基于 Linux 的 C++

第十四讲　线程编程

提　纲

线程基本概念

线程管理

- 线程创建，线程撤销，线程属性，线程局部存储，线程清除

线程同步机制

- 互斥、死锁、信号量、条件变量

C++11线程库

线程基本概念

线程的定义

- 线程是比进程更小的程序执行单位
- 多个线程可共享全局数据，也可使用专有数据

Linux线程支持史

- 1996年，LinuxThreads：基本符合POSIX标准，但效率低下，问题多多
- 2003年，内核2.6：提供线程支持库NPTL（Native POSIX Thread Library for Linux）

线程基本概念

内核线程

- 操作系统内核支持多线程调度与执行
- 内核线程使用资源较少，仅包括内核栈和上下文切换时需要的保存寄存器内容的空间

轻量级进程（lightweight process，LWP）

- 由内核支持的独立调度单元，调度开销小于普通的进程
- 系统支持多个轻量级进程同时运行，每个都与特定的内核线程相关联

线程基本概念

用户线程

- 建立在用户空间的多个用户级线程，映射到轻量级进程后调度执行
- 用户线程在用户空间创建、同步和销毁，开销较低
- 每个线程具有独特的ID

使用说明

- 线程功能不属于C/C++标准库，链接时需用-pthread选项
- 线程功能属于C++11标准库，可用C++11提供的thread类定义线程对象，C++11标准库同时提供基本的线程同步机制

线程基本概念

进程与线程的比较

- 线程空间不独立，有问题的线程会影响其他线程；进程空间独立，有问题的进程一般不会影响其他进程
- 创建进程需要额外的性能开销
- 线程用于开发细颗粒度并行性，进程用于开发粗颗粒度并行性
- 线程容易共享数据，进程共享数据必须使用进程间通讯机制

线程管理

- 线程创建
- 线程函数参数与返回值
- 线程ID
- 线程属性
- 线程撤销
- 线程局部存储
- 线程清除

线程创建

线程创建函数

- 头文件：“pthread.h”
- 原型：int pthread_create(pthread_t * thread, const pthread_attr_t * attr, void * (*start_routine)(void *), void * arg);

线程创建流程

- 定义指向pthread_t对象的指针对象，pthread_t对象用于存储新线程的ID
- 定义指向线程属性pthread_attr_t对象的指针对象；线程属性对象控制线程与程序其他部分（可能是其他线程）的交互；如果传递NULL，则使用缺省属性构造新线程

线程创建

线程创建流程

- 定义指向线程函数的指针对象，使其指向固定格式的线程函数
- 实现线程函数；线程函数的参数和返回值均为哑型指针；需要传递多个参数时，打包成单个void*型的指针对象
- 线程退出时使用返回值将数据传递给主调线程；多个结果同样可以打包传递

线程创建

线程创建说明

- pthread_create()函数在线程创建完毕后立即返回，它并不等待线程结束
- 原线程与新线程如何执行与调度有关，程序不得依赖线程先后执行的关系
- 可以使用同步机制确定线程的先后执行关系

线程退出方式

- 线程函数结束执行
- 调用pthread_exit()函数显式结束
- 被其他线程撤销

线程创建

```
#include <pthread.h>
#include <iostream>
void *  PrintAs( void * unused )
{
  while( true )    std::cerr << 'a';
  return NULL;
}
void *  PrintZs( void * unused )
{
  while( true )    std::cerr << 'z';
  return NULL;
}
int  main()
{
  pthread_t  thread_id;
  pthread_create( &thread_id, NULL, &PrintAs, NULL );
  PrintZs( NULL );
  return 0;
}
```

线程函数参数

```
#include <pthread.h>
#include <iostream>

class InfoPrinted
{
public:
  InfoPrinted( char c, int n ) : _c(c), _n(n)  {  }
  void  Show() const  {  for( int i = 0; i < _n; i++ )  std::cerr << _c;  }
private:
  char _c;
  int _n;
};

void *  PrintInfo( void * info )
{
  InfoPrinted *  p = reinterpret_cast<InfoPrinted *>( info );
  if( p )    p->Show();
  return NULL;
}
```

线程函数参数

```
// 注意：本程序大部分情况下不会输出任何结果
int  main()
{
  pthread_t  tid1, tid2;
  // 构造InfoPrinted类的动态对象，作为线程函数参数传递给线程tid1
  // 输出100个 ‘a’
  InfoPrinted *  p = new InfoPrinted( 'a', 100 );
  pthread_create( &tid1, NULL, &PrintInfo, reinterpret_cast<void *>( p ) );
  // 构造InfoPrinted类的动态对象，作为线程函数参数传递给线程tid2
  // 输出100个 ‘z’
  InfoPrinted *  q = new InfoPrinted( 'z', 100 );
  pthread_create( &tid2, NULL, &PrintInfo, reinterpret_cast<void *>( q ) );
  // 使用本注释行替换上述线程，可以看到输出结果，可能仅有部分输出
  // PrintInfo( reinterpret_cast<void *>( q ) );
  return 0;
}
```

线程函数参数

存在的问题：一般不会产生任何输出

- **子线程需要使用主线程的数据，如果主线程结束，子线程如何访问这些数据？**

解决方案：使用pthread_join()函数，等待子线程结束

- **原型：int pthread_join(pthread_t thread, void ** retval);**
- **参数：thread为pthread_t类型的线程ID；retval接收线程返回值，不需要接收返回值时传递NULL**

线程函数参数

```
// 注意：无法确定两个线程的执行顺序，多次输出结果可能不同
int  main()
{
  pthread_t  tid1, tid2;

  InfoPrinted *  p = new InfoPrinted( 'a', 100 );
  pthread_create( &tid1, NULL, &PrintInfo, reinterpret_cast<void *>( p ) );

  InfoPrinted *  q = new InfoPrinted( 'z', 100 );
  pthread_create( &tid2, NULL, &PrintInfo, reinterpret_cast<void *>( q ) );

  // 等待子线程结束
  pthread_join( tid1, NULL );
  pthread_join( tid2, NULL );

  return 0;
}
```

线程函数返回值

```
#include <pthread.h>
#include <cmath>
#include <iostream>

void *  IsPrime( void * n )
{
  unsigned int  p = reinterpret_cast<unsigned int>( n );
  unsigned int  i = 3u, t = (unsigned int)sqrt( p ) + 1u;
  if( p == 2u )
    return reinterpret_cast<void *>( true );
  if( p % 2u == 0u )
    return reinterpret_cast<void *>( false );
  while( i <= t )
  {
    if( p % i == 0u )
      return reinterpret_cast<void *>( false );
    i += 2u;
  }
  return reinterpret_cast<void *>( true );
}
```

线程函数返回值

```
// 使用g++ main.cpp –pthread –lm –fpermissive编译
// 以防止编译器将void*到int的转型当作错误
int  main()
{
  pthread_t  tids[8];
  bool  primalities[8];
  int i;
  for( i = 0; i < 8; i++ )
    pthread_create( &tids[i], NULL, &IsPrime, reinterpret_cast<void *>( i+2 ) );
  for( i = 0; i < 8; i++ )
    pthread_join( tids[i], reinterpret_cast<void **>( &primalities[i] ) );
  for( i = 0; i < 8; i++ )
    std::cout << primalities[i] << " ";
  std::cout << std::endl;
  return 0;
}
```

线程ID

pthread_equal()函数：确认两个线程是否相同

- **原型：int pthread_equal(pthread_t t1, pthread_t t2);**

pthread_self()函数：返回当前线程的ID

- **原型：pthread_t pthread_self();**
- **示例：if(!pthread_equal(pthread_self(), other_tid))　pthread_join(other_tid, NULL);**

线程属性

线程属性：精细调整线程的行为

设置线程属性的流程

- 创建pthread_attr_t类型的对象
- 调用pthread_attr_init()函数初始化线程的缺省属性，传递指向该线程属性对象的指针
  - 原型：int pthread_attr_init(pthread_attr_t * attr);
- 对线程属性进行必要修改
- 调用pthread_create()函数时传递指向线程属性对象的指针

线程属性

设置线程属性的流程

- 调用pthread_attr_destroy()函数清除线程属性对象，pthread_attr_t对象本身没有被销毁，因而可以调用pthread_attr_init()函数再次初始化
  - 原型：int pthread_attr_destroy(pthread_attr_t * attr);

线程属性说明

- 单一线程属性对象可以用于创建多个线程
- 线程创建后，继续保留线程属性对象本身并没有意义
- 对大多数Linux程序，线程最重要的属性为分离状态（detach state）

线程属性

线程分类

- 可联线程（joinable thread）：缺省设置，终止时并不自动清除（类似僵尸进程），主线程必须调用pthread_join()获取其返回值，此后才能清除
- 分离线程（detached thread）：结束时自动清除，不能调用pthread_join()进行线程同步
- 可联线程可通过pthread_detach()函数分离，分离线程不能再次联结
  - 原型：int pthread_detach(pthread_t thread);

线程属性

pthread_attr_setdetachstate()函数：设置线程分离属性

- 原型：int pthread_attr_setdetachstate (pthread_attr_t * attr, int detachstate);
- 传递线程属性对象指针和分离线程设置参数 PTHREAD_CREATE_DETACHED

pthread_attr_getdetachstate()函数：获取线程分离属性

- 原型：int pthread_attr_getdetachstate (pthread_attr_t * attr, int * detachstate);

线程属性

```
#include <pthread.h>
// 线程函数
void *  ThreadFunc( void * arg ) { ... }
int  main()
{
  pthread_attr_t  attr;
  pthread_t  thread;
  // 初始化线程属性
  pthread_attr_init( &attr );
  // 设置线程属性的分离状态
  pthread_attr_setdetachstate( &attr, PTHREAD_CREATE_DETACHED );
  // 创建线程
  pthread_create( &thread, &attr, &ThreadFunc, NULL );
  // 清除线程属性对象
  pthread_attr_destroy( &attr );
  // 无需联结该线程
  return 0;
}
```

线程撤销

pthread_cancel()函数：撤销线程

- 原型：int pthread_cancel(pthread_t thread);
- 已撤销的线程可以联结，且必须联结，以释放其资源，除非其为分离线程

线程撤销类型与状态

- 异步可撤销：在其执行的任何时刻都可撤销
- 同步可撤销：线程可撤销，但撤销操作首先进入队列排队，在线程执行到特定撤销点时才可撤销
- 不可撤消：撤销不可撤消线程的企图被系统忽略，且没有任何消息反馈

线程撤销

pthread_setcanceltype()函数：设置线程的撤销类型

- 原型： int pthread_setcanceltype(int type, int * oldtype);
- 参数：type为撤销类型，oldtype用于保存原始线程撤销类型，NULL表示不保存
- PTHREAD_CANCEL_ASYNCHRONOUS：线程异步可撤销
- PTHREAD_CANCEL_DEFERRED：线程同步可撤销，即延迟到下一撤销点时撤销

线程撤销

pthread_setcancelstate()函数：设置线程的撤销状态

- 原型： int pthread_setcancelstate(int state, int * oldstate);
- 第一个参数state为可撤销状态，第二个参数oldstate用于保存原始线程可撤销状态，NULL表示不保存
- PTHREAD_CANCEL_ENABLE：线程可撤销
- PTHREAD_CANCEL_DISABLE：线程不可撤销
- 线程的撤销状态可多次设置

线程撤销

pthread_testcancel()函数：设置撤销点

- 原型：void pthread_testcancel();
- 在线程函数中调用pthread_testcancel()函数设置撤销点
- 建议：周期性地设置撤销点，保证线程函数内部每隔一些代码就有一个撤销点，以保证资源能够正确释放

使用撤销状态构造临界区（critical section）

- 临界区：要么全部执行，要么一条都不执行的代码段
- 设置线程的撤销状态，线程一旦进入临界区，就必须等到离开临界区，才可以被撤销

线程撤销

```
// 账户转账
void Transfer( double * accounts, int from, int to, double amount )
{
  int ocs;
  // 数据有效性检查代码在此，确保转账操作合法有效

  // 将线程设置为不可撤销的，进入临界区
  pthread_setcancelstate( PTHREAD_CANCEL_DISABLE, &ocs );

  accounts[to] += amount;
  accounts[from] -= amount;

  // 恢复线程的撤销状态，离开临界区
  pthread_setcancelstate( ocs, NULL );
}
```

线程局部存储

线程局部存储（thread local storage，TLS）：每个线程的独有数据

- 线程特定数据（thread-specific data）
- 进程的多个线程通过全局堆共享全局数据对象
- 每个线程拥有独立的栈

让线程拥有数据的独立副本：不能简单赋值或读取

- pthread_key_create()函数：为线程特定数据创建一个键
- 参数：第一个为指向pthread_key_t类型变量的指针（每个线程都可以使用它访问自己的独立数据副本）；第二个参数为指向线程清除函数的指针，如果不存在，传递NULL
- pthread_setspecific()函数：设置对应键的值
- pthread_getspecific()函数：读取对应键的值

线程局部存储

```
#include <pthread.h>
#include <stdio.h>
static pthread_key_t  tlk;    //  关联线程日志文件指针的键
void  WriteToThreadLog( const char * msg )
{
  FILE *  fp = ( FILE * )pthread_getspecific( tlk );
  fprintf( fp, "%d: %s\n", (int)pthread_self(), msg );
}
void  CloseThreadLog( void * fp )
{
  fclose( ( FILE * )fp );
}
void *  ThreadFunc( void * args )
{
  char  filename[255];
  FILE *  fp;
  //  生成与线程ID配套的日志文件名
  sprintf( filename, "thread%d.log", (int)pthread_self() );
  fp = fopen( filename, "w" );
```

线程局部存储

```
  // 设置线程日志文件指针与键的局部存储关联
  pthread_setspecific( tlk, fp );
  // 向日志中写入数据，不同的线程会写入不同的文件
  WriteToThreadLog( "Thread starting..." );
  return NULL;
}
int  main()
{
  int  i;
  pthread_t  threads[8];
  // 创建键，使用CloseThreadLog()函数作为其清除程序
  pthread_key_create( &tlk, CloseThreadLog );
  for( i = 0; i < 8; ++i )
    pthread_create( &threads[i], NULL, ThreadFunc, NULL );
  for( i = 0; i < 8; ++i )
    pthread_join( threads[i], NULL );
  pthread_key_delete( tlk );
  return 0;
}
```

线程清除

线程清除函数：回调函数，单void*参数，无返回值

- 目的：销毁线程退出或被撤销时未释放的资源

pthread_cleanup_push()函数：注册线程清除函数

- 原型： void pthread_cleanup_push(void (*routine)(void*), void * arg);
- 参数：routine为指向线程清除函数的函数指针，arg为传递给回调函数的附加数据对象

pthread_cleanup_pop()函数：取消线程清除函数注册

- 原型： void pthread_cleanup_pop(int execute);
- 参数：整型值，非0调用回调函数释放资源，0不释放

线程清除

```
#include <malloc.h>
#include <pthread.h>
void *  AllocateBuffer( size_t size )
{
  return malloc( size );
}
void  DeallocateBuffer( void * buffer )
{
  free( buffer );
}
void  DoSomeWork()
{
  void *  temp_buffer = AllocateBuffer( 1024 );
  //  注册清除处理函数
  pthread_cleanup_push( DeallocateBuffer, temp_buffer );
  //  此处可以调用pthread_exit()退出线程或者撤销线程
  //  取消注册，传递非0值，实施清除任务
  pthread_cleanup_pop( 1 );
}
```

线程清除

C++的问题

- 对象的析构函数在线程退出时可能没有机会被调用，因而线程栈上的数据未清除
- 如何保证线程资源被正确释放？

解决方法

- 定义异常类，线程在准备退出时引发异常，然后在异常处理中退出线程执行
- 引发异常时，C++确保析构函数被调用

线程清除

```
#include <pthread.h>
class EThreadExit  {
public:
  EThreadExit( void * ret_val ) : _thread_ret_val(ret_val)  {  }
  //  实际退出线程，使用对象构造时的返回值
  void* DoThreadExit ()  {  pthread_exit( _thread_ret_val );  }
private:
  void *   _thread_ret_val;
};
void *  ThreadFunc( void * arg )
{
  try  {
    if( 线程需要立即退出 )
      throw EThreadExit(  线程返回值 );
  }
  catch( const EThreadExit & e )  {
    e.DoThreadExit();    //  执行线程实际退出动作
  }
  return NULL;
}
```

线程同步机制

资源竞争

互　斥

死　锁

信号量

条件变量

资源竞争

编程任务

- 存在一个任务队列，多个并发线程同时处理这些任务。每个线程在完成某项任务后，检查任务队列中是否有新任务。如果有，就处理该任务，并将该任务从任务队列中删除。
- 假设：两个线程碰巧完成各自任务，但队列中只有一个任务。
- 可能发生的情况：第一个线程发现任务队列非空，准备接收该任务，但没有完成全部设置。此时，操作系统碰巧中断该线程。第二个线程获得了执行，也发现任务队列非空，同样准备接收该任务，但发现已无法正确设置任务队列。
- 最坏情况：第一个线程已经从队列中摘取了任务，但是还没有将任务队列设置为空，第二个线程对任务队列的访问导致段错误，系统崩溃。

资源竞争

```
// 有问题的程序代码
#include <list>
struct Job;
std::list<Job *>  job_queue;
// 线程函数
void *  DequeueJob( void * arg )
{
  if( !job_queue.empty() )
  {
    Job *  job = job_queue.front();
    job_queue.pop_front();
    ProcessJob( job );
    delete job,  job = NULL;
  }
  return NULL;
}
```

互　斥

互斥（mutex）定义与性质：MUTial EXclusion

- 相互独占锁，与二元信号量类似
- 一次只有一个线程可以锁定一个数据对象，并访问
- 只有该线程释放锁定，其他线程才能访问该数据对象

pthread_mutex_init()函数：初始化互斥

- 原型：int pthread_mutex_init(pthread_mutex_t * mutex, const pthread_mutexattr_t * mutexattr);
- 参数： mutex为互斥对象，mutexattr为互斥属性对象，NULL表示使用缺省属性
- 可使用预定义宏PTHREAD_MUTEX_INITIALIZER初始化互斥

互　斥

pthread_mutex_destroy()函数：销毁互斥

- 原型：int pthread_mutex_destroy(pthread_mutex_t * mutex);

pthread_mutex_lock()函数：互斥加锁

- 原型：int pthread_mutex_lock(pthread_mutex_t * mutex);
- 如果无法锁定，则调用将阻塞，至该互斥被解除锁定状态

pthread_mutex_trylock()函数：互斥加锁

- 原型：int pthread_mutex_trylock(pthread_mutex_t * mutex);
- 如果无法锁定，则立即返回，不阻塞

pthread_mutex_unlock()函数：互斥解锁

- 原型：int pthread_mutex_unlock(pthread_mutex_t * mutex);

互　斥

使用互斥的流程

- 定义pthread_mutex_t类型的变量，将其地址作为第一个参数传给pthread_mutex_init()函数；初始化函数只需调用一次
- 锁定或尝试锁定该互斥；获得访问权后，执行正常程序代码；并在执行完毕后解锁

互斥属性

- pshared属性：进程共享属性
  - 取值：PTHREAD_PROCESS_SHARED（跨进程共享），PTHREAD_PROCESS_PRIVATE（本进程内部共享）
- type属性：互斥类型

互　斥

互斥type属性

- PTHREAD_MUTEX_NORMAL：普通锁
  - 被某个线程锁定后，其他请求加锁的线程将等待
  - 容易导致死锁
  - 解锁被其他线程锁定或已解锁的互斥，将导致不可预期的后果
- PTHREAD_MUTEX_ERRORCHECK：检错锁
  - 线程对已被其他线程锁定的互斥加锁，将返回EDEADLK
- PTHREAD_MUTEX_RECURSIVE：递归锁
  - 允许线程对互斥多次加锁；解锁次数必须与加锁次数匹配
- PTHREAD_MUTEX_DEFAULT：默认锁
  - 实现上可能为上述三种之一

互　斥

互斥属性函数

- 初始化互斥属性对象：int pthread_mutexattr_init(pthread_mutexatt_t * attr);
- 销毁互斥属性对象：int pthread_mutexattr_destroy(pthread_mutexatt_t * attr);
- 获取pshared属性：int pthread_mutexattr_getpshared(const pthread_mutex_t * mutex, int * pshared);
- 设置pshared属性：int pthread_mutexattr_setpshared(pthread_mutex_t * mutex, int pshared);
- 获取type属性：int pthread_mutexattr_gettype(const pthread_mutex_t * mutex, int * type);
- 设置type属性：int pthread_mutexattr_settype(pthread_mutex_t * mutex, int type);

互　斥

```
// 完整程序代码
#include <pthread.h>
#include <iostream>
#include <list>
struct Job  {
  Job( int x = 0, int y = 0) : x(x), y(y)  {  }
  int x, y;
};
// 一般要求临界区代码越短越好，执行时间越短越好，使用C++ STL可能并不是好选择
std::list<Job *>   job_queue;
pthread_mutex_t   job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;
// 此处作业处理工作仅为示例，简单输出线程ID和作业内容信息
void  ProcessJob( Job * job )
{
  std::cout << "Thread " << (int)pthread_self();
  std::cout << " processing (" << job->x << ", " << job->y << ")\n";
}
```

互　斥

```
// 处理作业时需要加锁
void *  DequeueJob( void * arg )
{
  while( true )  {
    Job *  job = NULL;
    pthread_mutex_lock( &job_queue_mutex );
    if( !job_queue.empty() )    {
      job = job_queue.front();       // 获取表头元素
      job_queue.pop_front();         // 删除表头元素
    }
    pthread_mutex_unlock( &job_queue_mutex );
    if( !job )    break;
    ProcessJob( job );
    delete job,  job = NULL;
  }
  return NULL;
}
```

互　斥

```
// 作业入队时需要加锁
void *  EnqueueJob( void * arg )
{
  Job * job = reinterpret_cast< Job * >( arg );
  pthread_mutex_lock( &job_queue_mutex );    // 锁定互斥
  job_queue.push_back( job );

  // 入队时也输出线程ID和作业内容信息
  std::cout << "Thread " << (int)pthread_self();
  std::cout << " enqueueing (" << job->x << ", " << job->y << ")\n";

  pthread_mutex_unlock( &job_queue_mutex );    // 解锁
  return NULL;
}
```

互　斥

```
int  main()
{
  int  i;
  pthread_t  threads[8];
  for( i = 0; i < 5; ++i )
  {
    Job *  job = new Job( i+1, (i+1)*2 );
    pthread_create( &threads[i], NULL, EnqueueJob, job );
  }
  for( i = 5; i < 8; ++i )
    pthread_create( &threads[i], NULL, DequeueJob, NULL );
  for( i = 0; i < 8; ++i )
    pthread_join( threads[i], NULL );
  return 0;
}
```

死　锁

死锁：资源被竞争占用，且无法释放

处理策略：更改互斥类型

- 创建互斥属性pthread_mutexattr_t型的对象
- 调用pthread_mutexattr_init()函数初始化互斥属性对象，传递其地址
- 调用pthread_mutexattr_setkind_np()函数设置互斥类型，函数第一个参数为指向互斥属性对象的指针，第二个参数为PTHREAD_MUTEX_RECURSIVE_NP（递归互斥）或PTHREAD_MUTEX_ERRORCHECK_NP（检错互斥）
- 调用pthread_mutexattr_destroy()函数销毁互斥属性对象

信号量

问题：如何确保任务队列中有任务可以做？

- 如果队列中没有任务，线程可能退出，后续任务出现时，没有线程可以执行它

POSIX标准信号量：头文件“semaphore.h”

- 用于多个线程的同步操作
- 操作方法比进程信号量简单

初始化信号量

- 原型：int sem_init(sem_t * sem, int pshared, unsigned int value);
- 参数：sem为信号量对象，pshared为共享属性，value为信号量初始值

信号量

等待信号量：P操作

- 原型：int sem_wait(sem_t * sem);
- 原型：int sem_trywait(sem_t * sem);
- 原型：int sem_timewait(sem_t * sem, const struct timespec * abs_timeout);
- 说明： sem_wait()在无法操作时阻塞， sem_trywait()则立即返回，sem_timewait()与sem_wait()类似，但有时间限制

发布信号量：V操作

- 原型：int sem_post(sem_t * sem);

销毁信号量

- 原型：int sem_destroy(sem_t * sem);

作业队列

```
// 完整程序代码
#include <pthread.h>
#include <iostream>
#include <list>
struct Job  {
  Job( int x = 0, int y = 0) : x(x), y(y)  {  }
  int x, y;
};
std::list<Job *>   job_queue;
pthread_mutex_t   job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;

// 控制作业数目的信号量
sem_t   job_queue_count;

void  ProcessJob( Job * job )
{
  std::cout << "Thread " << (int)pthread_self();
  std::cout << " processing (" << job->x << ", " << job->y << ")\n";
}
```

作业队列

```
// 处理作业时需要加锁
void *  DequeueJob( void * arg )
{
  while( true )  {
    Job *  job = NULL;
    sem_wait( &job_queue_count );   // 等待作业队列中有新作业
    pthread_mutex_lock( &job_queue_mutex );
    if( !job_queue.empty() )  {
      job = job_queue.front();          // 获取表头元素
      job_queue.pop_front();            // 删除表头元素
    }
    pthread_mutex_unlock( &job_queue_mutex );
    if( !job )    break;
    ProcessJob( job );
    delete job,  job = NULL;
  }
  return NULL;
}
```

作业队列

```
// 作业入队时需要加锁
void *  EnqueueJob( void * arg )
{
  Job *  job = reinterpret_cast< Job * >( arg );
  pthread_mutex_lock( &job_queue_mutex );    // 锁定互斥
  job_queue.push_back( job );

  // 新作业入队，递增信号量
  sem_post( &job_queue_count );

  // 入队时也输出线程ID和作业内容信息
  std::cout << "Thread " << (int)pthread_self();
  std::cout << " enqueueing (" << job->x << ", " << job->y << ")\n";

  pthread_mutex_unlock( &job_queue_mutex );    // 解锁
  return NULL;
}
```

作业队列

```
int  main()
{
  int  i;
  pthread_t  threads[8];
  if( !job_queue.empty() )    job_queue.clear();
  sem_init( &job_queue_count, 0, 0 );            //  初始化，非进程共享，初始值0
  for( i = 0; i < 5; ++i )
  {
    Job *  p = new Job( i+1, (i+1)*2 );
    pthread_create( &threads[i], NULL, EnqueueJob, p );
  }
  for( i = 5; i < 8; ++i )
    pthread_create( &threads[i], NULL, DequeueJob, NULL );
  for( i = 0; i < 8; ++i )
    pthread_join( threads[i], NULL );            //  等待线程终止，无作业时线程被阻塞
  sem_destroy( &job_queue_count );               //  销毁作业信号量
  return 0;
}
```

条件变量

条件变量的功能与目的

- 互斥用于同步线程对共享数据对象的访问
- 条件变量用于在线程间同步共享数据对象的值

初始化条件变量

- 原型：int pthread_cond_init(pthread_cond_t * cond, const pthread_condattr_t * cond_attr);
- 可使用宏PTHREAD_COND_INITIALIZER代替

销毁条件变量

- 原型：int pthread_cond_destroy(pthread_cond_t * cond);

条件变量

广播条件变量

- 原型：int pthread_cond_broadcast(pthread_cond_t * cond);
- 以广播方式唤醒所有等待目标条件变量的线程

唤醒条件变量

- 原型：int pthread_cond_signal(pthread_cond_t * cond);

等待条件变量

- 原型：int pthread_cond_wait(pthread_cond_t * cond, pthread_mutex_t * mutex);
- 参数：mutex为互斥，以确保函数操作的原子性

C++11线程库

支持平台无关的并行程序开发

库：atomic、thread、mutex、condition_variable、future

- **thread：std::thread类与std::this_thread名空间**
- **mutex：互斥相关类，包括std::mutex系列类，std::lock_guard类、std::unique_lock类及其他型式和函数**
- **condition_variable：条件变量类，包括std::condition_variable类与std::condition_variable_any类**
- **atomic：std::atomic类与std::atomic_flag类，另外还有一套C风格的原子型式和原子操作函数**

C++11线程库

库（续）

- future：包含两个承诺类（std::promise类、std::packaged_task类）、两个期许类（std::future类、std::shared_future类）及相关型式和函数

参考文献

- Anthony Williams. *C++ Concurrency in Action, Practical Multithreading.* Manning Publications, 2012. See Also: https://www.gitbook.com/book/chenxiaowei/cpp_concurrency_in_action/details.

线程类

线程类：thread

- 支持的线程函数无参数和返回值型式的特别要求，有无参数均可，返回值有无亦可

与Linux线程机制相比，C++11线程类更易用

线程局部存储使用thread_local关键字

可派生自己的thread类，但实现上需特别注意

- 线程类应支持移动语义，但不应支持拷贝语义

线程类

常用线程类成员函数

- 判断线程是否可联：bool thread::joinable();
- 等待线程结束：void thread::join();
- 分离线程：void thread::detach();

定义于名空间this_thread的线程管理函数

- 获取线程ID：thread::id get_id();
- 在处于等待状态时，让调度器选择其他线程执行：void yield();
- 阻塞当前线程指定时长：template<typename _Rep, typename _Period> void sleep_for(const chrono::duration<_Rep, _Period>& __rtime);
- 阻塞当前线程至指定时点：template<typename _Clock, typename _Duration> void sleep_until(const chrono::time_point<_Clock, _Duration>& __atime);

线程类

```
// 无参数线程函数
#include <iostream>
#include <thread>

void  ThreadFunc()
{
  std::cout << "Thread ID: " << std::this_thread::get_id() << std::endl;
}

int  main()
{
  std::thread  t( &ThreadFunc );          // 创建线程对象并运行
  t.join();                               // 等待线程结束
  return 0;
}
```

线程类

```
// 带双参数的线程函数
#include <iostream>
#include <thread>
void ThreadFunc( int a, int b )
{
  std::cout << "Thread ID: " << std::this_thread::get_id() << std::endl;
  std::cout << a << " + " << b << " = " << a + b << std::endl;
}
int main()
{
  int m = 10, n = 20;
  // C++11标准库使用可变参数的模板形式参数列表，线程函数参数个数任意
  std::thread t( &ThreadFunc, m, n );
  t.join();
  return 0;
}
```

线程类

```
// 带双参数的函子对象
#include <iostream>
#include <thread>
class Functor {
public:
  void operator()( int a, int b ) {
    std::cout << "Thread ID: " << std::this_thread::get_id() << std::endl;
    std::cout << a << " + " << b << " = " << a + b << std::endl;
  }
};
int main()
{
  int m = 10, n = 20;
  std::thread t( Functor(), m, n );
  t.join();
  return 0;
}
```

线程类

```
// 使用std::bind()函数绑定对象及其普通成员函数
#include <iostream>
#include <thread>
class Worker {
public:
  Worker( int a = 0, int b = 0 ) : _a(a), _b(b) { }
  void ThreadFunc() { ..... }
private: int _a, _b;
};

int main()
{
  Worker worker( 10, 20 );
  std::thread t( std::bind( &Worker::ThreadFunc, &worker ) );
  t.join();
  return 0;
}
```

互斥类

基本互斥：mutex类

- 核心成员函数lock()、try_lock()和unlock()
- 上述成员函数无参数，无返回值

递归互斥：recursive_mutex类

- 允许单个线程对互斥进行多次加锁与解锁处理

定时互斥：timed_mutex类

- 在某个时段里或者某个时刻前获取互斥
- 当线程在临界区操作的时间非常长，可以用定时锁指定时间

定时递归互斥：recursive_timed_mutex类

- 综合timed_mutex和recursive_mutex

共享定时互斥：shared_timed_mutex类（C++14）

互斥类

```
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>
std::mutex  x;
void  ThreadFunc()
{
  x.lock();
  std::cout << std::this_thread::get_id() << " is entering..." << std::endl;
  std::this_thread::sleep_for( std::chrono::seconds( 3 ) );
  std::cout << std::this_thread::get_id() << " is leaving..." << std::endl;
  x.unlock();
}
int  main()
{
  std::vector<std::thread *>  v( 8 );
  for( int i = 0; i < 8; i++ )    v[i] = new std::thread( ThreadFunc );
  for( int i = 0; i < 8; i++ )    v[i]->join();
}
```

互斥类

互斥的问题：容易导致死锁

- 若某个线程在临界区内的操作导致异常，有可能无法解锁，从而导致其他线程被永久阻塞
- 若临界区代码有多路分支，其中部分分支提前结束，但没有执行解锁操作，其他线程依然被永久阻塞
- 当多个线程同时申请多个资源时，加锁次序不同也可能导致死锁

资源获取即初始化（resource acquisition is initialization, RAII）

- 使用互斥对象管理类模板自动管理资源

互斥类

基于作用域的锁管理类模板：std::lock_guard

- 构造时是否加锁可选，不加锁时假定当前线程已获得锁的所有权，析构时自动解锁，所有权不可转移，对象生存期内不允许手动加锁和解锁

独一锁管理类模板：std::unique_lock

- 构造时是否加锁可选，对象析构时如果持有锁会自动解锁，所有权可转移，对象生存期内允许手动加锁和解锁

共享锁管理类模板：std::shared_lock（C++14）

- 用于管理可转移和共享所有权的互斥对象

互斥类

互斥管理策略

- 延迟：std::defer_lock，构造互斥管理对象时延迟加锁操作
- 尝试：std::try_to_lock，构造互斥管理对象时尝试加锁操作，但不阻塞线程，互斥不可用时立即返回
- 接收：std::adopt_lock，假定当前线程已获得互斥所有权，不再加锁
- 缺省行为：构造互斥管理对象时没有传递管理策略标签参数，阻塞当前线程至成功获得互斥

互斥类

互斥的解锁时机

- 当使用C++11的互斥自动管理策略时，只有析构互斥管理对象时才自动释放互斥，因此要特别注意互斥的持有时间；若线程持有互斥的时间过长，有可能极大降低程序效率
- 解决方案：使用复合语句块或专用辅助函数封装临界区操作；动态创建互斥管理对象，并尽早动态释放

多个互斥的竞争访问

- 多个线程对多个互斥加锁时保持顺序一致性，以避免可能的死锁
- 使用std::lock()或std::try_lock()

互斥类

```
// 使用互斥管理策略类重新实现线程函数
template< typename T > class Worker
{
public:
  explicit Worker( int no, T a = 0, T b = 0 ) : _no(no), _a(a), _b(b)  {  }
  void  ThreadFunc( T * r )
  {
    {   // 使用复合语句块封装临界区操作，块结束时即释放局部对象
      std::lock_guard<std::mutex>  locker( x );    //  构造对象的同时加锁
      *r = _x + _y;
    }   //  无需手工解锁，locker对象在析构时自动解锁
    std::cout << "Thread No: " << _no << std::endl;
    std::cout << _a << " + " << _b << " = " << _a + _b << std::endl;
  }
private:
  int  _no;
  T  _a, _b;
};
```

互斥类

```
// 转账处理示例
#include <iostream>
#include <mutex>
#include <thread>

class Account
{
public:
  explicit Account( double balance ) : _balance(balance)  {  }
  double  GetBalance()  {  return _balance; }
  void  Increase( double amount )  {  _balance += amount;  }
  void  Decrease( double amount )  {  _balance -= amount;  }
  std::mutex &  GetMutex()  {  return _x; }
private:
  double  _balance;
  std::mutex  _x;
};
```

互斥类

```
// 避免死锁，使用std::lock()函数锁定多个互斥，不同的锁定顺序不会导致死锁
// 加锁时有可能引发异常，std::lock()函数会处理该异常
// 将解锁此前已加锁的部分互斥，然后重新引发该异常
void Transfer( Account & from, Account & to, double amount )
{
  std::unique_lock<std::mutex>  locker1( from.GetMutex(), std::adopt_lock );
  std::unique_lock<std::mutex>  locker2( to.GetMutex(), std::adopt_lock );
  std::lock( from.GetMutex(), to.GetMutex() );
  from.Decrease( amount );    to.Increase( amount );
}
int main()
{
  Account  a1( 100.0 ), a2( 200.0 );
  // 线程参数采用值传递机制，如果要传递引用，调用std::ref()函数
  std::thread  t1( Transfer, std::ref( a1 ), std::ref( a2 ), 10.0 );
  std::thread  t2( Transfer, std::ref( a2 ), std::ref( a1 ), 20.0 );
  t1.join();    t2.join();
  return 0;
}
```

条件变量类

std::condition_variable类

- 必须与std::unique_lock配合使用

std::condition_variable_any类

- 更加通用的条件变量，可以与任意型式的互斥锁配合使用，相比前者使用时会有额外的开销

多线程通信同步原语

- 阻塞一个或多个线程至收到来自其他线程的通知，超时或发生虚假唤醒
- 两者具有同样的成员函数，且在等待条件变量前都必须要获得相应的锁

条件变量类

成员函数notify_one()：通知一个等待线程

- 原型：void motify_one() noexcept;

成员函数notify_all()：通知全部等待线程

- 原型：void motify_one() noexcept;

成员函数wait()：阻塞当前线程至被唤醒

- 原型：template<typename Lock> void wait(Lock & lock);
- 原型：template<typename Lock, typename Predicate> void wait(Lock & lock, Predicate p);

条件变量类

成员函数wait_for()：阻塞至被唤醒或超过指定时长

- 原型： template<typename Lock, typename Rep, typename _Period> cv_status wait_for(Lock& lock, const chrono::duration<Rep, Period>& rtime);
- 原型：template<typename Lock, typename Rep, typename Period, typename Predicate> bool wait_for(Lock& lock, const chrono::duration<Rep, Period>& rtime, Predicate p);

成员函数wait_until()：阻塞至被唤醒或到达指定时点

- 原型：template<typename Lock, typename Clock, typename Duration> cv_status wait_until(Lock & lock, const chrono::time_point<Clock, Duration>& atime);
- 原型：template<typename Lock, typename Clock, typename Duration, typename Predicate> bool wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& atime, Predicate p);

条件变量类

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
std::mutex  x;
std::condition_variable  cond;
bool  ready = false;
bool  IsReady()  {  return ready;  }
void  Run( int no )
{
  std::unique_lock<std::mutex>  locker( x );
  while( !ready )              //  若标志位非true，阻塞当前线程
    cond.wait( locker );    //  解锁并睡眠，被唤醒后重新加锁
  //  以上两行代码等价于cond.wait( locker, &IsReady );
  //  第二个参数为谓词，亦可使用函子实现
  std::cout << "thread " << no << '\n';
}
```

条件变量类

```
int  main()
{
  std::thread  threads[8];
  for( int i = 0; i < 8; ++i )
    threads[i] = std::thread( Run, i );
  std::cout << "8 threads ready...\n";
  {
    std::unique_lock<std::mutex>  locker( x );    //  互斥加锁
    ready = true;                 //  设置全局标志位为true
    cond.notify_all();            //  唤醒所有线程
  }    //  离开作用域，自动解锁；可将此复合语句块实现为函数
  //  基于区间的循环结构，对属于threads数组的所有元素t，执行循环体
  for( auto & t: threads )
    t.join();
  return 0;
}
```

原子型式

使用方法

- 使用atomic模板定义原子对象
- 使用预定义标准原子型式：atomic_bool、atomic_char、atomic_int、atomic_uint、atomic_long、atomic_wchar_t等等

意义：轻量级，支持单变量上的原子操作

原子型式

操　作	atomic_ flag	atomic <bool>	atomic <int_t>	atomic <T*>	atomic <other_t>
test_and_set	√				
clear	√				
is_lock_free		√	√	√	√
load		√	√	√	√
store		√	√	√	√
exchange		√	√	√	√
compare_exchange_weak, compare_exchange_strong		√	√	√	√
fetch_add, +=			√	√	
fetch_sub, -=			√	√	
fetch_or, |=			√		
fetch_and, &=			√		
fetch_xor, ^=			√		
++, --			√	√	

原子型式

```
#include <atomic>
#include <iostream>
#include <thread>
int  n = 0;
std::atomic<int>  a( 0 );
void  AddAtomically( int m ) { while( m-- ) a.fetch_add( 1 ); }
void  Add( int m ) { while( m-- ) ++n; }
int  main()
{
  std::thread  ts1[8], ts2[8];
  for( auto & t: ts1 ) t = std::move( std::thread( AddAtomically, 1000000 ) );
  for( auto & t: ts2 ) t = std::move( std::thread( Add, 1000000 ) );
  for( auto & t: ts1 ) t.join();
  for( auto & t: ts2 ) t.join();
  // 输出结果：a值固定，而n值多次运行结果可能不同
  std::cout << "a = " << a << std::endl;
  std::cout << "n = " << n << std::endl;
  return 0;
}
```

期许与承诺

线程返回值

- 为支持跨平台，thread类无属性字段保存线程函数的返回值

解决方案

- 使用指针型式的函数参数
- 使用期许：std::future类模板
- 使用承诺：std::promise类模板

指针型式参数

```
// 使用指针作为函数参数，获取线程计算结果
#include <iostream>
#include <vector>
#include <tuple>
#include <thread>
#include <mutex>
std::mutex x;
// 劳工线程类模板，处理T型数据对象
template< typename T >  class Worker
{
public:
  explicit Worker( int no, T a = 0, T b = 0 ) : _no(no), _a(a), _b(b)  {  }
  void  ThreadFunc( T * r )  {  x.lock();    *r = _a + _b;    x.unlock();  }
private:
  int  _no;              // 线程编号，非线程ID
  T  _a, _b;             // 保存在线程中的待处理数据
};
```

指针型式参数

```
int main()
{
  //  定义能够存储8个三元组的向量v，元组首元素为指向劳工对象的指针
  //  次元素保存该劳工对象计算后的结果数据，尾元素为指向劳工线程对象的指针
  //  向量中的每个元素都表示一个描述线程运行的线程对象，
  //  该线程对象对应的执行具体任务的劳工对象，及该劳工对象运算后的返回值
  std::vector< std::tuple<Worker<int>*, int, std::thread*> >  v( 8 );

  //  构造三元组向量，三元编号顺次为0、1、2
  for( int i = 0; i < 8; i++ )
    v[i] = std::make_tuple( new Worker<int>( i, i+1, i+2 ), 0, nullptr );

  //  输出处理前结果；使用std::get<n>(v[i])获取向量的第i个元组的第n个元素
  //  三元编号顺次为0、1、2，因而1号元保存的将是劳工对象运算后的结果
  for( int i = 0; i < 8; i++ )
    std::cout << "No. " << i << ": result = " << std::get<1>(v[i]) << std::endl;
```

指针型式参数

```
// 创建8个线程分别计算
for( int i = 0; i < 8; i++ )
{
 // 将劳工类成员函数绑定为线程函数，对应劳工对象绑定为执行对象
 // 将构造线程对象时传递的附加参数作为被绑定的线程函数的第一个参数
 // auto表示由编译器自动推断f的型式
 auto  f = std::bind( &Worker<int>::ThreadFunc,
   std::get<0>( v[i] ), std::placeholders::_1 );

 // 动态构造线程对象，并保存到向量的第i个三元组中
 // 传递三元组的1号元地址，即将该地址作为线程函数的参数
 // 线程将在执行时将结果写入该地址
 // 此性质由绑定函数std::bind()使用占位符std::placeholders::_1指定
 // 线程对象为2号元，即三元组的最后一个元素
 std::get<2>( v[i] ) = new std::thread( f, &std::get<1>( v[i] ) );
}
```

指针型式参数

```
    for( int i = 0; i < 8; i++ )
    {
      // 等待线程结束
      std::get<2>( v[i] )->join();
      // 销毁劳工对象
      delete std::get<0>( v[i] ),    std::get<0>( v[i] ) = nullptr;
      // 销毁线程对象
      delete std::get<2>( v[i] ),    std::get<2>( v[i] ) = nullptr;
    }
    // 输出线程计算后的结果
    for( int i = 0; i < 8; i++ )
      std::cout << "No. " << i << ": result = " << std::get<1>(v[i]) << std::endl;
    return 0;
}
```

期　许

std::future类模板

- 目的：获取异步操作结果，延迟引发线程的异步操作异常

使用方法

- 定义期许模板类的期许对象
- 使用std::async()函数的返回值初始化
- 调用期许对象的成员函数get()获取线程返回值

期　许

```
// 使用期许对象获取线程返回值
#include <iostream>
#include <exception>
#include <thread>
#include <future>

unsigned long int  CalculateFactorial( short int n )
{
  unsigned long int  r = 1;
  if( n > 20 )
    throw std::range_error( "The number is too big." );
  for( short int i = 2; i <= n; i++ )
    r *= i;
  return r;
}
```

期　许

```
int main()
{
  short int  n = 20;
  // 启动异步线程，执行后台计算任务，并返回std::future对象
  std::future<unsigned long int>  f = std::async( CalculateFactorial, n );
  try  {
    // 获取线程返回值，若线程已结束，立即返回，否则等待该线程计算完毕
    // 若线程引发异常，则延迟到std::future::get()或std::future::wait()调用时引发
    unsigned long int  r = f.get();
    std::cout << n << "! = " << r << std::endl;
  }
  catch( const std::range_error & e )  {
    std::cerr << e.what() << std::endl;
  }
  return 0;
}
```

承　诺

std::promise类模板

- 目的： 承诺对象允许期许对象获取线程对象创建的线程返回值

使用方法

- 创建承诺std::promise<T>对象
- 获取该承诺对象的相关期许std::future<T>对象
- 创建线程对象，并传递承诺对象
- 线程函数内部通过承诺模板类的成员函数set_value()、set_value_at_thread_exit()、set_exception()或set_exception_at_thread_exit()设置值或异常
- 通过期许对象等待并获取异步操作结果

承　诺

```
// 使用承诺对象设置线程返回值
#include <iostream>
#include <exception>
#include <thread>
#include <future>

unsigned long int CalculateFactorial( short int n )
{
  unsigned long int r = 1;
  if( n > 20 )    throw std::range_error( "The number is too big." );
  for( short int i = 2; i <= n; i++ )    r *= i;
  return r;
}

// CalculateFactorial()函数的包装函数原型
void DoCalculateFactorial(
  std::promise<unsigned long int> && promise, short int n );
```

承　诺

```
int main()
{
  short int  n = 6;
  std::promise<unsigned long int>  p;    //  创建承诺对象
  std::future<unsigned long int>  f = p.get_future();    //  获取相关期许对象
  //  启动线程，执行CalculateFactorial()函数的包装函数
  std::thread  t( DoCalculateFactorial, std::move( p ), n );
  t.detach();
  try
  {
    unsigned long int  r = f.get();
    std::cout << n << "! = " << r << std::endl;
  }
  catch( std::range_error & e )
    std::cerr << e.what() << std::endl;
  return 0;
}
```

承　诺

```
// CalculateFactorial()函数的包装函数实现
void DoCalculateFactorial(
  std::promise<unsigned long int> && promise, short int n )
{
  try
  {
    // 设置线程返回值，供期许对象获取
    promise.set_value( CalculateFactorial( n ) );
  }
  catch( ... )
  {
    // 捕获全部异常，并在期许获取线程返回值时重新引发
    promise.set_exception( std::current_exception() );
  }
}
```

编程实践

14.1 考虑作业队列的容量限制，修改代码，实现标准的生产者—消费者模型。设作业队列最多容量amount个作业，有m个接收作业的线程，有n个处理作业的线程。

14.2 实现Linux互斥、信号量和条件变量的封装类，并使用上述同步机制实现线程池类。提示：（1）线程池功能与实现类似于进程池。（2）以作业型作为模板形式参数实现作业处理线程池类和作业处理类。（3）线程函数为静态函数，要访问类的非静态成员，可以定义类的静态对象或者传递对象指针，在线程函数中通过该对象指针访问其成员。（4）可以参考C++11架构。