基础5-Posix线程编程指南

线程创建

1．1 线程与进程

相对进程而言，线程是一个更加接近于执行体的概念，它可以与同进程中的其他线程共享数据，但拥有自己的栈空间，拥有独立的执行序列。在串行程序基础上引入线程和进程是为了提高程序的并发度，从而提高程序运行效率和响应时间。

线程和进程在使用上各有优缺点：线程执行开销小，但不利于资源的管理和保护；而进程正相反。同时，线程适合于在SMP机器上运行，而进程则可以跨机器迁移。

1．2 创建线程

POSIX通过pthread_create()函数创建线程，API定义如下：

```
1  int  pthread_create(pthread_t  *  thread, pthread_attr_t * attr,
2  void * (*start_routine)(void *), void * arg)
```

与fork()调用创建一个进程的方法不同，pthread_create()创建的线程并不具备与主线程（即调用pthread_create()的线程）同样的执行序列，而是使其运行start_routine(arg)函数。thread返回创建的线程ID，而attr是创建线程时设置的线程属性（见下）。pthread_create()的返回值表示线程创建是否成功。尽管arg是void *类型的变量，但它同样可以作为任意类型的参数传给start_routine()函数；同时，start_routine()可以返回一个void *类型的返回值，而这个返回值也可以是其他类型，并由pthread_join()获取。

1．3 线程创建属性

pthread_create()中的attr参数是一个结构指针，结构中的元素分别对应着新线程的运行属性，主要包括以下几项：

__detachstate，表示新线程是否与进程中其他线程脱离同步，如果置位则新线程不能用pthread_join()来同步，且在退出时自行释放所占用的资源。缺省为PTHREAD_CREATE_JOINABLE状态。这个属性也可以在线程创建并运行以后用pthread_detach()来设置，而一旦设置为PTHREAD_CREATE_DETACH状态（不论是创建时设置还是运行时设置）则不能再恢复到PTHREAD_CREATE_JOINABLE状态。

__schedpolicy，表示新线程的调度策略，主要包括SCHED_OTHER（正常、非实时）、SCHED_RR（实时、轮转法）和SCHED_FIFO（实时、先入先出）三种，缺省为SCHED_OTHER，后两种调度策略仅对超级用户有效。运行时可以用过pthread_setschedparam()来改变。

__schedparam，一个struct sched_param结构，目前仅有一个sched_priority整型变量表示线程的运行优先级。这个参数仅当调度策略为实时（即SCHED_RR或SCHED_FIFO）时才有效，并可以在运行时通过pthread_setschedparam()函数来改变，缺省为0。

__inheritsched，有两种值可供选择：PTHREAD_EXPLICIT_SCHED和PTHREAD_INHERIT_SCHED，前者表示新线程使用显式指定调度策略和调度参数（即attr中的值），而后者表示继承调用者线程的值。缺省为PTHREAD_EXPLICIT_SCHED。

__scope，表示线程间竞争CPU的范围，也就是说线程优先级的有效范围。POSIX的标准中定义了两个值：PTHREAD_SCOPE_SYSTEM和PTHREAD_SCOPE_PROCESS，前者表示与系统中所有线程一起竞争CPU时间，后者表示仅与同进程中的线程竞争CPU。目前LinuxThreads仅实现了PTHREAD_SCOPE_SYSTEM一值。

pthread_attr_t结构中还有一些值，但不使用pthread_create()来设置。

为了设置这些属性，POSIX定义了一系列属性设置函数，包括pthread_attr_init()、pthread_attr_destroy()和与各个属性相关的pthread_attr_get---/pthread_attr_set---函数。

1．4 线程创建的Linux实现

我们知道，Linux的线程实现是在核外进行的，核内提供的是创建进程的接口do_fork()。内核提供了两个系统调用__clone()和fork()，最终都用不同的参数调用do_fork()核内API。当然，要想实现线程，没有核心对多进程（其实是轻量级进程）共享数据段的支持是不行的，因此，do_fork()提供了很多参数，包括CLONE_VM（共享内存空

间）、CLONE_FS（共享文件系统信息）、CLONE_FILES（共享文件描述符表）、CLONE_SIGHAND（共享信号句柄表）和CLONE_PID（共享进程ID，仅对核内进程，即0号进程有效）。当使用fork系统调用时，内核调用do_fork()不使用任何共享属性，进程拥有独立的运行环境，而使用pthread_create()来创建线程时,则最终设置了所有这些属性来调用__clone()，而这些参数又全部传给核内的do_fork()，从而创建的"进程"拥有共享的运行环境，只有栈是独立的，由__clone()传入。

Linux线程在核内是以轻量级进程的形式存在的，拥有独立的进程表项，而所有的创建、同步、删除等操作都在核外pthread库中进行。pthread库使用一个管理线程（__pthread_manager()，每个进程独立且唯一）来管理线程的创建和终止，为线程分配线程ID，发送线程相关的信号（比如Cancel），而主线程（pthread_create()）的调用者则通过管道将请求信息传给管理线程。

线程取消

2．1 线程取消的定义

一般情况下，线程在其主体函数退出的时候会自动终止，但同时也可以因为接收到另一个线程发来的终止（取消）请求而强制终止。

2．2 线程取消的语义

线程取消的方法是向目标线程发Cancel信号，但如何处理Cancel信号则由目标线程自己决定，或者忽略、或者立即终止、或者继续运行至Cancelation-point（取消点），由不同的Cancelation状态决定。

线程接收到CANCEL信号的缺省处理（即pthread_create()创建线程的缺省状态）是继续运行至取消点，也就是说设置一个CANCELED状态，线程继续运行，只有运行至Cancelation-point的时候才会退出。

2．3 取消点

根据POSIX标准，pthread_join()、pthread_testcancel()、pthread_cond_wait()、pthread_cond_timedwait()、sem_wait()、sigwait()等函数以及read()、write()等会引起阻塞的系统调用都是Cancelation-point，而其他pthread函数都不会引起Cancelation动作。但是pthread_cancel的手册页声称，由于LinuxThread库与C库结合得不好，因而目前C库函数都不是Cancelation-point；但CANCEL信号会使线程从阻塞的系统调用中退出，并置EINTR错误码，因此可以在需要作为Cancelation-point的系统调用前后调用pthread_testcancel()，从而达到POSIX标准所要求的目标，即如下代码段：

```
1  pthread_testcancel();
2      retcode = read(fd, buffer, length);
3      pthread_testcancel();
```

2．4 程序设计方面的考虑

如果线程处于无限循环中，且循环体内没有执行至取消点的必然路径，则线程无法由外部其他线程的取消请求而终止。因此在这样的循环体的必经路径上应该加入pthread_testcancel()调用。

2．5 与线程取消相关的pthread函数

int pthread_cancel(pthread_t thread)
发送终止信号给thread线程，如果成功则返回0，否则为非0值。发送成功并不意味着thread会终止。

int pthread_setcancelstate(int state, int *oldstate)
设置本线程对Cancel信号的反应，state有两种值：PTHREAD_CANCEL_ENABLE（缺省）和PTHREAD_CANCEL_DISABLE，分别表示收到信号后设为CANCLED状态和忽略CANCEL信号继续运行；old_state如果不为NULL则存入原来的Cancel状态以便恢复。

int pthread_setcanceltype(int type, int *oldtype)
设置本线程取消动作的执行时机，type由两种取值：PTHREAD_CANCEL_DEFFERED和PTHREAD_CANCEL_ASYCHRONOUS，仅当Cancel状态为Enable时有效，分别表示收到信号后继续运行至下一个取消点再退出和立即执行取消动作（退出）；oldtype如果不为NULL则存入运来的取消动作类型值。

void pthread_testcancel(void)
检查本线程是否处于Canceld状态，如果是，则进行取消动作，否则直接返回。

概念及作用

在单线程程序中，我们经常要用到"全局变量"以实现多个函数间共享数据。在多线程环境下，由于数据空间是共享的，因此全局变量也为所有线程所共有。但有时应用程序设计中有必要提供线程私有的全局变量，仅在某个线程中有效，但却可以跨多个函数访问，比如程序可能需要每个线程维护一个链表，而使用相同的函数操作，最简单的办法就是使用同名而不同变量地址的线程相关数据结构。这样的数据结构可以由Posix线程库维护，称为线程私有数据（Thread-specific Data，或TSD）。

创建和注销

Posix定义了两个API分别用来创建和注销TSD：

```
1  int pthread_key_create(pthread_key_t *key, void (*destr_function) (void *))
```

该函数从TSD池中分配一项，将其值赋给key供以后访问使用。如果destr_function不为空，在线程退出（pthread_exit()）时将以key所关联的数据为参数调用destr_function()，以释放分配的缓冲区。

不论哪个线程调用pthread_key_create()，所创建的key都是所有线程可访问的，但各个线程可根据自己的需要往key中填入不同的值，这就相当于提供了一个同名而不同值的全局变量。在LinuxThreads的实现中，TSD池用一个结构数组表示：

```
1  static struct pthread_key_struct pthread_keys[PTHREAD_KEYS_MAX] = { { 0, NULL } };
```

创建一个TSD就相当于将结构数组中的某一项设置为"in_use"，并将其索引返回给*key，然后设置destructor函数为destr_function。

注销一个TSD采用如下API：

```
1  int pthread_key_delete(pthread_key_t key)
```

这个函数并不检查当前是否有线程正使用该TSD，也不会调用清理函数（destr_function），而只是将TSD释放以供下一次调用pthread_key_create()使用。在LinuxThreads中，它还会将与之相关的线程数据项设为NULL（见"访问"）。

访问

TSD的读写都通过专门的Posix Thread函数进行，其API定义如下：

```
1  int  pthread_setspecific(pthread_key_t  key,  const   void  *pointer)
2  void * pthread_getspecific(pthread_key_t key)
```

写入（pthread_setspecific()）时，将pointer的值（不是所指的内容）与key相关联，而相应的读出函数则将与key相关联的数据读出来。数据类型都设为void *，因此可以指向任何类型的数据。

在LinuxThreads中，使用了一个位于线程描述结构（_pthread_descr_struct）中的二维void *指针数组来存放与key关联的数据，数组大小由以下几个宏来说明：

```
1  #define PTHREAD_KEY_2NDLEVEL_SIZE       32
2  #define PTHREAD_KEY_1STLEVEL_SIZE   \
3  ((PTHREAD_KEYS_MAX + PTHREAD_KEY_2NDLEVEL_SIZE - 1)
4  / PTHREAD_KEY_2NDLEVEL_SIZE)
5      其中在/usr/include/bits/local_lim.h中定义了PTHREAD_KEYS_MAX为1024，
6      因此一维数组大小为32。而具体存放的位置由key值经过以下计算得到：
7  idx1st = key / PTHREAD_KEY_2NDLEVEL_SIZE
8  idx2nd = key % PTHREAD_KEY_2NDLEVEL_SIZE
```

也就是说，数据存放与一个32×32的稀疏矩阵中。同样，访问的时候也由key值经过类似计算得到数据所在位置索引，再取出其中内容返回。

使用范例

以下这个例子没有什么实际意义，只是说明如何使用，以及能够使用这一机制达到存储线程私有数据的目的。

```
#include <stdio.h>
#include <pthread.h>
pthread_key_t   key;
void echomsg(int t)
{
        printf("destructor excuted in thread %d,param=%d\n",pthread_self(),t);
}
void * child1(void *arg)
{
        int tid=pthread_self();
        printf("thread %d enter\n",tid);
        pthread_setspecific(key,(void *)tid);
        sleep(2);
        printf("thread %d returns %d\n",tid,pthread_getspecific(key));
        sleep(5);
}
void * child2(void *arg)
{
        int tid=pthread_self();
        printf("thread %d enter\n",tid);
        pthread_setspecific(key,(void *)tid);
        sleep(1);
        printf("thread %d returns %d\n",tid,pthread_getspecific(key));
        sleep(5);
}
int main(void)
{
        int tid1,tid2;
        printf("hello\n");
        pthread_key_create(&key,echomsg);
        pthread_create(&tid1,NULL,child1,NULL);
        pthread_create(&tid2,NULL,child2,NULL);
        sleep(10);
        pthread_key_delete(key);
        printf("main thread exit\n");
        return 0;
}
```

给例程创建两个线程分别设置同一个线程私有数据为自己的线程ID，为了检验其私有性，程序错开了两个线程私有数据的写入和读出的时间，从程序运行结果可以看出，两个线程对TSD的修改互不干扰。同时，当线程退出时，清理函数会自动执行，参数为tid。

互斥锁

尽管在Posix Thread中同样可以使用IPC的信号量机制来实现互斥锁mutex功能，但显然semphore的功能过于强大了，在Posix Thread中定义了另外一套专门用于线程同步的mutex函数。

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。

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。

```
1 int pthread_mutex_lock(pthread_mutex_t *mutex)
2 int pthread_mutex_unlock(pthread_mutex_t *mutex)
3 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。

条件变量

条件变量是利用线程间共享的全局变量进行同步的一种机制，主要包括两个动作：一个线程等待"条件变量的条件成立"而挂起；另一个线程使"条件成立"（给出条件成立信号）。为了防止竞争，条件变量的使用总是和一个互斥锁结合在一起。

1. 创建和注销

条件变量和互斥锁一样，都有静态动态两种创建方式，静态方式使用PTHREAD_COND_INITIALIZER常量，如下：
pthread_cond_t cond=PTHREAD_COND_INITIALIZER

动态方式调用pthread_cond_init()函数，API定义如下：
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr)

尽管POSIX标准中为条件变量定义了属性，但在LinuxThreads中没有实现，因此cond_attr值通常为NULL，且被忽略。

注销一个条件变量需要调用pthread_cond_destroy()，只有在没有线程在该条件变量上等待的时候才能注销这个条件变量，否则返回EBUSY。因为Linux实现的条件变量没有分配什么资源，所以注销动作只包括检查是否有等待线程。API定义如下：
int pthread_cond_destroy(pthread_cond_t *cond)

2. 等待和激发

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t
*mutex, const struct timespec *abstime)
```

等待条件有两种方式：无条件等待pthread_cond_wait()和计时等待pthread_cond_timedwait()，其中计时等待方式如果在给定时刻前条件没有满足，则返回ETIMEOUT，结束等待，其中abstime以与time()系统调用相同意义的绝对时间形式出现，0表示格林尼治时间1970年1月1日0时0分0秒。

无论哪种等待方式，都必须和一个互斥锁配合，以防止多个线程同时请求pthread_cond_wait()（或pthread_cond_timedwait()，下同）的竞争条件（Race Condition）。mutex互斥锁必须是普通锁（PTHREAD_MUTEX_TIMED_NP）或者适应锁（PTHREAD_MUTEX_ADAPTIVE_NP），且在调用pthread_cond_wait()前必须由本线程加锁（pthread_mutex_lock()），而在更新条件等待队列以前，mutex保持锁定状态，并在线程挂起进入等待前解锁。在条件满足从而离开pthread_cond_wait()之前，mutex将被重新加锁，以与进入pthread_cond_wait()前的加锁动作对应。

激发条件有两种形式，pthread_cond_signal()激活一个等待该条件的线程，存在多个等待线程时按入队顺序激活其中一个；而pthread_cond_broadcast()则激活所有等待线程。

3. 其他

pthread_cond_wait()和pthread_cond_timedwait()都被实现为取消点，因此，在该处等待的线程将立即重新运行，在重新锁定mutex后离开pthread_cond_wait()，然后执行取消动作。也就是说如果pthread_cond_wait()被取消，mutex是保持锁定状态的，因而需要定义退出回调函数来为其解锁。

以下示例集中演示了互斥锁和条件变量的结合使用，以及取消对于条件等待动作的影响。在例子中，有两个线程被启动，并等待同一个条件变量，如果不使用退出回调函数（见范例中的注释部分），则tid2将在pthread_mutex_lock()处永久等待。如果使用回调函数，则tid2的条件等待及主线程的条件激发都能正常工作。

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
pthread_mutex_t mutex;
pthread_cond_t  cond;
void * child1(void *arg)
{
        pthread_cleanup_push(pthread_mutex_unlock,&mutex);  /* comment 1 */
        while(1){
                printf("thread 1 get running \n");
        printf("thread 1 pthread_mutex_lock returns %d\n",
pthread_mutex_lock(&mutex));
        pthread_cond_wait(&cond,&mutex);
                    printf("thread 1 condition applied\n");
        pthread_mutex_unlock(&mutex);
                    sleep(5);
    }
        pthread_cleanup_pop(0);     /* comment 2 */
}
void *child2(void *arg)
{
        while(1){
                sleep(3);               /* comment 3 */
                printf("thread 2 get running.\n");
        printf("thread 2 pthread_mutex_lock returns %d\n",
pthread_mutex_lock(&mutex));
        pthread_cond_wait(&cond,&mutex);
        printf("thread 2 condition applied\n");
        pthread_mutex_unlock(&mutex);
        sleep(1);
        }
}
int main(void)
{
        int tid1,tid2;
        printf("hello, condition variable test\n");
        pthread_mutex_init(&mutex,NULL);
        pthread_cond_init(&cond,NULL);
        pthread_create(&tid1,NULL,child1,NULL);
        pthread_create(&tid2,NULL,child2,NULL);
        do{
        sleep(2);                   /* comment 4 */
                pthread_cancel(tid1);       /* comment 5 */
                sleep(2);                   /* comment 6 */
        pthread_cond_signal(&cond);
    }while(1);
        sleep(100);
        pthread_exit(0);
}
```

如果不做注释5的pthread_cancel()动作，即使没有那些sleep()延时操作，child1和child2都能正常工作。注释3和注释4的延迟使得child1有时间完成取消动作，从而使child2能在child1退出之后进入请求锁操作。如果没有注释1和注释2的回调函数定义，系统将挂起在child2请求锁的地方；而如果同时也不做注释3和注释4的延时，child2能在child1完成取消动作以前得到控制，从而顺利执行申请锁的操作，但却可能挂起在pthread_cond_wait()中，因为其中也有申请mutex的操作。child1函数给出的是标准的条件变量的使用方式：回调函数保护，等待条件前锁定，pthread_cond_wait()返回后解锁。

条件变量机制不是异步信号安全的，也就是说，在信号处理函数中调用pthread_cond_signal()或者pthread_cond_broadcast()很可能引起死锁。

信号灯

信号灯与互斥锁和条件变量的主要不同在于"灯"的概念，灯亮则意味着资源可用，灯灭则意味着不可用。如果说后两中同步方式侧重于"等待"操作，即资源不可用的话，信号灯机制则侧重于点灯，即告知资源可用；没有等待线程的解锁或激发条件都是没有意义的，而没有等待灯亮的线程的点灯操作则有效，且能保持灯亮状态。当然，这样的操作原语也意味着更多的开销。

信号灯的应用除了灯亮/灯灭这种二元灯以外，也可以采用大于1的灯数，以表示资源数大于1，这时可以称之为多元灯。

1. 创建和注销

POSIX信号灯标准定义了有名信号灯和无名信号灯两种，但LinuxThreads的实现仅有无名灯，同时有名灯除了总是可用于多进程之间以外，在使用上与无名灯并没有很大的区别，因此下面仅就无名灯进行讨论。

int sem_init(sem_t *sem, int pshared, unsigned int value)
这是创建信号灯的API，其中value为信号灯的初值，pshared表示是否为多进程共享而不仅仅是用于一个进程。LinuxThreads没有实现多进程共享信号灯，因此所有非0值的pshared输入都将使sem_init()返回-1，且置errno为ENOSYS。初始化好的信号灯由sem变量表征，用于以下点灯、灭灯操作。

int sem_destroy(sem_t * sem)
被注销的信号灯sem要求已没有线程在等待该信号灯，否则返回-1，且置errno为EBUSY。除此之外，LinuxThreads的信号灯注销函数不做其他动作。

2. 点灯和灭灯

```
1  int sem_post(sem_t * sem)
```

点灯操作将信号灯值原子地加1，表示增加一个可访问的资源。

```
1  int sem_wait(sem_t * sem)
2  int sem_trywait(sem_t * sem)
```

sem_wait()为等待灯亮操作，等待灯亮（信号灯值大于0），然后将信号灯原子地减1，并返回。sem_trywait()为sem_wait()的非阻塞版，如果信号灯计数大于0，则原子地减1并返回0，否则立即返回-1，errno置为EAGAIN。

3. 获取灯值

```
1  int sem_getvalue(sem_t * sem, int * sval)
```

读取sem中的灯计数，存于*sval中，并返回0。

4. 其他

sem_wait()被实现为取消点，而且在支持原子"比较且交换"指令的体系结构上，sem_post()是唯一能用于异步信号处理函数的POSIX异步信号安全的API。

异步信号

由于LinuxThreads是在核外使用核内轻量级进程实现的线程，所以基于内核的异步信号操作对于线程也是有效的。但同时，由于异步信号总是实际发往某个进程，所以无法实现POSIX标准所要求的"信号到达某个进程，然后再由该进程将信号分发到所有没有阻塞该信号的线程中"原语，而是只能影响到其中一个线程。

POSIX异步信号同时也是一个标准C库提供的功能，主要包括信号集管理（sigemptyset()、sigfillset()、sigaddset()、sigdelset()、sigismember()等）、信号处理函数安装（sigaction()）、信号阻塞控制（sigprocmask()）、被阻塞信号查询（sigpending()）、信号等待(sigsuspend())等，它们与发送信号的kill()等函数配合就能实现进程间异步信号功能。LinuxThreads围绕线程封装了sigaction()何raise()，本节集中讨论LinuxThreads中扩展的异步信号函数，包括pthread_sigmask()、pthread_kill()和sigwait()三个函数。毫无疑问，所有POSIX异步信号函数对于线程都是可用的。

int pthread_sigmask(int how, const sigset_t *newmask, sigset_t *oldmask)
设置线程的信号屏蔽码，语义与sigprocmask()相同，但对不允许屏蔽的Cancel信号和不允许响应的Restart信号进行了保护。被屏蔽的信号保存在信号队列中，可由sigpending()函数取出。

int pthread_kill(pthread_t thread, int signo)
向thread号线程发送signo信号。实现中在通过thread线程号定位到对应进程号以后使用kill()系统调用完成发送。

int sigwait(const sigset_t *set, int *sig)
挂起线程，等待set中指定的信号之一到达，并将到达的信号存入*sig中。POSIX标准建议在调用sigwait()等待信号以前，进程中所有线程都应屏蔽该信号，以保证仅有sigwait()的调用者获得该信号，因此，对于需要等待同步的异步信号，总是应该在创建任何线程以前调用pthread_sigmask()屏蔽该信号的处理。而且，调用sigwait()期间，原来附接在该信号上的信号处理函数不会被调用。

如果在等待期间接收到Cancel信号，则立即退出等待，也就是说sigwait()被实现为取消点。

其他同步方式

除了上述讨论的同步方式以外，其他很多进程间通信手段对于LinuxThreads也是可用的，比如基于文件系统的IPC（管道、Unix域Socket等）、消息队列（Sys.V或者Posix的）、System V的信号灯等。只有一点需要注意，LinuxThreads在核内是作为共享存储区、共享文件系统属性、共享信号处理、共享文件描述符的独立进程看待的。