 Linux线程锁详解pthread\_mutexattr\_t

**1． 创建和销毁**

　　有两种方法创建互斥锁，静态方式和动态方式。

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LinuxThreads实现中，pthread\_mutex\_t是一个结构，而PTHREAD\_MUTEX\_INITIALIZER则是一个结构常量。

**动态方式**是采用pthread\_mutex\_init()函数来初始化互斥锁，

API定义如下： int pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*mutexattr) 其中mutexattr用于指定互斥锁属性（见下），如果为NULL则使用缺省属性。

　　pthread\_mutex\_destroy ()用于**注销一个互斥锁**，

API定义如下： int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex) 销毁一个互斥锁即意味着释放它所占用的资源，且要求锁当前处于开放状态。由于在Linux中，互斥锁并不占用任何资源，因此LinuxThreads中的 pthread\_mutex\_destroy()除了检查锁状态以外（锁定状态则返回EBUSY）没有其他动作。

**2． 互斥锁属性**

**互斥锁的属性在创建锁的时候指定**，在LinuxThreads实现中仅有一个锁类型属性，不同的锁类型在试图对一个已经被锁定的互斥锁加锁时表现不同。当前（glibc2.2.3,linuxthreads0.9）有四个值可供选择：

　　\* PTHREAD\_MUTEX\_TIMED\_NP，这是缺省值，也就是普通锁。当一个线程加锁以后，其余请求锁的线程将形成一个等待队列，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。

　　\* PTHREAD\_MUTEX\_RECURSIVE\_NP，嵌套锁，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。

　　\* PTHREAD\_MUTEX\_ERRORCHECK\_NP，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\_TIMED\_NP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。

　　\* PTHREAD\_MUTEX\_ADAPTIVE\_NP，适应锁，动作最简单的锁类型，仅等待解锁后重新竞争。

**#include <pthread.h>**

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

**3． 锁操作**

　　锁操作主要包括加锁pthread\_mutex\_lock()、解锁pthread\_mutex\_unlock()和测试加锁 pthread\_mutex\_trylock()三个，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。

对于普通锁和适应锁类型，解锁者可以是同进程内任何线程；而检错锁则必须由加锁者解锁才有效，否则返回EPERM；对于嵌套锁，文档和实现要求必须由加锁者解锁，但实验结果表明并没有这种限制，这个不同目前还没有得到解释。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。

**int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex)**

**int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex)**

**int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex)**

　　pthread\_mutex\_trylock()语义与pthread\_mutex\_lock()类似，不同的是在锁已经被占据时返回EBUSY而不是挂起等待。

4． 其他

　　POSIX 线程锁机制的Linux实现都不是取消点，因此，延迟取消类型的线程不会因收到取消信号而离开加锁等待。值得注意的是，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，或者设置了异步取消类型，则必须在退出回调函数中解锁。

　　这个锁机制同时也不是异步信号安全的，也就是说，不应该在信号处理过程中使用互斥锁，否则容易造成死锁。

　　互斥锁属性使用互斥锁（互斥）可以使线程按顺序执行。通常，互斥锁通过确保一次只有一个线程执行代码的临界段来同步多个线程。互斥锁还可以保护单线程代码。

　　要更改缺省的互斥锁属性，可以对属性对象进行声明和初始化。通常，互斥锁属性会设置在应用程序开头的某个位置，以便可以快速查找和轻松修改。表 4–1 列出了用来处理互斥锁属性的函数。

　　表 4–1 互斥锁属性例程

|  |  |
| --- | --- |
| **操作** | **相关函数说明** |
| 初始化互斥锁属性 对象 | pthread\_mutexattr\_init 语法 |
|  |  |
| 销毁互斥锁属性对象 | pthread\_mutexattr\_destroy 语法 |
| 设置互斥锁范围 | pthread\_mutexattr\_setpshared 语法 |
| 获取互斥锁范围 | pthread\_mutexattr\_getpshared 语法 |
| 设置互斥锁的类型属性 | pthread\_mutexattr\_settype 语法 |
| 获取互斥锁的类型属性 | pthread\_mutexattr\_gettype 语法 |
| 设置互斥锁属性的协议 | pthread\_mutexattr\_setprotocol 语法 |
| 获取互斥锁属性的协议 | pthread\_mutexattr\_getprotocol 语法 |
| 设置互斥锁属性的优先级上限 | pthread\_mutexattr\_setprioceiling 语法 |
| 获取互斥锁属性的优先级上限 | pthread\_mutexattr\_getprioceiling 语法 |
| 设置互斥锁的优先级上限 | pthread\_mutex\_setprioceiling 语法 |
| 获取互斥锁的优先级上限 | pthread\_mutex\_getprioceiling 语法 |
| 设置互斥锁的强健属性 | pthread\_mutexattr\_setrobust\_np 语法 |
| 获取互斥锁的强健属性 | pthread\_mutexattr\_getrobust\_np 语法 |