14 | 标准库：如何使用互斥量等技术协调线程运行？

2022-01-14 于航

《深入C语言和程序运行原理》 课程介绍 >

讲述：于航

时长 15:19 大小 14.04M

你好，我是于航。

在上一讲中，我主要介绍了有关并发编程的一些基础知识，并通过一个简单的例子展示了如何在 C 语言中进行线程创建等基本操作。同时我也向你介绍了，数据竞争、竞态条件，以及指令重排等因素，都在如何影响着多线程应用的执行正确性。那么，有哪些方法可以辅助我们解决这些问题呢？

今天我们就来看看 C 语言为并发编程提供的几大利器：互斥量、原子操作、条件变量，以及线程本地变量。

使用互斥量

从本质上来看，互斥量（Mutex）其实就是一把锁。一个线程在访问某个共享资源前，需要先对互斥量进行加锁操作。此时，其他任何想要再对互斥量进行加锁的线程都会被阻塞，直至当前线程释放该锁。而当锁被释放后，所有之前被阻塞的线程都开始继续运行，并再次重复之前的步骤，开始“争夺”可以对互斥量进行加锁的名额。通过这种方式，我们便可以保证每次在对多线程共享的资源进行操作时，都仅只有一个线程。

在 C 语言中，我们可以通过头文件 threads.h 提供的，以 “mtx_” 为前缀的相关接口来使用互斥量的能力。你应该还记得我在上一讲中提到的，那段存在数据竞争的 C 示例代码。这里我对它进行了改写，如下所示：

复制代码

```c
#include <threads.h>
#include <stdio.h>
#define THREAD_COUNT 10
#define THREAD_LOOP 100000000
mtx_t mutex;
long counter = 0;
int run(void* data) {
  for (int i = 0; i < THREAD_LOOP; i++) {
    mtx_lock(&mutex);  // 对互斥量加锁，
    counter++;
    mtx_unlock(&mutex);  // 释放一个互斥量；
  }
  printf("Thread %d terminates.\n", *((int*) data));
  return thrd_success;
}
int main(void) {
#ifndef __STDC_NO_THREADS__
  int ids[THREAD_COUNT];
  mtx_init(&mutex, mtx_plain);  // 创建一个简单、非递归的互斥量对象；
  thrd_t threads[THREAD_COUNT];
  for (int i = 0; i < THREAD_COUNT; i++) {
    ids[i] = i + 1;
    thrd_create(&threads[i], run, ids + i);
  }
  for (int i = 0; i < THREAD_COUNT; i++)
    thrd_join(threads[i], NULL);
  printf("Counter value is: %ld.\n", counter);
  mtx_destroy(&mutex);  // 销毁一个互斥量对象；
#endif
  return 0;
}
```

可以看到，在代码的第 19 行，我们使用 mtx_init 函数创建了一个基本类型（mtx_plain）的互斥量对象（下文中简称互斥量）。紧接着，在 run 函数内部，对变量 counter 进行值累加操作前，我们需要通过 mtx_lock 函数，来对之前创建的互斥量进行加锁操作。同样地，当进程使用完共享变量后，还需要通过 mtx_unlock 函数对互斥量进行解锁，来让其他线程有机会继续对共享变量进行处理。最后，在代码的第 28 行，程序退出前，我们销毁了之前创建的互斥量。

总的来看，**在 C 语言中，互斥量可以被分为三种类型：mtx_plain、mtx_recursive 与 mtx_timed**。

其中，mtx_plain 为最简单类型的互斥量，我们可以对它进行基本的加锁和解锁，但不能将其用在需要“重复加锁”的场景（比如函数的递归调用）中。这是因为，即使当前线程拥有该锁，对同一个 mtx_plain 互斥量的再次加锁也会导致该线程被阻塞。而此时，便会产生死锁的问题，即当前线程等待自己解锁后才能够再次进行加锁，而想要解锁，则需要让线程先加锁以完成当前功能的执行。

相反，mtx_recursive 类型的互斥量也被称为“可重入互斥量（Reentrant Mutex）”，顾名思义，它可以被用在需要重复加锁的场景中。该类型互斥量可以被同一个线程重复锁定多次，而不会阻塞线程。但相应地，对它的完全解锁也需要执行对应多次的 mtx_unlock。

而最后一种是 mtx_timed 类型的互斥量，它具有特殊的“超时属性”。这意味着，通过配合使用 mtx_timedlock 函数，我们可以实现“带超时限制的互斥量加锁”，即线程在尝试给对应互斥量加锁时，只会以阻塞的方式等待一定时间。若超过给定时间后仍未给互斥量成功上锁，则线程继续执行。

除了上面提到过的函数，C 标准库还提供了另外两个与“互斥”有关的函数。这里，我将它们整理在了下面的表格中，供你参考。

函数名	功能描述
mtx_trylock	锁住指定互斥量或直接返回
call_once	仅调用指定方法一次，即使它从多个线程中被调用

极客时间

利用互斥锁能够帮助我们解决数据竞争问题，但在某些对性能要求更加严苛的场景下，它可能并非最好的选择。接下来，让我们来看看另一种可以避免数据竞争的方式，原子操作。

使用原子操作

原子是化学反应中不可被继续分割的基本微粒，那么顾名思义，“原子操作”的意思就是操作本身无法再被划分为更细的步骤。当我们在多个不同线程中对共享资源进行原子操作时，编译器和 CPU 将会保证这些操作的正确执行，即同一时刻只会有一个线程在进行这些操作。而只有在该线程将整个操作全部执行完毕后，其他线程才可以继续执行同样的操作。

类似地，通过 C11 提供的名为 stdatomic.h 的头文件，我们可以方便地使用这些原子操作能力。比如，在下面这段代码中，我们便通过这种方式，解决了上一讲中那个实例的数据竞争问题。

复制代码

```
#include <threads.h>
#include <stdio.h>
#include <stdatomic.h>
#define THREAD_COUNT 10
#define THREAD_LOOP 100000000
#if !defined(__STDC_NO_ATOMICS__)
_Atomic long counter = 0;  // 定义一个原子类型全局变量，用来记录线程的累加值；
#endif
int run(void* data) {
  for (int i = 0; i < THREAD_LOOP; i++)
    atomic_fetch_add_explicit(&counter, 1, memory_order_relaxed);  // 使用原子加
  printf("Thread %d terminates.\n", *((int*) data));
  return thrd_success;
}
int main(void) {
#if !defined(__STDC_NO_THREADS__) || !defined(__STDC_NO_ATOMICS__)
```

```
17   int ids[THREAD_COUNT];
18   thrd_t threads[THREAD_COUNT];
19   for (int i = 0; i < THREAD_COUNT; i++) {
20     ids[i] = i + 1;
21     thrd_create(&threads[i], run, ids + i);
22   }
23   for (int i = 0; i < THREAD_COUNT; i++)
24     thrd_join(threads[i], NULL);
25   printf("Counter value is: %ld.\n", counter);
26 #endif
27   return 0;
28 }
```

与使用线程控制相关接口类似，我们也需要通过名为 **STDC_NO_ATOMICS** 的宏，来判断编译器是否对原子操作提供支持。可以看到，我们分别在代码的第 6 行与第 16 行进行了相应的预处理判断。

接下来，在代码的第 7 行，我们使用 C11 新引入的 _Atomic 关键字，修饰了原有的全局变量 counter，以将它定义为一个原子类型（这里也可以直接使用 C 标准库为我们封装好的宏 atomic_long）。

紧接着，在 run 函数内部，代码的第 11 行，我们使用名为 atomic_fetch_add_explicit 的函数来完成对 counter 变量的累加过程。该函数为我们提供了一种原子累加操作，可以使线程在进行数据累加时独占整个变量。除此之外，你还需要注意：通过该函数的第三个参数，我们还可以指定当前操作需要满足的内存顺序。

在上一讲中我提到，由于编译器和处理器可能会采用指令重排来优化程序的运行效率，因此，当在多核 CPU 上运行存在线程间数据依赖的多线程应用时，程序的正确性可能会出现问题。那么，怎样解决这个问题呢？我们来看下面这段代码。这里，我通过指定各个原子操作的具体内存顺序，修复了上一讲最后一小节中提到的例子。

复制代码

```
1  #include <threads.h>
2  #include <stdio.h>
3  #include <stdatomic.h>
4  #if !defined(__STDC_NO_ATOMICS__)
5  atomic_int x = 0, y = 0;
6  #endif
7  int run(void* v) {
8    atomic_store_explicit(&x, 10, memory_order_relaxed);
9    atomic_store_explicit(&y, 20, memory_order_release);
```

```
}
int observe(void* v) {
  while(atomic_load_explicit(&y, memory_order_acquire) != 20);
  printf("%d", atomic_load_explicit(&x, memory_order_relaxed));
}
int main(void) {
#if !defined(__STDC_NO_THREADS__) || !defined(__STDC_NO_ATOMICS__)
  thrd_t threadA, threadB;
  thrd_create(&threadA, run, NULL);
  thrd_create(&threadB, observe, NULL);
  thrd_join(threadA, NULL);
  thrd_join(threadB, NULL);
#endif
  return 0;
}
```

可以看到，我们修改了线程 run 和 observe 中对原子类型变量 x 和 y 的读写操作。其中，函数 atomic_load_explicit 用来读取某个原子类型变量的值；而对它们的修改，则使用函数 atomic_store_explicit 进行。除此之外，这两个函数都支持通过它们的最后一个参数，来指定相应操作需要遵循的内存顺序。

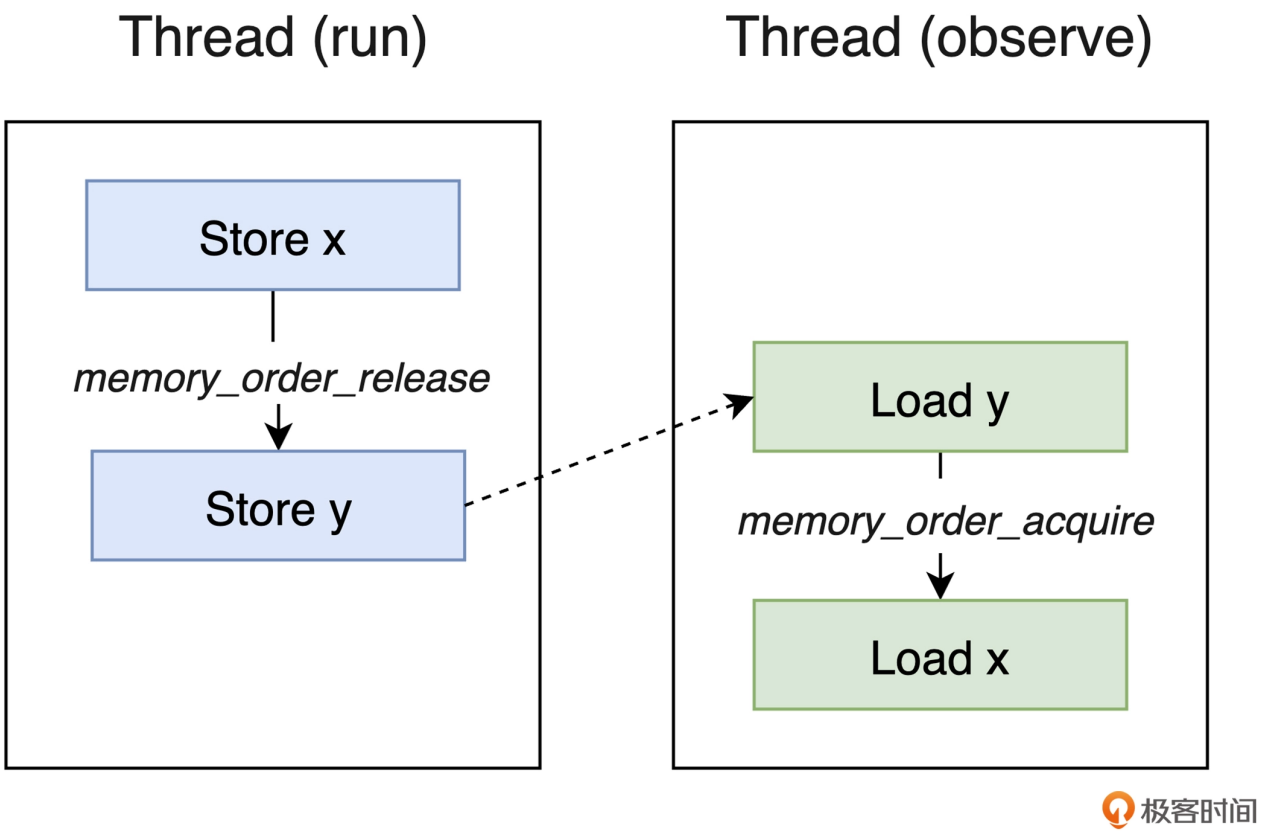
在这段修改后的代码中，一共使用到了三种不同的内存顺序（对应三个枚举值）。首先，我们来看看它们的具体定义。为了方便你观察，我将这些信息整理在了下面的表格中。

内存顺序（枚举值）	具体定义
memory_order_relaxed	不对执行顺序做任何保证
memory_order_release	必须完成所有之前的写操作，才能执行本条写操作
memory_order_acquire	所有后续的读操作，必须在本条读指令完成后才能执行

相信看过这三种内存顺序的定义后，你已经对它们的作用有个大致了解了。其中，对于使用了 memory_order_relaxed 的操作，我们并不需要对它们的执行顺序做任何保证。相反，编译器和处理器可以按照需求进行适当的优化。

而在 run 线程中，为了保证对变量 x 的修改过程一定发生在变量 y 的值被修改前，我们便需要使用 memory_order_release 来限制对变量 y 的修改，一定要在之前的所有修改都完

成后再进行。同样地，对于 observe 线程来说，为了防止处理器提前将变量 x 的值放入缓存，这里，我们也需要通过 memory_order_acquire，来保证对变量 y 进行的读操作一定会比对变量 x 的读操作先发生。你可以通过下面这张图片，直观地理解上面我们提到的各个操作之间的执行关系。



除了我们在上面的例子中用到的三种内存顺序外，C 语言还提供了另外 3 种不同的内存顺序，供我们在不同的场景中使用。如果想了解关于它们的更多信息，你可以参考这个链接。

总的来看，C11 通过 stdatomic.h 头文件为我们提供了大量可用于原子操作的相关类型、宏，以及函数。相较于使用互斥量，原子操作可以让我们更加清晰和方便地抽象并行代码，而不需要频繁进行加锁与释放锁的操作。

不仅如此，从执行性能角度，原子操作的执行通常直接依赖于 CPU 提供的相应的原子机器指令，比如在 x86-64 平台上，atomic_fetch_add_explicit 函数对应的 `lock add` 指令。而使用互斥量则需要让线程阻塞，还要频繁进行上下文切换，因此与之相比，原子操作的性能通常会更好。

这里，我将一些与原子操作相关的常用标准库函数整理在了下面的表格中，供你参考。你也可以点击这个链接，查看更多信息。

函数名	功能描述
atomic_flag_test_and_set	将一个 atomic_flag 的值置为真，并返回旧值
atomic_flag_clear	将一个 atomic_flag 的值设为假
atomic_init	初始化一个已经存在的原子对象
atomic_is_lock_free	检测指定对象是否是 lock-free 的
atomic_exchange	原子地交换两个值
atomic_compare_exchange_weak	比较并原子地交换两个值（允许伪失败）
atomic_compare_exchange_strong	比较并原子地交换两个值
atomic_signal_fence	在线程和信号处理程序之间建立内存栅栏
atomic_thread_fence	在线程之间建立内存栅栏

使用条件变量

条件变量是一种常用的线程同步机制。通过上一小节的例子，你会发现，在多线程应用中，存在着一种十分常见的线程间同步模式，即某个线程的执行依赖于另一个线程对数据首先进行的预处理。在上面的例子中，observe 线程中某段逻辑的执行需要等待 run 线程将原子变量 y 的值变为 20。这里，我们通过“忙等待（Busy Waiting）”的方式实现了这个效果。

用忙等待虽然可以达到我们的预期，但这是一种十分“昂贵”的方式，甚至会被认为是一种反模式，应该避免使用。这是因为，忙等待需要让线程反复检查某个条件是否为真，因此，需要浪费大量宝贵的 CPU 资源在无用的活动上。那么，有没有更好的办法，既可以尽量减少处理器资源的浪费，又能够解决线程间数据依赖的问题呢？答案是有的，这个方法就是使用条件变量。

来看下面这个例子：

```
#include <threads.h>
#include <stdio.h>
mtx_t mutex;
cnd_t cond;  // 定义一个条件变量；
int done = 0;
int run(void* data) {
  mtx_lock(&mutex);
  done = 1;
  cnd_signal(&cond);  // 通知等待中的线程；
  mtx_unlock(&mutex);
  return thrd_success;
}
int main(void) {
#ifndef __STDC_NO_THREADS__
  mtx_init(&mutex, mtx_plain);
  cnd_init(&cond);  // 初始化条件变量；
  thrd_t thread;
  thrd_create(&thread, run, NULL);
  mtx_lock(&mutex);
  while (done == 0) {
    cnd_wait(&cond, &mutex);  // 让当前线程进入等待队列；
  }
  mtx_unlock(&mutex);
  printf("The value of done is: %d", done);
  mtx_destroy(&mutex);
  cnd_destroy(&cond);  // 销毁条件变量；
#endif
  return 0;
}
```

这段代码的基本逻辑与上一小节的例子类似。从第 23 行开始的代码，执行前需要等待 run 线程首先将全局变量 done 的值修改为 1。代码的第 15~16 行，我们初始化了需要使用的互斥量对象与条件变量对象。在 main 线程对应代码的第 19~23 行，我们使用了与条件变量相关的函数 cnd_wait。该函数在被调用时，需要当前线程获得一个互斥锁，并将其作为实参传递给它，函数调用后锁会被释放。同时，所有执行到此处的线程都将被阻塞。

接下来，让我们把目光移到 run 线程。

在 run 线程代码的第 8 行，我们将变量 done 的值修改为 1。紧接着，通过调用函数 cnd_signal，run 线程得以“通知”所有之前被阻塞在函数 cnd_wait 处的线程，来让它们中的一个可以继续运行。当然，在这个例子中，我们只有 main 函数对应的一个线程。此时，互斥量将被重新上锁，main 线程将继续执行接下来的指令。在代码的第 24~26 行，

它打印出了全局变量 done 的值，并销毁了互斥量与条件变量对象。最后，程序执行完毕。

可以看到，实际上，**条件变量为我们提供了一种线程间的“通知”能力**。某个线程可以在完成了某件事情后，通知并唤醒等待线程，让其继续工作，完成接下来的任务。而在这个过程中，我们不需要通过忙等待的方式，让线程频繁查询标志量。因此，CPU 资源得到了更好的利用。

这里，我向你提一个小问题：为什么我们在代码的第 20 行使用 `while` 语句，而不是 `if` 语句呢？欢迎在评论区告诉我你的答案。

在并发编程中，条件变量是一个十分强大的武器。通过它，我们可以进一步实现监视器（Monitor）、管程等工具和同步原语。而且，它也可以很好地解决经典的生产者 - 消费者问题。如果你对这部分内容感兴趣，可以参考《C++ Concurrency in Action》和《现代操作系统》等书，来进行更加深入的学习。虽然它们并不会专门介绍基于 C 语言的并发编程，但其中的很多概念，甚至 C++ 接口，与 C 语言都是类似和相通的。

除了上述代码中用到的条件变量方法外，C 标准库还提供了另外两个常用函数。我将它们整理在了下面的表格中，供你参考。

函数名	功能描述
cnd_broadcast	唤醒所有在等待某个条件变量的线程
cnd_timedwait	带超时限制的 cnd_wait

最后，让我们再回过头来，看看与线程直接相关的另一个内容，线程本地变量。

使用线程本地变量

除了可以共享存在于进程内的全局变量外，线程还可以拥有属于它自己的线程本地变量（TLS）。

顾名思义，线程本地变量的值仅能够在某个具体线程的生存期内可用。变量的实际存储空间会在线程开始时分配，线程结束时回收。线程不会对这些变量的读写操作产生数据竞争。我们来看一个例子：

复制代码

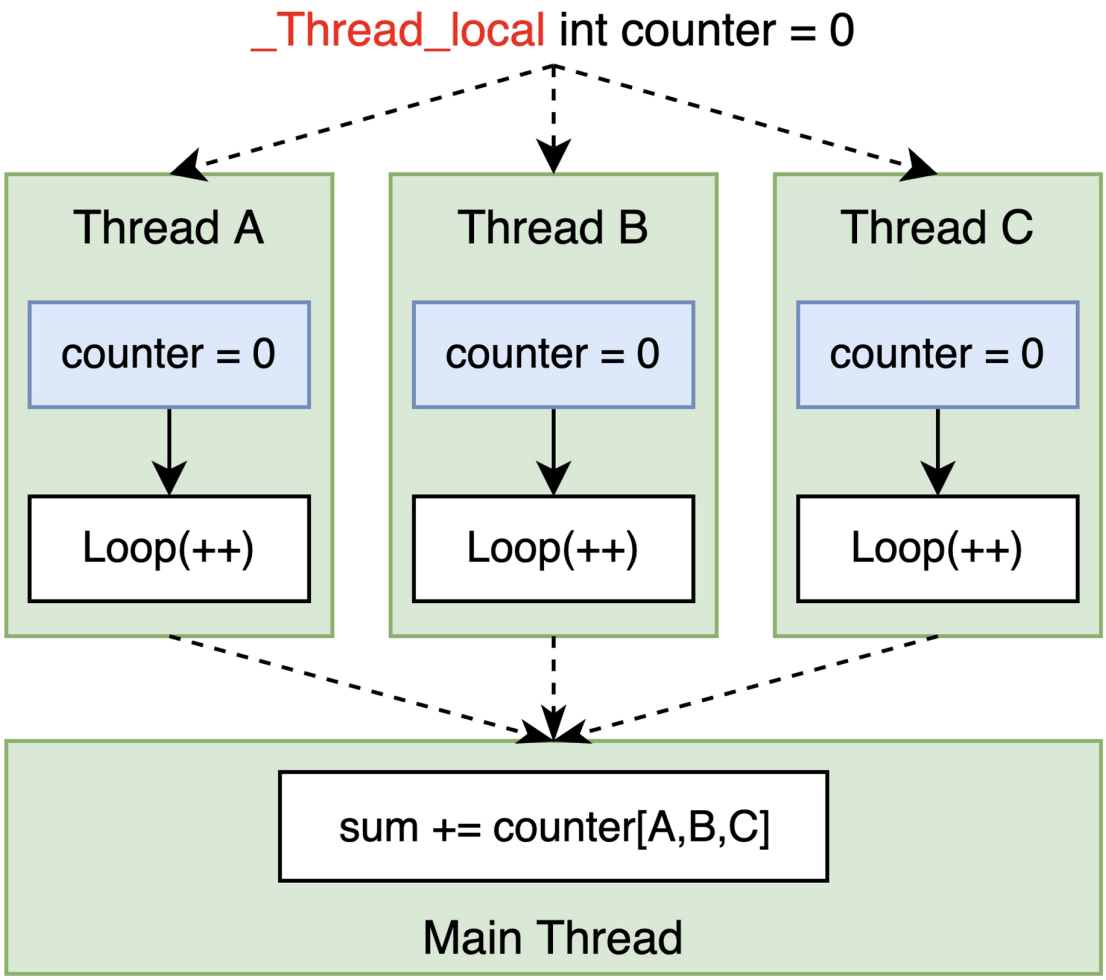
```
1  #include <stdio.h>
2  #include <threads.h>
3  #include <stdatomic.h>
4  #define THREAD_COUNT 10
5  #define THREAD_LOOP 10000
6  _Thread_local int counter = 0;  // 定义线程本地变量；
7  int run(void *data) {
8    for (int i = 0; i < THREAD_LOOP; ++i)
9      counter += 1;  // 更新当前线程所属的 counter 变量值；
10   return counter;
11 }
12 int main(int argc, char const *argv[]) {
13   thrd_t threads[THREAD_COUNT];
14   int sum = 0, result = 0;
15   for (int i = 0; i < THREAD_COUNT; ++i)
16     thrd_create(&threads[i], run, NULL);
17   for (int i = 0; i < THREAD_COUNT; ++i) {
18     thrd_join(threads[i], &result);
19     sum += result;  // 累加每个线程的计算值；
20   }
21   printf("The value of count is %d.\n", sum);
22   return 0;
23 }
```

可以看到，这段代码的逻辑十分简单：我们创建了 10 个（对应 THREAD_COUNT）线程，让它们同时对全局变量 counter 进行累加，并持续 10000 次（对应 THREAD_LOOP）。然后，在 main 线程的最后，我们将累加后的值打印了出来。

看到这里，相信你的第一感觉肯定是：应该通过互斥锁或原子操作等方式，来防止多个线程在对 counter 变量进行修改时产生数据竞争。但在这里，我却没有这样做，而是采用了一种更加便捷的方式。这一切，都要得益于线程本地变量的存在。

在代码的第 6 行，我们使用 _Thread_local 关键字（也可以使用宏 thread_local），将全局变量 counter 标记为线程本地变量。这意味着，每个线程都会在创建时生成仅属于当前线程的变量 counter。因此，当本线程在对 counter 变量进行累加时，便不会受到其他线程的影响。而当线程退出时，通过代码第 18 行的 thrd_join，我们得以在 main 线程中

将每个结束线程返回的，各自的 counter 值再进行统一累加，从而得到最后的计算结果。你可以通过下图来直观地理解这个过程。



总之，线程本地变量为我们提供了另一种可以避免数据竞争的方式。除此之外，它也可以被用来存储线程独有的一些信息，比如 errno 的值。

我们在上面代码中使用的是**以关键字来定义线程本地变量的方式**，除此之外，标准库还提供了一系列的函数，可以实现同样的目的。但不同之处在于，通过 tss_create 等函数来创建线程本地变量时，还可以为其指定对应的析构函数。这样，当线程退出时，便可以确保相应的线程本地资源（比如堆内存）能够以正确的方式被清理。这里，我将相关的函数列在了下面的表格中，供你参考。你也可以点击这个链接查看更多信息。

函数名	功能描述
tss_create	创建新的线程本地变量，并指定其析构函数
tss_get	读取一个线程本地变量的值
tss_set	修改一个线程本地变量的值
tss_delete	销毁一个线程本地变量，并释放其资源

总结

好了，讲到这里，今天的内容也就基本结束了。最后我来给你总结一下。

在本讲中，我主要介绍了有关互斥量、原子操作、条件变量，以及线程本地变量的相关内容。合理地使用这些方式，我们就可以避免多线程应用经常会遇到的，由于数据竞争、竞态条件，以及指令重排引起的问题。

其中，互斥量让我们可以通过对它进行**加锁与解锁**的方式，来限制多个线程的执行，以让它们有序地使用共享资源。在 C 语言中，互斥量被分为三种类型，mtx_plain 为最基本类型，mtx_recursive 可以被用在需要重复加锁的场景中，而 mtx_timed 则使得互斥量具有了超时属性。通过与 mtx_timedlock 结合使用，它可以让线程在给互斥量加锁时，只尝试有限的一段时间。

原子操作是一种更便捷的可以用来避免数据竞争的方式。通过使用 `_Atomic` 关键字，我们可以将变量定义为原子类型。而当线程访问该类型变量时，便可**按照“不可分割”的形式，一次性完成整个操作**。不仅如此，在进行原子操作时，还可以同时指定操作需要满足的内存顺序。原子操作的实现通常依赖于所在平台的特殊机器指令，而 C 标准库则通过直接提供常用同步原语的方式，帮我们屏蔽了这些细节。

条件变量提供了线程间的通知能力。它可以让线程在完成某件事情后，通知需要进行后续处理的等待线程，从而让具有数据依赖关系的线程以一种更加高效的方式进行同步。除此之外，条件变量还可被用于实现监视器、管程等更多复杂的同步机制。

最后，线程本地变量也是一种可用于解决数据竞争的常用方式。具体的操作是在 C 代码中，为全局变量添加 `_Thread_local` 关键字。**这样，就会仅在线程创建时，才生成仅属**

于当前线程的本地同名变量。 因此，当前线程对该变量的修改便不会被其他线程影响。

思考题

x86-64 指令集中的 mfence、lfence 与 sfence 指令，它们具体有什么作用呢？试着查找资料了解一下，并在评论区分享你的发现。

今天的课程到这里就结束了，希望可以帮助到你，也希望你在下方的留言区和我一起讨论。同时，欢迎你把这节课分享给你的朋友或同事，我们一起交流。

上一篇 13 | 标准库：你需要了解的 C 并发编程基础知识有哪些？

下一篇 课堂答疑（一）| 前置篇、C 核心语法实现篇问题集锦

精选留言 (1)

写留言

liu_liu
2022-01-14

使用 while 的原因是，当阻塞的线程被重新调度运行时，done 的值可能被改变了，不是预期值。