Posix 线程编程指南

一 线程创建与取消

这是一个关于 Posix 线程编程的专栏。作者在阐明概念的基础上，将向您详细讲述 Posix 线程库 API。本文是第一篇将向您讲述线程的创建与取消。

线程创建

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。

2. 创建线程

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

```
int   pthread_create(pthread_t   *   thread, pthread_attr_t * attr,
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。

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。

4. 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

1. 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. 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。

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：

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

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。

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。

二 线程私有数据

这是一个关于 Posix 线程编程的专栏。作者在阐明概念的基础上，将向您详细讲述 Posix 线程库 API。本文是第二篇将向您讲述线程的私有数据。

1. 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）。

2. 创建和注销

Posix 定义了两个 API 分别用来创建和注销 TSD：

```
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：

```
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：

```
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"）。

3. 访问

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

```
int   pthread_setspecific(pthread_key_t   key,   const    void   *pointer)
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：

```
#define PTHREAD_KEY_2NDLEVEL_SIZE        32
#define PTHREAD_KEY_1STLEVEL_SIZE    \
((PTHREAD_KEYS_MAX + PTHREAD_KEY_2NDLEVEL_SIZE - 1)
/ PTHREAD_KEY_2NDLEVEL_SIZE)
```

其中在/usr/include/bits/local_lim.h 中定义了 PTHREAD_KEYS_MAX 为 1024，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：

```
idx1st = key / PTHREAD_KEY_2NDLEVEL_SIZE
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。

4. 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。

三 线程同步

这是一个关于 Posix 线程编程的专栏。作者在阐明概念的基础上，将向您详细讲述 Posix 线程库 API。本文是第三篇将向您讲述线程同步。

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 Linux Threads 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 Linux Threads 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。

```
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。

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

```
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。

```
int sem_wait(sem_t * sem)
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

```
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。

四 线程终止

这是一个关于 Posix 线程编程的专栏。作者在阐明概念的基础上，将向您详细讲述 Posix 线程库 API。本文是第四篇将向您讲述线程中止。

线程终止方式

一般来说，Posix 的线程终止有两种情况：正常终止和非正常终止。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，这是可预见的退出方式；非正常终止是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，这种退出方式是不可预见的。

线程终止时的清理

不论是可预见的线程终止还是异常终止，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。最经常出现的情形是资源独占锁的使用：线程为了访问临界资源而为其加上锁，但在访问过程中被外界取消，如果线程处于响应取消状态，且采用异步方式响应，或者在打开独占锁以前的运行路径上存在取消点，则该临界资源将永远处于锁定状态得不到释放。外界取消操作是不可预见的，因此的确需要一个机制来简化用于资源释放的编程。
在 POSIX 线程 API 中提供了一个 pthread_cleanup_push()/pthread_cleanup_pop()函数对用于自动释放资源--从 pthread_cleanup_push()的调用点到 pthread_cleanup_pop()之间的程序段中的终

止动作（包括调用 pthread_exit()和取消点终止）都将执行 pthread_cleanup_push()所指定的清理函数。API 定义如下：

```
void pthread_cleanup_push(void (*routine) (void   *),  void *arg)
void pthread_cleanup_pop(int execute)
```

pthread_cleanup_push()/pthread_cleanup_pop()采用先入后出的栈结构管理，void routine(void *arg)函数在调用 pthread_cleanup_push()时压入清理函数栈，多次对 pthread_cleanup_push()的调用将在清理函数栈中形成一个函数链，在执行该函数链时按照压栈的相反顺序弹出。execute 参数表示执行到 pthread_cleanup_pop()时是否在弹出清理函数的同时执行该函数，为 0 表示不执行，非 0 为执行；这个参数并不影响异常终止时清理函数的执行。
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)                                   \
  { struct _pthread_cleanup_buffer _buffer;                                 \
    _pthread_cleanup_push (&_buffer, (routine), (arg));
#define pthread_cleanup_pop(execute)                                        \
    _pthread_cleanup_pop (&_buffer, (execute)); }
```

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。在下面的例子里，当线程在"do some work"中终止时，将主动调用 pthread_mutex_unlock(mut)，以完成解锁动作。

```
pthread_cleanup_push(pthread_mutex_unlock, (void *) &mut);
pthread_mutex_lock(&mut);
/* do some work */
pthread_mutex_unlock(&mut);
pthread_cleanup_pop(0);
```

必须要注意的是，如果线程处于 PTHREAD_CANCEL_ASYNCHRONOUS 状态，上述代码段就有可能出错，因为 CANCEL 事件有可能在 pthread_cleanup_push()和 pthread_mutex_lock()之间发生，或者在 pthread_mutex_unlock()和 pthread_cleanup_pop()之间发生，从而导致清理函数 unlock 一个并没有加锁的 mutex 变量，造成错误。因此，在使用清理函数的时候，都应该暂时设置成 PTHREAD_CANCEL_DEFERRED 模式。为此，POSIX 的 Linux 实现中还提供了一对不保证可移植的 pthread_cleanup_push_defer_np()/pthread_cleanup_pop_defer_np()扩展函数，功能与以下代码段相当：

```
{ int oldtype;
 pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &oldtype);
 pthread_cleanup_push(routine, arg);
 ...
 pthread_cleanup_pop(execute);
 pthread_setcanceltype(oldtype, NULL);
 }
```

线程终止的同步及其返回值

一般情况下，进程中各个线程的运行都是相互独立的，线程的终止并不会通知，也不会影响其他线程，终止的线程所占用的资源也并不会随着线程的终止而得到释放。正如进程之间可以用 wait()系统调用来同步终止并释放资源一样，线程之间也有类似机制，那就是 pthread_join()函数。

```
void pthread_exit(void *retval)
int pthread_join(pthread_t th, void **thread_return)
int pthread_detach(pthread_t th)
```

pthread_join()的调用者将挂起并等待 th 线程终止，retval 是 pthread_exit()调用者线程（线程 ID 为 th）的返回值，如果 thread_return 不为 NULL，则*thread_return=retval。需要注意的是一个线程仅允许唯一的一个线程使用 pthread_join()等待它的终止，并且被等待的线程应该处于可 join 状态，即非 DETACHED 状态。
如果进程中的某个线程执行了 pthread_detach(th)，则 th 线程将处于 DETACHED 状态，这使得 th 线程在结束运行时自行释放所占用的内存资源，同时也无法由 pthread_join()同步，pthread_detach()执行之后，对 th 请求 pthread_join()将返回错误。
一个可 join 的线程所占用的内存仅当有线程对其执行了 pthread_join()后才会释放，因此为了避免内存泄漏，所有线程的终止，要么已设为 DETACHED，要么就需要使用 pthread_join()来回收。

关于 pthread_exit()和 return

理论上说，pthread_exit()和线程宿体函数退出的功能是相同的，函数结束时会在内部自动调用 pthread_exit()来清理线程相关的资源。但实际上二者由于编译器的处理有很大的不同。
在进程主函数（main()）中调用 pthread_exit()，只会使主函数所在的线程（可以说是进程的主线程）退出；而如果是 return，编译器将使其调用进程退出的代码（如_exit()），从而导致进程及其所有线程结束运行。
其次，在线程宿主函数中主动调用 return，如果 return 语句包含在 pthread_cleanup_push()/pthread_cleanup_pop()对中，则不会引起清理函数的执行，反而会导致 segment fault。

五 杂项

这是一个关于 Posix 线程编程的专栏。作者在阐明概念的基础上，将向您详细讲述 Posix 线程库 API。本文是第五篇将向您讲述 pthread_self()、pthread_equal()和 pthread_once()等杂项函数。
在 Posix 线程规范中还有几个辅助函数难以归类，暂且称其为杂项函数，主要包括 pthread_self()、pthread_equal()和 pthread_once()三个，另外还有一个 LinuxThreads 非可移植性扩展函数 pthread_kill_other_threads_np()。本文就介绍这几个函数的定义和使用。

获得本线程 ID

pthread_t pthread_self(void)
本函数返回本线程的标识符。
在 LinuxThreads 中，每个线程都用一个 pthread_descr 结构来描述，其中包含了**线程状态、线程 ID** 等所有需要的数据结构，此函数的实现就是在线程栈帧中找到本线程的 pthread_descr 结构，然后返回其中的 p_tid 项。
pthread_t 类型在 LinuxThreads 中定义为无符号长整型。

判断两个线程是否为同一线程

int pthread_equal(pthread_t thread1, pthread_t thread2)
判断两个线程描述符是否指向同一线程。在 LinuxThreads 中，线程 ID 相同的线程必然是同一个线程，因此，这个函数的实现仅仅判断 thread1 和 thread2 是否相等。

仅执行一次的操作

int pthread_once(pthread_once_t *once_control, void (*init_routine) (void))
本函数使用初值为 PTHREAD_ONCE_INIT 的 once_control 变量保证 init_routine()函数在本进程执行序列中仅执行一次。

```
#include <stdio.h>
#include <pthread.h>
pthread_once_t   once=PTHREAD_ONCE_INIT;
void      once_run(void)
{
        printf("once_run in thread %d\n",pthread_self());
}
void * child1(void *arg)
{
        int tid=pthread_self();
        printf("thread %d enter\n",tid);
        pthread_once(&once,once_run);
        printf("thread %d returns\n",tid);
}
void * child2(void *arg)
{
        int tid=pthread_self();
        printf("thread %d enter\n",tid);
        pthread_once(&once,once_run);
        printf("thread %d returns\n",tid);
}
int main(void)
```

```
{
        int tid1,tid2;
        printf("hello\n");
        pthread_create(&tid1,NULL,child1,NULL);
        pthread_create(&tid2,NULL,child2,NULL);
        sleep(10);
        printf("main thread exit\n");
        return 0;
}
```

once_run()函数仅执行一次，且究竟在哪个线程中执行是不定的，尽管 pthread_once(&once,once_run)出现在两个线程中。

LinuxThreads 使用互斥锁和条件变量保证由 pthread_once()指定的函数执行且仅执行一次，而 once_control 则表征是否执行过。如果 once_control 的初值不是 PTHREAD_ONCE_INIT（LinuxThreads 定义为 0），pthread_once()的行为就会不正常。在 LinuxThreads 中，实际"一次性函数"的执行状态有三种：NEVER（0）、IN_PROGRESS（1）、DONE（2），如果 once 初值设为 1，则由于所有 pthread_once()都必须等待其中一个激发"已执行一次"信号，因此所有 pthread_once()都会陷入永久的等待中；如果设为 2，则表示该函数已执行过一次，从而所有 pthread_once()都会立即返回 0。

pthread_kill_other_threads_np()

void pthread_kill_other_threads_np(void)

这个函数是 LinuxThreads 针对本身无法实现的 POSIX 约定而做的扩展。POSIX 要求当进程的某一个线程执行 exec*系统调用在进程空间中加载另一个程序时，当前进程的所有线程都应终止。由于 LinuxThreads 的局限性，该机制无法在 exec 中实现，因此要求线程执行 exec 前手工终止其他所有线程。pthread_kill_other_threads_np()的作用就是这个。

需要注意的是，pthread_kill_other_threads_np()并没有通过 pthread_cancel()来终止线程，而是直接向管理线程发"进程退出"信号，使所有其他线程都结束运行，而不经过 Cancel 动作，当然也不会执行退出回调函数。尽管 LinuxThreads 的实验结果与文档说明相同，但代码实现中却是用的 __pthread_sig_cancel 信号来 kill 线程，应该效果与执行 pthread_cancel()是一样的，其中原因目前还不清楚。