**clone的fork与pthread\_create创建线程有何不同&pthread多线程编程的学习小结**

转载[rock\_joker](https://me.csdn.net/rock_joker) 最后发布于2017-05-25 09:32:36 阅读数 4047  收藏

展开

  进程是一个指令执行流及其执行环境，其执行环境是一个系统资源的集合，这些资源在Linux中被抽

象成各种数据对象：进程控制块、虚存空间、文件系统，文件I/O、信号处理函数。所以创建一个进程的

过程就是这些数据对象的创建过程。

  在调用系统调用fork创建一个进程时，子进程只是完全复制父进程的资源，这样得到的子进程独立于

父进程，具有良好的并发性，但是**二者之间的通讯需要通过专门的通讯机制，如：pipe，fifo，System V**

**IPC机制等，另外通过fork创建子进程系统开销很大，需要将上面描述的每种资源都复制一个副本。这样**

**看来，fork是一个开销十分大的系统调用，这些开销并不是所有的情况下都是必须的**，比如某进程fork出

一个子进程后，其子进程仅仅是为了调用exec执行另一个执行文件，那么在fork过程中对于虚存空间的复

制将是一个多余的过程（由于Linux中是采取了copy-on-write技术，所以这一步骤的所做的工作只是虚存

管理部分的复制以及页表的创建，而并没有包括物理也面的拷贝）；另外，**有时一个进程中具有几个独立**

**的计算单元，可以在相同的地址空间上基本无冲突进行运算，但是为了把这些计算单元分配到不同的处理**

**器上，需要创建几个子进程，然后各个子进程分别计算最后通过一定的进程间通讯和同步机制把计算结果**

**汇总，这样做往往有许多格外的开销，而且这种开销有时足以抵消并行计算带来的好处**。

另 **进程是系统中程序执行和资源分配的基本单位。每个进程都拥有自己的  
数据段、代码段和堆栈段，这就造成了进程在进行切换等操作时都需要有比较负责的上下文  
切换等动作。为了进一步减少处理机的空转时间支持多处理器和减少上下文切换开销**

  这说明了把计算单元抽象到进程上是不充分的，这也就是许多系统中都引入了线程的概念的原因。在

讲述线程前首先介绍以下vfork系统调用，vfork系统调用不同于fork，用vfork创建的子进程共享地址空

间，也就是说子进程完全运行在父进程的地址空间上，子进程对虚拟地址空间任何数据的修改同样为父进

程所见。但是用vfork创建子进程后，父进程会被阻塞直到子进程调用exec或exit。这样的好处是在子进

程被创建后仅仅是为了调用exec执行另一个程序时，因为它就不会对父进程的地址空间有任何引用，所以

对地址空间的复制是多余的，通过vfork可以减少不必要的开销。

  在Linux中， fork和vfork都是调用同一个核心函数

do\_fork(unsigned long clone\_flag, unsigned long usp, structpt\_regs)

其中clone\_flag包括CLONE\_VM, CLONE\_FS, CLONE\_FILES, CLONE\_SIGHAND,CLONE\_PID，CLONE\_VFORK

等等标志位，任何一位被置1了则表明创建的子进程和父进程共享该位对应的资源。所以在vfork的实现中

，cloneflags = CLONE\_VFORK | CLONE\_VM |SIGCHLD，这表示子进程和父进程共享地址空间，同时

do\_fork会检查CLONE\_VFORK，如果该位被置1了，子进程会把父进程的地址空间锁住，直到子进程退出或

执行exec时才释放该锁。

在讲述clone系统调用前先简单介绍线程的一些概念。

线程是在进程的基础上进一步的抽象，也就是说一个进程分为两个部分：线程集合和资源集合。线程

是进程中的一个动态对象，它应该是一组独立的指令流，进程中的所有线程将共享进程里的资源。但是线

程应该有自己的私有对象：比如程序计数器、堆栈和寄存器上下文。

线程分为三种类型：

内核线程、轻量级进程和用户线程。

**内核线程：**

它的创建和撤消是由内核的内部需求来决定的，用来负责执行一个指定的函数，一个内核线程不需要

和一个用户进程联系起来。它共享内核的正文段核全局数据，具有自己的内核堆栈。它能够单独的被调度

并且使用标准的内核同步机制，可以被单独的分配到一个处理器上运行。内核线程的调度由于不需要经过

态的转换并进行地址空间的重新映射，因此在内核线程间做上下文切换比在进程间做上下文切换快得多。

**轻量级进程：**

轻量级进程是核心支持的用户线程，它在一个单独的进程中提供多线程控制。这些轻量级进程被单独

的调度，可以在多个处理器上运行，每一个轻量级进程都被绑定在一个内核线程上，而且在它的生命周期

这种绑定都是有效的。轻量级进程被独立调度并且共享地址空间和进程中的其它资源，但是每个LWP都应

该有自己的程序计数器、寄存器集合、核心栈和用户栈。

**用户线程：**

用户线程是通过线程库实现的。它们可以在没有内核参与下创建、释放和管理。线程库提供了同步和

调度的方法。这样进程可以使用大量的线程而不消耗内核资源，而且省去大量的系统开销。用户线程的实

现是可能的，因为用户线程的上下文可以在没有内核干预的情况下保存和恢复。每个用户线程都可以有自

己的用户堆栈，一块用来保存用户级寄存器上下文以及如信号屏蔽等状态信息的内存区。库通过保存当前

线程的堆栈和寄存器内容载入新调度线程的那些内容来实现用户线程之间的调度和上下文切换。

内核仍然负责进程的切换，因为只有内核具有修改内存管理寄存器的权力。用户线程不是真正的调度

实体，**内核对它们一无所知，而只是调度用户线程下的进程或者轻量级进程，这些进程再通过线程库函数**

**来调度它们的线程。当一个进程被抢占时，它的所有用户线程都被抢占，当一个用户线程被阻塞时，它会**

**阻塞下面的轻量级进程，如果进程只有一个轻量级进程，则它的所有用户线程都会被阻塞。**

明确了这些概念后，来讲述Linux的线程和clone系统调用。

在许多实现了MT的操作系统中（如：Solaris，Digital Unix等）， 线程和进程通过两种数据结构来

抽象表示： 进程表项和线程表项，一个进程表项可以指向若干个线程表项， 调度器在进程的时间片内再

调度线程。 但是在Linux中没有做这种区分， 而是统一使用task\_struct来管理所有进程/线程，只是

线程与线程之间的资源是共享的，这些资源可是是前面提到过的：虚存、文件系统、文件I/O以及信号处

理函数甚至PID中的几种。

也就是说Linux中，每个线程都有一个task\_struct，所以线程和进程可以使用同一调度器调度。其实

Linux核心中，轻量级进程和进程没有质上的差别，因为Linux中进程的概念已经被抽象成了计算状态加资

源的集合，这些资源在进程间可以共享。如果一个task独占所有的资源，则是一个HWP，如果一个task和

其它task共享部分资源，则是LWP。

clone系统调用就是一个创建轻量级进程的系统调用：

int clone(int (\*fn)(void \* arg), void \*stack, int flags, void \*arg);

其中fn是轻量级进程所执行的过程，stack是轻量级进程所使用的堆栈，flags可以是前面提到的

CLONE\_VM, CLONE\_FS, CLONE\_FILES, CLONE\_SIGHAND,CLONE\_PID的组合。Clone和fork，vfork在实现时

都是调用核心函数do\_fork。

do\_fork(unsigned long clone\_flag, unsigned long usp, structpt\_regs)；

和fork、vfork不同的是，fork时clone\_flag = SIGCHLD；

vfork时clone\_flag = CLONE\_VM | CLONE\_VFORK | SIGCHLD；

而在clone中，clone\_flag由用户给出。

下面给出一个使用clone的例子。

Void \* func(int arg)

{

    .. . . . .

}

int main()

{

    int clone\_flag, arg;

    .. . . . .

    clone\_flag = CLONE\_VM | CLONE\_SIGHAND | CLONE\_FS |

    CLONE\_FILES;

    stack = (char \*)malloc(STACK\_FRAME);

    stack += STACK\_FRAME;

    retval = clone((void \*)func, stack, clone\_flag, arg);

    .. . . . .

}

**看起来clone的用法和pthread\_create有些相似，两者的最根本的差别在于clone是创建一个LWP，对**

**核心是可见的，由核心调度，而pthread\_create通常只是创建一个用户线程，对核心是不可见的，由线程**

**库调度。**

linux的pthread\_create最终调用clone,pthread\_create调用clone,并把开辟一个stack作为参数

thread 建立, 同步,销毁等由线程库负责,

**pthread多线程编程整理**

**1 Introduction**

不用介绍了吧…

**2 Thread Concepts**

1.     Thread由下面部分组成：

a.     Thread ID

b.     Stack

c.     Policy

d.     Signal mask

e.     Errno

f.      Thread-Specific Data

**3 Thread Identification**

1.     pthread\_t用于表示Thread ID，具体内容根据实现的不同而不同，有可能是一个Structure，因此不能将其看作为整数

2.     pthread\_equal函数用于比较两个pthread\_t是否相等

|  |
| --- |
| ＃i nclude <pthread.h>    int pthread\_equal(pthread\_t tid1, pthread\_t tid2) |

3.     pthread\_self函数用于获得本线程的thread id

|  |
| --- |
| ＃i nclude <pthread.h>    pthread \_t pthread\_self(void); |

**4 Thread Creation**

1.     创建线程可以调用pthread\_create函数：

|  |
| --- |
| ＃i nclude <pthread.h>    int pthread\_create(         pthread\_t \*restrict tidp,         const pthread\_attr\_t \*restrict attr,         void \*(\*start\_rtn)(void \*), void \*restrict arg); |

a.     pthread\_t \*restrict tidp：返回最后创建出来的Thread的Thread ID

b.     const pthread\_attr\_t \*restrict attr：指定线程的Attributes，后面会讲道，现在可以用NULL

c.     void \*(\*start\_rtn)(void \*)：指定线程函数指针，该函数返回一个void \*，参数也为void\*

d.     void \*restrict arg：传入给线程函数的参数

e.     返回错误值。

2.     pthread函数在出错的时候不会设置errno，而是直接返回错误值

3.     在Linux 系统下面，在老的内核中，由于Thread也被看作是一种特殊，可共享地址空间和资源的Process，因此在同一个Process中创建的不同 Thread具有不同的Process ID（调用getpid获得）。而在新的2.6内核之中，Linux采用了NPTL(Native POSIX Thread Library)线程模型（可以参考<http://en.wikipedia.org/wiki/Native_POSIX_Thread_Library>和<http://www-128.ibm.com/developerworks/linux/library/l-threading.html?ca=dgr-lnxw07LinuxThreadsAndNPTL>），在该线程模型下同一进程下不同线程调用getpid返回同一个PID。

4.     不能对创建的新线程和当前创建者线程的运行顺序作出任何假设

**5 Thread Termination**

1.     exit, \_Exit, \_exit用于中止当前进程，而非线程

2.     中止线程可以有三种方式：

a.     在线程函数中return

b.     被同一进程中的另外的线程Cancel掉

c.     线程调用pthread\_exit函数

3.     pthread\_exit和pthread\_join函数的用法：

a.     线程A调用pthread\_join(B, &rval\_ptr)，被Block，进入Detached状态（如果已经进入Detached状态，则pthread\_join函数返回EINVAL）。如果对B的结束代码不感兴趣，rval\_ptr可以传NULL。

b.     线程B调用pthread\_exit(rval\_ptr)，退出线程B，结束代码为rval\_ptr。注意rval\_ptr指向的内存的生命周期，不应该指向B的Stack中的数据。

c.     线程A恢复运行，pthread\_join函数调用结束，线程B的结束代码被保存到rval\_ptr参数中去。如果线程B被Cancel，那么rval\_ptr的值就是PTHREAD\_CANCELLED。

两个函数原型如下：

|  |
| --- |
| ＃i nclude <pthread.h>    void pthread\_exit(void \*rval\_ptr);    int pthread\_join(pthread\_t thread, void \*\*rval\_ptr); |

4.     一个Thread可以要求另外一个Thread被Cancel，通过调用pthread\_cancel函数：

|  |
| --- |
| ＃i nclude <pthread.h>    void pthread\_cancel(pthread\_t tid) |

该函数会使指定线程如同调用了pthread\_exit(PTHREAD\_CANCELLED)。不过，指定线程可以选择忽略或者进行自己的处理，在后面会讲到。此外，该函数不会导致Block，只是发送Cancel这个请求。

5.     线程可以安排在它退出的时候，某些函数自动被调用，类似atexit()函数。需要调用如下函数：

|  |
| --- |
| ＃i nclude <pthread.h>    void pthread\_cleanup\_push(void (\*rtn)(void \*), void \*arg);  void pthread\_cleanup\_pop(int execute); |

这两个函数维护一个函数指针的Stack，可以把函数指针和函数参数值push/pop。执行的顺序则是从栈顶到栈底，也就是和push的顺序相反。

在下面情况下pthread\_cleanup\_push所指定的thread cleanup handlers会被调用：

a.     调用pthread\_exit

b.     相应cancel请求

c.     以非0参数调用pthread\_cleanup\_pop()。（如果以0调用pthread\_cleanup\_pop()，那么handler不会被调用

有一个比较怪异的要求是，由于这两个函数可能由宏的方式来实现，因此这两个函数的调用必须得是在同一个Scope之中，并且配对，因为在pthread\_cleanup\_push的实现中可能有一个{，而 pthread\_cleanup\_pop可能有一个}。因此，一般情况下，这两个函数是用于处理意外情况用的，举例如下：

|  |
| --- |
| void \*thread\_func(void \*arg)  {      pthread\_cleanup\_push(cleanup, “handler”)        // do something        Pthread\_cleanup\_pop(0);      return((void \*)0)；  } |

6.     进程函数和线程函数的相关性：

|  |  |  |
| --- | --- | --- |
| **Process Primitive** | **Thread Primitive** | **Description** |
| fork | pthread\_create | 创建新的控制流 |
| exit | pthread\_exit | 退出已有的控制流 |
| waitpid | pthread\_join | 等待控制流并获得结束代码 |
| atexit | pthread\_cleanup\_push | 注册在控制流退出时候被调用的函数 |
| getpid | pthread\_self | 获得控制流的id |
| abort | pthread\_cancel | 请求非正常退出 |

7.     缺省情况下，一个线程A的结束状态被保存下来直到pthread\_join为该线程被调用过，也就是说即使线程A已经结束，只要没有线程B调用 pthread\_join(A)，A的退出状态则一直被保存。而当线程处于Detached状态之时，党线程退出的时候，其资源可以立刻被回收，那么这个退出状态也丢失了。在这个状态下，无法为该线程调用pthread\_join函数。我们可以通过调用pthread\_detach函数来使指定线程进入 Detach状态：

|  |
| --- |
| ＃i nclude <pthread.h>    int pthread\_detach(pthread\_t tid); |

通过修改调用pthread\_create函数的attr参数，我们可以指定一个线程在创建之后立刻就进入Detached状态

**6 Thread Synchronization**

1.     互斥量：Mutex

a.     用于互斥访问

b.     类型：pthread\_mutex\_t，必须被初始化为PTHREAD\_MUTEX\_INITIALIZER（用于静态分配的mutex，等价于 pthread\_mutex\_init(…, NULL)）或者调用pthread\_mutex\_init。Mutex也应该用pthread\_mutex\_destroy来销毁。这两个函数原型如下：（attr的具体含义下一章讨论）

|  |
| --- |
| ＃i nclude <pthread.h>    int pthread\_mutex\_init(         pthread\_mutex\_t \*restrict mutex,         const pthread\_mutexattr\_t \*restrict attr)    int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex); |

c.     pthread\_mutex\_lock 用于Lock Mutex，如果Mutex已经被Lock，该函数调用会Block直到Mutex被Unlock，然后该函数会Lock Mutex并返回。pthread\_mutex\_trylock类似，只是当Mutex被Lock的时候不会Block，而是返回一个错误值EBUSY。 pthread\_mutex\_unlock则是unlock一个mutex。这三个函数原型如下：

|  |
| --- |
| ＃i nclude <pthread.h>    int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);    int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);    int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex); |

2.     读写锁：Reader-Writer Locks

a.     多个线程可以同时获得读锁(Reader-Writer lock in read mode)，但是只有一个线程能够获得写锁(Reader-writer lock in write mode)

b.     读写锁有三种状态

                                          i.    一个或者多个线程获得读锁，其他线程无法获得写锁

                                         ii.    一个线程获得写锁，其他线程无法获得读锁

                                        iii.    没有线程获得此读写锁

c.     类型为pthread\_rwlock\_t

d.     创建和关闭方法如下：

|  |
| --- |
| ＃i nclude <pthread.h>    int pthread\_rwlock\_init(         pthread\_rwlock\_t \*restrict rwlock,         const pthread\_rwlockattr\_t \*restrict attr)    int pthread\_rwlock\_destroy(pthread\_rwlock\_t \*rwlock); |

e.     获得读写锁的方法如下：

|  |
| --- |
| ＃i nclude <pthread.h>    int pthread\_rwlock\_rdlock(pthread\_rwlock\_t \*rwlock);    int pthread\_rwlock\_wrlock(pthread\_rwlock\_t \*rwlock);    int pthread\_rwlock\_unlock(pthread\_rwlock\_t \*rwlock);    int pthread\_rwlock\_tryrdlock(pthread\_rwlock\_t \*rwlock);    int pthread\_rwlock\_trywrlock(pthread\_rwlock\_t \*rwlock); |

pthread\_rwlock\_rdlock：获得读锁

pthread\_rwlock\_wrlock：获得写锁

pthread\_rwlock\_unlock：释放锁，不管是读锁还是写锁都是调用此函数

注意具体实现可能对同时获得读锁的线程个数有限制，所以在调用 pthread\_rwlock\_rdlock的时候需要检查错误值，而另外两个pthread\_rwlock\_wrlock和 pthread\_rwlock\_unlock则一般不用检查，如果我们代码写的正确的话。

3.     Conditional Variable：条件

a.     条件必须被Mutex保护起来

b.     类型为：pthread\_cond\_t，必须被初始化为PTHREAD\_COND\_INITIALIZER（用于静态分配的条件，等价于pthread\_cond\_init(…, NULL)）或者调用pthread\_cond\_init

|  |
| --- |
| ＃i nclude <pthread.h>    int pthread\_cond\_init(         pthread\_cond\_t \*restrict cond,         const pthread\_condxattr\_t \*restrict attr)    int pthread\_cond\_destroy(pthread\_cond\_t \*cond); |

c.     pthread\_cond\_wait 函数用于等待条件发生（=true）。pthread\_cond\_timedwait类似，只是当等待超时的时候返回一个错误值ETIMEDOUT。超时的时间用timespec结构指定。此外，两个函数都需要传入一个Mutex用于保护条件

|  |
| --- |
| ＃i nclude <pthread.h>    int pthread\_cond\_wait(         pthread\_cond\_t \*restrict cond,         pthread\_mutex\_t \*restrict mutex);    int pthread\_cond\_timedwait(         pthread\_cond\_t \*restrict cond,         pthread\_mutex\_t \*restrict mutex,         const struct timespec \*restrict timeout); |

d.     timespec结构定义如下：

|  |
| --- |
| struct timespec {         time\_t tv\_sec;       /\* seconds \*/         long   tv\_nsec;      /\* nanoseconds \*/  }; |

注意timespec的时间是绝对时间而非相对时间，因此需要先调用gettimeofday函数获得当前时间，再转换成timespec结构，加上偏移量。

e.     有两个函数用于通知线程条件被满足（=true）：

|  |
| --- |
| ＃i nclude <pthread.h>    int pthread\_cond\_signal(pthread\_cond\_t \*cond);    int pthread\_cond\_broadcast(pthread\_cond\_t \*cond); |

两者的区别是前者会唤醒单个线程，而后者会唤醒多个线程。

在传统的Unix模型中，当一个进程需要由另一个实体执行某件事时，该进程派生（fork）一个子进程，让子进程去进行处理。Unix下的大多数网络服务器程序都是这么编写的，即父进程接受连接，派生子进程，子进程处理与客户的交互。

虽然这种模型很多年来使用得很好，但是fork时有一些问题：

1. fork是昂贵的。内存映像要从父进程拷贝到子进程，所有描述字要在子进程中复制等等。目前有的Unix实现使用一种叫做写时拷贝（copy－on－write）的技术，可避免父进程数据空间向子进程的拷贝。尽管有这种优化技术，fork仍然是昂贵的。

2. fork子进程后，需要用进程间通信（IPC）在父子进程之间传递信息。Fork之前的信息容易传递，因为子进程从一开始就有父进程数据空间及所有描述字的拷贝。但是从子进程返回信息给父进程需要做更多的工作。

线程有助于解决这两个问题。线程有时被称为轻权进程（lightweight process），因为线程比进程“轻权”，一般来说，创建一个线程要比创建一个进程快10～100倍。

一个进程中的所有线程共享相同的全局内存，这使得线程很容易共享信息，但是这种简易性也带来了同步问题。

一个进程中的所有线程不仅共享全局变量，而且共享：进程指令、大多数数据、打开的文件（如描述字）、信号处理程序和信号处置、当前工作目录、用户ID和组ID。但是每个线程有自己的线程ID、寄存器集合（包括程序计数器和栈指针）、栈（用于存放局部变量和返回地址）、error、信号掩码、优先级。在**[Linux](http://lib.csdn.net/base/linux" \o "Linux知识库)**中线程编程符合Posix.1标准，称为Pthreads。所有的pthread函数都以pthread\_开头。以下先讲述5个基本线程函数，在调用它们前均要包括pthread.h头文件。然后再给出用它们编写的一个TCP客户/服务器程序例子。

第一个函数：

|  |
| --- |
| int pthread\_create (pthread\_t ＊tid,const pthread\_attr\_t ＊attr,void ＊ (＊func)(void ＊),void ＊arg)； |

一个进程中的每个线程都由一个线程ID（thread ID）标识，其数据类型是pthread\_t（常常是unsigned int）。如果新的线程创建成功，其ID将通过tid指针返回。

每个线程都有很多属性：优先级、起始栈大小、是否应该是一个守护线程等等，当创建线程时，我们可通过初始化一个pthread\_attr\_t变量说明这些属性以覆盖缺省值。我们通常使用缺省值，在这种情况下，我们将attr参数说明为空指针。

最后，当创建一个线程时，我们要说明一个它将执行的函数。线程以调用该函数开始，然后或者显式地终止（调用pthread\_exit）或者隐式地终止（让该函数返回）。函数的地址由func参数指定，该函数的调用参数是一个指针arg，如果我们需要多个调用参数，我们必须将它们打包成一个结构，然后将其地址当作唯一的参数传递给起始函数。

在func和arg的声明中，func函数取一个通用指针（void ＊）参数，并返回一个通用指针（void ＊），这就使得我们可以传递一个指针（指向任何我们想要指向的东西）给线程，由线程返回一个指针（同样指向任何我们想要指向的东西）。调用成功，返回0，出错时返回正Exxx值。Pthread函数不设置errno。

第二个函数：

|  |
| --- |
| int pthread\_join(pthread\_t tid,void ＊＊status); |

该函数等待一个线程终止。把线程和进程相比，pthread\_creat类似于fork，而 pthread\_join类似于waitpid。我们必须要等待线程的tid，很可惜，我们没有办法等待任意一个线程结束。如果status指针非空，线程的返回值（一个指向某个对象的指针）将存放在status指向的位置。

第三个函数：

|  |
| --- |
| pthread\_t pthread\_self(void); |

线程都有一个ID以在给定的进程内标识自己。线程ID由pthread\_creat返回，我们可以pthread\_self取得自己的线程ID。

第四个函数：

|  |
| --- |
| int **pthread\_detach**(pthread\_t tid); |

线程或者是可汇合的（joinable）或者是脱离的（detached）。当可汇合的线程终止时，其线程ID和退出状态将保留，直到另外一个线程调用pthread\_join。脱离的线程则像守护进程：当它终止时，所有的资源都释放，我们不能等待它终止。如果一个线程需要知道另一个线程什么时候终止，最好保留第二个线程的可汇合性。**Pthread\_detach**函数将指定的线程变为脱离的。该函数通常被想脱离自己的线程调用，如：**pthread\_detach** (pthread\_self ( ));

第五个函数：

|  |
| --- |
| void pthread\_exit(void ＊status); |

该函数终止线程。如果线程未脱离，其线程ID和退出状态将一直保留到调用进程中的某个其他线程调用pthread\_join函数。指针status不能指向局部于调用线程的对象，因为线程终止时这些对象也消失。有两种其他方法可使线程终止：

1. 启动线程的函数（pthread\_creat的第3个参数）返回。既然该函数必须说明为返回一个void指针，该返回值便是线程的终止状态。

2. 如果进程的main函数返回或者任何线程调用了exit，进程将终止，线程将随之终止。

一.pthread\_create()之前的属性设置  
1．线程属性设置  
我们用pthread\_create函数创建一个线程，在这个线程中，我们使用默认参数，即将该函数的第二个参数设为NULL。的确，对大多数程序来说，使用默认属性就够了，但我们还是有必要来了解一下线程的有关属性。  
属性结构为pthread\_attr\_t，它同样在头文件pthread.h中定义，属性值不能直接设置，须使用相关函数进行操作，初始化的函数为**pthread\_attr\_init**，这个函数必须在pthread\_create函数之前调用。属性对象主要包括是否绑定、是否分离、  
堆栈地址、堆栈大小、优先级。默认的属性为非绑定、非分离、缺省的堆栈、与父进程同样级别的优先级。  
  
2．绑定  
关于线程的绑定，牵涉到另外一个概念：轻进程（LWP：Light Weight Process）。轻进程可以理解为内核线程，它位于用户层和系统层之间。系统对线程资源的分配、对线程的控制是通过轻进程来实现的，一个轻进程可以控制一个或多个线程。默认状况下，启动多少轻进程、哪些轻进程来控制哪些线程是由系统来控制的，这种状况即称为非绑定的。绑定状况下，则顾名思义，即某个线程固定的"绑"在一个轻进程之上。被绑定的线程具有较高的响应速度，这是因为CPU时间片的调度是面向轻进程的，绑定的线程可以保证在需要的时候它总有一个轻进程可用。通过设置被绑定的轻进程的优先级和调度级可以使得绑定的线程满足诸如实时反应之类的要求。  
　　设置线程绑定状态的函数为 pthread\_attr\_setscope，它有两个参数，第一个是指向属性结构的指针，第二个是绑定类型，它有两个取值： PTHREAD\_SCOPE\_SYSTEM（绑定的）和PTHREAD\_SCOPE\_PROCESS（非绑定的）。下面的代码即创建了一个绑定的线程。  
＃i nclude <pthread.h>  
pthread\_attr\_t attr;  
pthread\_t tid;  
/\*初始化属性值，均设为默认值\*/  
**pthread\_attr\_init**(&attr);  
pthread\_attr\_setscope(&attr, PTHREAD\_SCOPE\_SYSTEM);  
pthread\_create(&tid, &attr, (void \*) my\_function, NULL);  
  
3．线程分离状态                                                                                                                                         线程的分离状态决定一个线程以什么样的方式来终止自己。非分离的线程终止时，其线程ID和退出状态将保留，直到另外一个线程调用 pthread\_join.分离的线程在当它终止时，所有的资源将释放，我们不能等待它终止。                                                                                         设置线程分离状态的函数为 pthread\_attr\_setdetachstate（pthread\_attr\_t \*attr, int detachstate）。第二个参数可选为PTHREAD\_CREATE\_DETACHED（分离线程）和 PTHREAD \_CREATE\_JOINABLE（非分离线程）。这里要注意的一点是，如果设置一个线程为分离线程，而这个线程运行又非常快，它很可能在 pthread\_create函数返回之前就终止了，它终止以后就可能将线程号和系统资源移交给其他的线程使用，这样调用pthread\_create的线程就得到了错误的线程号。要避免这种情况可以采取一定的同步措施，最简单的方法之一是可以在被创建的线程里调用 pthread\_cond\_timewait函数，让这个线程等待一会儿，留出足够的时间让函数pthread\_create返回。设置一段等待时间，是在多线程编程里常用的方法。  
4．优先级                                                                                                                                                 它存放在结构sched\_param中。用函数pthread\_attr\_getschedparam和函数 pthread\_attr\_setschedparam进行存放，一般说来，我们总是先取优先级，对取得的值修改后再存放回去。下面即是一段简单的例子。  
  
＃i nclude <pthread.h>  
＃i nclude <sched.h>  
pthread\_attr\_t attr; pthread\_t tid;  
sched\_param param;  
int newprio=20;  
/\*初始化属性\*/  
**pthread\_attr\_init**(&attr);  
/\*设置优先级\*/  
pthread\_attr\_getschedparam(&attr, &param);    
param.sched\_priority=newprio;  
pthread\_attr\_setschedparam(&attr, &param);  
pthread\_create(&tid, &attr, (void \*)myfunction, myarg);  
  
二．线程数据处理                                                                                                                                 和进程相比，线程的最大优点之一是数据的共享性，各个进程共享父进程处沿袭的数据段，可以方便的获得、修改数据。但这也给多线程编程带来了许多问题。我们必须当心有多个不同的进程访问相同的变量。许多函数是不可重入的，即同时不能运行一个函数的多个拷贝（除非使用不同的数据段）。在函数中声明的静态变量常常带来问题，函数的返回值也会有问题。因为如果返回的是函数内部静态声明的空间的地址，则在一个线程调用该函数得到地址后使用该地址指向的数据时，别的线程可能调用此函数并修改了这一段数据。在进程中共享的变量必须用关键字volatile来定义，这是为了防止编译器在优化时（如gcc中使用-OX参数）改变它们的使用方式。为了保护变量，我们必须使用信号量、互斥等方法来保证我们对变量的正确使用。  
1．线程数据                                                                                                                                                 在单线程的程序里，有两种基本的数据：全局变量和局部变量。但在多线程程序里，还有第三种数据类型：线程数据（TSD: Thread-Specific Data）。它和全局变量很象，在线程内部，各个函数可以象使用全局变量一样调用它，但它对线程外部的其它线程是不可见的。例如我们常见的变量 errno，它返回标准的出错信息。它显然不能是一个局部变量，几乎每个函数都应该可以调用它；但它又不能是一个全局变量，否则在 A线程里输出的很可能是B线程的出错信息。要实现诸如此类的变量，我们就必须使用线程数据。我们为每个线程数据创建一个键，它和这个键相关联，在各个线程里，都使用这个键来指代线程数据，但在不同的线程里，这个键代表的数据是不同的，在同一个线程里，它代表同样的数据内容。  
　　和线程数据相关的函数主要有4个：创建一个键；为一个键指定线程数据；从一个键读取线程数据；删除键。  
　　创建键的函数原型为：  
　　int pthread\_key\_create \_\_P ((pthread\_key\_t \*\_\_key,void (\*\_\_destr\_function) (void \*)));                                                                                                                                                 第一个参数为指向一个键值的指针，第二个参数指明了一个destructor函数，如果这个参数不为空，那么当每个线程结束时，系统将调用这个函数来释放绑定在这个键上的内存块。这个函数常和函数pthread\_once ((pthread\_once\_t\*once\_control, void (\*initroutine) (void)))一起使用，为了让这个键只被创建一次。函数pthread\_once声明一个初始化函数，第一次调用pthread\_once时它执行这个函数，以后的调用将被它忽略。  
int pthread\_key\_delete(pthread\_key\_t \*key);  
该函数用于删除一个由pthread\_key\_create 函数调用创建的键。调用成功返回值为0，否则返回错误代码。  
在下面的例子中，我们创建一个键，并将它和某个数据相关联。我们要定义一个函数 createWindow，这个函数定义一个图形窗口（数据类型为Fl\_Window \*，这是图形界面开发工具FLTK中的数据类型）。由于各个线程都会调用这个函数，所以我们使用线程数据。  
/\* 声明一个键\*/  
pthread\_key\_t myWinKey;  
/\* 函数 createWindow \*/  
void createWindow ( void ) {  
Fl\_Window \* win;  
static pthread\_once\_t once= PTHREAD\_ONCE\_INIT;  
/\* 调用函数createMyKey，创建键\*/  
pthread\_once ( & once, createMyKey) ;  
/\*win指向一个新建立的窗口\*/  
win=new Fl\_Window( 0, 0, 100, 100, "MyWindow");  
/\* 对此窗口作一些可能的设置工作，如大小、位置、名称等\*/  
setWindow(win);  
/\* 将窗口指针值绑定在键myWinKey上\*/  
pthread\_setpecific ( myWinKey, win);  
}  
/\* 函数 createMyKey，创建一个键，并指定了destructor \*/  
void createMyKey ( void ) {  
pthread\_keycreate(&myWinKey, freeWinKey);  
}  
/\* 函数 freeWinKey，释放空间\*/  
void freeWinKey ( Fl\_Window \* win){  
delete win;  
}  
这样，在不同的线程中调用函数createMyWin，都可以得到在线程内部均可见的窗口变量，这个变量通过函数 pthread\_getspecific得到。在上面的例子中，我们已经使用了函数pthread\_setspecific来将线程数据和一个键绑定在一起。这两个函数的原型如下：  
　　  
int pthread\_setspecific \_\_P ((pthread\_key\_t \_\_key,\_\_const void \*\_\_pointer)); 该函数设置一个线程专有数据的值，赋给由pthread\_key\_create 创建的键，调用成功返回值为0，否则返回错误代码。  
void \*pthread\_getspecific \_\_P ((pthread\_key\_t \_\_key));                                                  该函数获得绑定到指定键上的值。调用成功，返回给定参数key 所对应的数据。如果没有数据连接到该键，则返回NULL。  
  
　　这两个函数的参数意义和使用方法是显而易见的。要注意的是，用pthread\_setspecific为一个键指定新的线程数据时，必须自己释放原有的线程数据以回收空间。这个过程函数pthread\_key\_delete用来删除一个键，这个键占用的内存将被释放，但同样要注意的是，它只释放键占用的内存，并不释放该键关联的线程数据所占用的内存资源，而且它也不会触发函数pthread\_key\_create中定义的destructor函数。线程数据的释放必须在释放键之前完成。  
2．互斥锁                                                                                                                                                 假设各个现成向同一个文件顺序写入数据，最后得到的结果是不可想象的。所以用互斥锁来保证一段时间内只有一个线程在执行一段代码。  
  
  
  
使用int pthread\_mutex\_lock锁住互斥锁，使用int pthread\_mutex\_unlock解琐。  
如果我们试图为一个已被其他线程锁住的互斥锁加锁，程序便会阻塞直到该互斥对象解锁。  
如果在共享内存中分配一个互斥锁，我们必须在运行时调用ptgread\_mutex\_init函数尽心初始化。  
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);  
}  
}  
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。  
需要注意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。  
3．条件变量  
互斥锁一个明显的缺点是它只有两种状态：锁定和非锁定。而条件变量通过允许线程阻塞和等待另一个线程发送信号的方法弥补了互斥锁的不足，它常和互斥锁一起使用。使用时，条件变量被用来阻塞一个线程，当条件不满足时，线程往往解开相应的互斥锁并等待条件发生变化。一旦其它的某个线程改变了条件变量，它将通知相应的条件变量唤醒一个或多个正被此条件变量阻塞的线程。这些线程将重新锁定互斥锁并重新[**测试**](http://lib.csdn.net/base/softwaretest)条件是否满足。一般说来，条件变量被用来进行线承间的同步。  
条件变量的结构为pthread\_cond\_t，函数pthread\_cond\_init（）被用来初始化一个条件变量。它的原型为：  
  
int pthread\_cond\_init \_\_P ((pthread\_cond\_t \*\_\_cond,\_\_const pthread\_condattr\_t \*\_\_cond\_attr));  
           
其中cond是一个指向结构pthread\_cond\_t的指针，cond\_attr是一个指向结构pthread\_condattr\_t的指针。结构 pthread\_condattr\_t是条件变量的属性结构，和互斥锁一样我们可以用它来设置条件变量是进程内可用还是进程间可用，默认值是 PTHREAD\_ PROCESS\_PRIVATE，即此条件变量被同一进程内的各个线程使用。注意初始化条件变量只有未被使用时才能重新初始化或被释放。  
在pthread中，条件变量是一个pthread\_cond\_t类型的变量，条件变量使用下面两个函数：  
  
pthread\_cond\_wait 函数用于阻塞，线程可以被函数pthread\_cond\_signal和函数    pthread\_cond\_broadcast唤醒，但是要注意的是，条件变量只是起阻塞和唤醒线程的作用，具体的判断条件还需用户给出，例如一个变量是否为0等等，这一点我们从后面的例子中可以看到。线程被唤醒后，它将重新检查判断条件是否满足，如果还不满足，一般说来线程应该仍阻塞在这里，被等待被下一次唤醒。这个过程一般用while语句实现。  
另一个用来阻塞线程的函数是pthread\_cond\_timedwait（）它比函数pthread\_cond\_wait（）多了一个时间参数，经历abstime段时间后，即使条件变量不满足，阻塞也被解除。  
函数pthread\_cond\_signal（）用来释放被阻塞在条件变量cond上的一个线程。  
函数pthread\_cond\_broadcast（pthread\_cond\_t \*cond）用来唤醒所有被阻塞在条件变量cond上的线程。这些线程被唤醒后将再次竞争相应的互斥锁，所以必须小心使用这个函数。  
下面是使用函数pthread\_cond\_wait（）和函数pthread\_cond\_signal（）的一个简单的例子:  
pthread\_mutex\_t count\_lock;  
pthread\_cond\_t count\_nonzero;  
unsigned count;  
decrement\_count　() {  
pthread\_mutex\_lock (&count\_lock);  
while(count==0)  
pthread\_cond\_wait( &count\_nonzero, &count\_lock);  
count=count -1;  
pthread\_mutex\_unlock (&count\_lock);  
}  
  
increment\_count(){  
pthread\_mutex\_lock(&count\_lock);  
if(count==0)  
pthread\_cond\_signal(&count\_nonzero);  
count=count+1;  
pthread\_mutex\_unlock(&count\_lock);  
}  
count 值为0时， decrement函数在pthread\_cond\_wait处被阻塞，并打开互斥锁count\_lock。此时，当调用到函数 increment\_count时，pthread\_cond\_signal（）函数改变条件变量，告知decrement\_count（）停止阻塞。

=================================================================================

pthread\_mutex\_lock

**函数名**

pthread\_mutex\_lock, pthread\_mutex\_trylock, pthread\_mutex\_unlock - lock and unlock a mutex

**SYNOPSIS**

概要

#include <[pthread.h](http://www.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html)>

int 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。

The *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]

=================================================================================

pthread\_join函数及linux线程

pthread\_join使一个线程等待另一个线程结束。

代码中如果没有pthread\_join主线程会很快结束从而使整个进程结束，从而使创建的线程没有机会开始执行就结束了。加入pthread\_join后，主线程会一直等待直到等待的线程结束自己才结束，使创建的线程有机会执行。

所有线程都有一个线程号，也就是Thread ID。其类型为pthread\_t。通过调用pthread\_self()函数可以获得自身的线程号。

下面说一下如何创建一个线程。

通过创建线程，线程将会执行一个线程函数，该线程格式必须按照下面来声明：

       void \* Thread\_Function(void \*)

创建线程的函数如下：

       int pthread\_create(pthread\_t \*restrict thread,

              const pthread\_attr\_t \*restrict attr,

              void \*(\*start\_routine)(void\*), void \*restrict arg);

下面说明一下各个参数的含义：

thread：所创建的线程号。

attr：所创建的线程属性，这个将在后面详细说明。

start\_routine：即将运行的线程函数。

art：传递给线程函数的参数。

下面是一个简单的创建线程例子：

|  |
| --- |
| #include <pthread.h>  #include <stdio.h>  /\* Prints x’s to stderr. The parameter is unused. Does not return. \*/  void\* print\_xs (void\* unused)  {  while (1)  fputc (‘x’, stderr);  return NULL;  }  /\* The main program. \*/  int main ()  {  pthread\_t thread\_id;  /\* Create a new thread. The new thread will run the print\_xs  function. \*/  pthread\_create (&thread\_id, NULL, &print\_xs, NULL);  /\* Print o’s continuously to stderr. \*/  while (1)  fputc (‘o’, stderr);  return 0;  } |

在编译的时候需要注意，由于线程创建函数在libpthread.so库中，所以在编译命令中需要将该库导入。命令如下：

gcc –o createthread –lpthread createthread.c

如果想传递参数给线程函数，可以通过其参数arg，其类型是void \*。如果你需要传递多个参数的话，可以考虑将这些参数组成一个结构体来传递。另外，由于类型是void \*，所以你的参数不可以被提前释放掉。

下面一个问题和前面创建进程类似，不过带来的问题回避进程要严重得多。如果你的主线程，也就是main函数执行的那个线程，在你其他县城推出之前就已经退出，那么带来的bug则不可估量。通过pthread\_join函数会让主线程阻塞，直到所有线程都已经退出。

int pthread\_join(pthread\_t thread, void \*\*value\_ptr);

thread：等待退出线程的线程号。

value\_ptr：退出线程的返回值。

下面一个例子结合上面的内容：

|  |
| --- |
| int main ()  {  pthread\_t thread1\_id;  pthread\_t thread2\_id;  struct char\_print\_parms thread1\_args;  struct char\_print\_parms thread2\_args;  /\* Create a new thread to print 30,000 x’s. \*/  thread1\_args.character = ’x’;  thread1\_args.count = 30000;  pthread\_create (&thread1\_id, NULL, &char\_print, &thread1\_args);  /\* Create a new thread to print 20,000 o’s. \*/  thread2\_args.character = ’o’;  thread2\_args.count = 20000;  pthread\_create (&thread2\_id, NULL, &char\_print, &thread2\_args);  /\* Make sure the first thread has finished. \*/  pthread\_join (thread1\_id, NULL);  /\* Make sure the second thread has finished. \*/  pthread\_join (thread2\_id, NULL);  /\* Now we can safely return. \*/  return 0;  } |

下面说一下前面提到的线程属性。

在我们前面提到，可以通过pthread\_join()函数来使主线程阻塞等待其他线程退出，这样主线程可以清理其他线程的环境。但是还有一些线程，更喜欢自己来清理退出的状态，他们也不愿意主线程调用pthread\_join来等待他们。我们将这一类线程的属性称为detached。如果我们在调用pthread\_create()函数的时候将属性设置为NULL，则表明我们希望所创建的线程采用默认的属性，也就是jionable。如果需要将属性设置为detached，则参考下面的例子：

|  |
| --- |
| #include <stdio.h>  #include <pthread.h>  void \* start\_run(void \* arg)  {          //do some work  }  int main()  {          pthread\_t thread\_id;          pthread\_attr\_t attr;          pthread\_attr\_init(&attr);          pthread\_attr\_setdetachstate(&attr,PTHREAD\_CREATE\_DETACHED);          pthread\_create(&thread\_id,&attr,start\_run,NULL);          pthread\_attr\_destroy(&attr);          sleep(5);          exit(0);  } |

在线程设置为joinable后，可以调用pthread\_detach()使之成为detached。但是相反的操作则不可以。还有，如果线程已经调用pthread\_join()后，则再调用pthread\_detach()则不会有任何效果。

线程可以通过自身执行结束来结束，也可以通过调用pthread\_exit()来结束线程的执行。另外，线程甲可以被线程乙被动结束。这个通过调用pthread\_cancel()来达到目的。

int pthread\_cancel(pthread\_t thread);

       函数调用成功返回0。

当然，线程也不是被动的被别人结束。它可以通过设置自身的属性来决定如何结束。

线程的被动结束分为两种，一种是异步终结，另外一种是同步终结。异步终结就是当其他线程调用 pthread\_cancel的时候，线程就立刻被结束。而同步终结则不会立刻终结，它会继续运行，直到到达下一个结束点（cancellation point）。当一个线程被按照默认的创建方式创建，那么它的属性是同步终结。

通过调用pthread\_setcanceltype()来设置终结状态。

int pthread\_setcanceltype(int type, int \*oldtype);

state：要设置的状态，可以为PTHREAD\_CANCEL\_DEFERRED或者为PTHREAD\_CANCEL\_ASYNCHRONOUS。

那么前面提到的结束点又是如何设置了？最常用的创建终结点就是调用pthread\_testcancel()的地方。该函数除了检查同步终结时的状态，其他什么也不做。

上面一个函数是用来设置终结状态的。还可以通过下面的函数来设置终结类型，即该线程可不可以被终结：

int pthread\_setcancelstate(int state, int \*oldstate);

       state：终结状态，可以为PTHREAD\_CANCEL\_DISABLE或者PTHREAD\_CANCEL\_ENABLE。具体什么含义大家可以通过单词意思即可明白。

最后说一下线程的本质。其实在Linux中，新建的线程并不是在原先的进程中，而是系统通过一个系统调用clone()。该系统copy了一个和原先进程完全一样的进程，并在这个进程中执行线程函数。不过这个copy过程和fork不一样。 copy后的进程和原先的进程共享了所有的变量，运行环境。这样，原先进程中的变量变动在copy后的进程中便能体现出来。

* [点赞 3](javascript:;)
* [收藏](javascript:;)
* [分享](javascript:;)

[[](https://blog.csdn.net/rock_joker)](https://blog.csdn.net/rock_joker)

[rock\_joker](https://blog.csdn.net/rock_joker)

发布了12 篇原创文章 · 获赞 96 · 访问量 11万+