# 目录

[目录 1](#_Toc21683089)

[第9章 线程 2](#_Toc21683090)

[9.1 Linux多线程概述 2](#_Toc21683091)

[9.1.1 多线程概述 2](#_Toc21683092)

[9.1.2 线程分类 2](#_Toc21683093)

[9.1.3 线程创建的Linux实现 2](#_Toc21683094)

[9.2 线程的相关函数 3](#_Toc21683095)

[9.2.1 pthread\_create() 创建线程 3](#_Toc21683096)

[9.2.2 pthread\_exit() 线程退出 3](#_Toc21683097)

[9.2.3 pthread\_join() 等待线程 3](#_Toc21683098)

[9.2.4 pthread\_cancel() 取消线程 3](#_Toc21683099)

[9.3 线程的清理 3](#_Toc21683100)

[9.3.1 清理函数压栈函数：pthread\_cleanup\_push 3](#_Toc21683101)

[9.3.2 清理函数出栈函数：pthread\_cleanup\_pop 3](#_Toc21683102)

[9.3.3 线程清理函数压栈、出栈函数原型定义说明 3](#_Toc21683103)

[9.4 线程的互斥 3](#_Toc21683104)

[9.4.1 互斥锁相关函数 3](#_Toc21683105)

[9.4.2 加锁注意事项（信号中不应该申请锁） 3](#_Toc21683106)

# 线程

## Linux多线程概述

### 多线程概述

进程是系统中程序执行和资源分配的基本单位。每个进程有自己的数据段、代码段和堆栈段。这就造成进程在进行切换等操作时都需要有比较复杂的上下文切换等动作。为了进一步减少处理器的空转时间支持多处理器和减少上下文切换开销，也就出现了线程。

线程通常叫做 “轻量级进程”。线程是在共享内存空间中并发执行的多道执行路径，是一个更加接近于执行体的概念，拥有独立的执行序列，是进程的基本调度单元，每个进程至少都有一个main线程(主线程)。它和同进程中的其他线程共享进程空间（堆代码、数据、文件描述符、信号等），只拥有自己的栈空间，大大减少了上下文切换的开销。进程和线程的关系如下图9.1所示。

线程和进程在使用上各有优缺点：线程执行开销小，占用的CPU资源少，线程之间的切换快，但不利于资源的管理和保护；而进程正相反。从可移植性来讲，多进程的可移植性要好些。

一个进程可以有多个线程，也就是有多个线程控制表及堆栈寄存器，但却共享一个用户地址空间。要注意的是，由于线程共享了进程的资源和地址空间，因此，任何线程对系统资源的操作都会给其他线程带来影响。

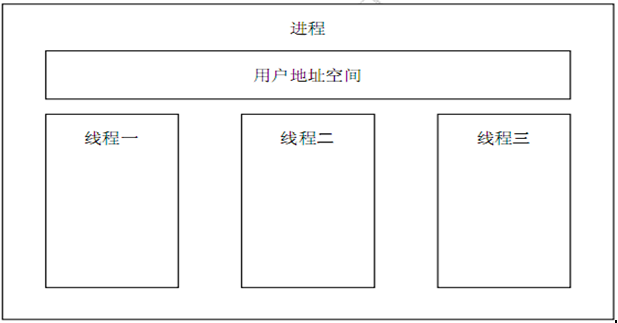


图9.1

### 线程分类

按调度者分为用户级线程和核心级线程。

用户级线程：主要解决上下文切换问题，调度算法和调度过程全部由用户决定，在运行时不需要特定的内核支持，缺点是无法发挥多处理器的优势。

核心级线程：允许不同进程中的线程按照同一相对优先调度方法调度，发挥多处理器的并发优势。

现在大多数系统都采用用户级线程和核心级线程并存的方法，一个用户级线程可以对应一个或多个核心级线程，也就是“一对一”或“一对多”模型。

### 线程创建的Linux实现

Linux的线程是通过**用户级**的函数库实现的，一般采用pthread线程库实现线程的访问和控制。它用第3方posix标准的pthread，具有良好的可移植性，编译的时候要在后面加上 -lpthread选项。

形如：（在Ubuntu）gcc -pthread test.c -o test

Compile and link with -pthread.

表6-1 线程和进程对比

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 创建 | 退出 | 等待 |
| 多进程 | fork() | exit() | wait() |
| 多线程 | pthread\_create() | pthread\_exit() | pthread\_join() |

## 线程的相关函数

头文件：#include <pthread.h>

### pthread\_create() 创建线程

|  |
| --- |
| 原型：int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr,  void \*(\*start\_routine) (void \*), void \*arg);  参数： pthread\_t \*thread 存放新线程ID （tid）缓存空间的地址  const pthread\_attr\_t \*attr 一般填NULL，新线程以默认方式创建  void \*(\*start\_routine) (void \*) 函数指针，指向新线程的任务函数  void \*arg 传递给新线程的任务函数的参数  返回值：成功，返回0  功能：创建一个新线程并指定新线程的任务函数，获取新线程的tid  备注：线程在以下情况发生时退出运行：  线程调用了pthread\_exit();  线程所对应的任务函数运行完毕  线程被取消pthread\_cancel()  主线程（main函数）运行结束 / 同一进程下的任一线程使用exit()，会导致该进程下的所有线程退出运行 |

创建新线程前，需要决定新线程的任务函数（即新线程启动后需要执行的操作）

示例：

|  |
| --- |
| dingwei@dingwei-virtual-machine:~/桌面/thread$ cat tid\_create\_test.c  #include <pthread.h>  #include <stdio.h>  #include <unistd.h>  #include <stdlib.h>  void\* task1(void \*arg);  int main(void)  {  pthread\_t tid;  pthread\_create(&tid,NULL,task1,(void \*)5);  printf("I am main thread\r\n");  sleep(1);  printf("main:the new tid = %ld\r\n",tid);  }  void\* task1(void \*arg)  {  printf("I am new thread\r\n");  printf("arg = %d\r\n",(int)arg);  }  dingwei@dingwei-virtual-machine:~/桌面/thread$ ./tid\_create\_test  I am main thread  I am new thread  arg = 5  main:the new tid = 139943046960896  主线程在创建子线程后，将继续运行主线程。  多线程运行，线程和进程类似（运行态、阻塞态、就绪态），需要考虑状态的切换 |

### pthread\_exit() 线程退出

线程退出方式有以下4种

1. 线程函数运行完之后，线程就退出。
2. 使用pthread\_exit函数退出，这是线程主动退出行为。
3. 使用pthread\_cancel函数取消线程。
4. 任何一个线程调用exit函数或者主线程调用运行结束，这将导致进程中的所有线程退出。

|  |
| --- |
| 原型：void pthread\_exit(void \*retval);  参数：void \*retval 线程结束时返回一个数据（可被其他线程使用pthread\_join()所捕获）  返回值：---  功能：结束当前线程 |

### pthread\_join() 等待线程

|  |
| --- |
| 原型：int pthread\_join(pthread\_t thread, void \*\*retval);  参数：pthread\_t thread 线程的tid  void \*\*retval 指向目标线程退出时pthread\_exit()的参数  返回值：成功，返回0  功能：等待目标线程结束运行 |

示例：主线程等待子线程结束运行

|  |
| --- |
| dingwei@dingwei-virtual-machine:~/桌面/thread$ cat thread\_join\_test.c  #include <pthread.h>  #include <stdio.h>  #include <unistd.h>  #include <stdlib.h>  void\* task1(void \*arg);  int main(void)  {  int ret;  pthread\_t tid;  pthread\_create(&tid,NULL,task1,(void \*)5);  printf("I am main thread\r\n");  printf("main:the new tid = %ld\r\n",tid);  printf("main:wait for new thread\r\n");  pthread\_join(tid,(void \*\*)&ret);  printf("main:new thread ends with ret = %d\r\n",ret);  }  void\* task1(void \*arg)  {  printf("I am new thread\r\n");  printf("arg = %d\r\n",(int)arg);  pthread\_exit((void \*)3);  printf("new thread:123\r\n");  }  dingwei@dingwei-virtual-machine:~/桌面/thread$ ./thread\_join\_test  I am main thread  main:the new tid = 139773353600768  main:wait for new thread  I am new thread  arg = 5  main:new thread ends with ret = 3 |

练习：编程实现，用3个子线程完成1~30报数

要求：显示的数字不重复

|  |
| --- |
| dingwei@dingwei-virtual-machine:~/桌面/thread$ cat multithread\_count.c  #include <pthread.h>  #include <stdio.h>  #include <unistd.h>  #include <stdlib.h>  void\* task\_count(void \*arg);  pthread\_t tid[3];  int cnt = 0;  int main(void)  {  int ret;  pthread\_create(&tid[0],NULL,task\_count,(void \*)1);  pthread\_create(&tid[1],NULL,task\_count,(void \*)2);  pthread\_create(&tid[2],NULL,task\_count,(void \*)3);  pthread\_join(tid[0],NULL);  pthread\_join(tid[1],NULL);  pthread\_join(tid[2],NULL);  printf("Main:ALL new threads end\r\n");  pthread\_exit(NULL);  }  void\* task\_count(void \*arg)  {  printf("I am new thread NO.%d\r\n",(int)arg);  while(cnt < 30)  {  cnt++;  printf("NO.%d : cnt = %d\r\n",(int) arg,cnt);  usleep(20000);  }  pthread\_exit(NULL);  }  运行程序时，发现报数时经常出现问题：  dingwei@dingwei-virtual-machine:~/桌面/thread$ ./multithread\_count  I am new thread NO.1  NO.1 : cnt = 1  I am new thread NO.2  NO.2 : cnt = 2  I am new thread NO.3  NO.3 : cnt = 3  NO.1 : cnt = 4  NO.3 : cnt = 5  NO.2 : cnt = 6  NO.1 : cnt = 7  NO.3 : cnt = 8  NO.2 : cnt = 9  NO.3 : cnt = 10  NO.1 : cnt = 11  NO.2 : cnt = 12  NO.1 : cnt = 13  NO.2 : cnt = 14  NO.3 : cnt = 15  NO.3 : cnt = 16  NO.2 : cnt = 17  NO.1 : cnt = 18  NO.2 : cnt = 19  NO.3 : cnt = 20  NO.1 : cnt = 21  NO.3 : cnt = 23  NO.2 : cnt = 22 （顺序出错）  NO.1 : cnt = 24  NO.2 : cnt = 25  NO.3 : cnt = 26  NO.1 : cnt = 27  NO.1 : cnt = 28  NO.3 : cnt = 29  NO.2 : cnt = 30  Main:ALL new threads end |

出现上述情况的原因是：

1. 线程间切换速度很快
2. 这些线程可能同时操作公共资源，但是没有对该操作进行保护，所以会导致一些问题，报数顺序出错或多个线程报同一个数。

所以当存在多线程操作公共资源时，一定要考虑线程的互斥，在上述问题中，cnt变量就是一种公共资源，而3个子线程之间存在竞争关系，需要控制同一时间只有一个子线程操作该cnt变量。

### pthread\_cancel() 取消线程

## 线程的清理

Posix的线程终止有两种情况：正常终止和非正常终止。

正常退出：线程主动调用pthread\_exit()或者从线程函数中return都将使线程正常退出。

非正常终止：线程在其他线程的干预下，或者由于自身运行出错（比如访问非法地址）而退出。

无论是哪种方式导致线程退出，都会存在资源释放的问题，在不考虑因运行出错而退出的前提下，如何保证线程终止时能顺利的释放掉自己所占用的资源，特别是锁资源，就是一个必须考虑解决的问题。

最常见的情形是资源独占锁的使用：线程为了访问临界共享资源而为其加上锁，但在访问过程中该线程被外界取消，或者发生了中断，则该临界资源将永远处于锁定状态得不到释放。

在POSIX线程API中提供了一对pthread\_cleanup\_push() / pthread\_cleanup\_pop()函数，用于自动释放资源：从pthread\_cleanup\_push()的调用点到pthread\_cleanup\_pop()之间的程序段中的终止动作都将执行 pthread\_cleanup\_push() 所指定的清理函数。man pthread\_cleanup\_pop

### 清理函数压栈函数：pthread\_cleanup\_push

**原型：**void pthread\_cleanup\_push(void (\*routine)(void \*),void \*arg);

**功能：**能在线程退出时指定清理函数，线程可以建立多个清理处理程序。处理程序记录在栈中，也就是说它们的执行顺序与它们注册时的顺序相反。它必须和pthread\_cleanup\_pop成对使用。

**参数：**

routine 线程清理函数指针

arg 当routine指向的线程清理函数调用时候传递的实际参数

**说明：**routine指向的线程清理函数执行时刻

　　1）线程调用pthread\_exit()函数，而不是直接return。

　　2）作为对取消线程请求(pthread\_cancel)的响应。

　　3）以非0参数调用pthread\_cleanup\_pop。

### 清理函数出栈函数：pthread\_cleanup\_pop

**原型：**void pthread\_cleanup\_pop(int execute);

**功能：**弹出线程栈当前最顶部的线程清理函数，并且根据参数 execute 决定是否执行它

**参数：**execute 是否执行弹出的线程清理函数；这个参数并不影响异常终止时清理函数的执行。

0：表示不执行弹出的清理函数；

非0：表示执行弹出的清理函数；

**说明：**pthread\_cleanup\_pop调用结果

1）如果线程只是由于简单的返回而终止的，则清除函数不会被调用。

2）如果pthread\_cleanup\_pop被传递0参数，则清除函数不会被调用，但是会清除处于栈顶的清理函数。

### 线程清理函数压栈、出栈函数原型定义说明

pthread\_cleanup\_push()/pthread\_cleanup\_pop()是以宏方式实现的，这是pthread.h中的宏定义：

|  |
| --- |
| #define pthread\_cleanup\_push(routine, arg) \  do { \  \_\_pthread\_cleanup\_class \_\_clframe (routine, arg)  #define pthread\_cleanup\_pop(execute) \  \_\_clframe.\_\_setdoit (execute); \  } while (0) |

可见，pthread\_cleanup\_push()带有一个"{"，而pthread\_cleanup\_pop()带有一个"}"，因此这两个函数必须成对出现，且必须位于程序的同一级别的代码段中才能通过编译。

execut非0值：按栈的顺序注销掉一个原来注册的清理函数的时候，会执行该函数；

0值：仅仅在线程调用pthread\_exit 函数或者其它线程对本线程调用pthread\_cancel函数时，才在弹出“清理函数”的同时执行该“清理函数”。

## 线程的互斥

在Posix Thread中定义了一套专门用于线程互斥的mutex函数。

mutex是一种简单的加锁的方法来控制对共享资源的存取，这个互斥锁只有两种状态（上锁和解锁），可以把互斥锁看作某种意义上的全局变量。（学习的核心：上锁和解锁操作。上锁：占用这个公共资源；解锁：释放占用的公共资源。）

因为多个线程共用进程的资源，要访问的是公共区间时（全局变量），当一个线程访问的时候，需要加上锁以防止另外的线程对它进行访问，以实现资源的独占。

在一个时刻只能有一个线程掌握某个互斥锁，拥有上锁状态的线程才能够对共享资源进行操作。

线程上锁一个已经上锁了的互斥锁，则该线程就会挂起，直到上锁的线程释放掉互斥锁为止。

互斥锁有三种类型：快速锁，嵌套锁（递归锁），检错锁。

互斥锁有两种创建（初始化）方法：静态方式（使用宏）和动态方式（用函数创建）。

互斥锁有五个函数：创建（初始化）锁，加锁，解锁 ，测试加锁，销毁锁。

### 互斥锁相关函数

使用互斥锁步骤：

一个互斥锁 对应 一种公共资源

主线程：

1. 创建并初始化互斥锁
2. 创建多个子线程
3. 等待子线程运行结束
4. 销毁互斥锁
5. 主线程结束运行

子线程：

1. 在操作公共资源之前上锁
2. 操作公共资源
3. 在操作公共资源之后解锁
4. 子线程结束运行

#### pthread\_mutex\_init() 动态创建线程互斥锁

原型：int pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*mutexattr)

功能：运行时动态创建一个线程互斥锁

参数：

mutex 线程互斥锁指针，

mutexattr 要给新创建的线程互斥锁设置的属性结构指针,如果为NULL则使用缺省属性，通常也设置为NULL。。

返回：失败：错误码（非0）； 成功：0

包含：#include <pthread.h>

Eg:

pthread\_mutex\_t lock;

int ret ;

ret = pthread\_mutex\_init(&lock,NULL);

if(ret != 0)

{  
 perror(“create mutex fail”);

exit(-1);

}

#### pthread\_mutex\_lock() 线程互斥锁加锁

原型：int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

功能：加锁，不论哪种类型的锁，都不可能被两个不同的线程同时得到，而必须等待解锁。在同一进程中的线程，如果加锁后没有解锁，则任何其他线程都无法再获得锁。

参数：mutex 指向要申请的互斥锁结构变量的指针

返回：返回：失败：错误码（非0）；成功：0

#### pthread\_mutex\_unlock() 线程互斥锁解锁

原型：int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

功能：解锁，根据不同的锁类型，可实现不同的行为：

快速锁：pthread\_mutex\_unlock解除锁定；

递规锁：pthread\_mutex\_unlock使锁上的引用计数减1；

检错锁：如果锁是本线程锁定的，则解除锁定，否则什么也不做。

返回：返回：失败：错误码（非0）；成功：0

#### pthread\_mutex\_destroy() 销毁线程互斥锁

原型：int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

功能：销毁线程互斥锁，释放它所占用的资源，且要求锁当前处于开放状态。

参数：mutex 指向要销毁的线程互斥锁结构变量的指针

返回：返回：失败：错误码（非0）；成功：0

#### pthread\_mutexattr\_settype() 修改线程互斥锁类型

原型：int pthread\_mutexattr\_settype(pthread\_mutexattr\_t \*attr, int type);

功能：修改线程互斥锁类型，使用修改后的属性来初始化线程互斥锁。

参数：

attr 指向线程互斥锁属性结构变量的指针；

type 可取值有以下：（缺省的互斥锁类型属性是：PTHREAD\_MUTEX\_DEFAULT）

PTHREAD\_MUTEX\_NORMAL：

>: 这种类型的互斥锁不会自动检测死锁。

>: 如果一个线程试图对一个互斥锁重复锁定，将会引起这个线程的死锁。

>: 如果试图解锁一个由别的线程锁定的互斥锁会引发不可预料的结果。

>: 如果一个线程试图解锁已经被解锁的互斥锁也会引发不可预料的结果。

PTHREAD\_MUTEX\_ERRORCHECK：

检错锁，如果同一个线程请求同一个锁，则返回EDEADLK，否则与PTHREAD\_MUTEX\_DEFAULT类型动作相同。这样就保证当不允许多次加锁时不会出现最简单情况下的死锁。

示例：初始化一个检错锁。

>: 这种类型的互斥锁会自动检测死锁。

>: 如果一个线程试图对一个互斥锁重复锁定，将会返回一个错误代码。

>: 如果试图解锁一个由别的线程锁定的互斥锁将会返回一个错误代码。

>: 如果一个线程试图解锁已经被解锁的互斥锁也将会返回一个错误代码。

pthread\_mutex\_t lock;

pthread\_mutexattr\_t mutexattr;

pthread\_mutexattr\_settype(&mutexattr, PTHREAD\_MUTEX\_ERRORCHECK);

pthread\_mutex\_init(&lock, &mutexattr);

PTHREAD\_MUTEX\_RECURSIVE：

嵌套锁（递归锁），允许同一个线程对同一个锁成功获得多次，并通过多次unlock解锁。如果是不同线程请求，则在加锁线程解锁时重新竞争。

>: 如果一个线程对这种类型的互斥锁重复上锁，不会引起死锁。

>: 一个线程对这类互斥锁的多次重复上锁必须由这个线程来重复相同数量的解锁，这样才能解开这个互斥锁，别的线程才能得到这个互斥锁。

>: 如果试图解锁一个由别的线程锁定的互斥锁将会返回一个错误代码。

>: 如果一个线程试图解锁已经被解锁的互斥锁也将会返回一个错误代码。

>: 这种类型的互斥锁只能是进程私有的（作用域属性为PTHREAD\_PROCESS\_PRIVATE）

示例：初始化一个嵌套锁。

pthread\_mutex\_t lock;

pthread\_mutexattr\_t mutexattr;

pthread\_mutexattr\_settype(&mutexattr, PTHREAD\_MUTEX\_RECURSIVE);

pthread\_mutex\_init(&lock, &mutexattr);

PTHREAD\_MUTEX\_DEFAULT：

这是缺省值（直接写NULL就是表示这个缺省值），也就是普通锁(或快速锁)。当一个线程加锁以后，其余请求锁的线程将形成一个阻塞等待队列，并在解锁后按优先级获得锁。这种锁策略保证了资源分配的公平性。

>: 这种类型的互斥锁不会自动检测死锁。

>: 如果一个线程试图对一个互斥锁重复锁定，将会引起不可预料的结果。

>: 如果试图解锁一个由别的线程锁定的互斥锁会引发不可预料的结果。

>: 如果一个线程试图解锁已经被解锁的互斥锁也会引发不可预料的结果。

>: POSIX标准规定，对于某一具体的实现，可以把这种类型的互斥锁定义为其他类型的互斥锁。

示例：初始化一个快速锁。

pthread\_mutex\_t lock;

pthread\_mutex\_init(&lock, NULL);

返回：失败：错误码（非0）；成功：0

#### PTHREAD\_MUTEX\_INITIALIZER 静态创建线程互斥锁

POSIX定义了一个宏PTHREAD\_MUTEX\_INITIALIZER来静态初始化互斥锁，方法如下：

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

在Linux Threads实现中，pthread\_mutex\_t是一个结构，PTHREAD\_MUTEX\_INITIALIZER是一个宏常量。

与上面的锁类型相对应，有3种类型的静态锁：

 快速静态锁pthread\_mutex\_t fastmutex = PTHREAD\_MUTEX\_INITIALIZER;

以下两个根据内核配置情况，不一定支持静态初始化。

 递规锁pthread\_mutex\_t recmutex = PTHREAD\_RECURSIVE\_MUTEX\_INITIALIZER\_NP;

 检错锁pthread\_mutex\_t errchkmutex = PTHREAD\_ERRORCHECK\_MUTEX\_INITIALIZER\_NP;

#### pthread\_mutex\_trylock() 线程互斥锁测试加锁

原型：int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);

功能：与pthread\_mutex\_lock()类似，如果互斥锁未被上锁则对其上锁，不同的是如锁已经被占用时返回EBUSY而不是挂起等待。

参数：mutex 指针要销毁的线程互斥锁结构变量的指针

返回：返回：失败：错误码（非0）；成功：0

使用该函数可以避免一个线程因重复对一个互斥锁上锁而导致 “死锁”的情况，使用该函数时一般要结合返回值决定若当前已上锁则进行哪些操作。

### 加锁注意事项（信号中不应该申请锁）

1. 如果线程在加锁后解锁前被取消，锁将永远保持锁定状态，因此如果在关键区段内有取消点存在，则必须在退出回调函数pthread\_cleanup\_push/pthread\_cleanup\_pop中解锁。
2. 同时不应该在信号处理函数中使用互斥锁，否则容易造成死锁。

**注意：**信号处理函数执行等级比主线程高，信号函数还没有执行完成前，主线程函数得不到运行，所以如果在信号函数中因为申请锁阻塞了，则主线程也就阻塞了。

### 互斥锁示例

示例：多线程使用互斥锁实现交替报数1~30

|  |
| --- |
| dingwei@dingwei-virtual-machine:~/桌面/thread$ cat mutex\_multithread\_count.c  #include <pthread.h>  #include <stdio.h>  #include <unistd.h>  #include <stdlib.h>  void\* task\_count(void \*arg);  pthread\_t tid[3];  pthread\_mutex\_t lock;  int cnt = 0;  int main(void)  {  int ret;    pthread\_mutex\_init(&lock,NULL);  pthread\_create(&tid[0],NULL,task\_count,(void \*)1);  pthread\_create(&tid[1],NULL,task\_count,(void \*)2);  pthread\_create(&tid[2],NULL,task\_count,(void \*)3);  pthread\_join(tid[0],NULL);  pthread\_join(tid[1],NULL);  pthread\_join(tid[2],NULL);  printf("Main:ALL new threads end\r\n");    pthread\_mutex\_destroy(&lock);  pthread\_exit(NULL);  }  void\* task\_count(void \*arg)  {  printf("I am new thread NO.%d\r\n",(int)arg);  while(1)  {  pthread\_mutex\_lock(&lock);//上锁  if(cnt < 30)  {  cnt++;  printf("NO.%d : cnt = %d\r\n",(int)arg,cnt);  pthread\_mutex\_unlock(&lock);//解锁  usleep(20000);//线程显示cnt后，休眠，允许其他线程被调度  }  else  {  pthread\_mutex\_unlock(&lock);//解锁  pthread\_exit(NULL);  }  }  pthread\_exit(NULL);  }  //上锁到解锁之间操作是被保护操作  dingwei@dingwei-virtual-machine:~/桌面/thread$ ./mutex\_multithread\_count  I am new thread NO.1  NO.1 : cnt = 1  I am new thread NO.3  NO.3 : cnt = 2  I am new thread NO.2  NO.2 : cnt = 3  NO.3 : cnt = 4  NO.1 : cnt = 5  NO.2 : cnt = 6  NO.2 : cnt = 7  NO.3 : cnt = 8  NO.1 : cnt = 9  NO.2 : cnt = 10  NO.3 : cnt = 11  NO.1 : cnt = 12  NO.3 : cnt = 13  NO.2 : cnt = 14  NO.1 : cnt = 15  NO.3 : cnt = 16  NO.2 : cnt = 17  NO.1 : cnt = 18  NO.3 : cnt = 19  NO.2 : cnt = 20  NO.1 : cnt = 21  NO.2 : cnt = 22  NO.1 : cnt = 23  NO.3 : cnt = 24  NO.1 : cnt = 25  NO.2 : cnt = 26  NO.3 : cnt = 27  NO.1 : cnt = 28  NO.2 : cnt = 29  NO.3 : cnt = 30  Main:ALL new threads end |

作业：编程实现模拟售票系统：

1. 车票ticket初值是20
2. 用2个线程表示 2个售票窗口任务，每次只能销售一张车票ticket，每次实现售票需要花费1s，并且每次售票后会显示当前剩余的票数
3. 用1个线程表示 1个补充票数任务，当检测到ticket==0时，就会将票数补到10。

补充：编写shell脚本的步骤

1. 创建一个文件 vim test.sh
2. 编写脚本

（运行一个脚本后会依次执行每个命令）

脚本的第一句用于说明脚本的类型（不可少），比如：#!/bin/sh

1. 由于脚本是命令的集合，所以不需要编译，直接运行即可
2. 由于使用vim创建文件时，不具有可执行权限，需要使用命令增加可执行权限

sudo chmod 0777 test.sh 或

sudo chmod u+x test.sh

1. 运行脚本./test.sh