**pthread\_mutex\_lock**

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

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

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

     描述 pthread\_mutex\_lock()函数锁住由mutex指定的mutex 对象。如果mutex已经被锁住，调用这个函数的线程阻塞直到mutex可用为止。这跟函数返回的时候参数mutex指定的mutex对象变成锁住状态， 同时该函数的调用线程成为该mutex对象的拥有者。

     如果mutex 对象的type是 PTHREAD\_MUTEX\_NORMAL，不进行deadlock detection(死锁检测)。企图进行relock 这个mutex会导致deadlock. 如果一个线程对未加锁的或已经unlock的mutex对象进行unlock操作，结果是不未知的。

     如果mutex类型是 PTHREAD\_MUTEX\_ERRORCHECK，那么将进行错误检查。如果一个线程企图对一个已经锁住的mutex进行relock，将返回一个错 误。如果一个线程对未加锁的或已经unlock的mutex对象进行unlock操作，将返回一个错误。

     如果mutex类型是 PTHREAD\_MUTEX\_RECURSIVE，mutex会有一个锁住次数（lock count）的概念。当一个线程成功地第一次锁住一个mutex的时候，锁住次数（lock count）被设置为1，每一次一个线程unlock这个mutex的时候，锁住次数（lock count）就减1。当锁住次数（lock count）减少为0的时候，其他线程就能获得该mutex锁了。如果一个线程对未加锁的或已经unlock的mutex对象进行unlock操作，将返 回一个错误。

    如果mutex类型是 PTHREAD\_MUTEX\_DEFAULT，企图递归的获取这个mutex的锁的结果是不确定的。unlock一个不是被调用线程锁住的mutex的结 果也是不确定的。企图unlock一个未被锁住的mutex导致不确定的结果。

    pthread\_mutex\_trylock()调用在参数mutex指定的mutex对象当前被锁住的时候立即返回，除此之外，pthread\_mutex\_trylock()跟pthread\_mutex\_lock()功能完全一样。

    pthread\_mutex\_unlock()函数释放有参数mutex指定的mutex对象的锁。如果被释放取决于该Mutex对象的类型属性。如果有多个线程为了获得该mutex锁阻塞，调用pthread\_mutex\_unlock()将是该mutex可用，一定的调度策略将被用来决定哪个线程可以获得该mutex锁。（在mutex类型为PTHREAD\_MUTEX\_RECURSIVE 的情况下，只有当lock count 减为0并且调用线程在该mutex上已经没有锁的时候）（翻译到这里，才觉得我的这个锁概念是多么模糊） 如果一个线程在等待一个mutex锁得时候收到了一个signal,那么在从signal handler返回的时候，该线程继续等待该mutex锁，就像这个线程没有被中断一样。

    返回值成功

    pthread\_mutex\_lock() 和 pthread\_mutex\_unlock() 返回0，否则返回一个错误的提示码

    pthread\_mutex\_trylock() 在成功获得了一个mutex的锁后返回0，否则返回一个错误提示码错误

    pthread\_mutex\_lock() 和 pthread\_mutex\_unlock()失败的时候 [EINVAL] mutex在生成的时候，它的protocol属性的值是 PTHREAD\_PRIO\_PROTECT，同时调用线程的优先级(priority)比该mutex的当前prority上限高

pthread\_mutex\_trylock() 函数在一下情况会失败：

[EBUSY] The mutex could not be acquired because it was already locked. mutex已经被锁住的时候无法再获取锁

The pthread\_mutex\_lock(), pthread\_mutex\_trylock() and pthread\_mutex\_unlock() functions may fail if:

[EINVAL] mutex指向的mutex未被初始化

[EAGAIN] Mutex的lock count(锁数量)已经超过 递归索的最大值，无法再获得该mutex锁

pthread\_mutex\_lock() 函数在一下情况下会失败：

[EDEADLK] 当前线程已经获得该mutex锁

pthread\_mutex\_unlock() 函数在以下情况下会失败：

 [EPERM] 当前线程不是该mutex锁的拥有者 所有的这些函数的错误返回值都不会是[EINTR]  
  
互斥锁  
  
互斥锁用来保证一段时间内只有一个线程在执行一段代码。必要性显而易见：假设各个线程向同一个文件顺序写入数据，最后得到的结果一定是灾难性的。  
  
我们先看下面一段代码。这是一个读/写程序，它们公用一个缓冲区，并且我们假定一个缓冲区只能保存一条信息。即缓冲区只有两个状态：有信息或没有信息。

void reader\_function ( void );  
void writer\_function ( void );   
  
char buffer;  
int buffer\_has\_item=0;  
pthread\_mutex\_t mutex;  
struct timespec delay;  
void main ( void ){  
pthread\_t reader;  
/\* 定义延迟时间\*/  
delay.tv\_sec = 2;  
delay.tv\_nec = 0;  
/\* 用默认属性初始化一个互斥锁对象\*/  
pthread\_mutex\_init (&mutex,NULL);  
pthread\_create(&reader, pthread\_attr\_default, (void \*)&reader\_function), NULL);  
writer\_function( );  
}  
  
void writer\_function (void){  
while(1){  
/\* 锁定互斥锁\*/  
pthread\_mutex\_lock (&mutex);  
if (buffer\_has\_item==0){  
buffer=make\_new\_item( );  
buffer\_has\_item=1;  
}  
/\* 打开互斥锁\*/  
pthread\_mutex\_unlock(&mutex);  
pthread\_delay\_np(&delay);  
}  
}  
  
void reader\_function(void){  
while(1){  
pthread\_mutex\_lock(&mutex);  
if(buffer\_has\_item==1){  
consume\_item(buffer);  
buffer\_has\_item=0;  
}  
pthread\_mutex\_unlock(&mutex);  
pthread\_delay\_np(&delay);  
}  
}

这里声明了互斥锁变量mutex，结构pthread\_mutex\_t为不公开的数据类型，其中包含一个系统分配的属性对象。函数 pthread\_mutex\_init用来生成一个互斥锁。NULL参数表明使用默认属性。如果需要声明特定属性的互斥锁，须调用函数 pthread\_mutexattr\_init。函数pthread\_mutexattr\_setpshared和函数 pthread\_mutexattr\_settype用来设置互斥锁属性。前一个函数设置属性pshared，它有两个取 值，PTHREAD\_PROCESS\_PRIVATE和PTHREAD\_PROCESS\_SHARED。前者用来不同进程中的线程同步，后者用于同步本进 程的不同线程。在上面的例子中，我们使用的是默认属性PTHREAD\_PROCESS\_ PRIVATE。后者用来设置互斥锁类型，可选的类型有PTHREAD\_MUTEX\_NORMAL、PTHREAD\_MUTEX\_ERRORCHECK、 PTHREAD\_MUTEX\_RECURSIVE和PTHREAD \_MUTEX\_DEFAULT。它们分别定义了不同的上所、解锁机制，一般情况下，选用最后一个默认属性。  
  
pthread\_mutex\_lock声明开始用互斥锁上锁，此后的代码直至调用pthread\_mutex\_unlock为止，均被上锁，即同一时间只 能被一个线程调用执行。当一个线程执行到pthread\_mutex\_lock处时，如果该锁此时被另一个线程使用，那此线程被阻塞，即程序将等待到另一 个线程释放此互斥锁。在上面的例子中，我们使用了pthread\_delay\_np函数，让线程睡眠一段时间，就是为了防止一个线程始终占据此函数。  
  
上面的例子非常简单，就不再介绍了，需要提出的是在使用互斥锁的过程中很有可能会出现死锁：两个线程试图同时占用两个资源，并按不同的次序锁定相应的互 斥锁，例如两个线程都需要锁定互斥锁1和互斥锁2，a线程先锁定互斥锁1，b线程先锁定互斥锁2，这时就出现了死锁。此时我们可以使用函数 pthread\_mutex\_trylock，它是函数pthread\_mutex\_lock的非阻塞版本，当它发现死锁不可避免时，它会返回相应的信 息，程序员可以针对死锁做出相应的处理。另外不同的互斥锁类型对死锁的处理不一样，但最主要的还是要程序员自己在程序设计注意这一点。