# 理解Linux线程

A computer is a state machine. Threads are for people who cannot program state machine.

-Alan Cox

古代神话小说中，很多妖怪都三头六臂，体现了古人的想象力，因为三头六臂，相当于同时可以调动更多的肢体投入战斗，从而战斗力加倍的效果。三头六臂的想法，折射到计算机上，就是多线程的思想。

在单CPU单Core的时代，纵然有多个线程，由于核数有限，也无法让效率加倍。2005年，Herb Sutter 发表了*The Free Lunch is over*一文，提到，传统的处理器厂商改善CPU性能的传统方法，如提升时钟速度和指令吞吐量，基本已经走到了尽头，处理器开始向超线程和多核架构靠拢。随着个人计算机和服务器核数越来越多，可以说多核的时代已来临，多线程编程的能力对程序员越来越重要。

当有多个CPU可用，就意味着有更多的计算资源可以调度，这是个时候，多线程就是一种选择了。对于单CPU单核而言，多线程完全失去了意义。有句名言说1个女人怀胎十月，能生出一个baby，但是10个女人也无法讲这个时间缩短成1个月。有位“缺德”的网友笑言，虽然无法将生出一个baby的时间缩短成1个月，但是如果有10个女人，同时怀孕，10个月后就可以得到10个baby。网友的笑话，反映了多线程的力量，既然有了更多CPU资源，利用多线程，就有可能在单位时间内完成更多的工作。

但是又不能简单地说，选择N个线程，必然会带来效能的N倍提升，这又是不对的。简单资源的累加，不一定带来效率的等比例提升。在真实的计算机世界中，情况可能复杂的读，比如我们的多线程是IO密集型的线程，CPU不是性能的瓶颈，可能我们多个线程都在等待DISK I/O的完成，对于只有一个存储设备的情况，这种情况尤为严重，除非每个线程的IO对应的存储设备不同，才能一定程度的缓解这种状况。多个线程的任务，某些操作不能并发，必须互相隔离，串行完成，这也会影响多线程程序的效率。

## 9.1线程与进程

线程和进程类似，都是允许应用程序并发执行多个任务的一种机制。以比较流行的数据库为例，MySQL采用的是多线程的架构，而PostgreSQL采用的是多进程的架构。

一个进程可能包含多个线程，传统意义上的进程，不过是多线程的一种特例，即该进程只包含一个线程。同一个进程的多个线程共享一份全局内存区域，包括初始化数据段，未初始化数据段和动态分配的堆内存段。这种全局内存的共享，使得多线程之间的数据共享非常的简单。我们简单定义全局变量，就能让多个线程都可以操作对应的数据。进程则不然，进程之间数据的并没有分享，进程之间必须采用一些进程间通信（inter-process communication）的手段,如消息队列,共享内存,信号量等，才能达到信息交换的目的。当然，没有免费的午餐，多线程共享一份地址空间和其他相关的属性，造成多线程一定程度的脆弱，因为只要有一个线程存在bug，比如访问了非法的地址空间，触发段错误（segment fault），就会危及该进程的所有线程。所以从这方面想，进程间的隔离做的更彻底,多进程的程序健壮性更好一些。

线程是CPU调度和分派的基本单位，同一个进程的多个线程可能在多个CPU上并发执行。前面提到多个线多个线程并发执行也是多线程编程的难点所在程共享一份全局内存数据，多个线程又可以并发执行，这就造成了，多个线程可能同时操作一份数据，从而造成数据的不一致。因此多线程引入了互斥量（mutex）、读写锁和条件变量（condition variable），来协调或者同步多线程的操作。线程之间需要同步的操作越多，表明彼此之间独立性越差，多个进程带来的效能的收益就越低。所以写多线程程序的时候，我们需要合理规划各个线程的任务，让线程之间的依赖或者干扰尽可能的减少，最大程度的发挥多线程的优势。多线程程序的关键，在于各个线程任务的分割。各个线程任务分配的合理，多线程程序就成功了一大半。各个线程任务分配不合理，线程之间互相牵制，互相依赖，互相干扰，必然造成逻辑复杂，效率低下，出现问题难以调试等弊端。如果存在这些弊端，哪怕后期在代码层面上的优化，消除bug，也无力回天，根本解决这些问题。

线程,在Linux操作系统中又被称为轻量级进程（Light Weight Process ,LWP) ，顾名思义，在消耗计算机资源方面,线程要比进程廉价不少。创建进程的代价要高于创建线程。Linux提供了fork()来创建进程，尽管现代Linux采用了写时复制（Copy-On-Write）技术，已经降低了开销，但是文件描述符和内存页表等依然需要复制，fork调用在时间上的开销不菲。而创建线程的开销就远远小于创建进程。有实验证明，创建线程比创建进程通常要快10倍甚至更多。虚拟内存总量越大，这个速度的差距就越大。原因就在于fork创建进程时需要拷贝的诸多属性，在线程间本来就是共享的，无需拷贝。除了创建以外，线程间的上下文切换的速度，消耗的时间要比不同进程之间切换要短。

多线程程序和多进程程序比较，对程序员的要求更高,需要程序员更加的小心。

首先，互斥量的存在有可能引入死锁（dead lock）：如果两个线程彼此持有一个互斥量而请求对方的互斥量，死锁就发生了。这是一种容易理解的死锁，程序员似乎不会犯这么愚蠢的错误。但是当我们线程数目比较多，互斥量数目也不少，防范和排查死锁问题就没有这么简单了。我们必须遵循一定的编程规范，才能防止死锁的发生。

其次，当线程拥有不同的优先级的时候，情况就会更加复杂，有可能带来优先级反转的问题,也就是较低优先级的线程不断执行,而高优先级的线程获取不到CPU资源,被饿死现象。一个比较有意思的案例是1997年火星探路者号由于优先级反转引发系统不断复位。

最后，时序问题。多个线程可能在多个CPU上同时执行，这决定了我们可能没有办法通过测试穷尽所有的时序组合，所以写多线程程序的时候，一定要小心。线程化的程序永远不能依赖巧合的时序来工作，它必须工作于所有可能的时序之下。

## 9.2 pthread库线程的实现

在信号与线程的关系一节，我们提到了POSIX标准对线程的信号机制的要求，POSIX标准对线程的要求有：

1. 查看进程列表时，相关的一组线程应该被展现成列表的一个节点
2. 发给进程（或者叫线程组更为恰当）的信号，将被同一个进程下的所有线程共享，并且被其中任何一个处理
3. 发给特定线程的信号，将被进程下的特定线程接收，只能由该线程处理
4. 当进程收到信号被停止（SIGSTOP）或者被继续（SIGCONT）时，对应的所有线程的状态应该一起变化。
5. 当进程收到一个致命信号时，进程里的所有线程全部退出。

LinuxThreads线程是早期出现的Linux线程实现。对照上面POSIX提出的要求，LinuxThreads应该自惭形秽，因为LinuxThreads除了第五条，其余的要求都没有达到。

1. 对于同一个线程的线程调用getpid()会返回不同的值，一些版本的ps命令会显示进程中的所有线程，并且他们的进程号也不相同。
2. 因为LinuxThreads线程的进程ID不同，所以，kill()或者sigqueue()向某进程发送信号，只能将信号递送给特定的线程，而不是POSIX标准提到的由目标进程下不阻塞该信号的任意线程来接收并处理。
3. LinuxThreads不支持收到信号，整个进程挂起的概念。SIGSTOP和SIGCONT信号只对进程中的一个线程有效。

LinuxThreads勉强实现了第五条，就是线程出现异常或者收到致命信号的时候，所有线程集体退出，但是实现的并不优美。

LinuxThreads存在管理进程。通过pthread\_create创建10个线程的时候，你会发现出现了11个线程（不算主线程），原因就是它会首先创建一个管理线程。第一次调用pthread\_create的时候，LinuxThreads发现管理线程不存在，于是会首先创建管理线程，然后由管理进程负责创建线程。对于管理进程创建的线程，getppid会返回管理进程的进程ID。当线程退出的时候，会向管理线程发送信号，管理线程会在信号处理函数中判断子线程是否是正常退出，如果不是，杀死所有线程，然后自杀。

主线程是管理线程的父进程，它退出的时候并不给管理线程发送信号。那怎么实现因为主进程异常或者收到致命信号导致整个线程组集体退出呢？管理进程会在大循环中通过getppid查看父进程的ID，如果父进程的进程ID是1，表示主进程已经退出了，管理进程会杀死所有线程之后，然后自杀。如果主进程是调用pthread\_exit退出呢？因为按照POSIX标准，这种情况下，其他线程应该继续运行。Linux的主进程退出并不是真正的退出，它在阻塞等待所有子线程的退出，待它们都退出以后主线程才真正的退出。

看了这里我们发现，线程的创建和退出，我们可以很明显看出管理线程成了LinuxThreads机制的枢纽。其实枢纽有个别名叫瓶颈。管理线程就是LinuxThreads的性能瓶颈。更糟糕的是，如果管理线程被杀死，那么其余的线程只能手工清理。

LinuxThreads机制除了上面的提到缺点，还有很多不符合标准的地方[[1]](#footnote-1)。所以后来开发了NPTL库。

NPTL（Native POSIX Threads Library）是目前Linux采用的线程实现，该实现的第一目标就是遵循POSIX标准[[2]](#footnote-2)。由glibc的大神Ulrich Drepper和内核开发的大神Ingo Molnar开发。从开发者我们可以看出，NPTL的实现，不但有glibc层面的改动，在Linux内核层面，为了支持NPTL也做了很多的工作：

首先是tgid概念的引入。内核进程描述符task\_struct增加了tgid（thread group id），对于进程的第一个线程（我们称为主线程或者初始线程），内核会分配pid，同时tgid等于pid。对于我们用后面用pthread\_create创建的线程，内核依然会分配新的pid，但是tgid等于主线程的tgid。我们看出来了，内核中的pid，称之为线程ID更为合适，而tgid称之为进程ID或者线程组ID更合适。事实上，我们在应用层调用getpid系统调用，返回的是tgid，所以同一个线程组的所有线程返回的结果一致。如果我们想获得内核的pid，我们应该用gettid系统调用，每一个线程，gettid系统调用返回的值各不相同。

通过这一点，POSIX标准的第一条ps时一组线程显示成一个节点就满足了。当然了，如果我们调用ps shell命令时，添加L参数（L代表LWP，Light Weight Process），就会把线程组里的线程也会显示出来。我们依然可以看到更详细的信息，同一个线程组的多个线程都会列出：

PID LWP TTY TIME CMD

1262 1262 ? 00:00:00 ntop

1262 1263 ? 00:00:00 ntop

1262 1264 ? 00:00:00 ntop

1262 1265 ? 00:00:00 ntop

1262 1266 ? 00:00:00 ntop

1262 1269 ? 00:00:00 ntop

1262 1279 ? 00:00:00 ntop

1262 1280 ? 00:00:00 ntop

1262 1663 ? 00:00:00 ntop

我们可以看到ntop进程是个多线程程序，进程ID或者线程组ID是1262。1262线程是主线程，（1263，1264，1265，1266，1269，1279，1280，1663）这些线程都是后创建线程。对于多线程程序，我们也可以从procfs查到它的所有线程：

manu@manu-hacks:~$ ls /proc/1262/task/

1262 1263 1264 1265 1266 1269 1279 1280 1663

其次是符合标准的信号处理机制。在信号一章中我们提到了内核的task\_struct维护了两套signal\_pending，一套是线程组所有线程共享的挂起信号，来应对发给进程或者线程组的信号（如kill），另一套是线程私有的挂起信号，应对发给特定线程的信号（通过系统调用tkill）。

再次是引入了新的系统调用exit\_group，可以终止进程中的所有线程。我们在日常编程中调用glibc中的exit，本质是调用exit\_group系统调用。我们在第四章已经详细讲述了exit\_group的实现，在此不赘述。对于pthread\_exit()调用的是\_exit系统调用，仅仅退出调用pthread\_exit的线程。

第四是引入了futex（fast userspace mutex）作为一种同步的机制，后面我们会解释这个机制。

第五，第六，第七……

NPTL 做了大量的工作，对标准的遵守更加严格，这也是我们Linux使用的thread实现。

值得一提的是，还有一个NGPT（Next Generation POSIX Threads），这个当年是IBM的开发人员开发的，当年是和NPTL竞争的项目。随着NPTL推出，NGPT放弃了开发。有意思的是NGPT采用的是M:N模型，也就是说M个用户线程对应N个内核调度实体，这种机制最大的问题是复杂。NPTL最初也曾考虑这种模型，但是过于复杂，对内核的改动太大，最关键的是没有必要，所以最终采取了1:1的方案，也就是我们今天的一个用户线程对应内核一个调度实体的方案。

## 9.3 pthread库接口介绍

在20世纪80年代末期，90年代早期存在各种不同的线程接口，1995年，POSIX.1c对POSIX线程API进行了标准化，这就是我们今天看到的pthread库的接口。

这些接口包括线程的创建，线程的退出，线程的取消，线程的分离，连接已经终止的线程，互斥量，读写锁，线程的条件等待等。

表9.1POSIX线程库的接口

|  |  |
| --- | --- |
| **POSIX 函数** | **函数功能描述** |
| pthread\_create | 创建一个线程 |
| pthread\_exit | 线程退出 |
| pthread\_self | 获取线程ID |
| pthread\_equal | 检查两个线程ID是否相等 |
| pthread\_join | 等待线程退出 |
| pthread\_detach | 设置线程状态为分离状态 |
| pthread\_cancel | 线程的取消（下章介绍） |
| pthread\_cleanup\_push  pthread\_cleanup\_pop | 线程退出，清理函数注册和执行  （下章介绍） |

上面提到的函数列表，是pthread的基本接口。当然多线程的程序不一定用到上述的所有接口。接下来分别介绍这些接口

### 9.3.1线程的创建和标识

首当其冲的接口是创建线程的接口。程序开始启动的时候，产生的进程只有一个线程，我们称之为主线程或者初始线程。对于单线程的进程而言，只存在主线程一个线程。如果我们想在主线程之外，再创建一个或者多个线程，我们就需要用到pthread\_create接口了。

**9.3.1.1 pthread\_create接口介绍**

pthread库提供了接口来创建线程：

#include <pthread.h>

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

const pthread\_attr\_t \*restrict attr,

void \*(\*start\_routine)(void\*),

void \*restrict arg);

pthread\_create会将创建的线程的ID填入第一个pthread\_t类型的变量。pthread\_t是用户态的线程ID，用来标识线程，我们在下一小节详细介绍线程的标识。

第二个参数是pthread\_attr\_t类型，顾名思义，我们可以指定我们创建的线程有什么特殊的属性，比如我们可以设定新建线程栈的大小，设置线程的调度策略等。如果我们队线程没什么特殊的设定，传递空指针NULL即可。至于存在哪些属性我们会在下一章讲解。

第三个参数是线程需要执行的函数。我们创建线程，是为了让线程执行一定的任务。线程创建成功之后，该线程就会执行start\_routine函数。

第四个参数新建线程的入参，新建线程如果正常工作，可能需要入参，那么主线程在调用pthread\_create的时候，可以讲入参的信息放入第四个参数传递给新建线程。

如果成功，pthread\_create返回0，如果不成功，pthread\_create返回一个非0的错误码。常见的错误有：

表9.2 pthread\_create的错误码及描述

|  |  |
| --- | --- |
| **错误码** | **描述** |
| EAGAIN | 系统资源不够，或者创建线程数超过系统对一个进程中线程总数的限制。 |
| EINVAL | 第二个参数attr值不对 |
| EPERM | 没有合适的权限来设置调度策略或者参数 |

**9.3.1.2线程标识**

POSIX threads创建线程的pthread\_create接口，会将创建的线程的线程ID赋给pthread\_t类型的变量tid，返回回来。从理论上讲，pthread\_t是一个实现相关的数据类型。没有保证一定是一个数值或者是指针，所以从理论意义上讲，我们无法打印pthread\_t，更无法判断这个值是否是合法值。

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

#include <string.h>

#include <errno.h>

#include <sys/syscall.h>

#include <sys/types.h>

void \*\*getEBP( int dummy )

{

void \*\*ebp = (void \*\*)&dummy -2 ;

return \*ebp ;

}

void\* thread\_work(void\* param)

{

pid\_t pid = getpid();

fprintf(stdout,"for thread: pid = %d,address of pid = %p\n",

pid,&pid);

pthread\_t tid = pthread\_self();

fprintf(stdout,"for thread: pthread\_self return %p\n",

(void \*)tid);

int dummy = 0;

void\*\* ebp = getEBP(dummy);

fprintf(stdout,"for thread: stack top is %p\n",

ebp);

int TID = syscall(SYS\_gettid);

fprintf(stdout,"for thread: gettid return %d\n",

TID);

fprintf(stdout,"for thread(%d): I will exit now\n",

TID);

pthread\_exit(NULL);

return NULL;

}

int main(int argc ,char\* argv[])

{

pthread\_t tid = 0;

int ret = pthread\_create(&tid,NULL,thread\_work,NULL);

if(ret != 0)

{

fprintf(stderr,"create thread failed ,return %d (%s)\n",

ret,strerror(errno));

return -1;

}

pid\_t pid = getpid();

fprintf(stdout,"for main thread: pid = %d address of pid is %p\n",

pid,&pid);

int dummy = 0;

void \*\*ebp = getEBP(dummy);

fprintf(stdout,"for main thread: stack top of main is 0x%p\n",ebp);

#ifdef JOIN\_THREAD

ret = pthread\_join(tid,NULL);

if(ret !=0 )

{

fprintf(stderr, "join thread %lu failed",

tid);

return -2;

}

#endif

tid = 0;

ret = pthread\_create(&tid,NULL,thread\_work,NULL);

if(ret != 0)

{

fprintf(stderr,"create thread failed ,return %d (%s)\n",

ret,strerror(errno));

return -1;

}

ret = pthread\_join(tid,NULL);

if(ret != 0)

{

fprintf(stderr, "join thread %lu failed",tid);

return -2;

}

return 0;

}

在应用层pthread\_t类型变量称为线程ID，唯一标识了线程。pthread\_create函数会将创建出来的线程的线程ID填入第一个参数。对于POSIX标准而言，在X86平台，pthread\_t本质是一个指针，指向pthread类型的结构体。

对于线程而言我们可以调用pthread\_self()来返回线程的pthread\_t类型的线程ID。

#include <pthread.h>

pthread\_t pthread\_self(void);

在同一个线程内，Pthread库提供了接口来比较两个线程ID是否对应者同一个线程

#include <pthread.h>

int pthread\_equal(pthread\_t t1, pthread\_t t2);

返回值是0的时候，表征两个线程是同一个线程。需要注意的是，同一个线程组内的线程，才有比较的意义，不同线程组内的两个线程，哪怕两者的pthread\_t值一样，也不是同一个线程，这是显而易见的。

更确切地说，pthread\_t类型的线程ID是glibc分配出来的一个栈上的地址，Pthread库仅仅保证同一个进程内，同一时刻，各个线程的ID是不同的，不能够保证同一进程的多个线程在不同时刻线程的ID是不同。如果线程组的一个线程退出了以后，再次调用pthread\_create来创建线程，线程ID重用的可能性是很大的。同一进程尚且如此，遑论不同进程之间。由此看来用pthread\_t类型的线程ID来标识一个线程是不合理的。

编译执行下范例代码：

gcc -o pthread\_id pthread\_id.c -DJOIN\_THREAD -lpthread

程序输出如下：

manu@manu-hacks:~/code/self/c/pthread$ ./pthread\_id

for main thread: pid = 4162 address of pid is 0xbff27bf0

for main thread: stack top of main is 0x0xbff27c08

for thread: pid = 4162,address of pid = 0xb752633c

for thread: pthread\_self return 0xb7526b40

for thread: stack top is 0xb7526358

for thread: gettid return 4163

for thread(4163): I will exit now

for thread: pid = 4162,address of pid = 0xb752633c

for thread: pthread\_self return 0xb7526b40

for thread: stack top is 0xb7526358

for thread: gettid return 4164

for thread(4164): I will exit now

从上面的例子我们可以看出两次调用pthread\_create返回的tid，都是同一个值0xb7526b40，在多线程进程地址空间中会分析线程栈缓存的机制，在那里就能理解为何两次调用pthread\_create返回相同的线程id。

我们前文提到了，引入线程组的概念之后，同一个进程的多个线程它们共享一份进程ID，即getpid()返回的值相同。但是每个线程在内核的task\_struct结构中有自己的线程ID。Linux系统调用gettid()可以返回线程的线程ID。可惜的是glibc没有提供gettid的封装，所以我们不得不迂回取得线程的线程ID。

<sys/syscall.h>

int TID = syscall(SYS\_gettid);

正如PID很适合标识进程，gettid系统调用返回的线程ID很适合作为线程的标示符：

* 返回类型是pid\_t类型，进程之间不会存在重复的线程ID，而且不同线程之间也不会重复，在任意时刻都是全局唯一的值。
* procfs中记录了线程的相关信息，我们可以查看/proc/pid/task/tid方便的查看线程对的信息。
* ps命令提供了查看线程信息的-L选项，我们可以通过查看LWP(Light Weighted Process)和NLWP(线程个数)，查看出同一个线程组的个数及线程ID的信息。

对于线程标识，另外一个比较有意思的功能是我们可以给线程起一个有意义名字，并且我们可以从procfs中获取到线程的名字，我们也能从ps命令中得到线程的名字，从而更好的辨识不同的线程。

Linux提供了prctl系统调用

#include <sys/prctl.h>

int prctl(int option, unsigned long arg2, unsigned long arg3 ,

unsigned long arg4, unsigned long arg5)

这个系统调用提供了ioctl类似的控制，当我们需要给线程设定名字的时候，我们只需要将option设为PR\_SET\_NAME,同时将我们想用的线程名字作为arg2传递给prctl系统调用，我们就能给线程命名。

下面我们提供了一个简单的例子，学习如何通过系统调用gettid获取线程ID，已经如何给线程命名。

#include <sys/prctl.h>

#include <stdarg.h>

#include <stdio.h>

void thread\_setnamev(const char\* namefmt, va\_list args)

{

char name[17];

vsnprintf(name, sizeof(name), namefmt, args);

prctl(PR\_SET\_NAME, name, NULL, NULL, NULL);

}

void thread\_setname(const char\* namefmt, ...)

{

va\_list args;

va\_start(args, namefmt);

thread\_setnamev(namefmt, args);

va\_end(args);

}

thread\_setname("BEAN-%d",num);

我们创建了四个线程，按照调用pthread\_create的顺序，将0，1，2，3作为参数传递给线程，我们会调用prctl给每个线程起一个名字：分别为BEAN-0，BEAN-1，BEAN-2，BEAB-3。

看下procfs下线程的情况：

manu@manu-hacks:~$ ll /proc/3454/task/

total 0

dr-xr-xr-x 7 manu manu 0 7月 27 10:20 ./

dr-xr-xr-x 9 manu manu 0 7月 27 10:18 ../

dr-xr-xr-x 6 manu manu 0 7月 27 10:20 3454/

dr-xr-xr-x 6 manu manu 0 7月 27 10:20 3455/

dr-xr-xr-x 6 manu manu 0 7月 27 10:20 3456/

dr-xr-xr-x 6 manu manu 0 7月 27 10:20 3457/

dr-xr-xr-x 6 manu manu 0 7月 27 10:20 3458/

我们可以调用ps来查看线程信息：

manu@manu-hacks:~$ ps -L -p 3454

PID LWP TTY TIME CMD

3454 3454 pts/0 00:00:00 pthread\_tid

3454 3455 pts/0 00:00:00 BEAN-0

3454 3456 pts/0 00:00:00 BEAN-1

3454 3457 pts/0 00:00:00 BEAN-2

3454 3458 pts/0 00:00:00 BEAN-3

进程下的所有线程的进程ID（PID），线程ID（LWP）都列了出来，有意思的是我们给线程起得名字也显示了出来供我们辨识不同的线程。

注意，对于线程而言，procfs中的cmdline仍然是最初的pthread\_tid名字，而不是我们定制的BEAN-X，但是我们给线程定制的名字藏在了procfs中的stat和status中：

manu@manu-hacks:~$ cat /proc/3454/task/3457/cmdline

**./pthread\_tid**manu@manu-hacks:~$ cat /proc/3454/task/3457/stat

3457 (**BEAN-2**) S 2558 3454 2558 34816 3454 1077952576 1 0 0 0 0 0 0 0 20 0 5 0 276886 35901440 77 4294967295 134512640 134515928 3213269440 3059679544 3078472740 0 0 0 0 3238498201 0 0 -1 1 0 0 0 0 0 134520576 134520912 136785920 3213271654 3213271668 3213271668 3213275118 0

manu@manu-hacks:~$ cat /proc/3454/task/3457/status

Name: **BEAN-2**

State: S (sleeping)

Tgid: 3454

我们从procfs文件系统中清楚的表达线程与进程的关系。一个进程下可能有多个线程。在最初启动的线程，我们称之为主线程或者启动线程，系统会为主线程分配线程组ID，对于主线程，线程ID就等于线程组ID。如果我们不在调用pthread\_create创建线程，那么，进程下只有一个线程。如果我们调用pthread\_create创建了线程，线程的getpid返回的是主线程的线程组ID，同时会为线程分配一个唯一的线程ID来唯一标识线程组里的该线程。

**9.3.1.3多线程进程的地址空间**

不同进程，彼此的地址空间是隔离的，各自有各自的虚拟地址空间。一个进程的虚拟地址空间和另一个进程的虚拟地址空间，会各自映射到物理内存，所以鸡同鸭讲，没有任何的关系。但是同一个进程下的多个线程，他们共用一套虚拟地址空间，它们在同一个屋檐下。本章序言提到，代码段，全局数据段，堆，以及动态共享库，都是所有线程共用的。对于每个线程而言，系统给每个线程分配了各自的栈，相当于每个线程都有了自己的生产车间。

下面我们执行pthread\_tid程序，看下内存地址空间的布局情况：

我们以32位的X86架构为例。



首先3G～4G内存地址属于kernel空间，用户态的程序不能访问这些地址，否则会出发段错误。

栈顶的地址也不是0xbfffffff，为了防范缓冲区溢出的攻击，内核引入了随机化策略（address-space layout randomization）：内核没有将栈和内存映射区域的起点固定，而是每次进程启动的时候，引入一个随机化的偏移，这样攻击者就无法依赖固定地址找到栈，增加了他们构建恶意代码攻击的难度。

随机化大小区域之后，就是栈空间。这里所说的栈空间是就是主线程的栈空间，线程栈空间不在此处。对于32位的X86系统，Linux内核有保证，保证栈的空间不小于128M。大部分主线程的栈用不了那么大的栈，除非程序设计有问题，比如太深的递归。

栈空间下面，又是一小块随机大小的区域，用来保护内存映射的基地址。

在下面就是mmap区域，或者内存映射区域。glibc以及需要的其他动态共享库会以mmap的方式加载在这个区域。对于多线程程序，pthread\_create创建的线程也在这个区域。当调用pthread\_create创建线程之初调用clone系统调用之前，glibc会调用ALLOCATE\_STACK分配一块栈空间。首先是线程栈的起始地址。Pthread库提供了pthread\_attr\_setstack接口设置线程栈的基地址及其栈大小，但是出于可移植性的考虑，强烈建议不要使用这个接口。再次是线程栈的大小，Pthread库提供了pthread\_attr\_setstacksize接口来指定线程栈的大小。如果用户不指定，会采用默认值，在Ubuntu系统上，默认值是8192。

ulimit –s

8192

如果用户没有指定线程栈的基地址，那么这个地址有glibc负责分配。对于初次创建线程，glibc会调用mmap函数在内存映射区域分配栈：

*glibc-2.17/nptl/allocatestack.c*

mem = mmap (NULL, size, prot,

             MAP\_PRIVATE | MAP\_ANONYMOUS | MAP\_STACK, -1, 0);

用strace跟踪系统调用：

mmap2(NULL, 8392704, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS|MAP\_STACK, -1, 0) = 0xb6d3c000

新创建的线程都会调用mmap来分配线程栈的空间，内存映射区域是向下增长的，所以先创建的线程，栈地址比较高，后创建的线程，栈地址比较低。

事实上，pthread\_t类型的线程ID，是mmap分配空间上的一个地址,在glibc中这个地址指向的是一个pthread类型的结构体，这个结构体，是pthread实现的核心数据结构。



后创建的线程，会调用mmap2系统调用创建了线程栈的地址比较低。但是这个结论也不一定。考虑我们线程的标识一节提到，同一个进程，创建的线程pthread\_self返回的线程ID可能完全一样。这就牵扯到pthread\_create中很有意思的一个机制，线程栈缓存。

如果创建的线程栈退出了，并且占据的资源被释放（被主线程join或者设置了detach状态），那么glibc可能会将为这个消亡线程分配的栈，分配给接下来要创建的栈，如果要新建的线程没有指定栈起始位置并且栈大小符合条件的话。从逻辑上讲，只要已消亡的栈的大小不小于新创建的栈的大小，就可以复用这块内存映射区域。但是glibc会尝试查找最合适的已经消亡线程的栈。所谓最合适，就是栈空间完全相等，如果不存在，退而求其次，就去寻找满足栈空间要求的最小的线程栈。如果发现最合适的栈比需要的栈空间4倍好要大，就放弃缓存的线程栈，调用mmap来分配线程栈。

用strace观察pthread\_id的执行，我们发现，第二次执行pthread\_create的时候，并没有触发mmap2系统调用，原因就是glibc调用get\_cached\_stack查找到了第一次调用pthread\_create时创建出来的线程栈。这也是两者调用pthread\_self返回同一个地址的原因。

前文提到过，仅仅靠glibc是不足以支持多线程的。为了支持多线程，内核做了很多的改动。Linux对clone系统调用做了很多的扩展来支持多线程的实现。

clone(child\_stack=0xb753c424, flags=CLONE\_VM|CLONE\_FS|CLONE\_FILES|CLONE\_SIGHAND|CLONE\_THREAD|CLONE\_SYSVSEM|CLONE\_SETTLS|CLONE\_PARENT\_SETTID|CLONE\_CHILD\_CLEARTID, parent\_tidptr=0xb753cba8, {entry\_number:6, base\_addr:0xb753cb40, limit:1048575, seg\_32bit:1, contents:0, read\_exec\_only:0, limit\_in\_pages:1, seg\_not\_present:0, useable:1}, child\_tidptr=0xb753cba8) = 25317

其实在Linux系统下，调用pthread\_create创建线程和fork创建进程，调用的系统调用都是clone，区别在于传入的flags的值。我们看到pthread\_create创建线程时，传递了很多的标志位，他们是什么含义，内核如何根据这些标志位，正确的创建线程。同一个进程下的多个线程如何共享文件描述，信号处理函数等属性。我们看下一节，clone系统调用及线程间资源的共享。

**9.3.1.4 clone系统调用及线程间资源的共享**

我们看到，glibc调用了clone系统调用通知内核创建线程。其中传递的标志位为：

flags=CLONE\_VM|CLONE\_FS|CLONE\_FILES|CLONE\_SIGHAND|CLONE\_THREAD|

CLONE\_SYSVSEM|CLONE\_SETTLS|CLONE\_PARENT\_SETTID|CLONE\_CHILD\_CLEARTID

我们可以看下Linux内核接收到这些标志位后，如何创建线程的内核部分：

**CLONE\_VM：**

这个标志位告诉内核，不要复制进程的虚拟内存页。新创建的线程和主线程共享一份虚拟内存页。vfork系统调用也有这个标志位。

尽管fork采用了写时复制（copy-on-write）的技术，但是进程仍然需要复制内存页表，这就造成了fork创建进程的代价要高于调用pthread\_create来创建线程的代价。

对于同一个进程下的多个线程，共用一份虚拟内存页，所以，对于全局变量，各个线程都可见，一个线程修改全局变量，另外一个线程也能发现全局变量的改动。不仅是全局变量，内存映射区域（mmap/munmap），或者堆上的空间，对于同意线程组的所有线程都是可见的。

**CLONE\_FS：**

这个标志位告诉内核，新创建的线程，和调用clone的主线程共享一份文件系统相关的信息，包括文件权限掩码，当前工作目录，及根目录。所谓共享是指，线程组内的任何一个线程执行了umask()，chroot()，chdir()，都会影响的其他线程。

对于fork调用而言，需要分配内核的fs\_struct结构，根据调用clone系统调用的进程的值，拷贝一份。

**CLONE\_FILES：**

这个标志位告诉内核，要共用一份文件描述符表。同一线程组内的任意线程打开的文件，管道，网络套接字（socket），另一线程也可以操作。

对于创建进程而言，很不幸，需要拷贝一份文件描述符表。

**CLONE\_SIGHAND：**

线程组里所有线程共用一份信号处理函数。所以，主线程执行调用signal函数或者sigaction函数注册的信号处理函数，对于后面创建的线程依然有效。

**CLONE\_THREAD：**

这个标志不是新朋友了，在信号一章中，我们已经和它打过交道了。POSIX标准要求，所有线程共享一个进程ID，即我们提到的所有线程执行getpid()返回同一个值。Linux引入了线程组的概念。同一线程组内所有线程共享一个相同的进程ID，即内核的TGID(thread group ID)。但是每个线程都有自己独特的线程标识符TID。前面章节已经详细提到，不在赘述。

p->pid = pid\_nr(pid);

p->tgid = p->pid;

if (clone\_flags & CLONE\_THREAD)

p->tgid = current->tgid;

**CLONE\_SETTLS：**

TLS，是Thread Local Storage的缩写，是指线程本地存储。

**CLONE\_PARENT\_SETTID：**

这个标志位和前面的有所不同，他和CLONE\_CHILD\_CLEARTID是类似的。内核会将新创建线程的线程ID写入到用户空间的一个地址。

int rc = ARCH\_CLONE (fct, STACK\_VARIABLES\_ARGS, clone\_flags,

pd, **&pd->tid**, TLS\_VALUE, &pd->tid);

这是glibc分配完线程栈空间之后，执行clone系统调用时候传入的参数。前文提到过pthread\_t是个指针，指向struct pthread 这样一个结构体。这个结构体中有一个变量叫做tid，保存kernel分配给线程的线程ID。clone系统调用把这个成员变量的地址作为第五个参数传递给内核，让内核在系统调用期间，返回之前将线程ID填入此处。

1519 nr = task\_pid\_vnr(p);

1520

1521 if (clone\_flags & CLONE\_PARENT\_SETTID)

1522 put\_user(nr, parent\_tidptr);

为什么这样做？

如果这个tid不在内核态赋值，那么以返回值的方式。但是clone先返回，后赋值，在clone之后，赋值之前，可能接收到终止信号。信号处理函数可能无法有效的访问这个值。

由于多线程之间共享了太多的资源，所以多线程编程要小心。比如多个线程往一个文件里面写，很有可能多个线程的内容互相交织在一起，无法阅读。资源共享给我们带来了方便，但是也带来了同步的需求。

### 9.3.2 线程的退出

有生就有灭，前面花了大量的篇幅介绍了线程的创建，这里我们介绍线程的终止。终止线程的运行，有以下方法

* 创建线程时的start\_routine函数执行了return，并且返回指定值
* 线程调用pthread\_exit;
* 调用了pthread\_cancel取消线程（后面详述）；
* 任意线程调用了exit()，或者主线程执行了在main函数执行了return语句，都会导致所有线程一并终止

值得注意的是pthread\_exit和start\_routine函数执行return是有区别的。在start\_routine调用的任何层级的函数执行pthread\_exit()都会引发线程退出，而return，只能是start\_routine函数内执行才能导致线程退出。

void\* start\_routine(void\* param)

{

…

foo();

bar();

return NULL;

}

void foo()

{

…

pthread\_exit(NULL);

}

如果foo函数执行了pthread\_exit，线程会立刻退出，后面的bar会没有机会执行。

下面我们看下pthread\_exit的接口定义：

#include <pthread.h>

void pthread\_exit(void \*value\_ptr);

value\_ptr是一个指针，作为线程的“临终遗言”，pthread\_join调用会接收这个地址，获取到退出线程的临终遗言。但是有个问题在里面，就是不能将线程的局部变量的地址，因为用户写的线程函数退出了，线程函数栈上的局部变量已经不复存在了（尽管线程栈可能还没有被销毁，见下一小节），线程的临终遗言可能无法被接收者读到。

void\* thread\_work(void\* param)

{

int ret = -1;

ret = whatever();

pthread\_exit(&ret);

}

上述用法一种典型的错误用法，因为当线程退出时，线程栈不复存在了，上面的ret变量也已经无法访问了。那我们如何正确的传递返回值呢？

* pthread\_exit((int\*) ret);
* 使用全局变量返回
* 将返回值填入到malloc出来堆上的空间

第一种一种tricky的做法，我们将返回值ret强制类型转换，接收方在把返回值强制传承int。但是这种方法不推荐使用。这种方法是奏效的但是太tricky，而且C标准没有承诺int型转成指针，然后从指针转成int型，数据保持不变。

第二种方法是可以做到，全局变量，pthread\_join也可以可见这个变量。但是全局变量用pthread\_exit返回有点奇怪，因为就是返回NULL，主进程依然可以看到全局变量的值。

第三种方法是用malloc，在堆上分配空间，然后将返回值填入其中。因为堆上的空间不会随着线程的退出而释放，所以pthread\_join可以取出返回值。切莫忘记该空间，引起内存泄露。

需要提出的是，如果主线程执行了pthread\_exit()，那么其他线程依然正常工作。主线程尚且如此，其他线程调用pthread\_exit()，也只影响自己线程，不影响其他线程。

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<string.h>

#include<sys/types.h>

#include<errno.h>

#include<pthread.h>

void\* thread\_work(void\* param)

{

fprintf(stderr,"enter thread %p\n",(void\*)pthread\_self());

sleep(40);

int \*ret = malloc(sizeof(int));

\*ret = 8;

fprintf(stderr,"thread exit\n");

pthread\_exit((void\*)(ret));

}

int main()

{

pthread\_t tid ;

fprintf(stderr,"the thread group pid is %d\n",getpid());

int ret = pthread\_create(&tid,NULL,thread\_work,NULL);

if(ret != 0)

{

fprintf(stderr,"create thread failed (%s)\n",strerror(errno));

return -1;

}

sleep(20);

#if defined MAIN\_EXIT

fprintf(stderr,"main thread will call exit\n");

exit(0);

#elif defined MAIN\_PTHREAD\_EXIT

fprintf(stderr,"main thread will pthread\_exit exit\n");

pthread\_exit(NULL);

#else

fprintf(stderr,"main thread begain to wait thread exit\n");

int\* thread\_ret = NULL;

pthread\_join(tid,(void\*)(&thread\_ret));

fprintf(stderr,"thread return value is %d\n",\*(int\*)thread\_ret);

free(thread\_ret);

#endif

return 0;

}

第一种情况是主线程阻塞，等待线程正常退出。线程将返回值填入malloc出来的空间，主进程调用pthread\_join接受该进程的返回值，然后显示出来，然后将动态分配的空间释放，防止内存泄露。

gcc -o pthread\_main\_exit pthread\_main\_exit.c -lpthread -Wall

./pthread\_main\_exit

the thread group pid is 4986

enter thread 0xb7f76b90

main thread begain to wait thread exit

thread exit

thread return value is 8

第二种情况是主线程调用exit()，我们会发现所有线程都退出了，包括尚未执行完的线程。

gcc -o pthread\_main\_exit pthread\_main\_exit.c -DMAIN\_EXIT -lpthread -Wall

./pthread\_main\_exit

the thread group pid is 5009

enter thread 0xb7f91b90

main thread will call exit

我们会发现，主线程退出后，子线程也不存在了。

第三种情况是主线程调用pthread\_exit()，我们会发现仅仅是主线程退出，子进程依然会继续执行，知道退出。

gcc -o pthread\_main\_exit pthread\_main\_exit.c -DMAIN\_PTHREAD\_EXIT -lpthread -Wall

./pthread\_main\_exit

the thread group pid is 5023

enter thread 0xb7f1ab90

main thread will pthread\_exit exit

**thread exit**

我们看到，主线程调用pthread\_exit()之后，线程依然继续执行，打印了thread exit之后，才退出。在主线程调用pthread\_exit之后，创建的线程继续执行之际，我们看ps信息如下：

PID LWP TTY TIME CMD

5023 5023 pts/3 00:00:00 pthread\_main\_ex <**defunct**>

5023 5024 pts/3 00:00:00 pthread\_main\_ex

### 9.3.3 pthread\_join和pthread\_detach解析

前面提到了，线程退出时有遗言的，它调用pthread\_exit()的时候，glibc会将遗言保存在哪里呢？

void

\_\_pthread\_exit (value)

void \*value;

{

**THREAD\_SETMEM (THREAD\_SELF, result, value);**

\_\_do\_cancel ();

}

strong\_alias (\_\_pthread\_exit, pthread\_exit)

我们提到了pthread\_t是个指针，指向了一个pthread类型的结构体。该结构体下面成员变量很多，其中有一个成员变量时void \*类型的result，进程退出时会将用户传递的指针保存在这个结构体中。

我们看到了，“遗言”保存在pthread结构体中，这就意味着，pthread结构体调用pthread\_exit()之后也没有被释放，否则遗言无法被传递出去。线程都调用了pthread\_exit()，可是线程对应空间却不能释放，这意味着内存泄露。的确如此，如果我们创建了很多线程，而不去调用pthread\_join()去连接已经终止的线程，也不调用pthread\_detach()去将线程设置成分离(detached)状态，那么，就意味着这部分栈空间无法回收。

笔者在工作期间遇到过这种bug。项目中存在一个FTP服务器进程，采用的是多线程架构。主线程负责监听端口，有用户来连，就创建一个线程为该用户服务。但是主进程没有调用pthread\_join(无法调用，主线程需要监听端口，不能阻塞在pthread\_join上)，也忘记将线程设置成可分离的状态，导致线程退出后，线程对应的内存空间不能释放。最终内存耗尽，无法创建线程响应新来的连接。

线程pthread结构体中有一个成员变量为：

struct pthread \*joinid；

这个值有三种可能性：

* NULL:没有执行过pthread\_join()，也没有执行过pthread\_detach()。一旦线程退出，无人收尸，导致内存泄露
* 线程自身的pthread\_t值：说明执行过pthread\_detach()，属于自我了断型，线程退出时，会释放自己的内存空间。
* 其他线程的pthread\_t值：有另外的线程调用了pthread\_join()，等待该线程退出。

第一种情况，已经提过了，我们接下来分别介绍后两种：

**9.3.3.1 pthread\_join接口**

pthread\_join接口负责等待线程终止，接受线程的“临终遗言”。它的定义如下：

#include <pthread.h>

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

第一个参数是要连接的线程，第二个参数用来接收退出线程的“遗言”。

需要强调的是线程之间的关系是平起平坐的关系，和进程调用fork()创建子进程不同，进程层级的关系比较明显，有父进程有子进程，父进程可以调用wait()等待子进程，但是子进程不能调用wait()等待子进程，而且，父进程是唯一有资格调用wait等待子进程退出的进程。线程不同，线程之间的关系是对等的，就像兄弟姐妹一样，A创建线程B，B创建线程C，ABC三者依然是对等关系，A可以调用pthread\_join()连接C，C也可以连接A。

pthread\_join会将终止线程的资源释放，主要是指内存空间：

int

pthread\_join (threadid, thread\_return)

pthread\_t threadid;

void \*\*thread\_return;

{

struct pthread \*pd = (struct pthread \*) threadid;

/\* Make sure the descriptor is valid. \*/

if (INVALID\_NOT\_TERMINATED\_TD\_P (pd))

/\* Not a valid thread handle. \*/

return ESRCH;

/\* Is the thread joinable?. \*/

if (IS\_DETACHED (pd))/\*如果线程处于已分离的状态，那么就不能连接了\*/

/\* We cannot wait for the thread. \*/

return EINVAL;

struct pthread \*self = THREAD\_SELF;

int result = 0;

LIBC\_PROBE (pthread\_join, 1, threadid);

/\* During the wait we change to asynchronous cancellation. If we

are canceled the thread we are waiting for must be marked as

un-wait-ed for again. \*/

pthread\_cleanup\_push (cleanup, &pd->joinid);

/\* Switch to asynchronous cancellation. \*/

int oldtype = CANCEL\_ASYNC ();

if ((pd == self

|| (self->joinid == pd

&& (pd->cancelhandling

& (CANCELING\_BITMASK | CANCELED\_BITMASK | EXITING\_BITMASK

| TERMINATED\_BITMASK)) == 0))

&& !CANCEL\_ENABLED\_AND\_CANCELED (self->cancelhandling))

/\* This is a deadlock situation. The threads are waiting for each

other to finish. Note that this is a "may" error. To be 100%

sure we catch this error we would have to lock the data

structures but it is not necessary. In the unlikely case that

two threads are really caught in this situation they will

deadlock. It is the programmer's problem to figure this

out. \*/

result = EDEADLK;

/\* Wait for the thread to finish. If it is already locked something

is wrong. There can only be one waiter. \*/

else if (\_\_builtin\_expect (atomic\_compare\_and\_exchange\_bool\_acq (&pd->joinid,

self,

NULL), 0))

/\* There is already somebody waiting for the thread. \*/

result = EINVAL;

else

/\* Wait for the child. \*/

lll\_wait\_tid (pd->tid);/\*在此处阻塞，直到等待的线程退出\*/

/\* Restore cancellation mode. \*/

CANCEL\_RESET (oldtype);

/\* Remove the handler. \*/

pthread\_cleanup\_pop (0);

if (\_\_builtin\_expect (result == 0, 1))

{

/\* We mark the thread as terminated and as joined. \*/

pd->tid = -1;

/\* Store the return value if the caller is interested. \*/

if (thread\_return != NULL)

\*thread\_return = pd->result;

/\* Free the TCB. \*/

\_\_free\_tcb (pd);/\*此处释放资源，也不一定调用munmap释放空间，可能添加到cache供下次分配线程栈时使用\*/

}

LIBC\_PROBE (pthread\_join\_ret, 3, threadid, result, pd->result);

return result;

}

上面的源码不难看出，pthread\_join的几种异常。

* 第一类错误码ESRCH比较简单，要连接的线程不存在，查无此人。
* 第二类错误码是EDEADLK，顾名思义是死锁，如果一个线程自己连接自己，自己给自己收尸，这就是传说中诈尸啊。另一种情况是两个线程互相连接对方。这种互相为对方收尸的情况明显也是不合理的，这两种情况都是不允许的。
* 第三类错误码是EINVAL，一种情况是说，该线程处于已分离的状态，拒绝任何人前来连接它，这厮比较绝，死前自焚，清理的干干净净，不劳其他线程操心，另一种情况就是已经有线程连接该线程了，你来晚了。我们后面讲线程退出，唤醒“收尸人”，会讲到，唤醒只能唤醒一个，所以，好几个线程为某个线程收尸的做法是不允许的。

一个难题是同一个线程组的线程，如何知道等待的线程已经退出？

前面讲clone标志位的时候，有一个标志位比较特别，CLONE\_CHILD\_CLEARTID，这个标志位在glibc代码中有段注释，

CLONE\_PARENT\_SETTID

The kernels writes the thread ID of the newly created thread

into the location pointed to by the fifth parameters to CLONE.

Note that it would be semantically equivalent to use

CLONE\_CHILD\_SETTID but it is be more expensive in the kernel.

CLONE\_CHILD\_CLEARTID

The kernels clears the thread ID of a thread that has called

sys\_exit() in the location pointed to by the seventh parameter

to CLONE.

glibc的pthread\_create函数会分配栈之后，会调用create\_thread函数，这段注释就位于这个函数之内。含义比较明确，调用clone的时候，会有七个参数，第七个参数是用户空间里的一个地址，更明确点说是关键数据结构struct pthread的成员变量tid的地址。

当线程退出的时候，内核负责将这个用户空间的地址上变量清空。不仅如此，内核还负责通知到等待这个线程通知的线程，线程退出时，内核通过如下函数调用sys\_exit()->\_\_mm\_exit()->mm\_release()将tid置成0，同时通知等待线程。

void mm\_release(struct task\_struct \*tsk, struct mm\_struct \*mm)

{

...

\* If we're exiting normally, clear a user-space tid field if

\* requested. We leave this alone when dying by signal, to leave

\* the value intact in a core dump, and to save the unnecessary

\* trouble otherwise. Userland only wants this done for a sys\_exit.

\*/

if (tsk->clear\_child\_tid) {

if (!(tsk->flags & PF\_SIGNALED) &&

atomic\_read(&mm->mm\_users) > 1) {

/\*

\* We don't check the error code - if userspace has

\* not set up a proper pointer then tough luck.

\*/

put\_user(0, tsk->clear\_child\_tid);//将用户空间的tid置零

sys\_futex(tsk->clear\_child\_tid, FUTEX\_WAKE,

**1**, NULL, NULL, 0);//唤醒调用pthread\_join等待这个线程退出的线程，注意第三个参数是1，只能唤醒1个，也就说说用户态只能有一个线程等待这个线程退出。

}

tsk->clear\_child\_tid = NULL;

}

}

为什么内核执行了sys\_futex，pthread\_join就获知了这件事？这要从lll\_wait\_tid谈起。这个函数本质调用的是futex系统调用，我们用strace观察pthread\_join也可以清楚地看出来：

futex(0xb7f81bd8, FUTEX\_WAIT, 4827, NULL) = 0

futex，是fast userspace mutex，快速用户区互斥。后面讲线程互斥的时候，我们会提到。简单地说，就是pthread\_join()等待在这把锁上，等待的线程退出的时候，会释放这把锁。交互的关键，就在这个用户态的tid地址上。

当等到线程退出的时候，glibc在pthread\_join()中调用了\_\_free\_tcb (pd)，释放线程相关的内存。

这么设计不仅仅是因为线程的返回值存储在线程相关的内存里。何时释放线程对应的内存空间，一个直观的想法是调用exit之前，这里面有个竞争条件在里面：如果我们释放了线程本地存储TLS的空间，在调用exit之前，收到了信号，而信号处理函数操作了线程的数据，这怎么办？所以NPTL的设计，将线程空间的清理放到了线程退出以后。

**9.3.3.2 pthread\_detach接口**

前面提到了，pthread\_join()会等待线程的终止，接收线程的返回值，同时释放线程的资源。除了这种方式释放线程的线程控制块资源外，还有pthread\_detach，将线程设置为分离状态。如果线程处于分离状态，那么线程退出的时候，或者线程被取消的时候，glibc会负责释放内存空间。

#include <pthread.h>

int pthread\_detach(pthread\_t thread);

前面讲解pthread\_join时候提到，如果线程处于已分离状态，就不能再调用pthread\_join，如果没人为该线程“收尸”，线程的内存空间如何释放？答案是自己释放。

pthread\_create的第三个函数是用户定义的函数，事实上，线程创建好了之后，执行的函数是glibc中的start\_thread函数，用户定义的函数只是start\_thread函数的一部分。

 create\_thread会将glibc的start\_thread函数传递给do\_clone然后传递给ARCH\_CLONE。我们可以看下start\_thread执行。

static int

start\_thread (void \*arg)

{

if (\_\_builtin\_expect (! not\_first\_call, 1))

{

/\* Store the new cleanup handler info. \*/

THREAD\_SETMEM (pd, cleanup\_jmp\_buf, &unwind\_buf);

if (\_\_builtin\_expect (pd->stopped\_start, 0))

{

int oldtype = CANCEL\_ASYNC ();

/\* Get the lock the parent locked to force synchronization. \*/

**lll\_lock (pd->lock, LLL\_PRIVATE);**

/\* And give it up right away. \*/

**lll\_unlock (pd->lock, LLL\_PRIVATE);**

CANCEL\_RESET (oldtype);

}

LIBC\_PROBE (pthread\_start, 3, (pthread\_t) pd, pd->start\_routine, pd->arg);

/\* Run the code the user provided. \*/

#ifdef CALL\_THREAD\_FCT

THREAD\_SETMEM (pd, result, CALL\_THREAD\_FCT (pd));

#else

//此处执行用户在pthred\_create中第三个参数对应的函数。

**THREAD\_SETMEM (pd, result, pd->start\_routine (pd->arg));**

#endif

}

……

/\* If the thread is detached free the TCB. \*/

**if (IS\_DETACHED (pd))**//线程退出时，如果线程处于已分离状态，

//需要释放线程的内存

**/\* Free the TCB. \*/**

**\_\_free\_tcb (pd);**

……

}

我们可以看出执行完用户定义的函数之后（该用户定义的线程函数里面的局部变量已经不复存在了，所以，我们不能用局部变量的地址作为返回值），start\_thread函数会判断线程是否处于已分离的状态，如果是，释放线程的内存空间。

很有意思的地方时lll\_lock和lll\_unlock这两行代码也很有意思。初看之下莫名其妙，为什么等待锁，然后又释放了这把锁？需要注意的是，start\_thread这个函数已经是新的线程执行的函数了，新的线程是独立的CPU调度的实体。新线程执行之前，还有一些东西需要设置，比如线程的CPU亲和力，再比如线程的进程调度策略和参数，这些设置好之前，新创建的线程不应该策马奔腾。所以调用pthread\_create的线程，在ARCH\_CLONE之前首先抢占了pd->lock这把锁，让新的线程就得等在下面语句上。

lll\_lock (pd->lock, LLL\_PRIVATE);

详情见do\_clone函数：

static int

do\_clone (struct pthread \*pd, const struct pthread\_attr \*attr,

int clone\_flags, int (\*fct) (void \*), STACK\_VARIABLES\_PARMS,

int stopped)

{

#ifdef PREPARE\_CREATE

PREPARE\_CREATE;

#endif

if (\_\_builtin\_expect (stopped != 0, 0))

/\* We make sure the thread does not run far by forcing it to get a

lock. We lock it here too so that the new thread cannot continue

until we tell it to. \*/

lll\_lock (pd->lock, LLL\_PRIVATE);//调用pthread\_create首先拿锁，防止

……

atomic\_increment (&\_\_nptl\_nthreads);

//fct是前文提到的start\_thread函数

int rc = ARCH\_CLONE (**fct**, STACK\_VARIABLES\_ARGS, clone\_flags,

pd, &pd->tid, TLS\_VALUE, &pd->tid);

……

}

## 9.4 互斥量

互斥量这个词，听着挺高大上，其实说起来很简单啦。前面讲到了，线程组里的很多线程共享了很多的资源，全局变量，信号处理函数，打开的文件等，线程就像同一个屋檐下的一家人，大家各自有各自的事情当然最好，比如老爸在客厅看电视，老妈在厨房炒菜，儿子在书房上网，并行不悖，互不干扰，十分和谐。但是也有不和谐的时候，比如你闹肚子上厕所，发现老爸在厕所里面，任你肚中翻江倒海，你也只能等着。说到底，资源有限，厕所里面马桶只有一个，一个人蹲的时候，另外一个人就得等着。这种场景对应到线程，就是线程的互斥机制，某些资源不能同时操作，操作之前先上锁，拿到锁才能操作。拿不到锁，就只能等待（阻塞等待），或者放弃。

**9.4.1 困境**

在多线程的模型中，我们经常会遇到多个线程都要操作一份共享数据的情况，如果一个线程要访问该共享数据，而另一个线程正在修改，可能会造成数据不一致的情况。下面我们看下简单的例子。

#define \_GNU\_SOURCE

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

#include <stdlib.h>

#include <sched.h>

#include <string.h>

#include <linux/unistd.h>

#include <sys/syscall.h>

#include <errno.h>

#include<linux/types.h>

#include<time.h>

#define INC\_TO 10000000 // one million...

int global\_int = 0;

pthread\_mutex\_t count\_lock = PTHREAD\_MUTEX\_INITIALIZER;

pid\_t gettid( void )

{

    return syscall( \_\_NR\_gettid );

}

void \*thread\_routine( void \*arg )

{

    int i;

    int proc\_num = (int)(long)arg;

    struct timeval tv\_begin,tv\_end;

    \_\_u64 timeinterval;

    cpu\_set\_t set;

    CPU\_ZERO( &set );

    CPU\_SET( proc\_num, &set );

    if (sched\_setaffinity( gettid(), sizeof( cpu\_set\_t ), &set ))

    {

        perror( "sched\_setaffinity" );

        return NULL;

    }

    gettimeofday(&tv\_begin,NULL);

    for (i = 0; i < INC\_TO; i++)

    {

        global\_int++;

        //\_\_sync\_fetch\_and\_add( &global\_int, 1 );

    }

    gettimeofday(&tv\_end,NULL);

    timeinterval = (tv\_end.tv\_sec - tv\_begin.tv\_sec)\*1000000 +(tv\_end.tv\_usec - tv\_begin.tv\_usec);

    fprintf(stderr,"proc\_num :%d cost %llu us\n", proc\_num,timeinterval);

    return NULL;

}

void \*thread\_routine2( void \*arg )

{

    int i;

    int proc\_num = (int)(long)arg;

    struct timeval tv\_begin,tv\_end;

    \_\_u64 timeinterval;

    cpu\_set\_t set;

    CPU\_ZERO( &set );

    CPU\_SET( proc\_num, &set );

    if (sched\_setaffinity( gettid(), sizeof( cpu\_set\_t ), &set ))

    {

        perror( "sched\_setaffinity" );

        return NULL;

    }

    gettimeofday(&tv\_begin,NULL);

    for(i = 0;i<INC\_TO;i++)

    {

        pthread\_mutex\_lock(&count\_lock);

        global\_int++;

        pthread\_mutex\_unlock(&count\_lock);

    }

    gettimeofday(&tv\_end,NULL);

    timeinterval = (tv\_end.tv\_sec - tv\_begin.tv\_sec)\*1000000

        +(tv\_end.tv\_usec - tv\_begin.tv\_usec);

    fprintf(stderr,"proc\_num :%d %llu us\n",

            proc\_num,timeinterval);

    return NULL;

}

int main()

{

    int procs = 0;

    int i;

    pthread\_t \*thrs;

    // Getting number of CPUs

    procs = (int)sysconf( \_SC\_NPROCESSORS\_ONLN );

//    procs = 1;

    if (procs < 0)

    {

        fprintf(stderr, "sysconf for CPU CORE num failed\n" );

        return -1;

    }

    thrs = malloc( sizeof( pthread\_t ) \* procs );

    if (thrs == NULL)

    {

        fprintf(stderr, "malloc thread id array failed\n" );

        return -1;

    }

    printf( "Starting %d threads...\n", procs );

    for (i = 0; i < procs; i++)

    {

        if (pthread\_create( &thrs[i], NULL, thread\_routine,

                    (void \*)(long)i ))

        {

            fprintf(stderr, "pthread\_create failed on %d thread (%s)\n",

i,strerror(errno));

            procs = i;

            break;

        }

    }

    for (i = 0; i < procs; i++)

        pthread\_join( thrs[i], NULL );

    free( thrs );

    printf( "After doing all the math, global\_int value is: %d\n", global\_int );

    printf( "Expected value is: %d\n", INC\_TO \* procs );

    return 0;

}

首先定义一个全局变量global\_int，初始化为0，然后判断我们的硬件CPU Core数N，然后创建N个线程，各自绑定在一个核上。每个线程执行的程序一样，让全局变量global\_int自加，循环10000000次。按照我的机器，一共有4个核，结束后，global\_int应该等于4\*10000000=40000000，看下执行情况：

Starting 4 threads...

proc\_num :0 cost 94331 us

proc\_num :1 cost 94065 us

proc\_num :2 cost 101741 us

proc\_num :3 cost 104771 us

After doing all the math, global\_int value is: 13971165

Expected value is: 40000000

首先，运行结束后，global\_int不是我们期待的结果，是一个很奇怪的数字。再次运行之后，是另一个数字。

造成这种情况的原因是操作不是原子的。当全局变量是5的时候，A线程取到全局变量的值，将变量加1，在平行宇宙的另一端，另一CPU在全局变量加1之前取到了值5 ，也加1，就会造成全局变量明明加了两次1，可变量的值是6，从而造成了结果的错误。

如果共享的数据仅仅是一个int型的数字的话，这根本不是什么问题，gcc提供了 \_\_sync\_fetch\_and\_add系列的函数，使加1编程原子操作，可以解决这个问题。可是实际编程中，共享的数据可能要比这个例子要复杂的多。举例说，共享的数据可能是个双向链表，如果一个线程删除链表的一个元素，同时另一个线程要链接在该元素的后面。无论删除还是添加都会牵扯到多个指针的变化，如果没有保护，我们的双向链表会遭到破坏。

如何解决这个问题呢，就像上厕所会锁门一样，POSIX引入了互斥量。严格来说，互斥量一种锁，满足以下三点条件：

* 原子性：无论多少个线程同时执行加锁操作，只能有1个线程加锁成功
* 单一性：如果一个线程已经持有了互斥量，其他线程都无法获得互斥量，直到持有互斥量的线程解锁，释放该互斥量；
* 非忙等待：如果互斥量已经被其他线程持有，申请互斥量的线程会陷入阻塞，但是不会消耗大量的CPU资源。

**9.4.2互斥量的初始化**

互斥量的缩写mutex，是英文mutual exclusive（互相排斥之意）的缩写。就像你的卫生间总是有锁的，对于这种不能并发访问的共享数据，你也需要有锁保护。

我已经掌握了藏私房钱的64招，我一定会是个好爸爸，但是，首先，你得，有个，老婆。正确的使用互斥量来保护共享数据，首先，你得，先定义和先初始化互斥量。

POSIX提供了两种初始化互斥量的方法。

首先是将PTHREAD\_MUTEX\_INITIALIZER赋给定义的互斥量。

#include <pthread.h>

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

如果我们的互斥量是动态分配的，或者要设定互斥量的属性，我们上面静态初始化的方法就不适用了，NPTL提供了另外的函数pthread\_mutex\_init()对互斥量进行动态的初始化：

int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex,

const pthread\_mutexattr\_t \*restrict attr);

对于第二个pthread\_mutexattr\_t指针的入参，是用来设定互斥量属性的。我们暂时不用管它，事实上大部分程序，我们会传递NULL，使用互斥量的默认属性。

调用pthread\_mutex\_init()之后，互斥量处于没有加锁的状态。

**9.4.3互斥量的销毁**

有生就有灭，我们创建了互斥量，当我们确认不在需要互斥量的时候，我们需要销毁它。

有三点需要注意的：

* 使用PTHREAD\_MUTEX\_INITIALIZER初始化的互斥量无需销毁
* 不要销毁一个已加锁的互斥量或者是被条件变量使用的互斥量（如调用了pthread\_cond\_wait，后面会讲，条件变量总是伴随互斥量）。
* 已经销毁的互斥量，要确保后面不会有线程尝试加锁

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

当互斥量处于已加锁的状态，或者和条件变量配合使用状态，会返回EBUSY错误码。

**9.4.3互斥量的加锁和解锁**

准备了这么多前戏，终于可以说说加锁和解锁了。POSIX提供了如下接口：

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

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

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

在调用pthread\_lock()的时候，可能遭遇一下几种情况：

* 如果互斥量处于未锁定的状态，则该函数会将互斥量锁定，同时返回成功给调用者；
* 如果其他线程已经对该互斥量加锁，那么pthread\_lock()调用会阻塞，直到该互斥量被解锁，并且该线程成功锁定互斥量（有可能该线程不是唯一阻塞等锁的线程）；

如果互斥量已经被解锁，但是线程没有竞争到该互斥量（有其他线程同时尝试加锁），就只能再次阻塞，等待下一轮解锁后，重新竞争。

* 如果调用pthread\_lock()线程之前已经调用过pthread\_lock()并且已经持有了互斥量，在默认情况下会发生死锁，再也没有线程能申请到该互斥量，从而导致整个线程组都不能正常工作。但是互斥量也可以设定高级属性，如果设定了属性为PTHREAD\_MUTEX\_ERRORCHECK，第二次调用pthread\_lock()会返回EDEADLK。

pthread\_mutex\_unlock是解锁函数。当线程临界区操作结束之后，应该调用解锁操作。如果忘记解锁，就会造成其他线程无法操作临界区。

在我们困境一节中，提供了thread\_routine2函数，这是一个加锁保护共享数据的版本，我们看下执行

Starting 4 threads...

proc\_num :3 8737187 us

proc\_num :1 8767394 us

proc\_num :2 8775942 us

proc\_num :0 8782070 us

After doing all the math, global\_int value is: 40000000

Expected value is: 40000000

首先执行结果是正确了，是预期的结果40000000，但是执行时间是不加锁的错误版本的80倍左右，事实上，大部分的时间。

下面介绍下加锁和解锁的注意事项：

1临界区做的事情尽可能的小：

pthread\_mutex\_lock(&mutex);

/\*this is critical section\*/

pthread\_mutex\_unlock(&mutex);

由于互斥量保护的临界区代码不能被多个线程并发执行，有可能其他线程会阻塞在互斥量上，从而造成效率的下降。线程数越多，临界区越大，阻塞线程执行的可能性就越大。因此，写互斥量保护的临界区代码时一定要慎之又慎，写每一行代码的时候，都确定一下，会不会引起其他线程长时间阻塞于此；对于可以不在临界区执行的代码，尽量移除。

2 预防死锁

对于互斥量而言，最臭名昭著的现象就是死锁（dead lock）了。最简单最好构造的死锁就是下面这种场景了。线程1已经成功拿到了互斥量1，正在申请互斥量2，而同时在另一个CPU Core上，线程2已经拿到了互斥量2，正在申请互斥量1。彼此占有对方正在申请的互斥量，结局就是谁也没办法拿到想要的互斥量，于是死锁发生了。

|  |  |
| --- | --- |
| Thread1 | Thread2 |
| pthread\_mutex\_lock(&mutex1); | pthread\_mutex\_lock(&mutex2); |
| /\*Operate on resource 1 \*/ | /\*Operate on resource 2\*/ |
| pthread\_mutex\_lock(&mutex2); | pthread\_mutex\_lock(&mutex1); |
| /\*Operate on resource 1 and resource 2\*/ | /\*Operate on resource 1 and resource 2\*/ |
| pthread\_mutex\_unlock(&mutex2); | pthread\_mutex\_unlock(&mutex1); |
| pthread\_mutex\_unlock(&mutex1); | pthread\_mutex\_unlock(&mutex2); |

上面的例子比较简单，但实际工程中死锁可能会发生在复杂的函数调用之中。可以想象随着程序复杂度增加，检测死锁并不像我们例子这么简单。

存在多个互斥量的情况下，避免死锁最简单的方法就是总是按照一定的先后顺序申请这多个互斥量。还是刚才的例子，如果每个线程都按照先申请互斥量1，再申请互斥量2的顺序，死锁就不会发生。有些互斥量有明显的层级关系，但是也有一些互斥量原本就没有特定的层级关系，不过没有关系，我们可以人为的干预，所有的线程必须遵循同样的顺序来申请互斥量。

另一种方法是尝试一下，如果取不到锁就回退。Linux提供了接口来表达这种思想：

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

int pthread\_mutex\_timedlock(pthread\_mutex\_t \*restrict mutex, const struct timespec \*restrict abs\_timeout);

这两个函数反应了这种尝试一下，不行就算了的思想。对于pthread\_mutex\_trylock()接口，如果互斥量已然被锁定，那么当即返回EBUSY错误，并不像pthread\_mutex\_lock()接口一样阻塞于此。

其中trylock不行就回退的思想有可能引发活锁（live lock）。考虑下面两个线程，线程1首先申请锁mutex\_a后，尝试申请mutex\_b，失败以后，释放mutex\_a进入下一轮循环，同时线程2会因为尝试申请mutex\_a失败，而释放mutex\_b，如果两个线程恰好一直保持这种节奏，可能在很长的时间内两者一次次的擦肩而过。当然这毕竟不是死锁，终究会有一个线程同时持有两把锁而结束这种情况。尽管如此，活锁现象的确会降低性能。

//线程1

void func1()

{

    int done = 0;

    while(!done)

{

        pthread\_mutex\_lock(&mutex\_a);

        if (pthread\_mutex\_trylock (&mutex\_b))

{

            counter++;

            pthread\_mutex\_unlock (&mutex\_b);

            pthread\_mutex\_unlock (&mutex\_a);

            done = 1;

        }

        else

{

            pthread\_mutex\_unlock (&mutex\_a);

        }

    }

}

// 线程 2

void func2()

{

    int done = 0;

    while(!done)

{

        pthread\_mutex\_lock (&mutex\_b);

        if (pthread\_mutex\_trylock (&mutex\_a))

{

            counter++;

            pthread\_mutex\_unlock (&mutex\_a);

            pthread\_mutex\_unlock (&mutex\_b);

            done = 1;

        }

        else

{

            pthread\_mutex\_unlock (&mutex\_b);

        }

    }

}

对于pthread\_mutex\_timedlock()接口，提供了一个时间参数abs\_timeout，如果申请互斥量的时候，互斥量已然被锁定，那么等待，如果到了abs\_timeout指定的时间，仍然没有申请到互斥量，那么返回ETIMEOUT错误。除此，这两个接口表现和pthread\_mutex\_lock一致。事实上，这两个接口表面看起来很有道理，实际上很少使用到这两个接口。

太多的互斥量或临界区，或者临界区太大，就会造成多线程大多数时间不能并发的执行，运行的效率可能比单线程的程序低，太少的互斥量，太小的临界区有可能带来各种错误。这也是很多人诟病多线程的主要原因。一个很有意思的比喻是这样的：在Linux系统上，多进程的程序是立体交通系统，虽然造价高，上坡下坡比较耗油，但是不堵车，多线程的程序属于平面交通系统（同一个地址空间，共享资源较多），但是红绿灯太多了，走走停停，容易堵车。这种比喻是有一定的道理的，所以设计多线程程序的时候，一定首先要从高处着眼，合理分派各个线程的职责，减少线程间的依赖和干涉，多花时间在设计流程上，减少线程间对共享数据的操作，减小临界区的个数和粒度，让多线程阻塞的情况减少，最大可能的并发执行。

互斥量的性能如何，互斥量是如何实现的？请看下章

**9.4.3互斥量的性能和futex机制**

互斥量的性能如何？如果竞争很少发生，或者更极端点，没有竞争的情况下，加锁解锁操作会不会拖垮程序的性能？互斥量是如何实现的？这是本节的主要内容。

互斥量使用了futex机制，futex是fast userspace mutual exclusive的缩写。快速，是如何做到的呢？不是每次加锁或者解锁都需要陷入内核执行系统调用。对于多线程来说，竞争并不是时时刻刻发生的。实际上有很多情况，并没有竞争发生，也不需要同步。我们考虑下面两种情况：

* 线程进入临界区的时候，并没有其他线程尝试进入临界区；
* 线程退出临界区的时候，可能并没有线程在等待进入临界区，

上述两种情况是很普遍的现象。

申请互斥量时，确定没有其他线程尝试进入临界区，陷入内核，执行系统调用查看并不存在的竞争是一种浪费。无独有偶，线程退出临界区，释放互斥量时，如果明明没有线程在等待互斥量，陷入内核执行系统调用，通知互斥量并不存在的等待者，这也是一种效率的浪费。

Linux的POSIX线程库有没有针对这个场景的优化呢？看下测试：

我们的测试场景是，

1 启动一个线程，无锁执行global\_int++，循环10000000，查看时间

global\_int++

2 启动一个线程，global\_int++用互斥量保护

pthread\_mutex\_lock(&mutex);

global\_int++;

pthread\_mutex\_unlock(&mutex);

两种情况下，都是单线程执行，不会有临界区的竞争，我们看下单纯的加锁和解锁操作导致性能下降了多少。

//不加锁不解锁的版本

Starting 1 threads...

proc\_num :0 cost 39185 us

After doing all the math, global\_int value is: 10000000

Expected value is: 10000000

//加锁和解锁的版本

Starting 1 threads...

proc\_num :0 306575 us

After doing all the math, global\_int value is: 10000000

Expected value is: 10000000

可以看到，单线程无竞争的比较，带互斥量保护的版本花费的时间是没有互斥花量的版本花费时间的八倍左右，考虑到临界区的大小仅仅是一个自加操作，是非常轻量级的操作，由此可见，在没有竞争的条件下，加锁和解锁时十分非常轻量级的操作。

Linux如何做到没有竞争情况下，优化加锁和解锁的？

Linux内核提供了futex系统调用：

#include <linux/futex.h>

#include <sys/time.h>

int futex(int \*uaddr, int op, int val, const struct timespec \*timeout,

int \*uaddr2, int val3);

这个系统调用根据op，有多种不同的含义，对于互斥量而言，简单理解有两种op

FUTEX\_WAIT：这个是个等待接口，如果用户空间地址uaddr对应的值等于第三个参数val，那么该线程阻塞在uaddr地址上，直到收到通知被唤醒；如果uaddr地址存储的值不等于第三个参数，当即返回EWOULDBLOCK。timeout参数是设定阻塞最大时间的，如果timeout参数是NULL，就始终等待。uaddr2参数和val3对于FUTEX\_WAIT没有意义。通过猜测也得知，这个接口是用来上锁操作的。

FUTEX\_WAKE:这个用来唤醒一个或者多个等待在uaddr地址上的线程。第三个参数是要唤醒的线程个数。如果值是INT\_MAX表示唤醒所有等待在uaddr上的线程。一般来说，这个值总是1或者INT\_MAX，表示唤醒1个线程或者所有线程。

和加法相比，陷入内核执行系统调用的代价是很高昂的，所以glibc在用户层和系统调用futex配合，对加锁和解锁做了优化。

对于锁的状态而言，锁处于一下三种状态：

* 0 互斥量处于尚未加锁的状态
* 1 互斥量已然处于加锁状态，但是没有线程等待该互斥量
* 2 互斥量已然加锁，并且有1个或者多个线程等待该互斥量

当锁处于状态0时，加锁时不需要陷入内核，因为不需要把自己添加到等待队列上。当锁处于1状态时，解锁时，不需要陷入内核通知其他线程，因为压根没有其他线程等待锁。

下面介绍实现机制

cmpxchg(val,a,b)是原子操作,含义是如果val等于a，那么b赋值给val，不过val是否等于a，都返回val的原始值。

class mutex

{

public:

mutex() : val(0) {}

void lock()

{

int c；

if(c = cmpxchg(val,0,1) != 0)

// 如果原始值是0，表示处于没加锁状态，将val改成1，直接返回

//原始值不是0，表示处于互斥量已被加锁，需要继续执行

do

{

if （c == 2 || cmpxchg(val, 1, 2) != 0)

{

//原始值等于2的时候，才等待

//如果执行futex\_wait时，val已经被改写，不等于2，则当即返回

futex\_wait(&val, 2);

}

// try (again) assuming the lock is free

} while （(c = cmpxchg(val, 0, 2))！= 0）;

//表示有线程unlock，可以但是不知道解锁后是1还是2，保守起见，写成2

}

void unlock()

{

if (atomic\_dec(val) != 1)//atomic\_dec作用是减1并返回的是原始值

{

// 原始值是2，有线程等待互斥量，才会进入

// 如果原始值是1，表示没有线程等待，没必要futex\_wake

val = 0;

futex\_wake(&val, 1);

}

}

private:

int val;

};

上面的代码非常的简短，但是却非常的复杂，包含的场景非常的多，我们从简单到复杂，分别讲述：

解锁的代码相对于加锁时比较简单的，而且要理解加锁，需要先理解解锁的流程

* 如果当前值等于2，那么表示有线程在等待互斥量，需要执行futex\_wakeup，唤醒一个线程（如果存在的话），同时将值改写为0。最多唤醒1个线程原因也比较简单：因为唤醒多个等待互斥量的线程，最终也只能有1个线程获得互斥量，其他线程只能继续进入睡眠状态，对于没有抢到锁的其他线程，被唤醒，然后再次休眠，是一种性能的浪费。
* 如果当前值是1，表示当前线程是唯一等待锁线程，将值改成0，潇洒退出即可。

加锁的代码比较复杂，我们由浅入深的分析：

* 首先如果值是0，我们用原子操作，将值改写成1，表示进入已加锁但是无线程等待互斥量的状态，潇洒返回，无需调用系统调用；由于我们将val原子地改成了1，纵然同时有线程请求锁资源，也不能像自己一样潇洒地返回。
* 如果继续执行，表示最初的值不是0，那么可能是1（已然加锁，当时没有线程等待），也可能是2（互斥量处于加锁状态，并且有线程在等待互斥量）。
  + 如果是2，那就不必说了，已经有线程等待，我们作为后来者，更得调用futex\_wait等待互斥量
  + 如果val是2，在执行futex\_wait之前，锁的持有者已经解锁，将val改写成0，那么futex\_wait会当即返回，走到while语句，去竞争锁资源。
  + 如果是1，表示自己是第一个或者第一批等待的倒霉鬼，原子操作cmpxchg(val,1,2)把val改写成2，调用futex\_wait,内核会将线程挂起安心等待,原子操作出也可能存在竞争，多个线程同时走到了cmpxchg，别的线程已经将val改写成2，不过没区别，这种情况，已然要调用futex\_wait，阻塞在此。
  + 如果val是1，但是cmpxchg(val, 1, 2) != 0原子操作执行之前，锁被所有者释放，val改写成了0，那么cmpxchg这个判断会失败，无需执行futex\_wait，流程会跳转到while，在那里争锁。
  + 如果val是1，cmpxchg(val,1,2)!=0将val改写成2，但是执行futex\_wait之前锁的所有者释放了锁，被改写成2，那么会调用futex\_wait但是系统调用不会阻塞，当即返回失败，就会执行后面的while语句。
  + while判断中会执行cmpxchg(val,0,2)。如果val是0，将val改写成2。注意，这条语句本质是个抢锁的语句。如前面提到的，有多重路径到达while语句中的cmpxchg，并不简单的是从futex\_wait中醒来这一种情况。如果多个线程通过不同路径走到了while中的cmpxchg，大家竞争，只有一个线程成功的将val从0改写成2，安然地退出while循环，其他线程竞争失败（如果存在竞争的话），继续执行循环。注意将val从0改成2，是一种保险或者说保守的做法，事实上，有些场景改写成1是合理的。比如只有两个线程，持有互斥量的线程解锁，等待线程从futex\_wait醒来时，其实val应该改写成1，因为只有原来的等待线程持有锁，锁上并没有等待者。但是我们不能冒这个风险，因为如果万一有其他等待线程，我们将val改写成1，持互斥量的线程解锁的时候，就会因为误以为没有等待者，而没有执行futex\_wake，从而陷入死锁的境地。

前面分析了futex机制，用户空间和内核系统调用futex共同配合完成了互斥量。在锁处于空闲状态时，加锁操作非常轻量，只是几条指令，在没有等待者的情况下，解锁操作也是非常轻量，仅仅是简单的原子自减加上一条比较语句。

## 9.5 读写锁

上一节提到了互斥量，对于临界区，不允许多个线程并发。对共享数据的操作，其实也不一定绝对不允许并发。如果大多数时间，线程只是读共享数据，很少改写共享数据（比如配置，绝大多数情况是读配置，改写配置的情况很少发生），其实多个线程读共享数据，我们是允许并发的，线程间完全互斥，反而影响效率。处于这种考虑，POSIX引入了读写锁。

读写锁比较简单，从下表可以看出，对于读读这种情况，读写锁做了优化，允许大家一起读。

|  |  |  |
| --- | --- | --- |
| 当前锁状态 | 读锁申请 | 写锁申请 |
| 无锁 | OK | OK |
| 读锁 | OK | 阻塞 |
| 写锁 | 阻塞 | 阻塞 |

我原本想把本节的内容放到下一节，线程的高级属性。原因是读写锁并没有像前面提到的那么美好，Bryan Cantrill和Jeff Bonwick在*Real-world Concurrency*提出的并发编程的建议中提到警惕读写锁（Be wary of readers-writer locks）。从我学习到的体会来说，读写锁有一下几个缺点

* 正确性：线程持有读锁，却不小心执行了更新共享数据的操作。这种情况就和没有锁保护一样，会带来共享数据的不一致。乍看之下，自己不会犯如此愚蠢的错误，我也相信大多数读者自己申请读锁，不会更新共享数据，但是随着项目的进行，代码的复杂，你（更有可能是其他维护代码的同事）可能会忘记此处是读锁，而添加了修改共享数据的代码。这就会带来数据的不一致。
* 性能：在临界区比较小的情况下，读写锁性能并没有什么优势。不能简单痴迷于多个读锁请求都可以不用block，也要考虑系统实现读写锁的代价，由于读写锁至少要维护当前读者的个数，所以性能不会好于普通的互斥量。
* 饿死：glibc读写锁的默认行为是读者优先的，也就是说，如果当前锁状态时读锁，纵然存在写锁等待者，已然允许新的读锁请求获得读锁。这种策略就决定了如果读锁源源不断的到来，写锁就得一直等待。这就是写锁饿死。虽然glibc提供了写者优先的策略，可以防范写锁饿死。但是如果写锁请求源源不断地到来，一定会发生读锁饿死，而且目前的实现无法避免。即读写锁只能用在写请求不多的情况。
* 死锁：读锁是可重入的，这可能会引起死锁。刚才提到了读者优先的策略，如果用户选择的不是默认的读者优先，只要有申请写锁的线程在等待，后来的读锁申请者就会被该写锁请求阻塞，这种情况下，读锁重入时，可能会发生死锁。

所以我们不能简单的理解为，读写锁是互斥量的优化，到底使用读写锁还是互斥锁，要根据实际情况，或者通过性能测试的比较，选择合适的锁。

我们讨论了这么多缺点，接下来我们学习读写锁相关的接口。

**9.5.1创建和销毁读写锁**

POSIX库提供了pthread\_rwlock\_t 类型来表示读写锁。和互斥量一样，提供了两种初始化的方法：

#include <[pthread.h](http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html)>

int pthread\_rwlock\_init(pthread\_rwlock\_t \*rwlock,

const pthread\_rwlockattr\_t \*attr);

int pthread\_rwlock\_destroy(pthread\_rwlock\_t \*rwlock);

pthread\_rwlock\_t rwlock=PTHREAD\_RWLOCK\_INITIALIZER;

对于静态变量可以采用PTHREAD\_RWLOCK\_INITIALIZER赋值的方式初始化，对于动态分配的读写锁，或者需要不使用读写锁默认属性的情况，需要用pthread\_rwlock\_init 函数进行初始化。如果第二个属性参数为NULL，那么采用默认属性。

何为默认属性？

1 锁的竞争范围是进程内竞争

2 读写锁的策略是读者优先。

下面是glibc中读写锁默认属性的定义：

static const struct pthread\_rwlockattr default\_attr =

  {

    .lockkind = **PTHREAD\_RWLOCK\_DEFAULT\_NP**,

    .pshared = PTHREAD\_PROCESS\_PRIVATE

  };

enum

{

  PTHREAD\_RWLOCK\_PREFER\_READER\_NP,

  PTHREAD\_RWLOCK\_PREFER\_WRITER\_NP,

  PTHREAD\_RWLOCK\_PREFER\_WRITER\_NONRECURSIVE\_NP,

**PTHREAD\_RWLOCK\_DEFAULT\_NP** = **PTHREAD\_RWLOCK\_PREFER\_READER\_NP**

};

所谓读者优先的策略，是指当前锁状态时读锁，如果线程申请读锁，此时纵然有写锁在等待队列上，仍然允许申请者获得读锁，而不是被写锁阻塞。

对于调用pthread\_rwlock\_init初始化的读写锁，在不需要读写锁的时候，需要调用pthread\_rwlock\_destroy销毁。

**9.5.2 读写锁的加锁和解锁**

读写锁又称共享-独占锁，有共享，有独占。

这是三个读锁上锁的接口

int pthread\_rwlock\_rdlock(pthread\_rwlock\_t \*rwlock);

int pthread\_rwlock\_tryrdlock(pthread\_rwlock\_t \*rwlock);

int pthread\_rwlock\_timedrdlock(pthread\_rwlock\_t \*rwlock,const struct timespec \*abstime);

这是三个写锁上锁的接口：

int pthread\_rwlock\_wrlock(pthread\_rwlock\_t \*rwlock);

int pthread\_rwlock\_trywrlock(pthread\_rwlock\_t \*rwlock);

int pthread\_rwlock\_timedwrlock(pthread\_rwlock\_t \*rwlock,const struct timespec \*abstime);

读锁用以共享模式。如果当前读写锁已经被某线程以读模式占有，那么调用pthread\_rwlock\_rdlock会立刻获得读锁；如果当前读写锁已经被某线程以写模式占有，那么调用pthread\_rwlock\_rdlock会阻塞于此。

写锁用的是独占模式。如果当前读写锁被某线程以写模式占有，不允许任何读锁请求通过，也不允许任何写锁请求通过，读锁请求和写锁请求都要阻塞与此，知道写锁释放持有的锁。

无论读锁还是写锁，锁的释放都是一个接口：

int pthread\_rwlock\_unlock (pthread\_rwlock\_t \*rwlock);

无论是读锁还是写锁，都提供了trylock的功能，当不能获得读锁或者写锁时，调用线程不会阻塞，会立即返回，错误码是EBUSY。

如果是读锁还是写锁都提供了限时等待，如果不能获取读写锁，会陷入阻塞，最多等待到abstime，如果仍然无法获得锁，返回，错误码是ETIMEOUT。

表面看，读写锁介绍到此处可以收工了，其实不然，读写锁是两种类型的锁，当它都存在时，它们之间的竞争关系如何。如果同时到来一大拨读锁请求和写锁请求，他们之间响应由什么特点。这是读写锁的策略决定的。

**9.5.3 读者优先和写者优先**

读写锁有属性是pthread\_rwlockattr\_t 类型，属性中有两个部分：lockkind和pshared。本节只讲lockkind。

所谓lockkind，表示读写锁表现出什么样的行为艺术。对于读写锁，目前有两种设计，一是读者优先，一是写者优先。

enum

{

  PTHREAD\_RWLOCK\_PREFER\_READER\_NP, //读者优先

  PTHREAD\_RWLOCK\_PREFER\_WRITER\_NP, //很唬人，但是也是读者优先

  PTHREAD\_RWLOCK\_PREFER\_WRITER\_NONRECURSIVE\_NP, //写者优先

**PTHREAD\_RWLOCK\_DEFAULT\_NP** = **PTHREAD\_RWLOCK\_PREFER\_READER\_NP**

};

前两个都是读者优先的策略,尤其是第二个货,名字起得很变态,叫什么PREFER\_WRITE，却干着挂羊头卖狗肉的勾当。只有第三个是写者优先的策略。从pthread\_rwlock\_init函数中我们可以看出端倪：

  rwlock->\_\_data.\_\_flags

    = iattr->lockkind == PTHREAD\_RWLOCK\_PREFER\_WRITER\_NONRECURSIVE\_NP;

只认PTHREAD\_RWLOCK\_PREFER\_WRITER\_NONRECURSIVE\_NP是写者优先，其他一律是读者优先。默认行为是读者优先。

说了半天读者优先写者优先，都快成绕口令了，到底何意？

前文提到过，如果当前锁的状态时读锁，存在写锁请求被阻塞，在写锁后面到来的读锁请求如何处理很重要。如果在写锁请求后面到来的读锁请求不被写锁请求阻塞，可以立即响应，写锁的下场可能会比较悲惨。读锁请求源源不断的到来，只要有一个读锁没完成，写锁就没份，好不容易熬走了金日成，金正日又来了，金正日还没熬走，下一任领袖是金正恩，写锁请求就会被活活饿死。这就是天杀的读者优先。

上面这种不排队，乱加塞的行为引起了写锁申请者的强烈不满：凭啥仅仅因为当前是读者拿锁，比我晚来的读锁申请者都不用排队，直接响应。所以glibc又实现了写者优先的策略。所谓写者优先是指，如果当前是读锁，有很多线程在共享读锁，这我管不着，但是一旦我申请写锁，在写锁请求后面到来的读锁请求统统阻塞，不能先于写请求拿到锁。

glibc如何做到这点？glibc引入了下面的变量

\_\_writer :写者锁持有者的线程ID（内核态的线程ID，而非pthread\_self返回的线程ID），

\_\_nr\_readers:当前共享读锁的个数

\_\_nr\_readers\_queued:被阻塞的读锁请求的个数。

\_\_nr\_writers\_queued:被阻塞的写锁请求个数

\_\_writer\_wakeup:唤醒写锁请求次数

\_\_readers\_wakeup:唤醒读锁请求次数,注意读锁请求都是批量唤醒的

OK，介绍这些之后，我们可以分析源码了：

int

\_\_pthread\_rwlock\_rdlock (rwlock)

     pthread\_rwlock\_t \*rwlock;

{

  int result = 0;

  LIBC\_PROBE (rdlock\_entry, 1, rwlock);

  /\* Make sure we are alone.  \*/

  lll\_lock (rwlock->\_\_data.\_\_lock, rwlock->\_\_data.\_\_shared);

  while (1)

    {

      /\*\_\_writer == 0 表示当前锁状态不是写锁 \*/

      if (rwlock->\_\_data.\_\_writer == 0

  /\*不存在被阻塞的写锁申请者（没有苦主），或者读者优先，

\*都可以拿到读锁，快乐的退出了\*/

  && (!rwlock->\_\_data.\_\_nr\_writers\_queued

      || PTHREAD\_RWLOCK\_PREFER\_READER\_P (rwlock)))

{

  /\*增加欢乐的读者锁持有者俱乐部成员的个数 \*/

  if (\_\_builtin\_expect (++rwlock->\_\_data.\_\_nr\_readers == 0, 0))

    {

      /\* Overflow on number of readers.  \*/

      --rwlock->\_\_data.\_\_nr\_readers;

      result = EAGAIN;

    }

  else

    LIBC\_PROBE (rdlock\_acquire\_read, 1, rwlock);

  break;

}

      /\* 如果已经是写锁持有者，再来无耻的申请读锁，这会引起死锁。\*/

      if (\_\_builtin\_expect (rwlock->\_\_data.\_\_writer

    == THREAD\_GETMEM (THREAD\_SELF, tid), 0))

{

  result = EDEADLK;

  break;

}

      /\* 锁没申请到，只能讲读锁排队者计数器++ \*/

      if (\_\_builtin\_expect (++rwlock->\_\_data.\_\_nr\_readers\_queued == 0, 0))

{

  /\* Overflow on number of queued readers.  \*/

  --rwlock->\_\_data.\_\_nr\_readers\_queued;

  result = EAGAIN;

  break;

}

      int waitval = rwlock->\_\_data.\_\_readers\_wakeup;

      /\* Free the lock.  \*/

      lll\_unlock (rwlock->\_\_data.\_\_lock, rwlock->\_\_data.\_\_shared);

      /\* 乖乖再此处等待 \*/

      lll\_futex\_wait (&rwlock->\_\_data.\_\_readers\_wakeup, waitval,

      rwlock->\_\_data.\_\_shared);

      /\* Get the lock.  \*/

      lll\_lock (rwlock->\_\_data.\_\_lock, rwlock->\_\_data.\_\_shared);

      --rwlock->\_\_data.\_\_nr\_readers\_queued;

    }

  /\* We are done, free the lock.  \*/

  lll\_unlock (rwlock->\_\_data.\_\_lock, rwlock->\_\_data.\_\_shared);

  return result;

}

读锁请求被阻塞有两种可能

* 当前锁持有者是写锁
* 写者优先，并且有写锁请求被阻塞，那么纵然当前是读锁，也要阻塞。

下面看申请写锁的流程：

int

\_\_pthread\_rwlock\_wrlock (rwlock)

pthread\_rwlock\_t \*rwlock;

{

int result = 0;

LIBC\_PROBE (wrlock\_entry, 1, rwlock);

/\* Make sure we are alone.  \*/

lll\_lock (rwlock->\_\_data.\_\_lock, rwlock->\_\_data.\_\_shared);

while (1)

{

/\* 如果当前既没有写锁持有者，也没有读锁持有者，我们就可以拿到写锁了 \*/

if (rwlock->\_\_data.\_\_writer == 0 && rwlock->\_\_data.\_\_nr\_readers == 0)

{

/\* 把自己的线程ID写入\_\_writer \*/

rwlock->\_\_data.\_\_writer = THREAD\_GETMEM (THREAD\_SELF, tid);

LIBC\_PROBE (wrlock\_acquire\_write, 1, rwlock);

break;//拿到锁可以退出了

}

/\*如果当前线程已经拿到写锁，再来申请写锁，此处会死锁所以返回EDEADLK\*/

if (\_\_builtin\_expect (rwlock->\_\_data.\_\_writer

== THREAD\_GETMEM (THREAD\_SELF, tid), 0))

{

result = EDEADLK;

break;

}

/\*既然拿不到锁，将写锁等待者队列计数器++\*/

if (++rwlock->\_\_data.\_\_nr\_writers\_queued == 0)

{

/\* Overflow on number of queued writers.  \*/

--rwlock->\_\_data.\_\_nr\_writers\_queued;

result = EAGAIN;

break;

}

int waitval = rwlock->\_\_data.\_\_writer\_wakeup;

/\* Free the lock.  \*/

lll\_unlock (rwlock->\_\_data.\_\_lock, rwlock->\_\_data.\_\_shared);

/\*乖乖等在此处，等待别处释放锁唤醒自己 \*/

lll\_futex\_wait (&rwlock->\_\_data.\_\_writer\_wakeup, waitval,

rwlock->\_\_data.\_\_shared);

/\* Get the lock.  \*/

lll\_lock (rwlock->\_\_data.\_\_lock, rwlock->\_\_data.\_\_shared);

/\* To start over again, remove the thread from the writer list.  \*/

--rwlock->\_\_data.\_\_nr\_writers\_queued;

}

/\* We are done, free the lock.  \*/

lll\_unlock (rwlock->\_\_data.\_\_lock, rwlock->\_\_data.\_\_shared);

return result;

}

写锁拿到锁的条件是当前没有人持有写锁并且当前没有人持有读锁。有一个条件不满足，就得陷入等待。

讨论到这个程度，解锁的行为其实是比较关键的，读锁和写锁都调用同一个接口pthread\_rwlock\_unlock，这个接口有怎样的行为。当同时存在被阻塞的读锁申请者和写锁申请者，解锁会唤醒哪个？让我们走进解锁源码一看究竟：

int

\_\_pthread\_rwlock\_unlock (pthread\_rwlock\_t \*rwlock)

{

LIBC\_PROBE (rwlock\_unlock, 1, rwlock);

lll\_lock (rwlock->\_\_data.\_\_lock, rwlock->\_\_data.\_\_shared);

if (rwlock->\_\_data.\_\_writer)

rwlock->\_\_data.\_\_writer = 0; /\*写锁释放时将\_\_writer改写成0\*/

else

--rwlock->\_\_data.\_\_nr\_readers;/\*读锁释放时，读锁持有者计数器--\*/

/\*写锁释放或者最后一个读锁持有者释放都会面临唤醒其他等待者的责任\*/

if (rwlock->\_\_data.\_\_nr\_readers == 0)

{

//有写锁等待者，先满足写锁申请者，注意只唤醒一个

if (rwlock->\_\_data.\_\_nr\_writers\_queued)

{

++rwlock->\_\_data.\_\_writer\_wakeup;

lll\_unlock (rwlock->\_\_data.\_\_lock, rwlock->\_\_data.\_\_shared);

lll\_futex\_wake (&rwlock->\_\_data.\_\_writer\_wakeup, **1**,

rwlock->\_\_data.\_\_shared);

return 0;

}

/\*没有阻塞的写锁申请者，就唤醒所有的读锁申请者\*/

else if (rwlock->\_\_data.\_\_nr\_readers\_queued)

{

++rwlock->\_\_data.\_\_readers\_wakeup;

lll\_unlock (rwlock->\_\_data.\_\_lock, rwlock->\_\_data.\_\_shared);

lll\_futex\_wake (&rwlock->\_\_data.\_\_readers\_wakeup, **INT\_MAX**,rwlock->\_\_data.\_\_shared);

return 0;

}

}

lll\_unlock (rwlock->\_\_data.\_\_lock, rwlock->\_\_data.\_\_shared);

return 0;

}

解锁流程可以看出，当写锁释放或者最后一个读锁释放时，需要唤醒阻塞在读写锁上的线程。唤醒的策略有写锁等待者，就唤醒写锁，没有写锁等待者就唤醒读锁等待者。

从这个流程上看，写锁存在自私的倾向，当读写锁状态是写锁期间，到来很多写请求和读请求，总是优先处理写请求。如果写锁请求源源不断地到来，一样会将读锁饿死。

通过上面的分析，可以看到，如果存在大量的读写请求，竞争非常激烈的条件下，读写锁存在很大的惯性，如果当前锁状态时读锁状态，在读者优先的策略下，几乎总是读锁请求得到响应，写锁被阻塞，出现写请求饿死的情况。解决的方法是设定成写者优先。如果当前的锁状态是写锁，而且写锁源源不断的带来，这时候，读请求会被饿死，而且无药可救。

下面我写了一个读写锁的程序，来证明这种惯性：

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define N\_THREAD 100

static int share\_cnt = 0;

static pthread\_rwlock\_t rwlock ;

void \*reader(void \*param)

{

    int i = (int) param;

    while(1)

    {

        pthread\_rwlock\_rdlock(&rwlock)  ;

        fprintf(stderr,"reader-%d: the share\_cnt = %d\n",i,share\_cnt);

        pthread\_rwlock\_unlock(&rwlock);

        //      sleep(1);

    }

    return NULL;

}

void \*writer(void \*param)

{

    int i = (int) param;

    while(1)

    {

        pthread\_rwlock\_wrlock(&rwlock)  ;

        share\_cnt++;

        fprintf(stderr,"writer-%d: the share\_cnt = %d\n",i,share\_cnt);

        pthread\_rwlock\_unlock(&rwlock);

        //      sleep(1);

    }

    return NULL;

}

int main()

{

    pthread\_t tid[N\_THREAD] ;

    pthread\_rwlockattr\_t  rwlock\_attr ;

    pthread\_rwlockattr\_init(&rwlock\_attr);

#ifdef WRITE\_FIRST

    pthread\_rwlockattr\_setkind\_np(&rwlock\_attr,PTHREAD\_RWLOCK\_PREFER\_WRITER\_NONRECURSIVE\_NP);

#endif

    pthread\_rwlock\_init(&rwlock,&rwlock\_attr);

    int i = 0;

    int ret = 0;

    pthread\_rwlock\_rdlock(&rwlock);

    for(i = 0;i < N\_THREAD; i++)

    {

        if(i%2 == 0)

        {

            ret = pthread\_create(&tid[i],NULL,reader,(void\*)i);

        }

        else

        {

            ret = pthread\_create(&tid[i],NULL,writer,(void\*)i);

        }

        if(ret != 0)

        {

            fprintf(stderr,"create thread %d failed \n",i);

            break;

        }

    }

    pthread\_rwlock\_unlock(&rwlock);

    while(i-- >0)

    {

        pthread\_join(tid[i],NULL);

    }

    pthread\_rwlockattr\_destroy(&rwlock\_attr);

    pthread\_rwlock\_destroy(&rwlock);

    return ret ;

}

我们创建100个线程，50个读线程50个写线程，读线程只读取share\_cnt的值，而写线程会将share\_cnt的值自加。由于while循环，所以属于读写竞争非常激烈的情况。创建线程之前，主线程会持有读写锁直到所有有线程创建完毕，然后主线程解锁，放任100个线程激烈的竞争读写锁。

如果我们采用读者优先的策略：

gcc –o rwlock\_race rwlock\_race.c –lpthread

会看到由于读线程源源不断地申请读锁，写锁活活被饿死，写线程根本捞不到机会执行。运行N秒之后，share\_cnt仍然是0。

如果我们加上采用写者优先的策略：

gcc –o rwlock\_race rwlock\_race.c –DWRITE\_FIRST –lpthread

情况就完全相反了，自从第一个写锁请求拿到锁之后，从此之后，读锁请求再也拿不到锁了，原因是总是有写锁请求，而写锁释放的时候，总是先唤醒写锁，表现出来很强大的惯性。原因我们从代码层面已经分析过了。

## 9.6 条件等待

条件等待，是POSIX提供的另一种线程间同步的方法。有句很风靡的话，叫做“待我长发及腰，少年你娶我可好？”，这句话稍加改动，改成“待你长发及腰，我来娶你可好？”就能解释了条件等待的精髓。对于一个线程而言，要想继续执行，要依赖某种条件，条件不满足，对该线程而言，它能做的事情就是等待，等待条件的发生，而这种条件，很可能被另外一个线程操控或者决定。当另外一个线程发现条件符合的时候（已然长发及腰），它会选择一个时机去通知等待在这个条件上的少年，哦不对，是线程。当然也有一家女，百家求的情况，百家都在等这家少女长发及腰，待少女长发及腰以后，少女可能有意中人，只通知了一家（pthread\_cond\_signal），也有可能少女是个bitch，满大街通知（pthread\_cond\_broadcast）……

前面说的这些，并非戏言，考虑生产者-消费者模型，如果任务队列处于空的状态，消费者线程就应该停工等待，一直等到队列不空为止。如果没有条件等待，我们的消费者代码可能会写成这样：

pthread\_mutex\_t m = PTHREAD\_MUTEX\_INITIALIZER;

int WaitForTrue()

{

pthread\_mutex\_lock(&m);

while (condition is false)//条件不满足

{

pthread\_mutex\_unlock(&m);//解锁等待其他线程改变共享数据

sleep(n);//睡眠n秒后再次加锁验证条件是否满足

pthread\_mutex\_lock(&m);

}

pthread\_mutex\_unlock(&m);

}

如果条件不满足，就只能睡眠。上面代码的问题是睡眠，纵然解锁之后，sleep之前，条件已经满足了，该线程已然需要睡眠n秒，而不能及时的获得通知。

很自然的，我们需要一种机制，线程在条件不满足的情况下，让出互斥量，让其他线程去折腾，线程再此处等待，等待条件的满足。线程之所以可以安心等待，依赖是其他线程的善良，他确信会有一个线程发现条件满足以后，想它发送信号，并且让出互斥量。如果其他线程不配合（不发信号，不让出互斥量），这个主动让出互斥量等待事件发生的线程就真的等的花儿都谢了。

下面我们详细讲述条件变量。

**9.6.1创建和销毁条件变量**

POSIX库用pthread\_cond\_t类型的变量来表示条件变量。条件变量不是一个值，我们无法给条件变量赋值。一个线程如果要等待某个事件的发生，或者某个条件的满足，那么这个线程需要的是条件变量：线程等待在条件变量上。

和互斥锁一样，使用之前要先初始化。互斥锁有静态初始化，条件变量也一样。简单地吧PTHREAD\_COND\_INITIALIZER赋值给pthread\_cond\_t类型的变量就行了：

pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;

对于动态分配的条件变量，或者对条件变量的属性有所定制，不使用默认属性的条件变量，我们需要用pthread\_cond\_init进行初始化：

int pthread\_cond\_init(pthread\_cond\_t \*cond,

const pthread\_condattr\_t \*attr);

如果采用默认属性，我们可以将NULL作为第二个参数。

对于pthread\_cond\_init初始化的条件变量，不要忘记调用pthread\_cond\_destroy销毁。

int pthread\_cond\_destroy(pthread\_cond\_t \*cond);

对于初始化和销毁注意一下三点：

* 永远不要采用用一个条件变量对另一个条件变量赋值，即pthread\_cond\_t cond\_b = cond\_a是不合法，行为是为定义的。
* 使用PTHREAD\_COND\_INITIALIZE静态初始化的条件变量，不需要销毁
* 调用pthread\_cond\_destroy销毁的条件变量可以调用pthread\_cond\_init重新初始化
* 不要引用已经销毁的条件变量，这样做的的行为时未定义的。

有了条件变量的初始化和销毁，我们可以进入正题，看下条件变量的等待和通知了。

**9.6.2等待和通知条件变量**

条件变量，天生就是与条件的满足与否相伴而生的。通常线程会对一个条件进行测试，如果条件不满足， 就等待（pthread\_cond\_wait），或者等待一段有限的时间（pthread\_cond\_timedwait）。

我们看下等待相关的API

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 abstime);

从接口上可以看出，条件等待总是和互斥量绑定在一起的。为什么这样设计？条件等待属于线程间同步的手段，如果只有一个线程，条件不满足，等待千年也是枉然，必然要有一个线程通过某些操作，改变了共享数据，使原先不满足的条件变得满足了，并且友好地通知等待在条件变量上的线程。条件不会无缘无故地突然满足了，必然牵扯到共享数据的变化。所以一定要有互斥锁。没有互斥锁，无法安全的获取和修改共享数据。

就算如此，先调用pthread\_mutex\_lock，发现条件不满足，解锁，然后等待在条件上就行了，为什么要把互斥锁作为参数传给pthread\_cond\_wait？就像下面所示代码：

//错误的设计

pthread\_mutex\_lock(&m)

while(condition\_is\_false)

{

pthread\_mutex\_unlock(&m);

//解锁之后，等待之前，可能条件已经满足，信号已经发出，该信号可能会被错过；

cond\_wait(&cv);

pthread\_mutex\_lock(&m);

}

原因在与，上面解锁和等待不是原子操作。如果解锁以后，调用cond\_wait之前，如果已经有其他线程获取到了互斥量，并且满足了条件，已经发出了通知信号，cond\_wait将错过这个信号，可能导致永远处于阻塞状态。所以解锁加等待必须是一个原子性的操作，确保已经注册到事件的等待队列之前，不应该有其他线程可以获得互斥量。

那先注册等待事件，后释放锁不行吗？注意条件等待是个阻塞型的接口，不单单是注册在事件的等待队列上，线程也会因此阻塞于此，导致锁无法释放，其他线程获取不到互斥量，也就无法通过改变共享数据，使等待的条件满足，这会造成死锁

下面为伪码显示了POSIX如何使用条件变量v和互斥量m来等待条件发生：

pthread\_mutex\_lock(&m);

while(condition\_is\_false)

pthread\_cond\_wait(&v,&m);//此处会阻塞

/\*如果代码运行到此处，表示我们等待的条件满足了，并且在此持有了互斥量\*/

/\*在满足条件的情况下，做你想做的事情。\*/

pthread\_mutex\_unlock(&m);

pthread\_cond\_wait函数只能有拥有互斥量的线程调用，当该函数返回的时候，系统会却确保该线再次持有互斥量，所以这个接口容易给人一种误解，就是该线程一直持有互斥量。事实上不是这样的。这个接口向系统声明了我的心在等待，永远在等待之后，就把互斥量给释放了。这样其他线程就有机会持有互斥量，操作共享数据，触发变化,使线程等待的条件满足。

既然互斥量和条件变量关系如此紧密，为什么不干脆将互斥量设计成条件变量的一部分。原因是，同一个互斥量上可能有不同的条件变量，比如说，有的线程希望队列不空的时候，发送信号，有的线程希望队列满的时候发送通知给它（为了创建更多的线程做消费者或者其他目的）。

pthread\_cond\_timedwait函数和pthread\_cond\_wait的工作方式几乎一样，只是指定了一个超时的时间。注意这个时间是绝对时间，而不是相对时间。如果最多等待2分钟，这个值应该是当前时间加上2分钟。

上面将互斥量和条件变量配合使用的示范代码有个很有意思的地方，就是用了while语句，醒来之后要再次判断，条件是否满足。

while(condition\_is\_false)

pthread\_cond\_wait(&v,&m);//此处会阻塞

为什么不写成：

if(condition\_is\_false)

pthread\_cond\_wait(&v,&m);//此处会阻塞

唤醒以后，再次检查条件是否满足，是不是多此一举？

答案是我们不得不如此。因为唤醒中存在虚假唤醒（spurious wakeup），换言之，就言之，条件尚未满足，pthread\_cond\_wait就返回了。在一些实现中，即使没有其他线程像条件变量发送信号，等待此条件变量的线程仍可能醒来。

看起来这像是个bug，但是它实实在在是存在的。为什么虚假唤醒会存在？一个原因是pthread\_cond\_wait是futex系统调用，属于阻塞型的系统调用，当系统调用被信号中断的时候，会返还-1，并且把errno置为EINTR。很多这种系统调用为了防止被信号中断都会重启系统调用，如下所示：

pid\_t r\_wait(int \*stat\_loc)

{

    int retval;

    while(((retval = wait(stat\_loc)) ==-1 && (errno == EINTR));

    return retval;

}

但是futex不一样，在futex返回之后，到重启系统调用之前，可能已经调用过pthread\_cond\_signal或者pthread\_cond\_broadcast。一旦错失，再次调用pthread\_cond\_wait可能就会无限制地等待下去。为了防止这种情况，宁可虚假唤醒，也不能再次调用pthread\_cond\_wait，以免陷入无穷的等待。

除了上面信号的因素外，存在条件满足了发送信号，等到调用pthread\_cond\_wait的线程得到CPU资源时，条件再次不满足了。好在无论是那种情况，醒来之后再次测试条件就可以解决虚假等待的问题。

条件等待，等于把控制权交给了别的线程，它信任别的线程会在合适的时机通知它，这是多大的信任啊。如果其他线程忘记发送信号，那么条件等待的线程就彻底悲剧了。

如何发送信号通知等待的线程呢？POSIX提供了两个接口：

int pthread\_cond\_signal(pthread\_cond\_t \*cond);

int pthread\_cond\_broadcast(pthread\_cond\_t \*cond);

pthread\_cond\_signal负责唤醒一个等待在条件变量上的线程，pthread\_cond\_broadcast顾名思义，广播唤醒所有等待在条件变量上的线程。

等一下，刚才讲解pthread\_cond\_wait的时候，提到，线程醒来时确保持有互斥量，为何广播能唤醒所有等待在条件变量上的线程。不是矛盾吗？

答案是不矛盾，所有的线程被广播唤醒了之后，集体争夺互斥锁，没抢到的继续睡。从内核中醒来，然后继续睡去，是一种性能的浪费。

使用通知机制来完成线程同步，代码范例如下：

//为让流程更加清晰，忽略了error handle

pthread\_mutex\_lock(&m);

/\*一些对共享数据的操作，会导致另一个线程等待的条件满足\*/

pthread\_cond\_signal(&cond);//此处可以为pthread\_cond\_broadcast

pthread\_mutex\_unlock(&m);

发送信号，通知等待在条件上的线程，然后解锁互斥量。

注意范例代码中先发送信号，然后解锁互斥量，这个顺序不是必须的，可以颠倒。标准允许任意顺序执行这两个调用。

有什么区别？

先通知条件变量，后解锁互斥量，效率会比先解锁，后通知条件变量低。因为先通知后解锁，执行pthread\_cond\_wait的线程可能在互斥量已然处于加锁状态的时候醒来，发现互斥量仍没解锁，就再次休眠，从而导致了多余的上下文切换。某些实现使用等待变形（wait morphing）来优化这个问题：并不真正的唤醒执行pthread\_cond\_wait的线程，而是将线程从条件变量的等待队列转移到互斥量的等待队列上，从而消灭了无谓的上下文切换。

glibc对pthread\_cond\_broadcast做了类似的优化，只唤醒一个线程，将其他线程从条件变量的等待队列搬移到了互斥量的等待队列。对实现细节感兴趣的可以参阅Ulrich Drepper的*Futexes are tricky*

先解锁，后通知条件变量虽然可能会有性能上的优势，但是会带来其他的问题。如果存在一个高优先级的进程等待在互斥量上，也等待在条件变量上，同时存在低优先级的进程只等待在互斥量上。一旦先解锁，低优先级的进程就可能抢先获得互斥量，待通知条件变量之后，高优先级的进程发现锁已经被低优先级的进程抢走了，无法执行。这是优先级翻转的典型案例。

## 9.7 总结

上面所有的接口，基本是多线程编程常用的接口，熟练掌握这些接口，就基本能够应对大多数情况下的多线程编程了。多线程还提供了很多高级的属性，下章学习。

1. *Linux/UNIX系统编程手册，*更详细地介绍了LinuxThreads的缺陷 [↑](#footnote-ref-1)
2. Redhat公司由Ulrich Drepper和Ingo Molnar完成的的*The Native POSIX Thread Library for Linux*文档 [↑](#footnote-ref-2)