthread_mutex_lock() 接受一个指向互斥对象的指针作为参数以将其锁定。如果碰巧已经锁定了互斥对象，调用者将进入睡眠状态。函数返回时，将唤醒调用者（显然）并且调用者还将保留该锁。函数调用成功时返回零，失败时返回非零的错误代码。

```
pthread_mutex_unlock(pthread_mutex_t *mutex)
```

pthread_mutex_unlock() 与 pthread_mutex_lock() 相配合，它把线程已经加锁的互斥对象解锁。始终应该尽快对已加锁的互斥对象进行解锁（以提高性能）。并且绝对不要对您未保持锁的互斥对象进行解锁操作（否则，pthread_mutex_unlock() 调用将失败并带一个非零的 EPERM 返回值）。

```
pthread_mutex_trylock(pthread_mutex_t *mutex)
```

当线程正在做其它事情的时候（由于互斥对象当前是锁定的），如果希望锁定互斥对象，这个调用就相当方便。调用 pthread_mutex_trylock() 时将尝试锁定互斥对象。如果互斥对象当前处于解锁状态，那么您将获得该锁并且函数将返回零。然而，如果互斥对象已锁定，这个调用也不会阻塞。当然，它会返回非零的 EBUSY 错误值。然后可以继续做其它事情，稍后再尝试锁定。

等待条件发生

互斥对象是线程程序必需的工具，但它们并非万能的。例如，如果线程正在等待共享数据内某个条件出现，那会发生什么呢？代码可以反复对互斥对象锁定和解锁，以检查值的任何变化。同时，还要快速将互斥对象解锁，以便其它线程能够进行任何必需的更改。这是一种非常可怕的方法，因为线程需要在合理的时间范围内频繁地循环检测变化。

在每次检查之间，可以让调用线程短暂地进入睡眠，比如睡眠三秒钟，但是因此线程代码就无法最快作出响应。真正需要的是这样一种方法，当线程在等待满足某些条件时使线程进入睡眠状态。一旦条件满足，还需要一种方法以唤醒因等待满足特定条件而睡眠的线程。如果能够做到这一点，线程代码将是非常高效的，并且不会占用宝贵的互斥对象锁。这正是 POSIX 条件变量能做的事！

而 POSIX 条件变量将是我下一篇文章的主题，其中将说明如何正确使用条件变量。到那时，您将拥有了创建复杂线程程序所需的全部资源，那些线程程序可以模拟工作人员、装配线等等。既然您已经越来越熟悉线程，我将在下一篇文章中加快进度。这样，在下一篇文章的结尾就能放上一个相对复杂的线程程序。说到等到条件产生，下次再见！

通用线程：POSIX 线程详解，第 3 部分

使用条件变量提高效率

本文是 POSIX 线程三部曲系列的最后一部分，Daniel 将详细讨论如何使用条件变量。条件变量是 POSIX 线程结构，可以让您在遇到某些条件时"唤醒"线程。可以将它们看作是一种线程安全的信号发送。Daniel 使用目前您所学到的知识实现了一个多线程工作组应用程序，本文将围绕着这一示例而进行讨论。

条件变量详解

在 上一篇文章结束时，我描述了一个比较特殊的难题：如果线程正在等待某个特定条件发生，它应该如何处理这种情况？它可以重复对互斥对象锁定和解锁，每次都会检查共享数据结构，以查找某个值。但这是在浪费时间和资源，而且这种繁忙查询的效率非常低。解决这个问题的最佳方法是使用 pthread_cond_wait() 调用来等待特殊条件发生。

了解 pthread_cond_wait() 的作用非常重要 -- 它是 POSIX 线程信号发送系统的核心，也是最难以理解的部分。

首先，让我们考虑以下情况：线程为查看已链接列表而锁定了互斥对象，然而该列表恰巧是空的。这一特定线程什么也干不了 -- 其设计意图是从列表中除去节点，但是现在却没有节点。因此，它只能：

锁定互斥对象时，线程将调用 pthread_cond_wait(&mycond,&mymutex)。pthread_cond_wait() 调用相当复杂，因此我们每次只执行它的一个操作。

pthread_cond_wait() 所做的第一件事就是同时对互斥对象解锁（于是其它线程可以修改已链接列表），并等待条件 mycond 发生（这样当 pthread_cond_wait() 接收到另一个线程的"信号"时，它将苏醒）。现在互斥对象已被解锁，其它线程可以访问和修改已链接列表，可能还会添加项。

此时，pthread_cond_wait() 调用还未返回。对互斥对象解锁会立即发生，但等待条件 mycond 通常是一个阻塞操作，这意味着线程将睡眠，在它苏醒之前不会消耗 CPU 周期。这正是我们期待发生的情况。线程将一直睡眠，直到特定条件发生，在这期间不会发生任何浪费 CPU 时间的繁忙查询。从线程的角度来看，它只是在等待 pthread_cond_wait() 调用返回。

现在继续说明，假设另一个线程（称作"2 号线程"）锁定了 mymutex 并对已链接列表添加了一项。在对互斥对象解锁之后，2 号线程会立即调用函数 pthread_cond_broadcast(&mycond)。此操作之后，2 号线程将使所有等待 mycond 条件变量的线程立即苏醒。这意味着第一个线程（仍处于 pthread_cond_wait() 调用中）现在将苏醒。

现在，看一下第一个线程发生了什么。您可能会认为在 2 号线程调用 pthread_cond_broadcast(&mymutex) 之后，1 号线程的 pthread_cond_wait() 会立即返回。不是那样！实际上，pthread_cond_wait() 将执行最后一个操作：重新锁定 mymutex。一旦 pthread_cond_wait() 锁定了互斥对象，那么它将返回并允许 1 号线程继续执行。那时，它可以马上检查列表，查看它所感兴趣的更改。

停止并回顾！

那个过程非常复杂，因此让我们先来回顾一下。第一个线程首先调用：

```
pthread_mutex_lock(&mymutex);
```

然后，它检查了列表。没有找到感兴趣的东西，于是它调用：

```
pthread_cond_wait(&mycond, &mymutex);
```

然后，pthread_cond_wait() 调用在返回前执行许多操作：

```
pthread_mutex_unlock(&mymutex);
```

它对 mymutex 解锁，然后进入睡眠状态，等待 mycond 以接收 POSIX 线程"信号"。一旦接收到"信号"（加引号是因为我们并不是在讨论传统的 UNIX 信号，而是来自 pthread_cond_signal() 或 pthread_cond_broadcast() 调用的信号），它就会苏醒。但 pthread_cond_wait() 没有立即返回 -- 它还要做一件事：重新锁定 mutex：

```
    pthread_mutex_lock(&mymutex);
```

pthread_cond_wait() 知道我们在查找 mymutex "背后"的变化，因此它继续操作，为我们锁定互斥对象，然后才返回。

pthread_cond_wait() 小测验

现在已回顾了 pthread_cond_wait() 调用，您应该了解了它的工作方式。应该能够叙述 pthread_cond_wait() 依次执行的所有操作。尝试一下。如果理解了 pthread_cond_wait()，其余部分就相当容易，因此请重新阅读以上部分，直到记住为止。好，读完之后，能否告诉我在调用 pthread_cond_wait() 之 *前*，互斥对象必须处于什么状态？pthread_cond_wait() 调用返回之后，互斥对象处于什么状态？这两个问题的答案都是"锁定"。既然已经完全理解了 pthread_cond_wait() 调用，现在来继续研究更简单的东西 -- 初始化和真正的发送信号和广播进程。到那时，我们将会对包含了多线程工作队列的 C 代码了如指掌。

初始化和清除

条件变量是一个需要初始化的真实数据结构。以下就初始化的方法。首先，定义或分配一个条件变量，如下所示：

```
    pthread_cond_t mycond;
```

然后，调用以下函数进行初始化：

```
    pthread_cond_init(&mycond,NULL);
```

瞧，初始化完成了！在释放或废弃条件变量之前，需要毁坏它，如下所示：

```
    pthread_cond_destroy(&mycond);
```

很简单吧。接着讨论 pthread_cond_wait() 调用。

等待

一旦初始化了互斥对象和条件变量，就可以等待某个条件，如下所示：

```
    pthread_cond_wait(&mycond, &mymutex);
```

请注意，代码在逻辑上应该包含 mycond 和 mymutex。一个特定条件只能有一个互斥对象，而且条件变量应该表示互斥数据"内部"的一种特殊的条件更改。一个互斥对象可以用许多条件变量（例如，cond_empty、cond_full、cond_cleanup），但每个条件变量只能有一个互斥对象。

发送信号和广播

对于发送信号和广播，需要注意一点。如果线程更改某些共享数据，而且它想要唤醒所有正在等待的线程，则应使用 pthread_cond_broadcast 调用，如下所示：

```
   pthread_cond_broadcast(&mycond);
```

在某些情况下，活动线程只需要唤醒第一个正在睡眠的线程。假设您只对队列添加了一个工作作业。那么只需要唤醒一个工作程序线程（再唤醒其它线程是不礼貌的！）：

```
    pthread_cond_signal(&mycond);
```

此函数只唤醒一个线程。如果 POSIX 线程标准允许指定一个整数，可以让您唤醒一定数量的正在睡眠的线程，那就更完美了。但是很可惜，我没有被邀请参加会议。

工作组

我将演示如何创建多线程工作组。在这个方案中，我们创建了许多工作程序线程。每个线程都会检查 wq（“工作队列”），查看是否有需要完成的工作。如果有需要完成的工作，那么线程将从队列中除去一个节点，执行这些特定工作，然后等待新的工作到达。

与此同时，主线程负责创建这些工作程序线程、将工作添加到队列，然后在它退出时收集所有工作程序线程。您将会遇到许多 C 代码，好好准备吧！

队列

需要队列是出于两个原因。首先，需要队列来保存工作作业。还需要可用于跟踪已终止线程的数据结构。还记得前几篇文章（请参阅本文结尾处的 参考资料）中，我曾提到过需要使用带有特定进程标识的 pthread_join 吗？使用“清除队列”（称作 "cq"）可以解决无法等待 *任何*已终止线程的问题（稍后将详细讨论这个问题）。以下是标准队列代码。将此代码保存到文件 queue.h 和 queue.c：

queue.h

```
/* queue.h
** Copyright 2000 Daniel Robbins, Gentoo Technologies, Inc.
** Author: Daniel Robbins
** Date: 16 Jun 2000
*/

typedef struct node {
  struct node *next;
} node;

typedef struct queue {
  node *head, *tail;
} queue;

void queue_init(queue *myroot);
void queue_put(queue *myroot, node *mynode);
node *queue_get(queue *myroot);
```

queue.c

```
/* queue.c
** Copyright 2000 Daniel Robbins, Gentoo Technologies, Inc.
** Author: Daniel Robbins
** Date: 16 Jun 2000
**
** This set of queue functions was originally thread-aware.  I
** redesigned the code to make this set of queue routines
** thread-ignorant (just a generic, boring yet very fast set of queue
** routines).  Why the change?  Because it makes more sense to have
** the thread support as an optional add-on.  Consider a situation
** where you want to add 5 nodes to the queue.  With the
** thread-enabled version, each call to queue_put() would
** automatically lock and unlock the queue mutex 5 times -- that's a
** lot of unnecessary overhead.  However, by moving the thread stuff
** out of the queue routines, the caller can lock the mutex once at
** the beginning, then insert 5 items, and then unlock at the end.
** Moving the lock/unlock code out of the queue functions allows for
** optimizations that aren't possible otherwise.  It also makes this
** code useful for non-threaded applications.
**
** We can easily thread-enable this data structure by using the
** data_control type defined in control.c and control.h.  */

#include <stdio.h>
#include "queue.h"

void queue_init(queue *myroot) {
  myroot->head=NULL;
  myroot->tail=NULL;
}

void queue_put(queue *myroot,node *mynode) {
  mynode->next=NULL;
  if (myroot->tail!=NULL)
    myroot->tail->next=mynode;
  myroot->tail=mynode;
  if (myroot->:head==NULL)
    myroot->head=mynode;
}

node *queue_get(queue *myroot) {
  //get from root
```

```
  node *mynode;
  mynode=myroot->head;
  if (myroot->head!=NULL)
    myroot->head=myroot->head->next;
  return mynode;
}
```

data_control 代码

我编写的并不是线程安全的队列例程，事实上我创建了一个“数据包装”或“控制”结构，它可以是任何线程支持的数据结构。看一下 control.h：

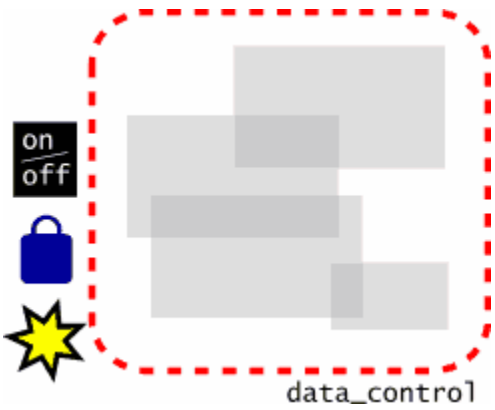
control.h

```
#include

typedef struct data_control {
  pthread_mutex_t mutex;
  pthread_cond_t cond;
  int active;
} data_control;
```

现在您看到了 data_control 结构定义，以下是它的视觉表示：

所使用的 data_control 结构



图像中的锁代表互斥对象，它允许对数据结构进行互斥访问。黄色的星代表条件变量，它可以睡眠，直到所讨论的数据结构改变为止。on/off 开关表示整数 "active"，它告诉线程此数据是否是活动的。在代码中，我使用整数 active 作为标志，告诉工作队列何时应该关闭。以下是 control.c：

control.c

```
/* control.c
** Copyright 2000 Daniel Robbins, Gentoo Technologies, Inc.
** Author: Daniel Robbins
```

```
** Date: 16 Jun 2000
**
** These routines provide an easy way to make any type of
** data-structure thread-aware.  Simply associate a data_control
** structure with the data structure (by creating a new struct, for
** example).  Then, simply lock and unlock the mutex, or
** wait/signal/broadcast on the condition variable in the data_control
** structure as needed.
**
** data_control structs contain an int called "active".  This int is
** intended to be used for a specific kind of multithreaded design,
** where each thread checks the state of "active" every time it locks
** the mutex.  If active is 0, the thread knows that instead of doing
** its normal routine, it should stop itself.  If active is 1, it
** should continue as normal.  So, by setting active to 0, a
** controlling thread can easily inform a thread work crew to shut
** down instead of processing new jobs.  Use the control_activate()
** and control_deactivate() functions, which will also broadcast on
** the data_control struct's condition variable, so that all threads
** stuck in pthread_cond_wait() will wake up, have an opportunity to
** notice the change, and then terminate.
*/

#include "control.h"

int control_init(data_control *mycontrol) {
  int mystatus;
  if (pthread_mutex_init(&(mycontrol->mutex),NULL))
    return 1;
  if (pthread_cond_init(&(mycontrol->cond),NULL))
    return 1;
  mycontrol->active=0;
  return 0;
}

int control_destroy(data_control *mycontrol) {
  int mystatus;
  if (pthread_cond_destroy(&(mycontrol->cond)))
    return 1;
  if (pthread_cond_destroy(&(mycontrol->cond)))
    return 1;
  mycontrol->active=0;
  return 0;
}
```

```
int control_activate(data_control *mycontrol) {
  int mystatus;
  if (pthread_mutex_lock(&(mycontrol->mutex)))
    return 0;
  mycontrol->active=1;
  pthread_mutex_unlock(&(mycontrol->mutex));
  pthread_cond_broadcast(&(mycontrol->cond));
  return 1;
}

int control_deactivate(data_control *mycontrol) {
  int mystatus;
  if (pthread_mutex_lock(&(mycontrol->mutex)))
    return 0;
  mycontrol->active=0;
  pthread_mutex_unlock(&(mycontrol->mutex));
  pthread_cond_broadcast(&(mycontrol->cond));
  return 1;
}
```

调试时间

在开始调试之前，还需要一个文件。以下是 dbug.h：

dbug.h

```

#define dabort() \
 {  printf("Aborting at line %d in source file %s\n",__LINE__,__FILE__);
abort(); }
```

此代码用于处理工作组代码中的不可纠正错误。

工作组代码

说到工作组代码，以下就是：

workcrew.c

```
#include <stdio.h>
#include <stdlib.h>
#include "control.h"
#include "queue.h"
#include "dbug.h"

/* the work_queue holds tasks for the various threads to complete. */
```

```
struct work_queue {
  data_control control;
  queue work;
} wq;


/* I added a job number to the work node.  Normally, the work node
   would contain additional data that needed to be processed. */

typedef struct work_node {
  struct node *next;
  int jobnum;
} wnode;

/* the cleanup queue holds stopped threads.  Before a thread
   terminates, it adds itself to this list.  Since the main thread is
   waiting for changes in this list, it will then wake up and clean up
   the newly terminated thread. */

struct cleanup_queue {
  data_control control;
  queue cleanup;
} cq;

/* I added a thread number (for debugging/instructional purposes) and
   a thread id to the cleanup node.  The cleanup node gets passed to
   the new thread on startup, and just before the thread stops, it
   attaches the cleanup node to the cleanup queue.  The main thread
   monitors the cleanup queue and is the one that performs the
   necessary cleanup. */

typedef struct cleanup_node {
  struct node *next;
  int threadnum;
  pthread_t tid;
} cnode;

void *threadfunc(void *myarg) {

  wnode *mywork;
  cnode *mynode;

  mynode=(cnode *) myarg;
```

```
  pthread_mutex_lock(&wq.control.mutex);

  while (wq.control.active) {
    while (wq.work.head==NULL && wq.control.active) {
      pthread_cond_wait(&wq.control.cond, &wq.control.mutex);
    }
    if (!wq.control.active)
      break;
    //we got something!
    mywork=(wnode *) queue_get(&wq.work);
    pthread_mutex_unlock(&wq.control.mutex);
    //perform processing...
    printf("Thread number %d processing job
%d\n",mynode->threadnum,mywork->jobnum);
    free(mywork);
    pthread_mutex_lock(&wq.control.mutex);
  }

  pthread_mutex_unlock(&wq.control.mutex);

  pthread_mutex_lock(&cq.control.mutex);
  queue_put(&cq.cleanup,(node *) mynode);
  pthread_mutex_unlock(&cq.control.mutex);
  pthread_cond_signal(&cq.control.cond);
  printf("thread %d shutting down...\n",mynode->threadnum);
  return NULL;

}

#define NUM_WORKERS 4

int numthreads;

void join_threads(void) {
  cnode *curnode;

  printf("joining threads...\n");

  while (numthreads) {
    pthread_mutex_lock(&cq.control.mutex);

    /* below, we sleep until there really is a new cleanup node.  This
       takes care of any false wakeups... even if we break out of
```

```
      pthread_cond_wait(), we don't make any assumptions that the
      condition we were waiting for is true.  */

    while (cq.cleanup.head==NULL) {
      pthread_cond_wait(&cq.control.cond,&cq.control.mutex);
    }

    /* at this point, we hold the mutex and there is an item in the
       list that we need to process.  First, we remove the node from
       the queue.  Then, we call pthread_join() on the tid stored in
       the node.  When pthread_join() returns, we have cleaned up
       after a thread.  Only then do we free() the node, decrement the
       number of additional threads we need to wait for and repeat the
       entire process, if necessary */

      curnode = (cnode *) queue_get(&cq.cleanup);
      pthread_mutex_unlock(&cq.control.mutex);
      pthread_join(curnode->tid,NULL);
      printf("joined with thread %d\n",curnode->threadnum);
      free(curnode);
      numthreads--;
  }
}


int create_threads(void) {
  int x;
  cnode *curnode;

  for (x=0; x<NUM_WORKERS; x++) {
    curnode=malloc(sizeof(cnode));
    if (!curnode)
      return 1;
    curnode->threadnum=x;
    if (pthread_create(&curnode->tid, NULL, threadfunc, (void *) curnode))
      return 1;
    printf("created thread %d\n",x);
    numthreads++;
  }
  return 0;
}

void initialize_structs(void) {
  numthreads=0;
```

```
  if (control_init(&wq.control))
    dabort();
  queue_init(&wq.work);
  if (control_init(&cq.control)) {
    control_destroy(&wq.control);
    dabort();
  }
  queue_init(&wq.work);
  control_activate(&wq.control);
}

void cleanup_structs(void) {
  control_destroy(&cq.control);
  control_destroy(&wq.control);
}


int main(void) {

  int x;
  wnode *mywork;

  initialize_structs();

  /* CREATION */

  if (create_threads()) {
    printf("Error starting threads... cleaning up.\n");
    join_threads();
    dabort();
  }

  pthread_mutex_lock(&wq.control.mutex);
  for (x=0; x<16000; x++) {
    mywork=malloc(sizeof(wnode));
    if (!mywork) {
      printf("ouch! can't malloc!\n");
      break;
    }
    mywork->jobnum=x;
    queue_put(&wq.work,(node *) mywork);
  }
  pthread_mutex_unlock(&wq.control.mutex);
  pthread_cond_broadcast(&wq.control.cond);
```

```
  printf("sleeping...\n");
  sleep(2);
  printf("deactivating work queue...\n");
  control_deactivate(&wq.control);
  /* CLEANUP  */

  join_threads();
  cleanup_structs();

}
```

代码初排

现在来快速初排代码。定义的第一个结构称作 "wq"，它包含了 data_control 和队列头。data_control 结构用于仲裁对整个队列的访问，包括队列中的节点。下一步工作是定义实际的工作节点。要使代码符合本文中的示例，此处所包含的都是作业号。

接着，创建清除队列。注释说明了它的工作方式。好，现在让我们跳过 threadfunc()、join_threads()、create_threads() 和 initialize_structs() 调用，直接跳到 main()。所做的第一件事就是初始化结构 -- 这包括初始化 data_controls 和队列，以及激活工作队列。

有关清除的注意事项

现在初始化线程。如果看一下 create_threads() 调用，似乎一切正常 -- 除了一件事。请注意，我们正在分配清除节点，以及初始化它的线程号和 TID 组件。我们还将清除节点作为初始自变量传递给每一个新的工作程序线程。为什么这样做？

因为当某个工作程序线程退出时，它会将其清除节点连接到清除队列，然后终止。那时，主线程会在清除队列中检测到这个节点（利用条件变量），并将这个节点移出队列。因为 TID（线程标识）存储在清除节点中，所以主线程可以确切知道哪个线程已终止了。然后，主线程将调用 pthread_join(tid)，并联接适当的工作程序线程。如果没有做记录，那么主线程就需要按任意顺序联接工作程序线程，可能是按它们的创建顺序。由于线程不一定按此顺序终止，那么主线程可能会在已经联接了十个线程时，等待联接另一个线程。您能理解这种设计决策是如何使关闭代码加速的吗（尤其在使用几百个工作程序线程的情况下）？

创建工作

我们已启动了工作程序线程（它们已经完成了执行 threadfunc()，稍后将讨论此函数），现在主线程开始将工作节点插入工作队列。首先，它锁定 wq 的控制互斥对象，然后分配 16000 个工作包，将它们逐个插入队列。完成之后，将调用 pthread_cond_broadcast()，于是所有正在睡眠的线程会被唤醒，并开始执行工作。此时，主线程将睡眠两秒钟，然后释放工作队列，并通知工作程序线程终止活动。接着，主线程会调用 join_threads() 函数来清除所有工作程序线程。

threadfunc()

现在来讨论 threadfunc()，这是所有工作程序线程都要执行的代码。当工作程序线程启动时，它会立即锁定工作队列互斥对象，获取一个工作节点（如果有的话），然后对它进行处理。如果没有工作，则调用 pthread_cond_wait()。您会注意到这个调用在一个非常紧凑的 while() 循环中，这是非常重要的。当从 pthread_cond_wait() 调用中苏醒时，决不能认为条件肯定发生了 -- 它 *可能*发生了，也可能没有发生。如果发生了这种情况，即错误地唤醒了线程，而列表是空的，那么 while 循环将再次调用 pthread_cond_wait()。

如果有一个工作节点，那么我们只打印它的作业号，释放它并退出。然而，实际代码会执行一些更实质性的操作。在 while() 循环结尾，我们锁定了互斥对象，以便检查 active 变量，以及在循环顶部检查新的工作节点。如果执行完此代码，就会发现如果 wq.control.active 是 0，while 循环就会终止，并会执行 threadfunc() 结尾处的清除代码。

工作程序线程的清除代码部件非常有趣。首先，由于 pthread_cond_wait() 返回了锁定的互斥对象，它会对 work_queue 解锁。然后，它锁定清除队列，添加清除代码（包含了 TID，主线程将使用此 TID 来调用 pthread_join()），然后再对清除队列解锁。此后，它发信号给所有 cq 等待者 (pthread_cond_signal(&cq.control.cond))，于是主线程就知道有一个待处理的新节点。我们不使用 pthread_cond_broadcast()，因为没有这个必要 -- 只有一个线程（主线程）在等待清除队列中的新节点。当它调用 join_threads() 时，工作程序线程将打印关闭消息，然后终止，等待主线程发出的 pthread_join() 调用。

join_threads()

如果要查看关于如何使用条件变量的简单示例，请参考 join_threads() 函数。如果还有工作程序线程，join_threads() 会一直执行，等待清除队列中新的清除节点。如果有新节点，我们会将此节点移出队列、对清除队列解锁（从而使工作程序可以添加清除节点）、联接新的工作程序线程（使用存储在清除节点中的 TID）、释放清除节点、减少"现有"线程的数量，然后继续。

结束语

现在已经到了"POSIX 线程详解"系列的尾声，希望您已经准备好开始将多线程代码添加到您自己的应用程序中。有关详细信息，请参阅 参考资料部分，这部分内容还包含了本文中使用的所有源码的 tar 文件。下一个系列中再见！