# 第11章 线程

## 11.1引言

在前面的章节中讨论了进程，学习了LINUX/UNIX进程的环境、进程间的关系以及控制进程的不同方式。可以看出在相关的进程间能够存在一定的共享。

本章将进一步深入考察进程，了解如何使用多个控制线程（或简称为线程）在单进程环境中执行多个任务。一个进程中的所有线程都可以访问该进程的组成部件，如文件描述符和内存。

无论何时，只要单个资源需要在多个用户间共享，就必须处理一致性问题。本章的最后将讨论目前可用的同步机制，该机制用以防止多个线程査看到不一致的共享资源。

## 11.2线程概念

典型的LINUX/UNIX进程可以看成只有一个控制线程：一个进程在同一时刻只做一件事情。有了多个控制线程以后，在程序设计时可以把进程设计成在同一时刻能够做不止一件事，每个线程处理各自独立的任务。这种方法有很多好处。

* 通过为每种事件类型的处理分配单独的线程，能够简化处理异步事件的代码。每个线程在进行事件处理时可以采用同步编程模式，同步编程模式要比异步编程模式简单得多。
* 多个进程必须使用操作系统提供的复杂机制才能实现内存和文件描述符的共享。而多个线程自动地可以访问相同的存储地址空间和文件描述符。
* 有些问题可以通过将其分解从而改善整个程序的吞吐量。在只有一个控制线程的情况下，单个进程需要完成多个任务时，实际上需要把这些任务串行化，有了多个控制线程，相互独立的任务的处理就可以交叉进行，只需要为每个任务分配一个单独的线程，当然只有在处理过程互不依赖的情况下，两个任务的执行才可以穿插进行。
* 交互的程序同样可以通过使用多线程实现响应时间的改善，多线程可以把程序中处理用户输入输出的部分与其他部分分开。

有些人把多线程的程序设计与多处理器系统联系起来，但是即使程序运行在单处理器上， 也能得到多线程编程模型的好处。处理器的数量并不影响程序结构，所以不管处理器的个数是多少，程序可以通过使用线程得以简化。而且，即使多线程程序在串行化任务时不得不阻塞，由于某些线程在阻塞的时候还有另外一些线程可以运行，所以多线程程序在单处理器上运行仍然能够改善响应时间和吞吐量。

线程包含了表示进程内执行环境必需的信息，其中包括进程中标识线程的线程ID、一组寄存器值、找、调度优先级和策略、信号屏蔽字、errno变量以及线程私有数据。进程的所有信息对该进程的所有线程都是共享的，包括可执行的程序文本、程序的全局内存和堆内存、栈以及文件描述符。

我们将要讨论的线程接口来自POSIX.1-2001。线程接口（也称为“pthread”或“POSIX线程”）在POSIX.1-2001中是一个可选特征。

## 11.3线程标识

就像每个进程有一个进程ID—样，每个线程也有一个线程ID。进程ID在整个系统中是唯一的，但线程ID不同，线程ID只在它所属的进程环境中有效。

回忆一下进程ID，它用pid\_t数据类型来表示，是一个非负整数。线程ID则用pthread\_t 数据类型来表示，实现的时候可以用一个结构来代表pthread\_t数据类型，所以可移植的操作 系统实现不能把它作为整数处理。因此必须使用函数来对两个线程ID进行比较。

|  |
| --- |
| #include <pthread.h>  int pthread equal (pthread\_t tid1, pthread\_t tid2);  返回值：若相等则返回非0值，否则返回0 |

用结构表示pthread\_t数据类型的后果是不能用一种可移植的方式打印该数据类型的值。在程序调试过程中打印线程ID有时是非常有用的，而在其他情况下通常不需要打印线程ID，因此有可能出现不可移植的调试代码，当然这也算不上是很大的局限性。

线程可以通过调用pthread\_sel f函数获得自身的线程ID。

|  |
| --- |
| #include <pthread.h>  pthread\_t pthread\_self(void);  返回值：调用线程的线程ID |

当线程需要识别以线程ID作为标识的数据结构时，pthread\_self函数可以与pthread\_equal函数一起使用。例如，主线程可能把工作任务放在一个队列中，用线程ID来 控制每个工作线程处理哪些作业。如图下图所示：

|  |
| --- |
|  |
| 工作队列实例 |

主线程把新的作业放到一个工作队列中，由三个工作线程组成的线程池从队列中移出作业，每个线程并不是任意地处理从队列顶端取出的作业，而是由主线程控制作业的分配，生线程在每个待处理作业的结构中放置处理该作业的线程ID，每个工作线程只能移出标有自己线程ID的作业。

## 11.4线程的创建

在传统的LINUX/UNIX进程模型中，每个进程只有一个控制线程。从概念上讲，这与基于线程的模型中每个进程只包含一个线程是相同的。在POSIX线程(pthread)的情况下，程序开始运行时，它也是以单进程中的单个控制线程启动的，在创建多个控制线程以前，程序的行为与传统的进程并没有什么区别。新增的线程可以通过调用pthread\_create函数创建。

|  |
| --- |
| #include <pthread.h>  int pthread\_create (pthread\_ t \*restrict tidp,  const pthread\_attr\_t \*restrict attr, void \* (\* start\_rtn) (void), void \*restrict arg);  返回值：若成功则返回0，否则返回错误编号 |

当pthread\_create成功返回时，由指向的内存单元被设置为新创建线程的线程ID。attr参数用于定制各种不同的线程属性。线程属性将在后面讨论，现在暂时把它设置为 null，创建默认属性的线程。

新创建的线程从函数的地址开始运行，该函数只有一个无类型指针参数arg，如果需要向函数传递的参数不止一个，那么需要把这些参数放到一个结构中，然后把这个结构的地址作为arg参数传入。

线程创建时并不能保证哪个线程会先运行：是新创建的线程还是调用线程。新创建的线程可以访问进程的地址空间，并且继承调用线程的浮点环境和信号屏蔽字，但是该线程的未决信号集被清除。

注意pthread函数在调用失败时通常会返回错误码，它们并不像其他的函数一样设置errno。每个线程都提供errno的副本，这只是为了与使用errno的现有函数兼容。在线程中，从函数中返回错误码更为清晰整洁，不需要依赖那些随着函数执行不断变化的全局状态， 因而可以把错误的范围限制在引起出错的函数中。

虽然没有可移植的方法打印线程ID，但是可以写一个小的测试程序来完成这个任务，以便 更深入地了解线程是如何工作的。下面的程序创建了一个线程并且打印进程ID、新线程的线程ID以及初始线程的线程ID。

|  |
| --- |
|  |

这个实例有两个特别之处，需要处理主线程和新线程之间的竞争。在本章后面的内容中将学习如何更好地处理这种竞争。首先是主线程需要休眠，如果主线程不休眠，它就可能退 出，这样在新线程有机会运行之前整个进程可能就已经终止了。这种行为特征依赖于操作系统中的线程实现和调度算法。

第二个特别之处在于新线程是通过调用pthread\_self函数获取自己的线程ID，而不是从 共享内存中读出或者从线程的启动例程中以参数的形式接收到。回忆pthread\_Create函数， 它会通过第一个参数（tidp）返回新建线程的线程ID。在本例里，主线程把新线程ID存放在ntid中，但是新建的线程并不能安全地使用它，如果新线程在主线程调用pthread\_ create 返回之前就运行了，那么新线程看到的是未经初始化的ntid的内容，这个内容并不是正确的线程ID。

运行程序，结果如下：

|  |
| --- |
| $ ./a.out  main thread： pid 14954 tid 134529024 (0x804c000)  new thread： pid 14954 tid 134530048 (0x804c400) |

正如所料，两个线程的进程ID相同，但线程ID不同。在FreeBSD上运行程序清单11-1中的程序， 得到：

相同的程序在Linux上运行得到的结果稍有不同：

$ ./a.out

new thread： pid 6628 tid 1026 (0x402)

main thread： pid 6626 tid 1024 (0x400)

Linux线程ID看起来更合理一些，但进程ID并不匹配。这与Linux的线程实现有关，Linux使用clone系统调用来实现pthread\_create。clone系统调用创建子进程，这个子进程可以共 享父进程一定数量的执行环境（如文件描述符和内存），这个数量是可配置的。

另外还需要注意，主线程的输出基本上出现在新建线程的输出之前，但Linux却不是这样的，所以不能在线程调度上做出任何假设。

## 11.5线程终止

如果进程中的任一线程调用了exit，\_Exit或者\_exit，那么整个进程就会终止。与此类似，如果信号的默认动作是终止进程，那么，把该信号发送到线程会终止整个进程。

单个线程可以通过下列三种方式退出，在不终止整个进程的情况下停止它的控制流。

* 线程只是从启动例程中返回，返回值是线程的退出码。
* 线程可以被同一进程中的其他线程取消。
* 线程调用pthread\_exit。

|  |
| --- |
| #include <pthread.h>  void pthread\_exit (void \*rval\_ptr); |

rvaL\_ptr是一个无类型指针，与传给启动例程的单个参数类似。进程中的其他线程可以通过调用pthread\_join函数访问到这个指针。

|  |
| --- |
| #include <pthread.h>  int pthread\_join (pthread\_t thread, void \*\*rval\_ptr);  返回值：若成功则返回0，否则返回错误编号 |

调用线程将一直阻塞，直到指定的线程调用pthread\_exit、从启动例程中返回或者被取消。如果线程只是从它的启动例程返回，rval\_ptr将包含返回码。如果线程被取消，由rval\_ptr指定的内存单元就置为PTHREAD\_CANCELED。

可以通过调用pthread\_join自动把线程置于分离状态，这样资源就可以恢复。如果线程已经处于分离状态，pthread\_join调用就会失败，返回EINVAL。

如果对线程的返回值并不感兴趣，可以把rval\_ptr置为NULL。在这种情况下，调用 pthread\_join函数将等待指定的线程终止，但并不获取线程的终止状态。

下面程序说明了如何获取已终止的线程的退出码。

|  |
| --- |
|  |

运行上面的程序，得到的结果是:

|  |
| --- |
|  |

可以看出，当一个线程通过调用pthread\_exit退出或者简单地从启动例程中返回时，进程中的其他线程可以通过调用pthread\_join函数获得该线程的退出状态。

pthread\_create和pthread\_exit函数的无类型指针参数能传递的数值可以不止一个，该指针可以传递包含更复杂信息的结构的地址，但是注意这个结构所使用的内存在调用者完成 调用以后必须仍然是有效的，否则就会出现无效或非法内存访问。例如，在调用线程的栈上分配了该结构，那么其他的线程在使用这个结构时内存内容可能已经改变了。又如，线程在自己的栈上分配了一个结构然后把指向这个结构的指针传给pthread\_exit，那么当调用 pthread\_join的线程试图使用该结构时，这个栈有可能已经被撤销，这块内存也已另作他用。

下面程序给出了用自动变量（分配在栈上）作为pthread\_exit的参数时出现的问题。

|  |
| --- |
|  |

程序的运行结果如下：

|  |
| --- |
|  |

当然，运行结果根据内存结构、编译器以及线程库的实现会有所不同。

可以看出，当主线程访问这个结构时，结构的内容（在线程的栈上分配）已经改变。注意第二个线程（tid2）的栈是如何覆盖第一个线程的栈的。为了解决这个问题，可以使用全局结构，或者用malloc函数分配结构。

线程可以通过调用pthread\_cancel函数来请求取消同一进程中的其他线程。

|  |
| --- |
| #include <pthread.h>  int pthread\_cancel (pthread\_t tid);  返回值：若成功则返回0，否则返回错误编号 |

在默认的情况下，pthread\_cancel函数会使得由标识的线程的行为表现为如同调用了参数为PTHREAD\_CANCELED的pthread\_exit函数，但是，线程可以选择忽略取消方式或是控制取消方式。相关内容将后面章节详细讨论。注意pthread\_cancel并不等待线程终止，它仅仅提出请求。

线程可以安排它退出时需要调用的函数，这与进程可以用atexit函数安排进 程退出时需要调用的函数是类似的。这样的函数称为线程清理处理程序（thread cleanup handler）。 线程可以建立多个清理处理程序。处理程序记录在栈中，也就是说它们的执行顺序与它们注册 时的顺序相反。

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

当线程执行以下动作时调用清理函数，调用参数为arg，清理函数rtn的调用顺序是由pthread\_cleanup\_push 函数来安排的。

* 调用 pthread\_exit 时。
* 响应取消请求时。
* 用非零 execute参数调用pthread\_cleanup\_pop时。

如果execute参数置为0，清理函数将不被调用。无论哪种情况，pthread\_cleanup\_pop 都将删除上次pthread\_cleanup\_push调用建立的清理处理程序。

这些函数有一个限制，由于它们可以实现为宏，所以必须在与线程相同的作用域中以匹配对的形式使用，pthread\_cleanup\_push的宏定义可包含字符{，在这种情况下对应的匹配字符}就要在pthread\_cleanup\_pop定义中出现。

下面程序显示了如何使用线程清理处理程序。虽然例子有人为编造之嫌，但说清楚了其中涉及的清理机制。注意，虽然并未打算要传一个非零参数给线程启动例程，还是需要把pthread\_cleanup\_pop调用和pthread\_cleanup\_push调用匹配起来，否则，程序编译可能通不过。

|  |
| --- |
|  |

运行程序会得到：

|  |
| --- |
|  |

从输出结果可以看出，两个线程都正确地启动和退出了，但是只调用了第二个线程的清理处理程序，所以如果线程是通过从它的启动例程中返回而终止的话，那么它的清理处理程序就不会被调用，还要注意清理处理程序是按照与它们安装时相反的顺序被调用的。

现在可以开始看出线程函数和进程函数之间的相似之处。下表总结了这些相似的函数。

|  |  |  |
| --- | --- | --- |
| 进程原语和线程原语的比较 | | |
| 进程原语 | 线程原语 | 描述 |
| fork | pthread\_create | 创建新的控制流 |
| exit | pthread\_exit | 从现有的控制流中退出 |
| waitpid | pthread\_join | 从控制流中得到退出状态 |
| atexit | pthread\_cancel\_push | 注册在退出控制流时调用的函数 |
| getpid | pthread\_self | 获取控制流的ID |
| abort | pthread\_cancel | 请求控制流的非正常退出 |

在默认情况下，线程的终止状态会保存到对该线程调用pthread\_join，如果线程已经处于分离状态，线程的底层存储资源可以在线程终止时立即被收回。当线程被分离时，并不能用pthread\_join函数等待它的终止状态。对分离状态的线程进行pthread\_join的调用会产生失败，返回EINVAL。pthread\_detach调用可以用于使线程进入分离状态。

|  |
| --- |
| #include <pthread.h>  int pthread\_detach (pthread\_t tid);  返回值：若成功则返回0，否则返回错误编号 |

后面章节将学习通过对传给pthread\_create函数的线程属性进行修改，创建一个已处于分离状态的线程。

## 11.6线程同步

当多个控制线程共享相同的内存时，需要确保每个线程看到一致的数据视图。如果每个线程使用的变量都是其他线程不会读取或修改的，那么就不存在一致性问题。同样地，如果变量是只读的，多个线程同时读取该变量也不会有一致性问题。但是，当某个线程可以修改变量，而其他线程也可以读取或者修改这个变量的时候，就需要对这些线程进行同步，以确保它们在访问变量的存储内容时不会访问到无效的数值。

当一个线程修改变量时，其他线程在读取这个变量的值时就可能会看到不一致的数据。在变量修改时间多于一个存储器访问周期的处理器结构中，当存储器读与存储器写这两个周期交叉时，这种潜在的不一致性就会出现。当然，这种行为是与处理器结构相关的，但是可移植的程序并不能对使用何种处理器结构做出假设。

下图描述了两个线程读写相同变量的假设例子。

|  |
| --- |
|  |
| 两个线程的交叉存储器周期 |

在这个例子中，线程A读取变量然后给这个变量赋予一个新的值，但写操作需要两个存储器周期。当线程B在这两个存储器写周期中间读取这个相同的变量时，它就会得到不一致的值。

为了解决这个问题，线程不得不使用锁，在同一时间只允许一个线程访问该变量。下图描述了这种同步。

|  |
| --- |
|  |
| 两个线程同步内存访问 |

如果线程B希望读取变量，它首先要获取锁，同样地，当线程A更新变量时，也需要获取这把同样的锁。因而线程B在线程A释放锁以前不能读取变量。

当两个或多个线程试图在同一时间修改同一变量时，也需要进行同步。考虑变量递增操作的情况如下图：

|  |
| --- |
|  |
| 两个非同步的线程对同一个变量做增量操作 |

增量操作通常可分为三步：

* 从内存单元读入寄存器。 ’
* 在寄存器中进行变量值的增加。
* 把新的值写回内存单元。

如果两个线程试图在几乎同一时间对同一个变量做增量操作而不进行同步的话，结果就可能出现不一致。变量可能比原来增加了1，也有可能比原来增加了2，具体是1还是2取决于第二个线程开始操作时获取的数值。如果第二个线程执行第一步要比第一个线程执行第三步早，第二个线程读到的初始值就与第一个线程一样，它为变量加1，然后再写回去，事实上没有实际的效果，总的来说变量只增加了1。

如果修改操作是原子操作，那么就不存在竞争。在前面的例子中，如果增加1只需要一个存储器周期，那么就没有竞争存在。如果数据总是以顺序一致的方式出现，就不需要额外的同步。当多个线程并不能观察到数据的不一致时，那么操作就是顺序一致的。在现代计算机系统中，存储器访问需要多个总线周期，多处理器的总线周期通常在多个处理器上是交叉的，所以无法保证数据是顺序一致的。

在顺序一致的环境中，可以把数据修改操作解释为运行线程的顺序操作步骤。可以把这样的情形描述为“线程A对变量增加了1，然后线程B对变量增加了1，所以变量的值比原来的大2”，或者描述为“线程B对变量增加了1，然后线程A对变量增加了1，所以变量的值比原来的大2”。这两个线程的任何操作顺序都不可能让变量出现除了上述值以外的其他数值。

除了计算机体系结构的因素以外，程序使用变量的方式也会引起竞争，也会导致不一致的情况发生。例如，可能会对某个变量加1，然后基于这个数值做出某种决定。增量操作这一步和做出决定这一步两者的组合并非原子操作，因而给不一致情况的出现提供了可能。

### 1.互斥量

可以通过使用pthread的互斥接口保护数据，确保同一时间只有一个线程访问数据。互斥量(mutex)从本质上说是一把锁，在访问共享资源前对互斥量进行加锁，在访问完成后释放互斥量上的锁。对互斥量进行加锁以后，任何其他试图再次对互斥量加锁的线程将会被阻塞直到当前线程释放该互斥锁。如果释放互斥锁时有多个线程阻塞，所有在该互斥锁上的阻塞线程都会变成可运行状态，第一个变为运行状态的线程可以对互斥量加锁，其他线程将会看到互斥锁依然被锁住，只能回去再次等待它重新变为可用。在这种方式下，每次只有一个线程可以向前执行。

在设计时需要规定所有的线程必须遵守相同的数据访问规则，只有这样，互斥机制才能正常工作。操作系统并不会做数据访问的串行化。如果允许其中的某个线程在没有得到锁的情况下也可以访问共享资源，那么即使其他的线程在使用共享资源前都获取了锁，也还是会出现数据不一致的问题。

互斥变量用pthread\_mutex\_t数据类型来表示，在使用互斥变量以前，必须首先对它进行初始化，可以把它置为常量PTHREAD\_MUTEX\_INITIALIZER （只对静态分配的互斥量），也可以通过调用pthread\_mutex\_init函数进行初始化。如果动态地分配互斥量（例如通过调用malloc函数），那么在释放内存前需要调用pthread\_mutex\_destroy。

|  |
| --- |
| #include <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);  返回值：若成功则返回0，否则返回错误编号 |

要用默认的属性初始化互斥量，只需把attr设置为NULL。非默认的互斥量属性将在后面讨论。

对互斥量进行加锁，需要调用pthread\_mutex\_lock，如果互斥量已经上锁，调用线程将阻塞直到互斥量被解锁。对互斥量解锁，需要调用pthread\_mutex\_unlock。

|  |
| --- |
| #include <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);  返回值：若成功则返回0，否则返回错误编号 |

如果线程不希望被阻塞，它可以使用pthread\_mutex\_trylock尝试对互斥量进行加锁。 如果调用pthread\_mutex\_trylock时互斥量处于未锁住状态，那么pthread\_mutex\_trylock将锁住互斥量，不会出现阻塞并返回0，否则pthread\_mutex\_trylock就会失败，不能锁住互斥量，而返回EBUSY。

下面程序演示了用于保护某个数据结构的互斥量。当多个线程需要访问动态分配的对象时，可以在对象中嵌入引用计数，确保在所有使用该对象的线程完成数据访问之前，该对象 内存空间不会被释放。

|  |
| --- |
|  |

在对引用计数加1、减1以及检查引用计数是否为0这些操作之前需要锁住互斥量。在 foo\_alloc函数将引用计数初始化为1时没必要加锁，因为在这个操作之前分配线程是唯一引用该对象的线程。但是在这之后如果要将该对象放到一个列表中，那么它就有可能被别的线程发现，因此需要首先对它加锁。

在使用该对象前，线程需要对这个对象的引用计数加1，当对象使用完毕时，需要对引用计数减1。当最后一个引用被释放时，对象所占的内存空间就被释放。

### 2.避免死锁

如果线程试图对同一个互斥量加锁两次，那么它自身就会陷入死锁状态，使用互斥量时，还有其他更不明显的方式也能产生死锁。例如，程序中使用多个互斥量时，如果允许一个线程一直占有第一个互斥量，并且在试图锁住第二个互斥量时处于阻塞状态，但是拥有第二个互斥量的线程也在试图锁住第一个互斥量，这时就会发生死锁。因为两个线程都在相互请求另一个线程拥有的资源，所以这两个线程都无法向前运行，于是就产生死锁。

可以通过小心地控制互斥量加锁的顺序来避免死锁的发生。例如，假设需要对两个互斥量A和B同时加锁，如果所有线程总是在对互斥量B加锁之前锁住互斥量A，那么使用这两个互斥量不会产生死锁（当然在其他的资源上仍可能出现死锁），类似地，如果所有的线程总是在锁住互斥量A之前锁住互斥量B，那么也不会发生死锁。只有在一个线程试图以与另一个线程相反的顺序锁住互斥量时，才可能出现死锁。

有时候应用程序的结构使得对互斥量加锁进行排序是很困难的，如果涉及了太多的锁和数据结构，可用的函数并不能把它转换成简单的层次，那么就需要采用另外的方法。可以先释放占有的锁，然后过一段时间再试。这种情况可以使用pthread \_mutex\_trylock 接口避免 死锁。如果已经占有某些锁而且pthread\_mutex\_trylock接口返回成功，那么就可以前进；但是，如果不能获取锁，可以先释放已经占有的锁，做好清理工作，然后过一段时间重新尝试。

下面这个程序修改了前面的程序，用以描述两个互斥量的使用方法（见程序清单11-6）。当同时需要两个互斥量时，总是让它们以相同的顺序加锁，以避免死锁。第二个互斥量维护着一个用于跟踪foo数据结构的散列列表。这样hashlock互斥量保护foo数据结构中的fh散列表和f\_next散列链字段。foo结构中的£\_lock互斥量保护对foo结构中的其他字段的访问。

|  |
| --- |
|  |

比较这两个程序，可以看出分配函数现在锁住散列列表锁，把新的结构添加到散列存储桶中，在对散列列表的锁解锁之前，先锁住新结构中的互斥量。因为新的结构是放在全局列表中的，其他线程可以找到它，所以在完成初始化之前，需要阻塞其他试图访问新结构的线程。

foo\_find函数锁住散列列表锁然后搜索被请求的结构。如果找到了，就增加其引用计数并返回指向该结构的指针。注意加锁的顺序是先在foo\_find函数中锁定散列列表锁，然后再在foo\_hold函数中锁定foo结构中的f\_lock互斥量。

现在有了两个锁以后，foo\_rele函数变得更加复杂。如果这是最后一个引用，因为将需要从散列列表中删除这个结构，就要先对这个结构互斥量进行解锁，才可以获取散列列表锁。然后重新获取结构互斥量。从上一次获得结构互斥量以来可能处于被阻塞状态，所以需要重新检查条件，判断是否还需要释放这个结构。如果其他线程在我们为满足锁顺序而阻塞时发现了这个结构并对其引用计数加1，那么只需要简单地对引用计数减1，对所有的东西解锁然后返回。

如此加、解锁太复杂，所以需要重新审视原来的设计。也可以使用散列列表锁来保护结构引用计数，使事情大大简化，结构互斥量可以用于保护foo结构中的其他任何东西。下面的程序反映了这种变化。

|  |
| --- |
|  |

注意，前面的程序相比，这个程序简单得多。两种用途使用相同的锁时，围绕散列列表和引用计数的锁的排序问题就随之不见了。多线程的软件设计经常要考虑这类折中处理方案。如果锁的粒度太粗，就会出现很多线程阻塞等待相同的锁，源自并发性的改善微乎其微。如果锁的粒度太细，那么过多的锁开销会使系统性能受到影响，而且代码变得相当复杂。作为一个程序员，需要在满足锁需求的情况下，在代码复杂性和优化性能之间找好平衡点。

### 3. 读写锁

读写锁与互斥量类似，不过读写锁允许更髙的并行性。互斥量要么是锁住状态要么是不加锁状态，而且一次只有一个线程可以对其加锁。读写锁可以有三种状态：读模式下加锁状态，写模式下加锁状态，不加锁状态。一次只有一个线程可以占有写模式的读写锁，但是多个线程可以同时占有读模式的读写锁。

当读写锁是写加锁状态时，在这个锁被解锁之前，所有试图对这个锁加锁的线程都会被阻塞。当读写锁在读加锁状态时，所有试图以读模式对它进行加锁的线程都可以得到访问权，但是如果线程希望以写模式对此锁进行加锁，它必须阻塞直到所有的线程释放读锁。虽然读写锁的实现各不相同，但当读写锁处于读模式锁住状态时，如果有另外的线程试图以写模式加锁，读写锁通常会阻塞随后的读模式锁请求。这样可以避免读模式锁长期占用，而等待的写模式锁请求一直得不到满足。

读写锁非常适合于对数据结构读的次数远大于写的情况。当读写锁在写模式下时，它所保护的数据结构就可以被安全地修改，因为当前只有一个线程可以在写模式下拥有这个锁。当读写锁在读模式下时，只要线程获取了读模式下的读写锁，该锁所保护的数据结构可以被多个获得读模式锁的线程读取。

读写锁也叫做共享-独占锁，当读写锁以读模式锁住时，它是以共享模式锁住的，当它以写模式锁住时，它是以独占模式锁住的。

与互斥量一样，读写锁在使用之前必须初始化，在释放它们底层的内存前必须销毁。

|  |
| --- |
| #include <pthread.h>  int pthread\_rwlock\_init (pthread\_rwlock\_t \*restrict rwlock,  const pthread\_rwlockattr\_t\*restrict attr)；  int p thread\_rwlock\_destroy (pthread\_rwlock\_t \* rwlock);  两者的返回值都是：若成功则返回0，否则返回错误编号 |

读写锁通过调用pthread\_rwlock\_init进行初始化。如果希望读写锁有默认的属性，可以传一个空指针给attr，读写锁的属性将在后面讨论。

在释放读写锁占用的内存之前，需要调用pthread\_rwlock\_destroy做清理工作。如果pthread\_rwlock\_init为读写锁分配了资源，pthread\_rwlock\_destroy将释放这些资源。如果在调用pthread\_rwlock\_destroy之前就释放了读写锁占用的内存空间，那么分配给这个锁的资源就丢失了。

要在读模式下锁定读写锁，需要调用pthread\_rwlock\_rdlock；要在写模式下锁定读写锁，需要调用pthread\_rwlock\_wrlock。不管以何种方式锁住读写锁，都可以调用pthread\_rwlock\_unlock进行解锁。

|  |
| --- |
| #include <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);  所有的返回值都是：若成功则返回0，否则返回错误编号 |

在实现读写锁的时候可能会对共享模式下可获取的锁的数量进行限制，所以需要检查pthread\_rwlock\_rdlock的返回值。即使pthread\_rwlock\_wrlock和 pthread\_ rwlock\_unlock有错误的返回值，如果锁设计合理的话，也不需要检查其返回值。错误返回值的定义只是针对不正确地使用读写锁的情况，例如未经初始化的锁，或者试图获取已拥有的锁从而可能产生死锁这样的错误返回等。

LINUX/UNIX同样定义了有条件的读写锁原语的版本。

|  |
| --- |
| #include <pthread.h>  int pthread\_rwlock\_tryrdlock(pthread\_rwlock\_t \* rwlock);  int pthread\_rwlock\_trywrlock(pthread\_rwlock\_t \* rwlock);  两者的返回值都是：若成功则返回0，否则返回错误编号 |

可以获取锁时，函数返回0，否则，返回错误EBUSY。这些函数可以用于前面讨论的遵守某种锁层次但还不能完全避免死锁的情况。

下面程序解释了读写锁的使用。作业请求队列由单个读写锁保护，以此实现多个工作线程获取由单个主线程分配给它们的作业。

|  |
| --- |
|  |

在这个例子中，不管什么时候需要增加一个作业到队列中或者从队列中删除作业，都用写模式锁住队列的读写锁。不管何时搜索队列，首先需要获取读模式下的锁，允许所有的工作线程并发地搜索队列。在这种情况下，只有线程搜索队列的频率远远高于增加或删除作业时，使用读写锁才可能改善性能。

工作线程只能从队列中读取与它们的线程ID匹配的作业。既然作业结构同一时间只能由一个线程使用，所以不需要额外加锁。 口

### 4.条件变置

条件变量是线程可用的另一种同步机制。条件变量给多个线程提供了一个会合的场所。条件变量与互斥量一起使用时，允许线程以无竞争的方式等待特定的条件发生。

条件本身是由互斥量保护的。线程在改变条件状态前必须首先锁住互斥量，其他线程在获得互斥量之前不会察觉到这种改变，因为必须锁定互斥量以后才能计算条件。

条件变量使用之前必须首先进行初始化，pthread\_cond\_t数据类型代表的条件变量可以用两种方式进行初始化，可以把常量PTHREAD\_COND\_INITIALIZER赋给静态分配的条件变量，但是如果条件变量是动态分配的，可以使用pthread\_cond\_init函数进行初始化。

在释放底层的内存空间之前，可以使用pthread\_mutex\_destroy函数对条件变量进行去除初始化（deinitialize）。

|  |
| --- |
| #include <pthread.h>  int pthread\_cond\_init(pthread\_cond\_t \*restrict cond,  pthread\_condattr\_t\* restrict attr);  int pthread\_cond\_destroy(pthread\_cond\_t \*cond);  两者的返回值都是：若成功则返回0，否则返回错误编号 |

除非需要创建一个非默认属性的条件变量，否则pthread\_cond\_init函数的attr参数可以设置为NULL，条件变量属性将在后面讨论。

使用pthread\_cond\_wait等待条件变为真，如果在给定的时间内条件不能满足，那么会 生成个代表出错码的返回变量。

|  |
| --- |
| #include <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 f const struct timespec\*restrict timeout);  两者的返回值都是：若成功则返回0，否则返回错误编号 |

传递给pthread\_cond\_wait的互斥量对条件进行保护，调用者把锁住的互斥量传给函数。函数把调用线程放到等待条件的线程列表上，然后对互斥量解锁，这两个操作是原子操作。这样就关闭了条件检查和线程进入休眠状态等待条件改变这两个操作之间的时间通道，这样线程就不会错过条件的任何变化。pthread\_cond\_wait返回时，互斥量再次被锁住。

pthread\_cond\_timedwait函数的工作方式与pthread\_cond\_wait函数相似，只是多了一个timeout。timeout值指定了等待的时间，它是通过timespec结构指定。时间值用秒数或者分秒数来表示，分秒数的单位是纳秒。

struct timespec {

time\_t tv\_sec; /\* seconds \*/

long tv\_nsec； /\* nanoseconds \*/

};

使用这个结构时，需要指定愿意等待多长时间，时间值是一个绝对数而不是相对数。例如，如果能等待3分钟，就需要把当前时间加上3分钟再转换到timespec结构，而不是把3分钟转换成timespec结构。

可以使用gettimeofday获取用timeval结构表示的当前时间，然后把这个时间转换成timespec结构。要得到timeout值的绝对时间，可以使用下面的函数：

|  |
| --- |
|  |

如果时间值到了但是条件还是没有出现，pthread\_cond\_timedwait将重新获取互斥量然后返回错误ETIMEDOUT。从pthread\_cond\_wait或者pthread\_cond\_timedwait调用成功返回时，线程需要重新计算条件，因为其他的线程可能已经在运行并改变了条件。

有两个函数可以用于通知线程条件已经满足。pthread\_cond一signal函数将唤醒等待该条件的某个线程，而pthread\_cond\_broadcast函数将唤醒等待该条件的所有线程。

|  |
| --- |
| #include <pthread.h>  int pthread\_cond\_signal(pthread\_cond\_t \*cond);  int pthread\_cond\_broadcast(pthread\_cond\_t \*cond);  两者的返回值都是：若成功则返回0，否则返回错误编号 |

调用pthread\_cond\_signal或者pthread\_cond\_broadcast，也称为向线程或条件发送信号。必须注意一定要在改变条件状态以后再给线程发送信号。

|  |
| --- |
|  |

条件是工作队列的状态。用互斥量保护条件，在while循环中判断条件。把消息放到工作队列时，需要占有互斥量，但向等待线程发送信号时并不需要占有互斥量。只要线程可以在调用cond\_signal之前把消息从队列中拖出，就可以在释放互斥量以后再完成这部分工作。因为是在while循环中检查条件，所以不会存在问题：线程醒来，发现队列仍为空，然后返回继续等待。如果代码不能容忍这种竞争，就需要在向线程发送信号的时候占有互斥量。

## 11.7小结

在本章中，介绍了线程的概念，讨论了现有的创建线程和销毁线程的相关接口，此外还介绍了线程同步问题，讨论了三种基本的同步机制：互斥、读写锁以及条件变量，了解了如何使用它们来保护共享资源。

# 第12章 线程控制

## 12.1引言

在上一章中，学习了线程和线程同步的基础知识。在本章中，将学习控制线程行为方面的详细内容，在前面的章节中对线程属性和同步原语属性都取其默认行为，忽略了这些属性的具体介绍。

接下来将介绍同一进程中的多个线程之间如何保持数据的私有性，最后讨论基于进程的系 统调用如何与线程进行交互。

## 12.2线程属性

上一章所有调用pthread\_create函数的例子中，传入的参数都是空指针，而不是指向pthread\_attr\_t结构的指针。可以使用pthread\_attr\_t结构修改线程默认属性，并把这些属性与创建的线程联系起来。可以使用pthread\_attr\_init函数初始化pthread\_attr\_t结构。调用pthread\_attr\_init以后，pthread\_attr\_t结构所包含的内容就是操作系统实现支持的线程所有属性的默认值。如果要修改其中个别属性的值，需要调用其他的函数，这方面的细节将在本节的后续内容中讨论。

|  |
| --- |
| #include <pthread.h>  int pthread\_attr\_init(pthread\_attr\_t \*attr);  int pthread\_attr\_destroy(pthread\_attr\_t \*attr);  两者的返回值都是：若成功则返回0，否则返回错误编号 |

如果要去除对pthread\_attr\_t结构的初始化，可以调用pthread\_attr\_destroy函数。如果pthread\_attr\_init实现时为属性对象分配了动态内存空间，pthread\_attr\_ destroy将会释放该内存空间。除此之外，pthread\_attr\_destroy还会用无效的值初始化属性对象，因此如果该属性对象被误用，将会导致pthread\_create函数返回错误。

pthread\_attr\_t结构对应用程序是不透明的，也就是说应用程序并不需要了解有关属性对象内部结构的任何细节，因而可以增强应用程序的可移植性。LINUX/UNIX沿用了这种模型，并且为查询和设置每种属性定义了独立的函数。

下表总结了LINUX/UNIX定义的线程属性。虽然LINUX/UNIX还为实时线程定义了额外的属性，但这里并不打算讨论这些属性。

|  |  |
| --- | --- |
| 线程属性 | |
| 名称 | 描述 |
| detachstate | 线程的分离状态属性 |
| guardsize | 线程找末尾的警戒缓冲区大小（字节数） |
| stackaddr | 线程栈的最低地址 |
| stacksize | 线程找的大小（字节数） |

前面介绍了分离线程的概念。如果对现有的某个线程的终止状态不感兴趣的话，可以使 用pthread\_detach函数让操作系统在线程退出时收回它所占用的资源。

如果在创建线程时就知道不需要了解线程的终止状态，则可以修改pthread\_attr\_t结构 中的detachstate线程属性，让线程以分离状态启动。可以使用pthread\_a ttr\_ setdetachstate函数把线程属性detachstate设置为下面的两个合法值之一：设置为PTHREAD\_CREATE\_DETACHED，以分离状态启动线程，或者设置为PTHREAD\_CREATE\_JOINABLE，正常启动线程，应用程序可以获取线程的终止状态。

|  |
| --- |
| #include <pthread.h>  int pthread\_attr\_getdetachstate(const pthread\_attr\_t \*restrict attr,  int \*detachstate);  int pthread\_attr\_setdetachstate (pthread\_attr\_t \*attr, int detachstate);  两者的返回值都是：若成功则返回0,否则返回错误编号 |

可以调用pthread\_attr\_getdetachstate函数获取当前的detachstate线程属性，第二个参数所指向的整数也许被设置为PTHREAD\_CREATE\_DETACHED，也可能设置为 PTHREAD\_CREATE\_JOINABLE，具体要取决于给定p thr e ad\_a t tr\_t结构中的属性值。

下面给出了一个以分离状态创建线程的函数。

|  |
| --- |
|  |

注意，这里忽略了pthread\_attr\_destroy函数调用的返回值。在这种情况下，由于对线程属性进行了合理的初始化，pthread\_attr\_destroy—般不会失败。但是如果pthread\_attr\_destroy确实出现了失败的情况，清理工作就会变得很困难：必须销毁刚刚创建的线程，而这个线程可能已经运行，并且与pthread\_attr\_destroy函数可能是异步执行的。忽略pthread\_attr\_destroy的错误返回可能出现的最坏情况是：如果pthread\_attr\_init分配了内存空间，这些内存空间会被泄漏。另一方面，如果pthread\_attr\_ init成功地对线程属性进行了初始化，但pthread\_attr\_destroy在做清理工作时却出现了失败，就没有任何补救策略，因为线程属性结构对应用来说是不透明的，可以对线程属性结构进行清理的唯一接口是pthread\_attr\_destroy，但它失败了。

LINUX/UNIX定义了线程栈属性的一些操作接口。虽然很多pthread实现中仍然提供两个早些时候的函数pthread\_attr\_getstackaddr和pthread\_attr\_setstackaddr，但在新版本的LINUX/UNIX中这两个函数已被标记为过时，线程栈属性的查询和修改一般是通过较新的函数pthread\_attr\_getstack和pthread\_attr\_setstack来进行。这些新的函数消除了老接口定义中存在的二义性。

|  |
| --- |
| #include <pthread.h>  int pthread\_attr\_gets tack (const pthread\_attr\_t \*restrict attr,  void \*\*restrict stackaddr, size\_t \* restrict stacksize)；  int pthread\_attr\_setstack (const pthread\_attr\_t \*attr,  void \* stackaddr, size\_t \* stacksize);  两者的返回値都是：若成功则返冋0，否则返回错误编号 |

这两个函数可以用于管理线程stackaddr属性，也可以用于管理statcksize线程属性。

对进程来说，虚拟地址空间的大小是固定的，进程中只有一个栈，所以它的大小通常不是问题。但对线程来说，同样大小的虚拟地址空间必须被所有的线程栈共享。如果应用程序使用了太多的线程，致使线程栈的累计大小超过了可用的虚拟地址空间，这时就需要减少线程默认的栈大小。另一方面，如果线程调用的函数分配了大量的自动变量或者调用的函数涉及很深的栈帧（stack frame），那么这时需要的栈大小可能要比默认的大。

如果用完了线程栈的虚拟地址空间，可以使用malloc或者mmap来为其他栈分配空间，并用pthread\_attr\_setstack函数来改变新建线程的栈位置。线程栈所占内存范围中可寻址的最低地址可以由stack\_addr参数指定，该地址与处理器结构相应的边界对齐。

stackaddr线程属性被定义为栈的内存单元的最低地址，但这并不必然是栈的开始位置。对于某些处理器结构来说，找是从高地址向低地址方向伸展的，那么对stackaddr线程属性就是栈的结尾而不是开始位置。

pthread\_attr\_getstackaddr和pthread\_attr\_setstackaddr的缺陷在于stackaddr参数并没有明确地指定。它可以解释为栈的开始地址，还可以解释成用作栈的内存范围的最低地址。在栈内存地址空间从高地址向低地址扩展的处理器结构中，如果stackaddr参数是栈地址空间的最低地址，那么就需要知道栈的大小才能确定栈的开始位置。pthread\_attr\_getstack和pthread\_attr\_setstack函数纠正了这些缺陷。

应用程序也可以通过pthread\_attr\_getstacksize和pthread\_attr\_setstacksize函数读取或设置线程属性stacksize。

|  |
| --- |
| #include <pthread.h>  int pthread\_attr\_getstacksize (const pthread\_attr\_t \*restrict attr,  sizet \* restrict stacksize);  int pthread\_attr\_setstacksize (pthread\_attr\_t \*attr, size\_t stacksize);  两者的返回值都是：若成功则返回0，否则返回错误编号 |

如果希望改变栈的默认大小，但又不想自己处理线程栈的分配问题，这时使用pthread\_attr\_setstacksize函数就非常有用。

线程属性控制着线程栈末尾之后用以避免栈溢出的扩展内存的大小。这个属性默认设置为PAGESIZE个字节。可以把线程属性设为0，从而不允许属性的这种特征行为发生：在这种情况下不会提供警戒缓冲区。同样地，如果对线程属性stackaddr作了修改，系统就会假设我们会自己管理栈，并使警戒栈缓冲区机制无效，等同于把线程属性设为0。

|  |
| --- |
| #include <pthread.h>  int pthread\_attr\_getguardsize(const pthread\_attr\_t\* restrict attr,  size\_t \* re strict guardsize)；  int pthread\_attr\_setguardsize (pthread\_attr\_t \*attr, size\_t guardsize);  两者的返回值都是：若成功则返回0,否则返回错误编号 |

如果线程属性被修改了，操作系统可能把它取为页大小的整数倍。如果线程的栈指针溢出到警戒区域，应用程序就可能通过信号接收到出错信息。

LINUX/UNIX还定义了其他的一些可选的线程属性作为实时线程可选属性的一部分，但在这里不讨论这些属性。

**更多的线程属性**

线程还有其他的一些属性，这些属性并没有在pthread\_attr\_t结构中表达：

可取消状态

可取消类型

并发度

可取消状态和可取消类型将在后面的章节中讨论。

并发度控制着用户级线程可以映射的内核线程或进程的数目。如果操作系统的实现在内核级的线程和用户级的线程之间保持一对一的映射，那么改变并发度并不会有什么效果，因为所有的用户级的线程都可能被调度到。但是，如果操作系统的实现让用户级线程到内核级线程或进程之间的映射关系是多对一的话，那么在给定时间内增加可运行的用户级线程数，可能会改善性能。pthread\_setconcurrency函数可以用于提示系统，表明希望的并发度。

|  |
| --- |
| #include <pthread.h>  int pthread\_getconcurrency(void);  返回值：当前的并发度  int pthread\_set concurrency (int level)；  返回值：若成功则返回0，否则返回错误编号 |

pthread\_getconcurrency函数返回当前的并发度。如果操作系统当前正控制着并发度（即之前没有调用过pthread\_setconcurrency函数），那么pthread\_getconcurrency将返回0。

pthread\_setconcurrency函数设定的并发度只是对系统的一个提示，系统并不保证请求的并发度一定会被采用。如果希望系统自己决定使用什么样的并发度，就把传入的参数level设为0。这样，应用程序调用level参数为0的pthread\_setconcurrency函数，就可以撤销在这之前level参数非零的pthread\_setconcurrency调用所产生的作用。

## 12.3同步属性

就像线程具有属性一样，线程的同步对象也有属性。本节讨论互斥量、读写锁和条件变量的属性。

### 1.互斥置属性

用pthread\_mutexattr\_init初始化pthread\_mutexattr\_t结构，用pthread\_mutexattr\_destroy来对该结构进行回收。

|  |
| --- |
| #include <pthread.h>  int pthread\_mutexattr\_init(pthread\_mutexattr\_t \*attr);  int pthread\_mutexattr\_destroy(pthread\_mutexattr\_t \*attr);  返回值：若成功则返回0，否则返回错误编号 |

pthread\_mutexattr\_init函数用默认的互斥量属性初始化pthread\_mutexattr\_t结构。值得注意的两个属性是进程共享属性和类型属性。

进程共享属性是可选的。在进程中，多个线程可以访问同一个同步对象，这是默认的行为。在这种情况下，进程共享互斥量属性需设置为PTHREAD\_PROCESS\_PRIVATE。

以后可能会用到这样的机制，允许相互独立的多个进程把同一个内存区域映射到它们各自独立的地址空间中。就像多个线程访问共享数据一样，多个进程访问共享数据通常也需要同步。如果进程共享互斥量属性设置为PTHREAD\_PROCESS\_SHARED，从多个进程共享的内存区域中分配的互斥量就可以用于这些进程的同步。

得到它的进程共享属性，可以用pthread\_mutexattr\_setpshared函数修改进程共享属。

|  |
| --- |
| #include <pthread.h>  int pthread\_mutexattr\_getpshared(const pthread\_mutexattr\_t \*restrict attr,  int \* restrict pshared);  int pthread\_mutexattr\_setpshared (pthread\_mutexattr\_t \*attr,  int pshared)；  返回值：若成功则返回0，否则返回错误编号 |

进程共享互斥量属性设为PTHREAD\_PROCESS\_PRIVATE时，允许pthread线程库提供更加有效的互斥量实现，这在多线程应用程序中是默认的情况。在多个进程共享多个互斥量的情况下，pthread线程库可以限制开销较大的互斥量实现。

类型互斥量属性控制着互斥量的特性。LINUX/UNIX定义了四种类型。 PTHREAD\_MUTEX\_NORMAL类型是标准的互斥量类型，并不做任何特殊的错误检査或死锁检测。PTHREAD\_ MUTEX\_ERRORCHECK互斥量类型提供错误检查。

PTHREAD\_MUTEX\_RECURSIVE互斥量类型允许同一线程在互斥量解锁之前对该互斥量进行多次加锁。用一个递归互斥量维护锁的计数，在解锁的次数和加锁次数不相同的情况下不会释放锁。所以如果对一个递归互斥量加锁两次，然后对它解锁一次，这个互斥量依然处于加锁状态，在对它再次解锁以前不能释放该锁。

最后，PTHREAD\_MUTEX\_DEFAULT类型可以用于请求默认语义。操作系统在实现它的时候可以把这种类型自由地映射到其他类型。例如，在Linux中，这种类型映射为普通的互斥量类型。

四种类型的行为如下表所示。“不占用时解锁”这一栏指的是一个线程对被另一个线程加锁的互斥量进行解锁的情况，“在已解锁时解锁”这一栏指的是当一个线程对已经解锁的互斥量进行解锁时将会发生的情况，这通常是编码错误所致。

|  |  |  |  |
| --- | --- | --- | --- |
| 互斥量类型行为 | | | |
| 互斥量类型 | 没解锁时再加锁？ | 不占用时解锁？ | 在解锁时解锁？ |
| PTHREAD\_MUTEX\_NORMAL | 死锁 | 未定义 | 定义 |
| PTHREAD\_MUTEX\_ERRORCHECK | 返回错误 | 返回错误 | 返回错误 |
| PTHREAD\_MUTEX\_RECURSIVE | 允许 | 返回错误 | 返回错误 |
| PTHREAD\_MUTEX\_DEFAULT | 未定义 | 未定义 | 未定义 |

可以用pthread\_mutex\_attr\_gettype函数得到互斥量类型属性，用pthread\_mutexattr\_settype函数修改互斥量类型属性。

|  |
| --- |
| #include <pthread.h>  int pthread\_mutexattr\_gettype(const pthread\_mutexattr\_t \*  restrict attr, int ^restrict type);  int pthread\_mutexattr\_settype(pthread\_mutexattr\_t \*attr, int type);  两者的返回值都是：若成功则返回0，否则返回错误编号 |

回忆前面章节，互斥量用于保护与条件变量关联的条件。在阻塞线程之前，pthread\_cond\_wait和pthread\_cond\_timedwait函数释放与条件相关的互斥量，这就允许其他线程获取互斥量、改变条件、释放互斥量并向条件变量发送信号。既然改变条件时必须占有互斥量，所以使用递归互斥量并不是好的办法。如果递归互斥量被多次加锁，然后用在调用pthread\_cond\_wait函数中，那么条件永远都不会得到满足，因为pthread\_cond\_wait 所做的解锁操作并不能释放互斥量。

如果需要把现有的单线程接口放到多线程环境中，递归互斥量是非常有用的，但由于程序兼容性的限制，不能对函数接口进行修改。然而由于递归锁的使用需要一定技巧，它只应在没有其他可行方案的情况下使用。

下图解释了递归锁看似解决并发问题。假设func1和func2是函数库中现有的函数，其接口不能改变，因为存在调用这两个接口的应用程序，而且应用程序不能改动。

|  |
| --- |
|  |
| 使用递归锁的情况 |

为了保持接口跟原来相同，可以把互斥量嵌入到数据结构中，把这个数据结构的地址（x）作为参数传入。这种方案只有在为该数据结构提供了分配函数时才可行，所以应用并不知道数据结构的大小（假设在其中增加互斥量后必须扩大该数据结构的大小）。

如果在最初定义教据结构时，预留了足够的可填充字段，允许把一些填充字段替换成互斥量，那么这种方法也是可行的。不过，大多数程序员并不善于预测未来，所以这不是普遍可行的经验。

如果func1和func2函数都必须操作这个结构，而且可能会有多个线程同时访问该数据结构，那么func1和func2必须在操作数据以前对互斥量加锁。当func1必须调用func2时，如果互斥量不是递归类型，那么就会出现死锁。如果能在调用func2之前释放互斥量，在func2 返回后重新获取互斥量，那么就可以避免使用递归互斥量，但这也给其他的线程提供了机会，其他的线程可能在func1执行期间得到互斥量的控制权，修改这个数据结构。这也许是不可接受的，当然具体的情况要取决于互斥量试图提供什么样的保护。

下图显示了这种情况下使用递归互斥量的另一种替代方法。

|  |
| --- |
|  |
| 避免使用递归锁的情况 |

通过提供func2函数的私有 版本（称之为func2\_locked函数），可以保持func1和func2函数接口不变，并且避免使用递归互斥量。要调用func2\_locked函数，必须占有嵌入到数据结构中的互斥量，这个数据结构的地址是作为参数传入的。func2\_locked的函数体包含func2的副本，func2现在只是用以获取互斥量，调用func2\_locked,最后释放互斥量。

如果并不一定要保持库函数掊口不变，就可以在每个函数中另外再加一个参数，以表明这个结构是否被调用者锁定。但是，如果可能的话，保持接口不变通常是更好的选择，这样可以避免实现过程中人为加入的东西对原有接口产生不良影响。

提供函数的加锁版本和不加锁版本，这样的策略在简单的情况下通常是可行的。在比较复杂的情况下，例如库需要调用库以外的函数，而且可能会再次回调库中的函数时，就需要依赖递归锁。

下面程序解释了有必要使用递归互斥量的另一种情况。这里，有一个“超时”（timeout）函数，它允许另一个函数可以安排在未来的某个时间运行。假设线程并不是很昂贵的资源，可以为每个未决的超时函数创建一个线程。线程在时间未到时将一直等待，时间到了以后就调用请求的函数。

|  |
| --- |
|  |

如果不能创建线程，或者安排函数运行的时间已过，问题就出现了。在这种情况下，要从当前环境中调用之前请求运行的函数，因为函数要获取的锁和现在占有的锁是同一个，除非该锁是递归的，否则就会出现死锁。

这里使用前面的makethread函数以分离状态创建线程。希望函数在未来的某个时间运行，而且不希望一直等待线程结束。

可以调用sleep等待超时到达，但它提供的时间粒度是秒级的，如果希望等待的时间不是整数秒，需要用nanosleep函数，它提供了类似的功能。

timeout的调用者需要占有互斥量来检査条件，并且把retry函数安排为原子操作。retry函数试图对同一个互斥量进行加锁，因此除非互斥量是递归的，否则如果timeout函数直接调用retry就会导致死锁。

### 2.读写锁属性

读写锁与互斥量类似，也具有属性。用pthread\_rwlockattr\_init初始化pthread\_rwlockattr\_t结构，用pthread\_rwlockattr\_destroy回收结构。

|  |
| --- |
| #include <pthread.h>  int pthread\_rwlockattr\_init (pthread\_rwlockattr\_t \*attr);  int pthread\_rwlockattr\_destroy (pthread\_rwlockattr\_t \*attr);  两者的返回值都是：若成功则返回0，否则返回错误编号 |

读写锁支持的唯一属性是进程共享属性，该属性与互斥量的进程共享属性相同。就像互斥量的进程共享属性一样，用一对函数来读取和设置读写锁的进程共享属性。

|  |
| --- |
| #include <pthread.h>  int pthread\_rwlockattr\_getpshared(const pthread\_rwlockattr\_t \*restrict attr,  int \* restrict pshared);  int pthread\_rwlockattr\_setpshared(pthread rwlockattr\_t \*attr,  int pshared);  两者的返回值都是：若成功则返回0，否则返回错误编号 |

虽然LINUX/UNIX只定义了一个读写锁属性，但不同平台的实现可以自由地定义额外的、非标准的属性。

### 3.条件变量属性

条件变量也有属性。与互斥量和读写锁类似，有一对函数用于初始化和回收条件变量属性。

|  |
| --- |
| #include <pthread.h>  int pthread\_condattr\_init(pthread\_condattr\_t \*);  int pthread\_condattr\_destroy (pthread\_condattr\_t \*attr);  两者的返回值都是：若成功则返回0，否则返回错误编号 |

与其他的同步原语一样，条件变量支持进程共享属性。

|  |
| --- |
| #include <pthread.h>  int pthread\_condattr\_getpshared(const pthread\_condattr t \*restrict attr,  int \* restrict pshared);  int pthread\_condattr\_setpshared (pthread\_condattr\_t \*attr,  int pshared);  两者的返回值都是：若成功则返回0，否则返回错误编号 |

## 12.4重入

在前面讨论了可重入函数和信号处理程序。在遇到重入问题时线程与信号处理程序类似。有了信号处理程序和线程，多个控制线程在同一时间可能潜在地调用同一个函数。

如果一个函数在同一时刻可以被多个线程安全地调用，就称该函数是线程安全的。一般来说除了下表列出的函数以外，LINUX/UNIX提供的其他函数都保证是线程安全的。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| LINUX/UNIX中不能保证线程安全的函数 | | | | |
| asctime | ecvt | gethostent | getutxline | putc\_unlocked |
| basename | encrypt | getlogin | gmtime | putchar\_unlocked |
| catgets | endgrent | getnetbyaddr | hcreate | putenv |
| crypt | endpwent | getnetbyname | hdestroy | pututxline |
| ctime | endutxent | getnetent | hsearch | rand |
| dbm\_clearerr | fcvt | getopt | inet\_ntoa | readdir |
| dbm\_close | ftw | getprotobyname | 164a | setenv |
| dbm\_delete | gcvt | getprotobynuraber | 1gamma | setgrent |
| dbm\_error | getc\_unlocked | getprotoent | 1gammaf | setkey |
| dbm\_fetch | getchar\_unlocked | getpwent | lgammal | setpwent |
| dbm\_firstkey | getdate | getpwnam | localeconv | setutxent |
| dbm\_nextkey | getenv | getpwuid | localtime | strerror |
| dbm\_open | getgrent | getservbyname | lrand48 | strtok |
| dbm\_store | getgrgid | get servbyport | mrand48 | ttyname |
| dirname | getgrnam | getservent | nftw | unsetenv |
| dlerror | gethostbyaddr | getutxent | nl\_langinfo | wcstombs |
| drand48 | gethostbyname | getutxid | ptsname | wctomb |

另外，ctermid和tmpnam函数在参数传入空指针时并不能保证是线程安全的。 类似地，wcrtomb和wcsrtombs函数如果参数mbstate\_t传入的是空指针的话，也不能保证它们是线程安全的。

操作系统实现支持线程安全函数这一特性时，对其中的一些非线程安全函数，它会提供可替代的线程安全版本，下表列出了这些函数的线程安全版本。很多函数并不是线程安全的，因为他们返回的数据是存放在静态的内存缓冲区中。通过修改接口，要求调用者自己提供缓冲区可以使函数变为线程安全的。

|  |  |
| --- | --- |
| 替代的线程安全函数 | |
| acstime\_r | gmtime\_r |
| ctime\_r | localtime\_r |
| getgrgid\_r | rand\_r |
| getgrnam\_r | readdir\_r |
| getlogin\_r | strerror\_r |
| getpwnam\_r | strtok\_r |
| getpwuid\_r | ttyname\_r |

下表中列出的函数的命名方式与他们的非线程安全版本的名字相似，只不过在名字最后加了\_r,以表明这个版本是可重入的。

如果一个函数对多个线程来说是可重入的，则说这个函数是线程安全的，但这并不能说明对信号处理程序来说该函数也是可重入的。如果函数对异步信号处理程序的重入是安全的，那么就可以说函数是异步-信号安全的。

LINUX/UNIX还提供了以线程安全的方式管理FILE对象的方法。可以使用flockfile和ftrylockfile获取与给定FILE对象关联的锁。这个锁是递归的，当占有这把锁的时候，还可以再次获取该锁，这并不会导致死锁。虽然这种锁的具体实现并无规定，但要求所有操作FILE对象的标准I/O例程表现得就像它们内部调用了flockf ile和funlockfile一样。

|  |
| --- |
| #include <stdio.h>  int ftrylockfile (FILE \*fp);  返回值：若成功则返回0，否则返回非0值  void flockf ile (FILE \*fp);  void funlockf ile (FILE \*fp); |

虽然标准的I/O例程从它们各自的内部数据结构这一角度出发，可能是以线程安全的方式实现的，但有时把锁开放给应用程序仍然是非常有用的。这允许应用程序把多个对标准I/O函数的调用组合成原子序列。当然，在处理多个file对象时，需要注意可能出现的死锁，并且需要对所有的锁仔细地排序。

如果标准I/O例程都获取它们各自的锁，那么在做一次一个字符的I/O操作时性能就会出现 严重的下降。在这种情况下，需要对每一个字符的读或写操作进行获取锁和释放锁的动作。为了避免这种开销，出现了不加锁版本的基于字符的标准I/O例程。

|  |
| --- |
| #include <stdio.h>  int getchar\_unlocked(void>;  int getc\_unlocked (FILE \*fp);  两者的返回值都是：若成功则返回下一个字符，若已到达文件结尾或出错则返回EOF  int putchar\_unlocked(int c);  int putc\_unlocked(int c, FILE \*fp);  两者的返回值都是：若成功则返回c，若出错则返回EOF |

除非被flockfile（或ftrylockfile）和funlockfile的调用包围，否则尽量不要调用这四个函数，因为它们会导致不可预期的结果（即由多个控制线程非同步地访问数据所引 起的种种问题）。

一旦对FILE对象进行加锁，就可以在释放锁之前对这些函数进行多次调用。这样就可以在多次的数据读写上分摊总的加解锁的开销。

下面程序给出了getenv的一个可能的实现。因为所有调用getenv的线程返回的字符串都存放在同一个静态缓冲区中，所以这个版本不是可重入的。如果两个线程同时调用这个函数，就会看到不一致的结果。

|  |
| --- |
|  |

下面给出getenv的可重入版本，这个版本命名为getenv\_r。它pthread\_once函数（后面会讨论）来确保每个进程只调用一次thread\_init函数。

|  |
| --- |
|  |

要使getenv\_r重入，需要改变接口，调用者必须自己提供缓冲区，这样每个线程可以使用各自不同的缓冲区从而避免其他线程的干扰。但是注意这还不足以使为线程安全的，要使getenv\_r成为线程安全的，需要在搜索请求的字符串时保护环境不被修改。我们可以使用互斥量，通过getenv\_r和putenv函数对环境列表的访问进行序列化。

可以使用读写锁，从而允许对getenv\_r多次并发访问，但并发性的增强可能并不会在很大程度上改善程序的性能。这里面有两个原因：首先，环境列表通常不会很长，所以扫描列表时并不需要长时间地占有互斥量；其次，对getenv和putenv的调用不是频繁发生的，所以改善它们的性能并不会对程序的整体性能产生很大的影响。

即使把成线程安全的，也并不意味着它对信号处理程序是可重入的。如果使用的是非递归的互斥量，当线程从信号处理程序中调用getenv\_r时，就有可能出现死锁。如果信号处理程序在线程执行getenv\_r中断了该线程，由于这时已经占有加锁的env\_mutex，这样其他线程试图对这个互斥量的加锁就会被阻塞，最终导致线程进入死锁状态。所以，必须使用递归互斥量阻止其他线程改变当前正査看的数据结构，同时还要阻止来自信号处理程序的死锁。问题是pthread函数并不保证是异步信号安全的，所以不能把pthread函数用于其他函数，让该函数成为异步信号安全的。

## 12.5线程私有数据

线程私有数据（也称线程特定数据）是存储和査询与某个线程相关的数据的一种机制。把这种数据称为线程私有数据或线程特定数据的原因是，希望每个线程可以独立地访问数据副本，而不需要担心与其他线程的同步访问问题。

线程模型促进了进程中数据和属性的共享，许多人在设计线程模型时会遇到各种麻烦。但在这样的模型中，为什么还要提出一些合适的用于阻止共享的接口呢？其中有两个原因：

第一，有时候需要维护基于每个线程的数据。因为线程ID并不能保证是小而连续的整数，所以不能简单地分配一个线程数据数组，用线程ID作为数组的索引。即使线程ID确实是小而连续的整数，可能还希望有一些额外的保护，以防止某个线程的数据与其他线程的数据相混淆。

采用线程私有数据的第二个原因是：它提供了让基于进程的接口适应多线程环境的机制。一个很明显的实例就是errno。回忆前面对errno的讨论，（线程出现）以前的接口把 errno 定义为进程环境中全局可访问的整数。系统调用和库例程在调用或执行失败时设置errno，把它作为操作失败时的附属结果。为了让线程也能够使用那些原本基于进程的系统调用和库例程，errno被重新定义为线程私有数据。这样，一个线程做了设置errno的操作并不会影响进程中其他线程的errno值。

回顾前面所学可知，进程中的所有线程都可以访问进程的整个地址空间。除了使用寄存器以外，线程没有办法阻止其他线程访问它的数据，线程私有数据也不例外。虽然底层的实现部分并不能阻止这种访问能力，但管理线程私有数据的函数可以提髙线程间的数据独立性。

在分配线程私有数据之前，需要创建与该数据关联的键。这个键将用于获取对线程私有数据的访问权。使用pthread\_key\_create创建一个键。

|  |
| --- |
| #include <pthread.h>  int pthread\_key\_create(pthread key t \*, void (\*destructor) (void \*));  返回值：若成功则返回0，否则返回错误编号 |

创建的键存放在指向的内存单元，这个键可以被进程中的所有线程使用，但每个线程把这个键与不同的线程私有数据地址关联。创建新键时，每个线程的数据地址设为null值。

除了创建键以外，pthread\_key\_create可以选择为该键关联析构函数，当线程退出时， 如果数据地址已经被置为非null数值，那么析构函数就会被调用，它唯一的参数就是该数据地址。如果传入的destructor参数为null，就表明没有析构函数与键关联。当线程调用pthread\_exit或者线程执行返回，正常退出时，析构函数就会被调用，但如果线程调用了exit、 \_exit、\_Exit、abort或出现其他非正常的退出时，就不会调用析构函数。

线程通常使用malloc为线程私有数据分配内存空间，析构函数通常释放已分配的内存。如果线程没有释放内存就退出了，那么这块内存将会丢失，即线程所属进程出现了内存泄漏。

线程可以为线程私有数据分配多个键，每个键都可以有一个析构函数与它关联。各个键的析构函数可以互不相同，当然它们也可以使用相同的析构函数。每个操作系统在实现的时候可以对进程可分配的键的数量进行限制。

线程退出时，线程私有数据的析构函数将按照操作系统实现中定义的顺序被调用。析构函数可能会调用另一个函数，该函数可能会创建新的线程私有数据而且把这个数据与当前的键关联起来。当所有的析构函数都调用完成以后，系统会检査是否还有非null的线程私有数据值与键关联，如果有的话，再次调用析构函数。这个过程将会一直重复直到线程所有的键都为null 值线程私有数据，或者已经做了PTHREAD\_DESTRUCTOR\_ITERATIONS 中定义的最大次数的尝试。

对所有的线程，都可以通过调用pthread\_key\_delete来取消键与线程私有数据值之间的关联关系。

|  |
| --- |
| #include <pthread.h>  int pthread\_key\_delete(pthread\_key\_t \*key);  返回值：若成功则返回0，否则返回错误编号 |

注意调用pthread\_key\_delete并不会激活与键关联的析构函数。要释放任何与键对应的线程私有数据值的内存空间，需要在应用程序中采取额外的步骤。

需要确保分配的键并不会由于在初始化阶段的竞争而发生变动。下列代码可以导致两个线程都调用 pthread\_key\_create1：

|  |
| --- |
| void destructor(void \*);  pthread\_key\_t key;  int init\_done = 0;  int threadfunc(void \*arg)  {  if (!init\_done)  {  init\_done = 1;  err = pthread\_key\_create(&key, destructor);  }  …  } |

有些线程可能看到某个键值，而其他的线程看到的可能是另一个不同的键值，这取决于系统是如何调度线程的，解决这种竞争的办法是使用pthread\_once。

|  |
| --- |
| #include <pthread.h>  pthread\_once\_t initflag = PTHREAD\_ONCE\_INIT;  int pthread\_once(pthread\_once\_t \* initflag, void(\*initfn)(void));  返回值：若成功则返回0，否则返回错误编号 |

initflag必须是一个非本地变量（即全局变量或静态变量），而且必须初始化为PTHREAD. 0NCE\_INIT。

如果每个线程都调用pthread\_once，系统就能保证初始化例程initfn只被调用一次，即 在系统首次调用pthread\_once时。创建键时避免出现竞争的一个恰当的方法可以描述如下：

|  |
| --- |
| void destructor(void \*)；  pthread\_key\_t key；  pthread \_once\_t init\_done = PTHREAD\_0NCE\_INIT;  void thread\_init (void)  {  err = pthread\_key\_create(&key, destructor);  }  int threadfunc(void \*arg)  {  pthread\_once(&init\_done, thread\_init);  …  } |

键一旦创建，就可以通过调用pthread\_setspecific函数把键和线程私有数据关联起来。可以通过pthread\_getspecific函数获得线程私有数据的地址。

|  |
| --- |
| #include <pthread.h>  void \*pthread\_getspecific (pthread\_key\_t key);  返回值：线程私有数据值；若没有值与键关联则返回NULL  int pthread\_setspecific(pthread\_key\_t key, const void \* value);  返回值：若成功则返回0，否则返回错误编号 |

如果没有线程私有数据值与键关联，pthread\_getspecific将返回一个空指针，可以据此来确定是否需要调用pthread\_setspecific。

前面给出了getenv的两种版本。但是如果无法修改应用程序以直接使用新的接口会出现什么问题呢？这种情况下，可以使用线程私有数据来维护每个线程的数据缓冲区的副本，用于存放各自的返回字符串，代码如下。

|  |
| --- |
|  |

使用pthread\_once来确保只为将要使用的线程私有数据创建了一个键。如果pthread\_getspecific返回的是空指针，需要分配内存然后把键与该内存单元关联，否则如果返回的不是空指针，就使用pthread\_getspecific返回的内存单元。对析构函数，使用free来释放之前由malloc分配的内存。只有当线程私有数据值为非null时，析构函数才会被调用。

注意虽然这个版本的getenv是线程安全的，但它并不是异步-信号安全的。对信号处理程序而言，即使使用递归的互斥量，这个版本的getenv也不可能是可重入的，因为它调用了malloc，而malloc函数本身并不是异步-信号安全的。

## 12.6取消选项

有两个线程属性并没有包含在pthread\_attr\_t结构中，它们是可取消状态和可取消类型。这两个属性影响着线程在响应pthread\_cancel函数调用时所呈现的行为。

可取消状态属性可以是PTHREAD\_CANCEL\_ENABLE或PTHREAD\_CANCEL\_DISABLE。线程可以通过调用pthread\_setcancelstate修改它的可取消状态。

|  |
| --- |
| #include <pthread.h>  int pthread\_setcancelstate(int state, int \*oldstate);  返回值：若成功则返回0，否则返回错误编号 |

pthread\_setcancelstate把当前的可取消状态设置为state，把原来的可取消状态存放在由oldstate指向的内存单元中，这两步是原子操作。

pthread\_cancel调用并不等待线程终止，在默认情况下，线程在取消请求发出以后还是继续运行，直到线程到达某个取消点。取消点是线程检査是否被取消并按照请求进行动作的一个位置。LINUX/UNIX保证在线程调用下表中列出的任何函数时，取消点都会出现。

|  |  |  |  |
| --- | --- | --- | --- |
| LINUX/UNIX定义的取消点 | | | |
| accept | mq\_timedsend | putpmsg | sigsuspend |
| aio\_siispend | msgrcv | pwrite | sigtimedwait |
| clock\_nanosleep | msgsnd | read | sigwait |
| close | msync | readv | sigwaitinfo |
| connect | nanosleep | recv | sleep |
| creat | open | recvfrom | system |
| fcntl2 | pause | recvmsg | tcdrain |
| fsync | poll | select | usleep |
| getmsg | pread | sem\_timedwait | wait |
| getpmsg | pthread\_cond\_timedwait | sem\_wait | waitid |
| lockf | pthread\_cond\_wait | send | waitpid |
| mq\_receive | pthread\_join | sendmsg | write |
| mq\_send | pthread\_testcancel | sendto | writev |
| mq\_timedreceive | putmsg | sigpause |  |

线程启动时默认的可取消状态是PTHREAD\_CANCEL\_ENABLE。当状态设为PTHREAD\_CANCEL\_DISABLE时，对pthread\_cancel的调用并不会杀死线程；相反，取消请求对这个线程来说处于未决状态。当取消状态再次变为PTHREAD\_CANCEL\_ENABLE时，线程将在下一个取消点上对所有未决的取消请求进行处理。

如果应用程序在很长一段时间内都不会调用到取消点函数，那么可以调用pthread\_testcancel函数在程序中自己添加取消点。

|  |
| --- |
| #include <pthread.h>  void pthread\_testcancel(void); |

调用pthread\_testcancel时，如果有某个取消请求正处于未决状态，而且取消并没有置为无效，那么线程就会被取消。但是如果取消被置为无效时，pthread\_testcancel调用就没有任何效果。

这里所描述的默认取消类型也称为延迟取消。调用pthread\_cancel以后，在线程到达取消点之前，并不会出现真正的取消。可以通过调用pthread\_setcanceltype来修改取消类型。

|  |
| --- |
| #include <pthread.h>  int pthread\_set cancel type (int type, int \* old type)；  返回值：若成功则返回0，否则返回错误编号 |

type参数可以是PTHREAD\_CANCEL\_DEFERRED或PTHREAD\_CANCEL\_ASYNCHRONOUS,，pthread\_setcanceltype函数把取消类型设置为type，把原来的取消类型返回到oldtype指向的整型单元。

异步取消与延迟取消不同，使用异步取消时，线程可以在任意时间取消，而不是非得遇到取消点才能被取消。

## 12.7线程和信号

即使是在基于进程的编程模式中，信号的处理也可能是很复杂的。把线程引入编程范型，就使信号的处理变得更加复杂。

每个线程都有自己的信号屏蔽字，但是信号的处理是进程中所有线程共享的。这意味着尽管单个线程可以阻止某些信号，但当线程修改了与某个信号相关的处理行为以后，所有的线程都必须共享这个处理行为的改变。这样如果一个线程选择忽略某个信号，而其他的线程可以恢复信号的默认处理行为，或者为信号设置一个新的处理程序，从而可以撒销上述线程的信号选择。

进程中的信号是递送到单个线程的。如果信号与硬件故障或计时器超时相关，该信号就被发送到引起该事件的线程中去，而其他的信号则被发送到任意一个线程。

sigprocmask的行为在多线程的进程中并没有定义，线程必须使用pthread\_sigmask。

|  |
| --- |
| #include <signal.h>  int pthread\_sigmask (int how, const sigset\_t \*restrict set, sigset\_t \*restrict oset);  返回值：若成功则返回0，否则返回错误编号 |

pthread\_sigmask函数与sigprocmask函数基本相同，除了pthread\_sigmask工作在线程中，并且失败时返回错误码，而不像sigprocmask中那样设置errno并返回-1。

线程可以通过调用sigwait等待一个或多个信号发生。

|  |
| --- |
| #include <signal.h>  int sigwait (const sigset\_t \*restrict set, mt \*restrict signop);  返回值：若成功则返回0，否则返回错误编号 |

set参数指出了线程等待的信号集，signop指向的整数将作为返回值，表明发送信号的数量。

如果信号集中的某个信号在sigwait调用的时候处于未决状态，那么sigwait将无阻塞地返回，在返回之前，sigwait将从进程中移除那些处于未决状态的信号。为了避免错误动作发生，线程在调用sigwait之前，必须阻塞那些它正在等待的信号。sigwait函数会自动取消信号集的阻塞状态，直到有新的信号被递送。在返回之前，sigwait将恢复线程的信号屏蔽字。如果信号在sigwait调用的时候没有被阻塞，在完成对sigwait调用之前会出现一个时间窗，在这个窗口期，某个信号可能在线程完成sigwait调用之前就被递送了。

使用sigwait的好处在于它可以简化信号处理，允许把异步产生的信号用同步的方式处理。为了防止信号中断线程，可以把信号加到每个线程的信号屏蔽字中，然后安排专用线程作信号处理。这些专用线程可以进行函数调用，不需要担心在信号处理程序中调用哪些函数是安全的，因为这些函数调用来自正常的线程环境，而非传统的信号处理程序，传统信号处理程序通常会中断线程的正常执行。

如果多个线程在sigwait调用时，等待的是同一个信号，这时就会出现线程阻塞。当信号递送的时候，只有一个线程可以从sigwait中返回。如果信号被捕获（例如进程通过使用sigaction建立了一个信号处理程序），而且线程正在sigwait调用中等待同一信号，那么这时将由操作系统实现来决定以何种方式递送信号。在这种情况下，操作系统实现可以让sigwait返回，也可以激活信号处理程序，但不可能出现两者皆可的情况。

要把信号发送到进程，可以调用kill。要把信号发送到线程，可以调用pthread\_kill。

|  |
| --- |
| #include <signal.h>  int pthread\_kill (pthread\_t thread, int signo);  返回值：若成功则返回0，否则返回错误编号 |

可以传一个0值的来检查线程是否存在。如果信号的默认处理动作是终止该进程，那么把信号传递给某个线程仍然会杀掉整个进程。

注意闹钟定时器是进程资源，并且所有的线程共享相同的alarm。所以进程中的多个线程不可能互不干扰（或互不合作）地使用闹钟定时器。

回顾前面的程序，等待信号处理程序设置标志，从而表明主程序应该退出。唯一可运行的控制线程就是主线程和信号处理程序，所以阻塞信号足以避免错失标志修改。在线程中，需要使用互斥量来保护标志。程序如下。

|  |
| --- |
|  |

这里并不让信号处理程序中断主控线程，而是由专门的独立控制线程进行信号处理。改动quitflag的值是在互斥量的保护下进行的，这样主控线程不会在调用pthread\_cond\_signal时错失唤醒调用。在主控线程中使用相同的互斥量来检查标志的值，并且原子地释放斥量，等待条件的发生。

注意在主线程开始时阻塞SIGINT和SIGQUIT。当创建线程进行信号处理时，新建线程继承了现有的信号屏蔽字。因为sigwait会解除信号的阻塞状态，所以只有一个线程可以用于信号的接收。这使得对主线程进行编码时不必担心来自这些信号的中断。

运行这个程序可以得到与前面程序类似的输出结果：

|  |
| --- |
|  |

Linux线程是以独立进程实现的，使用clone（2）共享资源。因此Linux线程在遇到信号时的行为与其他UNIX操作系统实现不同。在POSIX.1线程模型中，异步信号被发送到进程以后，进程中当前没有阻塞该信号的某个线程就被选中，接收信号。在Linux上，异步信号发送到特定的线程，而且因为每个线程是作为独立的进程执行的，系统就不能选择当前没有阻塞该信号的线程。这样一来，结果就是线程可能不会注意到该信号。因此，如果信号产生于终端驱动程序，这样的信号是通知到进程组的，上面的程序就可以工作；但是如果试图用kill把信号发送给进程时，在Linux上就不能如预期的那样工作。

## 12.8线程和fork

当线程调用fork时，就为子进程创建了整个进程地址空间的副本。前面讨论的写时复制，子进程与父进程是完全不同的进程，只要两者都没有对内存做出改动，父进程和子进程之间还可以共享内存页的副本。

子进程通过继承整个地址空间的副本，也从父进程那里继承了所有互斥量、读写锁和条件变量的状态。如果父进程包含多个线程，子进程在fork返回以后，如果紧接着不是马上调用exec的话，就需要清理锁状态。

在子进程内部只存在一个线程，它是由父进程中调用fork的线程的副本构成的。如果父进程中的线程占有锁，子进程同样占有这些锁。问题是子进程并不包含占有锁的线程的副本，所以子进程没有办法知道它占有了哪些锁并且需要释放哪些锁。

如果子进程从fork返回以后马上调用某个exec函数，就可以避免这样的问题。这种情况下，老的地址空间被丢弃，所以锁的状态无关紧要。但如果子进程需要继续做处理工作的话，这种方法就行不通，还需要使用其他的策略。

要清除锁状态，可以通过调用pthread\_atfork函数建立fork处理程序。

|  |
| --- |
| #include <pthread.h>  int pthread\_atfork (void (\*prepare) (void) , void (\*parent) (void), void (\*child) (void));  返回值：若成功则返回0，否则返回错误编号 |

用pthread\_atfork函数最多可以安装三个帮助清理锁的函数。prepare fork处理程序由父进程在fork创建子进程前调用，这个fork处理程序的任务是获取父进程定义的所有锁。parent fork处理程序是在fork创建了子进程以后，但在fork返回之前在父进程环境中调用的，这个fork处理程序的任务是对fork处理程序获得的所有锁进行解锁。child fork处理程序在fork返回之前在子进程环境中调用，与parcnf fork处理程序一样，c/u\_W fork处理程序也必须释放prepare fork处理程序获得的所有锁。

注意不会出现加锁一次解锁两次的情况，虽然看起来也许会出现。当子进程地址空间创建的时候，它得到了父进程定义的所有锁的副本。因为prepare fork处理程序获取所有的锁，父进程中的内存和子进程中的内存内容在开始的时候是相同的。当父进程和子进程对他们的锁的副本进行解锁的时候，新的内存是分配给子进程的，父进程的内存内容被复制到子进程的内存中（写时复制），所以就会陷入这样的假象，看起来父进程对它所有的锁的副本进行了加锁，子进程对它所有的锁的副本进行了加锁。父进程和子进程对在不同内存位置的重复的锁都进行了解锁操作，就好像出现了下列的事件序列：

* 父进程获得所有的锁。
* 子进程获得所有的锁。
* 父进程释放它的锁。
* 子进程释放它的锁。

可以多次调用pthread\_atfork函数从而设置多套fork处理程序。如果不需要使用其中某个处理程序，可以给特定的处理程序参数传入空指针，这样它就不会起任何作用。使用多个fork处理程序时，处理程序的调用顺序并不相同。pamiJ和c/h\_W fork处理程序是以它们注册时的顺序进行调用的，而prepare fork处理程序的调用顺序与它们注册时的顺序相反。这样可以允许多个模块注册它们自己的fork处理程序，并且保持锁的层次。

例如，假设模块A调用模块B中的函数，而且每个模块有自己的一套锁。如果锁的层次是A在B之前，模块B必须在模块A之前设置fork处理程序。当父进程调用fork时，就会执行以下的步骤，假设子进程在父进程之前运行。

(1) 调用模块A的prepare fork处理程序获取模块A的所有锁。

(2) 调用模块B的prepare fork处理程序获取模块B的所有锁。

(3) 创建子进程。

(4) 调用模块B中的child fork处理程序释放子进程中模块B的所有锁。

(5) 调用模块A中的child fork处理程序释放子进程中模块A的所有锁。

(6) fork函数返回到子进程。

(7) 调用模块B中的parent fork处理程序释放父进程中模块B的所有锁。

(8) 调用模块A中的parent fork处理程序来释放父进程中模块A的所有锁。

(9) fork函数返回到父进程。

如果fork处理程序是为了清理锁状态，那么又由谁来负责清理条件变量的状态呢？在有些操作系统的实现中，条件变量可能并不需要做任何清理。但是有些操作系统实现把锁作为条件变量实现的一部分，这种情况下的条件变量就需要清理。问题是目前不存在这样的接口允许做锁的清理工作，如果锁是嵌入到条件变量的数据结构中的，那么在调用fork之后就不能使用条件变量，因为还没有可移植的方法对锁进行状态清理。另外，如果操作系统的实现是使用全局锁保护进程中所有的条件变量数据结构，那么操作系统实现本身可以在fork库例程中做清理锁的工作，但是应用程序不应该依赖操作系统实现中这样的细节。

下面的程序描述了如何使用pthread\_atfork和fork处理程序。

|  |
| --- |
|  |

程序中定义了两个互斥量，lock1和lock2，prepare fork处理程序获取这两把锁，child fork处理程序在子进程环境中释放锁，parent fork处理程序在父进程中释放锁。

运行该程序，得到如下输出：

|  |
| --- |
|  |

可以看出，prepare fork处理程序在调用fork以后运行，child fork处理程序在fork调用返回到子进程之前运行，parent fork处理程序在fork调用返回给父进程前运行。

## 12.9线程和I/O

在讨论文件的时候介绍了pread和pwrite函数，这些函数在多线程环境下是非常有帮助的，因为进程中的所有线程共享相同的文件描述符。

考虑两个线程，在同一时间对同一个文件描述符进行读写操作。

|  |  |
| --- | --- |
| 线程A | 线程B |
| lseek(fd, 300, SEEK\_SET);  read(fd, buf1, 100); | lseek(fd, 700, SEEK\_SET);  read(fd, buf2, 100); |

如果线程A执行lseek，然后线程B在线程A调用read之前调用lseek，那么两个线程最终会读取同一条记录。很显然这不是我们希望的。

为了解决这个问题，可以使用pread,使偏移量的设定和数据的读取成为一个原子操作。

|  |  |
| --- | --- |
| 线程A | 线程B |
| pread(fd, buff1, 100, 300); | pread(fd, buf2, 100, 700); |

使用pread可以确保线程A读取偏移量为300的记录，而线程B读取偏移量为700的记录。可以使用pwrite来解决并发线程对同一文件进行写操作的问题。

## 12.10小结

在LINUX/UNIX系统中，线程提供了分解并发任务的一种替代模型。线程促进了独立控制线程之间的共享，但也带来了它特有的同步问题。本章中，我们考查了如何调整线程和它们的同步原语，讨论了线程的可重入性，还学习了线程如何与其他面向进程的系统调用进行交互。