第4章 线程管理：ACE的同步和线程管理机制

ACE拥有许多不同的用于创建和管理多线程程序的类。在这一章里，我们将查看ACE中的一些线程管理机制。在一开始，我们将查看那些简单的线程包装类，它们的管理功能很少。但是，随着内容的进展，我们将查看ACE_Thread_Manager中的更为强大的管理机制。ACE还拥有一组非常全面的处理线程同步的类。这些类也将在本章讲述。

4.1 创建和取消线程

在不同的平台上，有着若干不同的用于线程管理的接口。其中包括POSIX pthreads接口、Solaris线程、Win32线程等等。这些接口提供了相同或是相似的功能，但是它们的API的差别却极为悬殊。这就导致了困难、麻烦和易错的编程，因为应用程序员必须熟悉不同平台上的若干接口。而且，这样写下的程序，是不可移植和不灵活的。

ACE_Thread提供了对OS的线程调用的简单包装，这些调用处理线程创建、挂起、取消和删除等问题。它提供给应用程序员一个简单易用的接口，可以在不同的线程API间移植。ACE_Thread是非常“瘦”的包装，有着很少的开销。其大多数方法都是内联的，因而等价于对底层OS专有线程接口的直接调用。ACE_Thread中的所有方法都是静态的，而且该类一般不进行实例化。

下面的例子演示怎样使用ACE_Thread包装类创建、生成和联接（join）线程。

例4-1

```
#include "ace/Thread.h"
#include "ace/Synch.h"

static int number=0;
static int seed = 0;

static void* worker(void *arg)
{
ACE_UNUSED_ARG(arg);
ACE_DEBUG((LM_DEBUG,"Thread (%t) Created to do some work"));
::number++;
ACE_DEBUG((LM_DEBUG," and number is %d\n",::number));

//Let the other guy go while I fall asleep for a random period
//of time
ACE_OS::sleep(ACE_OS::rand()%2);
```

```
//Exiting now
ACE_DEBUG((LM_DEBUG,
      "\t\t Thread (%t) Done! \t The number is now: %d\n",number));

return 0;
}

int main(int argc, char *argv[])
{
if(argc<2)
{
      ACE_DEBUG((LM_DEBUG,"Usage: %s <number of threads>\n", argv[0]));
      ACE_OS::exit(1);
}

ACE_OS::srand(::seed);

//setup the random number generator
int n_threads= ACE_OS::atoi(argv[1]);

//number of threads to spawn
ACE_thread_t *threadID = new ACE_thread_t[n_threads+1];
ACE_hthread_t *threadHandles = new ACE_hthread_t[n_threads+1];
if(ACE_Thread::spawn_n(threadID, //id's for each of the threads
            n_threads, //number of threads to spawn
            (ACE_THR_FUNC)worker, //entry point for new thread
            0, //args to worker
            THR_JOINABLE | THR_NEW_LWP, //flags
            ACE_DEFAULT_THREAD_PRIORITY,
            0, 0, threadHandles)==-1)
      ACE_DEBUG((LM_DEBUG,"Error in spawning thread\n"));

//spawn n_threads
for(int i=0; i<n_threads; i++)
      ACE_Thread::join(threadHandles[i]);

//Wait for all the threads to exit before you let the main fall through
//and have the process exit.
return 0;
}
```

在这个简单的例子中，创建了n_thread个工作者线程。每个线程都执行程序中定义的worker()函数。线程是通过使用ACE_Thread::spawn_n()调用创建的。要作为线程的执行启动点调用的函数的指针（在此例中为worker()函数）被作为参数传入该调用中。要注意的重点是ACE_Thread::spawn_n()要求所有的线程启动函数（方法）必须是静态的或全局的（就如同直接使用OS线程API时所要求的一样）。

一旦工作者函数启动，它将全局变量number的值加1，报告它的当前值，然后进入休眠状态，以把处理器让给其他线程。sleep()休眠一段随机长度的时间。在线程醒来后，它将number的当前值通知给用户，然后退出worker()函数。

一旦线程从它的启动函数返回，它在线程库上隐含地发出线程exit()调用并退出。这样一旦“掉出”worker()函数，工作者线程也就退出了。负责创建工作者线程的主线程，在退出之前“等待”所有其他的线程完成它们的执行并退出。当主线程退出时（通过退出main()函数），整个进程也将被销毁。这之所以会发生是因为每当线程退出main()函数时，都会隐含地调用exit(3c)函数。因此，如果主线程没有被强制等待其他线程结束，当它死掉时，进程将会被自动销毁，并在它的所有工作者线程完成工作**之前**销毁它们！

上面所说的等待是通过使用ACE_Thread::join()调用来完成的。该方法的参数是你想要主线程与之联接的线程的句柄（ACE_hthread_t）。

在此例中有若干事实值得注意。首先，在该类中没有可供我们调用的管理功能，用以在内部记住应用所派生的线程的ID。这使得我们难以联接（join()）、杀死（kill()）或是一般性地管理我们派生的线程。在本章后面讲述的ACE_Thread_Manager缓解了这些问题，一般说来，应该使用ACE_Thread_Manager而不是线程包装API。

其次，在程序中没有使用同步原语来保护全局数据。在此例中，它们并不是必须的，因为所有的线程都只对全局变量执行一次加操作。但是在实际应用中，为了保护所有共享互斥数据（全局或静态变量），比如说全局的number变量，“锁”将会是必需的。

4.2 ACE同步原语

ACE有若干可用于同步目的的类。这些类可划分为以下范畴：

- ACE Lock类属
- ACE Guard类属
- ACE Condition类属
- 杂项ACE Synchronization类

4.2.1 ACE Lock类属

锁类属包含的类包装简单的锁定机制，比如互斥体、信号量、读/写互斥体和令牌。在这一类属中可用的类在表4-1中显示。每个类名后都有对用法和用途的简要描述：

名字	描述
ACE_Mutex	封装互斥机制（根据平台，可以是process-...

ACE_Mutex	封装互斥机制（根据平台，可以是mutex_t、pthread_mutex_t等等）的包装类，用于提供简单而有效的机制来使对共享资源的访问序列化。它与二元信号量（binary semaphore）的功能相类似。可被用于线程和进程间的互斥。
ACE_Thread_Mutex	可用于替换ACE_Mutex，专用于线程同步。
ACE_Process_Mutex	可用于替换ACE_Mutex，专用于进程同步。
ACE_NULL_Mutex	提供了ACE_Mutex接口的“无为”（do-nothing）实现，可在不需要同步时用作替换。
ACE_RW_Mutex	封装读者/作者锁的包装类。它们是分别为读和写进行获取的锁，在没有作者在写的时候，多个读者可以同时进行读取。
ACE_RW_Thread_Mutex	可用于替换ACE_RW_Mutex，专用于线程同步。
ACE_RW_Process_Mutex	可用于替换ACE_RW_Mutex，专用于进程同步。
ACE_Semaphore	这些类实现计数信号量，在有固定数量的线程可以同时访问一个资源时很有用。在OS不提供这种同步机制的情况下，可通过互斥体来进行模拟。
ACE_Thread_Semaphore	应被用于替换ACE_Semaphore，专用于线程同步。
ACE_Process_Semaphore	应被用于替换ACE_Semaphore，专用于进程同步。
ACE_Token	提供“递归互斥体”（recursive mutex），也就是，当前持有某令牌的线程可以多次重新获取它，而不会阻塞。而且，当令牌被释放时，它确保下一个正阻塞并等待此令牌的线程就是下一个被放行的线程。
ACE_Null_Token	令牌接口的“无为”（do-nothing）实现，在你知道不会出现多个线程时使用。
ACE_Lock	定义锁定接口的接口类。一个纯虚类，如果使用的话，必须承受虚函数调用开销。
ACE_Lock_Adapter	基于模板的适配器，允许将前面提到的任意一种锁定机制适配到ACE_Lock接口。

表4-1 ACE锁类属中的类

表4-1中描述的类都支持同样的接口。但是，在任何继承层次中，这些类都是**互不关联**的。在ACE中，锁通常用模板来参数化，因为，在大多数情况下，使用虚函数调用的开销都是不可接受的。使用模板使得程序员可获得相当程度的灵活性。他可以在编译时（但不是在运行时）选择他想要使用的的锁定机制的类型。然而，在某些情形中，程序员仍可能需要使用动态绑定和替换（substitution）；对于这些情况，ACE提供了ACE_Lock和ACE_Lock_Adapter类。

4.2.1.1 使用互斥体类

互斥体实现了“*互相排斥*”（mutual exclusion）同步的简单形式（所以名为互斥体(mutex)）。互斥体禁止多个线程同时进入受保护的代码“*临界区*”（critical section）。因此，在任意时刻，只有一个线程被允许进入这样的代码保护区。

任何线程在进入临界区之前，必须*获取*（acquire）与此区域相关联的互斥体的所有权。如果已有另一线程拥有了临界区的互斥体，其他线程就不能再进入其中。这些线程必须等待，直到当前的属主线程*释放*（release）该互斥体。

什么时候需要使用互斥体呢？互斥体用于保护共享的易变代码，也就是，全局或静态数据。这样的数据必须通过互斥体进行保护，以防止它们在多个线程同时访问时损坏。

下面的例子演示ACE_Thread_Mutex类的使用。注意在此处很容易用ACE_Mutex替换ACE_Thread_Mutex类，因为它们拥有同样的接口。

例4-2

```
#include "ace/Synch.h"
#include "ace/Thread.h"

//Arguments that are to be passed to the worker thread are passed
//through this struct.
struct Args
{
public:
Args(int iterations): mutex_(),iterations_(iterations){}
ACE_Thread_Mutex mutex_;
int iterations_;
};

//The starting point for the worker threads
static void* worker(void*arguments)
{
Args *arg= (Args*) arguments;
for(int i=0;i<arg->iterations_;i++)
{
```

```
      ACE_DEBUG((LM_DEBUG,
            "(%t) Trying to get a hold of this iteration\n"));

      //This is our critical section
      arg->mutex_.acquire();
      ACE_DEBUG((LM_DEBUG,"(%t) This is iteration number %d\n",i));
      ACE_OS::sleep(2);

      //simulate critical work
      arg->mutex_.release();
}

return 0;
}

int main(int argc, char*argv[])
{
if(argc<2)
{
      ACE_OS::printf("Usage: %s <number_of_threads>
            <number_of_iterations>\n", argv[0]);
      ACE_OS::exit(1);
}

Args arg(ACE_OS::atoi(argv[2]));

//Setup the arguments
int n_threads = ACE_OS::atoi(argv[1]);

//determine the number of threads to be spawned.
ACE_thread_t *threadID = new ACE_thread_t[n_threads+1];
ACE_hthread_t *threadHandles = new ACE_hthread_t[n_threads+1];
if(ACE_Thread::spawn_n(threadID, //id's for each of the threads
                        n_threads, //number of threads to spawn
                        (ACE_THR_FUNC)worker, //entry point for new thread
                        0, //args to worker
                        THR_JOINABLE | THR_NEW_LWP, //flags
                        ACE_DEFAULT_THREAD_PRIORITY,
                        0, 0, threadHandles)==-1)
      ACE_DEBUG((LM_DEBUG,"Error in spawning thread\n"));
```

```
//spawn n_threads
for(int i=0; i<n_threads; i++)
      ACE_Thread::join(threadHandles[i]);

//Wait for all the threads to exit before you let the main fall through
//and have the process exit.
return 0;
}
```

在上面的例子中，ACE_Thread包装类用于生成多个线程来执行worker()函数，就如同在前面的例子里一样。Arg对象作为参数传入各个线程，在该对象中含有循环要执行的次数，以及将要使用的互斥体。

在此例中，一开始，每个线程立即进入for循环。一进入循环，线程就进入了临界区。在临界区内完成的工作使用ACE_Thread_Mutex互斥体对象进行保护。该对象由主线程作为参数传给工作者线程。临界区控制是通过在ACE_Thread_Mutex对象上发出acquire()调用，从而获取互斥体的所有权来完成的。一旦互斥体被获取，没有其他线程能够再进入这一代码区。临界区控制是通过使用release()调用来释放的。一旦互斥体的所有权被放弃，就会唤醒所有其他在等待的线程。这些线程随即相互竞争，以获得互斥体的所有权。第一个试图获取所有权的线程会进入临界区。

4.2.1.2 将锁和锁适配器（Lock Adapter）用于动态绑定

如前面所提到的，各种互斥体锁应被直接用于你的代码，或者，如果需要灵活性，作为模板参数来使用。但是，如果你需要动态地（也就是在运行时）改变你的代码所用锁的类型，就无法使用这些锁。

为应对这个问题，ACE拥有ACE_Lock和ACE_Lock_Adapter类，它们可用于这样的运行时替换（substitution）。

下面的例子演示ACE_Lock类和ACE_Lock_Adapter怎样为应用程序员提供方便，和锁定机制一起使用动态绑定和替换。

例4-3

```
#include "ace/Synch.h"
#include "ace/Thread.h"

//Arguments that are to be passed to the worker thread are passed
//through this class.
struct Args
{
public:
Args(ACE_Lock* lock,int iterations):
      mutex_(lock),iterations_(iterations){}
ACE_Lock* mutex_;
int iterations_;
```

```
};

//The starting point for the worker threads
static void* worker(void*arguments)
{
Args *arg= (Args*) arguments;
for(int i=0;i<arg->iterations_;i++)
{
      ACE_DEBUG((LM_DEBUG,
            "(%t) Trying to get a hold of this iteration\n"));

      //This is our critical section
      arg->mutex_->acquire();
      ACE_DEBUG((LM_DEBUG,"(%t) This is iteration number %d\n",i));
      ACE_OS::sleep(2);

      //simulate critical work
      arg->mutex_->release();
}

return 0;
}

int main(int argc, char*argv[])
{
if(argc<4)
{
      ACE_OS::printf("Usage: %s <number_of_threads>
            <number_of_iterations> <lock_type>\n", argv[0]);
      ACE_OS::exit(1);
}

//Polymorphic lock that will be used by the application
ACE_Lock *lock;

//Decide which lock you want to use at run time,
//recursive or non-recursive.
if(ACE_OS::strcmp(argv[3],"Recursive"))
      lock=new ACE_Lock_Adapter<ACE_Recursive_Thread_Mutex>;
else
      lock=new ACE_Lock_Adapter<ACE_Thread_Mutex>
```

```
//Setup the arguments
Args arg(lock,ACE_OS::atoi(argv[2]));


//spawn threads and wait as in previous examples..
}
```

在此例中，和前面的例子唯一的不同是ACE_Lock类是和ACE_Lock_Adapter一起使用的，以便能提供动态绑定。底层的锁定机制使用递归还是非递归互斥体，是在程序运行时由命令行参数决定的。使用动态绑定的好处是实际的锁定机制可以在运行时被替换。缺点是现在对锁的每次调用都需要负担额外的经由虚函数表的间接层次。

4.2.1.3 使用令牌（Token）

如表4-1中所提到的，ACE_Token类提供所谓的“递归互斥体”，它可以被最初获得它的同一线程进行多次重新获取。ACE_Token类还确保所有试图获取它的线程按严格的FIFO（先进先出）顺序排序。

递归锁允许同一线程多次获取同一个锁。线程不会因为试图获取它已经拥有的锁而死锁。这些类型的锁能在各种不同的情况下派上用场。例如，如果你用一个锁来维护跟踪流的一致性，你可能希望这个锁是递归的，因为某个方法可以调用一个跟踪例程，获取锁，被信号中断，然后再尝试获取这个跟踪锁。如果锁是非递归的，线程将会在这里锁住它自己。你会发现很多其他需要递归锁的有趣应用。重要的是要记住，你获取递归锁多少次，就**必须释放**它多少次。

在SunOS 5.x上运行例4-3，释放锁的线程常常也是重新获得它的线程（大约90%的情况是这样）。但是如果你采用ACE_Token类作为锁定机制来运行这个例子，每个线程都会轮流获得令牌，然后有序地把机会让给下一个线程。

尽管ACE_Token作为所谓的递归锁非常有用，它们实际上是更大的“令牌管理”构架的一部分。该构架允许你维护数据存储中数据的一致性。遗憾的是，这已经超出了此教程的范围。

例4-4

```
#include "ace/Token.h"
#include "ace/Thread.h"


//Arguments that are to be passed to the worker thread are passed
//through this struct.
struct Args
{
public:
Args(int iterations):
      token_(“myToken”),iterations_(iterations){}
ACE_Token token_;
```

```
int iterations_;
};

//The starting point for the worker threads
static void* worker(void*arguments)
{
Args *arg= (Args*) arguments;
for(int i=0;i<arg->iterations_;i++)
{
      ACE_DEBUG((LM_DEBUG,"(%t) Trying to get a hold of this iteration\n"));

      //This is our critical section
      arg->token_.acquire();
      ACE_DEBUG((LM_DEBUG,"(%t) This is iteration number %d\n",i));

      //work
      ACE_OS::sleep(2);
      arg->token_.release();
}

return 0;
}

int main(int argc, char*argv[])
{
//same as previous examples..
}
```

4.2.2 ACE守卫（Guard）类属

ACE中的守卫用于自动获取和释放锁。守卫类的对象定义一个代码块，在其上获取一个锁。在退出此代码块时，锁被自动释放。

ACE中的守卫类是一种模板，它通过所需锁定机制的类型来参数化。底层的锁可以是ACE Lock类属中的任何类，也就是，任何互斥体或锁类。它是这样工作的：对象的构造器获取锁，析构器释放锁。表4-2列出了ACE中可用的守卫：

名字	描述

ACE_Guard	自动在底层锁上调用acquire()和release()。任何ACE Lock类属中的锁都可以作为它的模板参数传入。
ACE_Read_Guard	自动在底层锁上调用acquire()和release()。
ACE_Write_Guard	自动在底层锁上调用acquire()和release()。

表4-2 ACE中的守卫

下面的例子演示怎样使用看守：

例4-5

```
#include "ace/Synch.h"
#include "ace/Thread.h"

//Arguments that are to be passed to the worker thread are passed
//through this class.
class Args
{
public:
Args(int iterations):
      mutex_(),iterations_(iterations){}
ACE_Thread_Mutex mutex_;
int iterations_;
};

//The starting point for the worker threads
static void* worker(void*arguments)
{
Args *arg= (Args*) arguments;
for(int i=0;i<arg->iterations_;i++)
{
      ACE_DEBUG((LM_DEBUG,"(%t) Trying to get a hold of this iteration\n"));
            ACE_Guard<ACE_Thread_Mutex> guard(arg->mutex_);
      {
            //This is our critical section
            ACE_DEBUG((LM_DEBUG,"(%t) This is iteration number %d\n",i));

            //work
```

```
            ACE_OS::sleep(2);
      }//end critical section
}

return 0;
}

int main(int argc, char*argv[])
{
//same as previous example
}
```

在上面的例子中，守卫在工作者线程中管理临界区。守卫对象从ACE_Guard模板类创建而来。守卫应使用的锁的类型被传给模板。通过将所需获取的实际锁对象经由守卫对象的构造器传入，程序创建守卫对象。该锁由ACE_Guard在内部自动获取，所以for循环中的区域就是受保护的临界区。一旦出了作用域，看守对象就会被自动删除，锁也随之被释放。

守卫是很有用的，因为它们保证一旦你获取了一个锁，你总是会释放它（当然，除非你的线程因为无法预料的情况而死掉）。在有许多不同的返回路径的复杂方法中，这被证明为是极其有用的。

4.2.3 ACE条件（Condition）类属

ACE_Condition类是针对OS条件变量原语的包装类。那么，到底什么是条件变量呢？

线程常常需要特定条件被满足才能继续它的操作。例如，设想线程需要在全局消息队列里插入消息。在插入任何消息之前，它必须检查在消息队列里是否有空闲空间。如果消息队列在“满”状态，它就什么也不能做，而必须进行休眠，过一会再重试。就是说，在访问全局资源之前，某个*条件*必须为真。然后，当另外的线程空出消息队列时，应该有方法通知或发信号给原来的线程：在消息队列里有位置了，现在应该再次尝试插入消息。这可以使用条件变量来完成。条件变量不是被用作互斥原语，而是用作特定条件已经满足的指示器。

在使用条件变量时，你的程序应该完成以下步骤：

- 获取全局资源（例如，消息队列）的锁（互斥体）。
- 检查条件（例如，消息队列里有空间吗？）。
- 如果条件失败，调用条件变量的wait()方法。等待在未来条件变为真。
- 当另一线程在全局资源上执行操作时，它发信号（signal()）给所有其他在此资源上测试条件的线程（例如，另一线程从消息队列中取出一个消息，然后通过条件变量发送信号，以使阻塞在wait()上的线程能够再尝试将它们的消息插入队列）。
- 在醒来之后，重新检查条件现在是否为真。如为真，则在全局资源上执行操作（例如，将消息插入全局消息队列）

需要特别注意的是，在阻塞在wait调用中之前，条件变量机制（也就是ACE_Cond）负责释放全局资源上的互斥体。如果没有进行此操作，将没有其他的线程能够在此资源上工作（该资源是条件改变的原因）。同样，一旦阻塞线程收到信号、重又醒来，它在检查条件之前会在内部重新获取锁。

下面的例子是对本章第一个例子的改写。如果你还记得，我们曾说过使用ACE_Thread::join()调用可以使主线程等待其他的线程结束。另一种达到同样目的的方法是使用条件变量，它使主线程在退出之前等待“所有线程已经结束”条件为真。最后一个线程可以通过条件变量发信号给等待中的主线程，通知它所有线程已经结束、而它是最后一个。随后主线程继续执行，退出应用并销毁进程。如下如示：

例4-6

```
#include "ace/Thread.h"
#include "ace/Synch.h"

static int number=0;
static int seed=0;

class Args
{
public:
Args(ACE_Condition<ACE_Thread_Mutex> *cond, int threads):
      cond_(cond), threads_(threads){}
ACE_Condition<ACE_Thread_Mutex> *cond_;
int threads_;
};

static void* worker(void *arguments)
{
Args *arg= (Args*)arguments;
ACE_DEBUG((LM_DEBUG,"Thread (%t) Created to do some work\n"));
::number++;

//Work
ACE_OS::sleep(ACE_OS::rand()%2);

//Exiting now
ACE_DEBUG((LM_DEBUG,
      "\tThread (%t) Done! \n\tThe number is now: %d\n",number));

//If all threads are done signal main thread that
//program can now exit
if(number==arg->threads_)
{
      ACE_DEBUG((LM_DEBUG,
            "(%t) Last Thread!\n All threads have done their job!
            Signal main thread\n"));
```

```
      arg->cond_->signal();
}

return 0;
}

int main(int argc, char *argv[])
{
if(argc<2)
{
      ACE_DEBUG((LM_DEBUG,
            "Usage: %s <number of threads>\n", argv[0]));
      ACE_OS::exit(1);
}

int n_threads=ACE_OS::atoi(argv[1]);

//Setup the random number generator
ACE_OS::srand(::seed);

//Setup arguments for threads
ACE_Thread_Mutex mutex;
ACE_Condition<ACE_Thread_Mutex> cond(mutex);
Args arg(&cond,n_threads);

//Spawn off n_threads number of threads
for(int i=0; i<n_threads; i++)
{
      if(ACE_Thread::spawn((ACE_THR_FUNC)worker,(void*)&arg,
                                        THR_DETACHED|THR_NEW_LWP)==-1)
            ACE_DEBUG((LM_DEBUG,"Error in spawning thread\n"));
}

//Wait for signal indicating that all threads are done and program
//can exit. The global resource here is “number” and the condition
//that the condition variable is waiting for is number==n_threads.
mutex.acquire();

while(number!=n_threads)
      cond.wait();
ACE_DEBUG((LM_DEBUG,"(%t) Main Thread got signal. Program
```

```
      exiting..\n"));

mutex.release();

ACE_OS::exit(0);
}
```

注意主线程首先获取一个互斥体，然后对条件进行测试。如果条件不为真，主线程就等待在此条件变量上。条件变量随即*自动*释放互斥体，并使主线程进入睡眠。条件变量总是像这样与互斥体一起使用。这是一种可如下描述的一般模式[1]：

while(expression NOT TRUE) wait on condition variable;

记住条件变量不是用于互斥，而是用于我们所描述的发送信号功能。

除了ACE_Condition类，ACE还包括ACE_Condition_Thread_Mutex类，它使用ACE_Thread_Mutex作为全局资源的底层锁定机制。

4.2.4 杂项同步类

除了上面描述的同步类，ACE还包括其他一些同步类，比如ACE_Barrier和ACE_Atomic_Op。

4.2.4.1 ACE中的栅栏（Barrier）

栅栏有一个好名字，因为它恰切地描述了栅栏应做的事情。一组线程可以使用栅栏来进行共同的相互同步。组中的每个线程各自执行，直到到达栅栏，就阻塞在那里。在所有相关线程到达栅栏后，它们就全部继续它们的执行。就是说，它们一个接一个地阻塞，等待其他的线程到达栅栏；一旦所有线程都到达了它们的执行路径中的“栅栏点”，它们就一起重新启动。

在ACE中，栅栏在ACE_Barrier类中实现。在栅栏对象被实例化时，它将要等待的线程的数目会作为参数传入。一旦到达执行路径中的“栅栏点”，每个线程都在栅栏对象上发出wait()调用。它们在这里阻塞，直到其他线程到达它们各自的“栅栏点”，然后再一起继续执行。当栅栏从相关线程那里接收了适当数目的wait()调用时，它就同时唤醒所有阻塞的线程。

下面的例子演示怎样通过ACE使用栅栏：

例4-7

```
#include "ace/Thread.h"
#include "ace/Synch.h"

static int number=0;
```

```
static int seed=0;

class Args
{
public:
Args(ACE_Barrier *barrier): barrier_(barrier){}
ACE_Barrier *barrier_;
};

static void*
worker(void *arguments)
{
Args *arg= (Args*)arguments;
ACE_DEBUG((LM_DEBUG,"Thread (%t) Created to do some work\n"));
::number++;

//Work
ACE_OS::sleep(ACE_OS::rand()%2);

//Exiting now
ACE_DEBUG((LM_DEBUG,
           "\tThread (%t) Done! \n\tThe number is now: %d\n",number));

//Let the barrier know we are done.
arg->barrier_->wait();
ACE_DEBUG((LM_DEBUG,"Thread (%t) is exiting \n"));

return 0;
}

int main(int argc, char *argv[])
{
if(argc<2)
{
      ACE_DEBUG((LM_DEBUG,"Usage: %s <number of threads>\n", argv[0]));
      ACE_OS::exit(1);
}

int n_threads=ACE_OS::atoi(argv[1]);
ACE_DEBUG((LM_DEBUG,"Preparing to spawn %d threads",n_threads));
```

```
//Setup the random number generator
ACE_OS::srand(::seed);

//Setup arguments for threads
ACE_Barrier barrier(n_threads);
Args arg(&barrier);

//Spawn off n_threads number of threads
for(int i=0; i<n_threads; i++)
{
      if(ACE_Thread::spawn((ACE_THR_FUNC)worker,
                        (void*)&arg,THR_DETACHED|THR_NEW_LWP)==-1)
            ACE_DEBUG((LM_DEBUG,"Error in spawning thread\n"));
}

//Wait for all the other threads to let the main thread
// know that they are done using the barrier
barrier.wait();
ACE_DEBUG((LM_DEBUG,"(%t)Other threads are finished. Program exiting..\n"));
ACE_OS::sleep(2);
}
```

在此例中，主线程创建一个栅栏，并将其传递给工作者线程。每个工作者线程都在就要退出前在栅栏上调用wait()，从而使它们在完成工作后和就要退出前阻塞住。主线程也在就要退出前阻塞。一旦所有线程（包括主线程）执行结束，它们就会一起退出。

4.2.4.2 原子操作（Atomic Op）

ACE_Atomic_Op类用于将同步透明地参数化进基本的算术运算中。ACE_Atomic_Op是一种模板类，锁定机制和需要参数化的类型被作为参数传入其中。ACE是这样来实现此机制的：重载所有算术操作符，并确保在操作前获取锁，在操作后释放它。运算本身被委托给通过模板传入的的类。

下面的例子演示此类的用法：

例4-8

```
#include "ace/Synch.h"

//Global mutable and shared data on which we will perform simple
//arithmetic operations which will be protected.
ACE_Atomic_Op<ACE_Thread_Mutex,int> foo;
```

```
//The worker threads will start from here.
static void* worker(void *arg)
{
ACE_UNUSED_ARG(arg);

foo=5;
ACE_ASSERT (foo == 5);

++foo;
ACE_ASSERT (foo == 6);

--foo;
ACE_ASSERT (foo == 5);

foo += 10;
ACE_ASSERT (foo == 15);

foo -= 10;
ACE_ASSERT (foo == 5);

foo = 5L;
ACE_ASSERT (foo == 5);

return 0;
}

int main(int argc, char *argv[])
{
//spawn threads as in previous examples
}
```

在上面的程序中，在foo变量上执行了若干简单的算术运算。在运算之后，执行了断言检查来保证变量的值是它“应该是”的值。

你也许想知道为什么这样的算术原语（比如像上例中那样）需要同步。你一定认为增减运算本来就是原子的。

但是，这些运算通常**不是**原子的。CPU有可能将指令划分为三个步骤：读变量、增加或减少变量的值，以及回写。在这样的情况下，如果没有使用原子操作，就可能发生下面的情况：

- 线程一读变量，增加它的值，还未及将新值写回，就被OS调换出去。

- 线程二读取变量的旧值，增加它并写回新的增加了的值。
- 线程一用它自己的值覆盖了线程二的增量。

即使没有使用同步原语，上面的例程也*可能*并不会出错。原因是这种情况下的线程是计算绑定的，OS不会先占（pre-empt）这样的线程。但是，这样编写的代码是不安全的，因为你不能依赖OS调度器的工作方式。在大多数环境中，任何情况下同步关系都是非确定性的（因为实时效应，像页面错误或定时器的使用；或是因为实际上有多个物理处理器）。

4.3 使用ACE_THREAD_MANAGER进行线程管理

在前面所有的例子中，我们一直使用ACE_Thread包装类来创建和销毁线程。但是，该包装类的功能比较有限。ACE_Thread_Manager提供了ACE_Thread中的功能的超集。特别地，它增加了管理功能，以使启动、取消、挂起和恢复一组相关线程变得更为容易。它用于创建和销毁成组的线程和任务（ACE_Task是一种比线程更高级的构造，可在ACE中用于进行多线程编程。我们将在后面再来讨论任务）。它还提供了这样的功能：发送信号给一组线程，或是在一组线程上等待，而不是像我们在前面的例子中所看到的那样，以一种不可移植的方式来调用join()。

下面的例子演示怎样使用ACE_Thread_Manager创建一组线程，然后等待它们的完成。

例4-9

```
#include "ace/Thread_Manager.h"
#include "ace/Get_Opt.h"

static void* taskone(void*)
{
ACE_DEBUG((LM_DEBUG,"Thread:(%t)started Task one! \n"));
ACE_OS::sleep(2);
ACE_DEBUG((LM_DEBUG,"Thread:(%t)finished Task one!\n"));
return 0;
}

static void* tasktwo(void*)
{
ACE_DEBUG((LM_DEBUG,"Thread:(%t)started Task two!\n"));
ACE_OS::sleep(1);
ACE_DEBUG((LM_DEBUG,"Thread:(%t)finished Task two!\n"));
return 0;
}

static void print_usage_and_die()
{
```

```
ACE_DEBUG((LM_DEBUG,"Usage program_name
      -a<num threads for pool1> -b<num threads for pool2>"));
ACE_OS::exit(1);
}

int main(int argc, char* argv[])
{
int num_task_1;
int num_task_2;

if(argc<3)
      print_usage_and_die();

ACE_Get_Opt get_opt(argc,argv,"a:b:");

char c;
while( (c=get_opt())!=EOF)
{
      switch(c)
      {
      case 'a':
            num_task_1=ACE_OS::atoi(get_opt.optarg);
            break;
      case 'b':
            num_task_2=ACE_OS::atoi(get_opt.optarg);
            break;
      default:
            ACE_ERROR((LM_ERROR,"Unknown option\n"));
            ACE_OS::exit(1);
      }
}

//Spawn the first set of threads that work on task 1.
if(ACE_Thread_Manager::instance()->spawn_n(num_task_1,
            (ACE_THR_FUNC)taskone,//Execute task one
            0, //No arguments
            THR_NEW_LWP, //New Light Weight Process
            ACE_DEFAULT_THREAD_PRIORITY,
            1)==-1) //Group ID is 1
      ACE_ERROR((LM_ERROR,
            "Failure to spawn first group of threads: %p \n"));
```

```
//Spawn second set of threads that work on task 2.
if(ACE_Thread_Manager::instance()->spawn_n(num_task_2,
                        (ACE_THR_FUNC)tasktwo,//Execute task one
                        0, //No arguments
                        THR_NEW_LWP, //New Light Weight Process
                        ACE_DEFAULT_THREAD_PRIORITY,
                        2)==-1)//Group ID is 2
      ACE_ERROR((LM_ERROR,
            "Failure to spawn second group of threads: %p \n"));

//Wait for all tasks in grp 1 to exit
ACE_Thread_Manager::instance()->wait_grp(1);
ACE_DEBUG((LM_DEBUG,"Tasks in group 1 have exited! Continuing \n"));

//Wait for all tasks in grp 2 to exit
ACE_Thread_Manager::instance()->wait_grp(2);

ACE_DEBUG((LM_DEBUG,"Tasks in group 2 have exited! Continuing \n"));
}
```

下一个例子演示ACE_Thread_Manager中的挂起、恢复和协作式取消机制：

例4-10

```
// Test out the group management mechanisms provided by the
// ACE_Thread_Manager, including the group suspension and resumption,
//and cooperative thread cancellation mechanisms.
#include "ace/Thread_Manager.h"
static const int DEFAULT_THREADS = ACE_DEFAULT_THREADS;
static const int DEFAULT_ITERATIONS = 100000;

static void *
worker (int iterations)
{
for (int i = 0; i < iterations; i++)
{
      if ((i % 1000) == 0)
      {
            ACE_DEBUG ((LM_DEBUG,
```

```
                        "(%t) checking cancellation before iteration %d!\n",
                        i));


            //Before doing work check if you have been canceled. If so don't

            //do any more work.

            if (ACE_Thread_Manager::instance ()->testcancel

                              (ACE_Thread::self ()) != 0)

            {

                        ACE_DEBUG ((LM_DEBUG,

                              "(%t) has been canceled before iteration
                              %d!\n",i));

                        break;

            }

      }

}


return 0;

}


int main (int argc, char *argv[])

{

int n_threads = argc > 1 ? ACE_OS::atoi (argv[1]) : DEFAULT_THREADS;

int n_iterations = argc > 2 ? ACE_OS::atoi (argv[2]) : DEFAULT_ITERATIONS;


ACE_Thread_Manager *thr_mgr = ACE_Thread_Manager::instance ();


//Create a group of threads n_threads that will execute the worker

//function the spawn_n method returns the group ID for the group of

//threads that are spawned. The argument n_iterations is passed back

//to the worker. Notice that all threads are created detached.

int grp_id = thr_mgr->spawn_n (n_threads, ACE_THR_FUNC (worker),

            (void *) n_iterations,

            THR_NEW_LWP | THR_DETACHED);


      // Wait for 1 second and then suspend every thread in the group.

ACE_OS::sleep (1);

ACE_DEBUG ((LM_DEBUG, "(%t) suspending group\n"));

if (thr_mgr->suspend_grp (grp_id) == -1)

      ACE_ERROR ((LM_DEBUG, "(%t) %p\n", "Could not suspend_grp"));


// Wait for 1 more second and then resume every thread in the

// group.
```

```
ACE_OS::sleep (1);
ACE_DEBUG ((LM_DEBUG, "(%t) resuming group\n"));
if (thr_mgr->resume_grp (grp_id) == -1)
      ACE_ERROR ((LM_DEBUG, "(%t) %p\n", "resume_grp"));

// Wait for 1 more second and then cancel all the threads.
ACE_OS::sleep (ACE_Time_Value (1));
ACE_DEBUG ((LM_DEBUG, "(%t) canceling group\n"));
if (thr_mgr->cancel_grp (grp_id) == -1)
      ACE_ERROR ((LM_DEBUG, "(%t) %p\n", "cancel_grp"));

// Perform a barrier wait until all the threads have shut down.
thr_mgr->wait ();

return 0;
}
```

在此例中创建了n_threads个线程来执行worker函数。每个线程在worker函数中执行n_iterations次循环。当这些线程在worker函数中执行循环时，主线程将挂起（suspend()）、恢复（resume()）、并最终取消它们。Worker函数中的每个线程将使用ACE_Thread_Manager的testcancel()来检查取消情况。

4.4 线程专有存储（Thread Specific Storage）

如果单线程程序希望创建一个变量，其值能在多个函数调用间持续，它可以静态或全局地分配该数据。如果这是个多线程程序，这个全局或静态数据对于所有线程来说都是一样的。这可能是，也可能不是我们所期望的。例如，伪随机数生成器可能需要静态的或全局的整型种子变量，它不会因为多个线程同时改变它的值而受到影响。但是，在另外的情形中，对于各个线程来说，可能需要不同的全局或静态数据。例如，设想一个多线程的GUI应用，在其中各个窗口都运行在独立的线程中，并各拥有一个输入端口，窗口从中接收事件输入。这样的输入端口必须在窗口的各个函数调用之间保持“持续”，并且还必须是窗口专有的或私有的。可使用线程专有存储来满足此需求。像输入端口这样的结构可放在线程专有存储中，并可像逻辑上的静态或全局变量一样被访问；而实际上它对线程来说是私有的。

传统上，线程专有存储通过让人迷惑的底层操作系统API来实现。在ACE中，TSS通过使用ACE_TSS模板类来实现。需要成为线程专有的类被传入ACE_TSS模板，然后可以使用C++的->操作符来调用它的全部公共方法。

下面的例子演示在ACE中使用线程专有存储是多么简单：

例4-11

```
#include "ace/Synch.h"
#include "ace/Thread_Manager.h"

class DataType
{
public:
DataType():data(0){}
void increment(){ data++;}
void set(int new_data){ data=new_data;}
void decrement(){ data--;}
int get(){return data;}

private:
int data;
};

ACE_TSS<DataType> data;

static void* thread1(void*)
{
data->set(10);
ACE_DEBUG((LM_DEBUG,"(%t)The value of data is %d \n",data->get()));
for(int i=0;i<5;i++)
      data->increment();
ACE_DEBUG((LM_DEBUG,"(%t)The value of data is %d \n",data->get()));
return 0;
}

static void * thread2(void*)
{
data->set(100);
ACE_DEBUG((LM_DEBUG,"(%t)The value of data is %d \n",data->get()));
      for(int i=0; i<5;i++)
      data->increment();
ACE_DEBUG((LM_DEBUG,"(%t)The value of data is %d \n",data->get()));
return 0;
}

int main(int argc, char*argv[])
{
```

```
//Spawn off the first thread

ACE_Thread_Manager::instance()->spawn((ACE_THR_FUNC)thread1,0,THR_NEW_LWP|

      THR_DETACHED);


//Spawn off the second thread

      ACE_Thread_Manager::instance()->spawn((ACE_THR_FUNC)thread2,0,THR_NEW_LWP|
      THR_DETACHED);


//Wait for all threads in the manager to complete.

ACE_Thread_Manager::instance()->wait();

ACE_DEBUG((LM_DEBUG,"Both threads done.Exiting.. \n"));

}
```

在上面的例子中，DataType类是在线程专有存储中创建的。随后程序使用->操作符来从函数thread1和thread2访问此类的方法，这两个函数分别在不同的线程中执行。第一个线程将私有数据变量设置为10，然后增加5，将它变为15。第二个线程使用它自己的私有数据变量，将它的值设为100，并增加5，变成105。尽管数据**看起来**是全局的，它实际上是线程专有的，而且两个线程分别打印出15和105，也说明了这一点。

尽可能使用线程专有存储有若干好处。如果全局或静态数据可放在线程专有存储中，就可将同步所导致的开销降到最低。这是使用TSS的主要好处。