Linux 线程实现机制分析-杨沙洲

本文作者为杨沙洲博士

国防科技大学计算机学院

2003 年 5 月 19 日

> 自从多线程编程的概念出现在 Linux 中以来，Linux 多线程应用的发展总是与两个问题脱不开干系：兼容性、效率。本文从线程模型入手，通过分析目前 Linux 平台上最流行的 LinuxThreads 线程库的实现及其不足，描述了 Linux 社区是如何看待和解决兼容性和效率这两个问题的。

一.基础知识：线程和进程

按照教科书上的定义，进程是资源管理的最小单位，线程是程序执行的最小单位。在操作系统设计上，从进程演化出线程，最主要的目的就是更好的支持SMP以及减小（进程/线程）上下文切换开销。

无论按照怎样的分法，一个进程至少需要一个线程作为它的指令执行体，进程管理着资源（比如cpu、内存、文件等等），而将线程分配到某个cpu上执行。一个进程当然可以拥有多个线程，此时，如果进程运行在SMP机器上，它就可以同时使用多个cpu来执行各个线程，达到最大程度的并行，以提高效率；同时，即使是在单cpu的机器上，采用多线程模型来设计程序，正如当年采用多进程模型代替单进程模型一样，使设计更简洁、功能更完备，程序的执行效率也更高，例如采用多个线程响应多个输入，而此时多线程模型所实现的功能实际上也可以用多进程模型来实现，而与后者相比，线程的上下文切换开销就比进程要小多了，从语义上来说，同时响应多个输入这样的功能，实际上就是共享了除cpu以外的所有资源的。

针对线程模型的两大意义，分别开发出了核心级线程和用户级线程两种线程模型，分类的标准主要是线程的调度者在核内还是在核外。前者更利于并发使用多处理器的资源，而后者则更多考虑的是上下文切换开销。在目前的商用系统中，通常都将两者结合起来使用，既提供核心线程以满足smp系统的需要，也支持用线程库的方式在用户态实现另一套线程机制，此时一个核心线程同时成为多个用户态线程的调度者。正如很多技术一样，"混合"通常都能带来更高的效率，但同时也带来更大的实现难度，出于"简单"的设计思路，Linux从一开始就没有实现混合模型的计划，但它在实现上采用了另一种思路的"混合"。

在线程机制的具体实现上，可以在操作系统内核上实现线程，也可以在核外实现，后者显然要求核内至少实现了进程，而前者则一般要求在核内同时也支持进程。核心级线程模型显然要求前者的支持，而用户级线程模型则不一定基于后者实现。这种差异，正如前所述，是两种分类方式

当核内既支持进程也支持线程时，就可以实现线程-进程的"多对多"模型，即一个进程的某个线程由核内调度，而同时它也可以作为用户级线程池的调度者，选择合适的用户级线程在其空间中运行。这就是前面提到的"混合"线程模型，既可满足多处理机系统的需要，也可以最大限度的减小调度开销。绝大多数商业操作系统（如Digital Unix、Solaris、Irix）都采用的这种能够完全实现POSIX1003.1c标准的线程模型。在核外实现的线程又可以分为"一对一"、"多对一"两种模型，前者用一个核心进程（也许是轻量进程）对应一个线程，将线程调度等同于进程调度，交给核心完成，而后者则完全在核外实现多线程，调度也在用户态完成。后者就是前面提到的单纯的用户级线程模型的实现方式，显然，这种核外的线程调度器实际上只需要完成线程运行栈的切换，调度开销非常小，但同时因为核心信号（无论是同步的还是异步的）都是以进程为单位的，因而无法定位到线程，所以这种实现方式不能用于多处理器系统，而这个需求正变得越来越大，因此，在现实中，纯用户级线程的实现，除算法研究目的以外，几乎已经消失了。

Linux内核只提供了轻量进程的支持，限制了更高效的线程模型的实现，但Linux着重优化了进程的调度开销，一定程度上也弥补了这一缺陷。目前最流行的线程机制LinuxThreads所采用的就是线程-进程"一对一"模型，调度交给核心，而在用户级实现一个包括信号处理在内的线程管理机制。Linux-LinuxThreads的运行机制正是本文的描述重点。

二.Linux 2.4内核中的轻量进程实现

最初的进程定义都包含程序、资源及其执行三部分，其中程序通常指代码，资源在操作系统层面上通常包括内存资源、IO资源、信号处理等部分，而程序的执行通常理解为执行上下文，包括对cpu的占用，后来发展为线程。在线程概念出现以前，为了减小进程切换的开销，操作系统设计者逐渐修正进程的概念，逐渐允许将进程所占有的资源从其主体剥离出来，允许某些进程共享一部分资源，例如文件、信号，数据内存，甚至代码，这就发展出轻量进程的概念。Linux内核在2.0.x版本就已经实现了轻量进程，应用程序可以通过一个统一的clone()系统调用接口，用不同的参数指定创建轻量进程还是普通进程。在内核中，clone()调用经过参数传递和解释后会调用do_fork()，这个核内函数同时也是fork()、vfork()系统调用的最终实现：

```
<linux-2.4.20/kernel/fork.c>

int do_fork(unsigned long clone_flags, unsigned long stack_start,
struct pt_regs *regs, unsigned long stack_size)
```

其中的clone_flags取自以下宏的"或"值：

```
<linux-2.4.20/include/linux/sched.h>
#define CSIGNAL      0x000000ff
/* signal mask to be sent at exit */
#define CLONE_VM    0x00000100
/* set if VM shared between processes */
#define CLONE_FS
/* set if fs info share
```

```
#define CLONE_FILES     0x00000400
/* set if open files shared between processes */
#define CLONE_SIGHAND  0x00000800
/* set if signal handlers and blocked signals shared */
#define CLONE_PID    0x00001000  /* set if pid shared */
#define CLONE_PTRACE  0x00002000
```

在do_fork()中，不同的clone_flags将导致不同的行为，对于LinuxThreads，它使用（CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND）参数来调用clone()创建"线程"，表示共享内存、共享文件系统访问计数、共享文件描述符表，以及共享信号处理方式。本节就针对这几个参数，看看Linux内核是如何实现这些资源的共享的。

1.CLONE_VM

do_fork()需要调用copy_mm()来设置task_struct中的mm和active_mm项，这两个mm_struct数据与进程所关联的内存空间相对应。如果do_fork()时指定了CLONE_VM开关，copy_mm()将把新的task_struct中的mm和active_mm设置成与current的相同，同时提高该mm_struct的使用者数目（mm_struct::mm_users）。也就是说，轻量级进程与父进程共享内存地址空间，由下图示意可以看出mm_struct在进程中的地位：

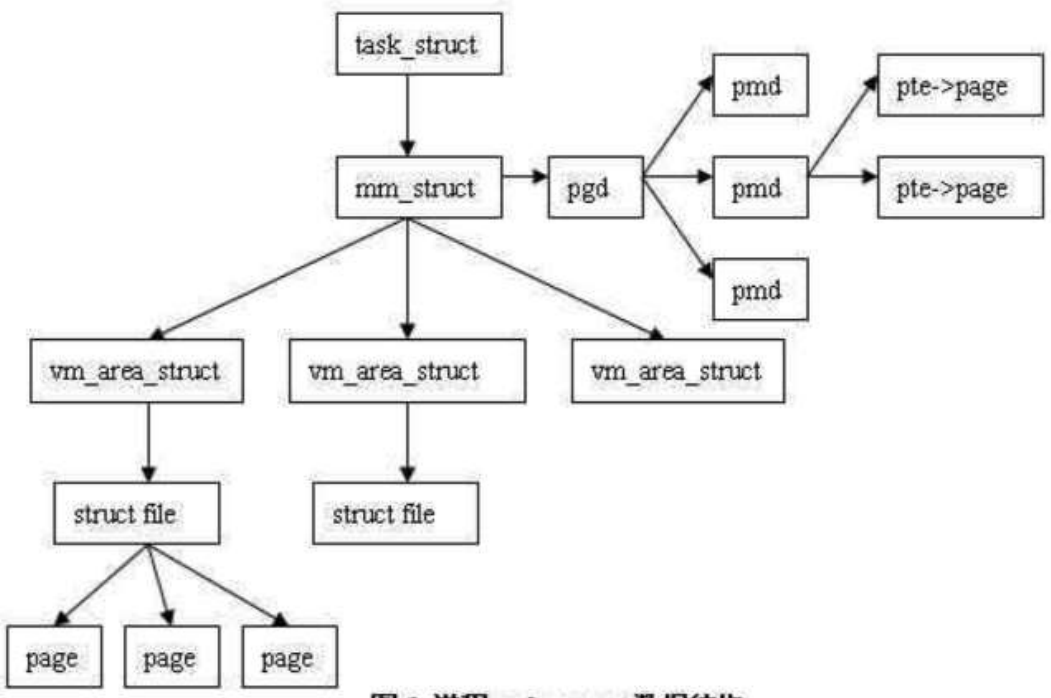

图 1 进程 task_struct 数据结构

2.CLONE_FS

task_struct中利用fs（struct fs_struct *）记录了进程所在文件系统的根目录和当前目录信息，do_fork()时调用copy_fs()复制了这个结构，而对于轻量级进程则仅增加fs->count计数，与父进程共享相同的

系统相关的信息，进程中任何一个线程改变当前目录、根目录等信息都将直接影响到其他线程。

3.CLONE_FILES

一个进程可能打开了一些文件，在进程结构task_struct中利用files（struct files_struct *）来保存进程打开的文件结构（struct file）信息，do_fork()中调用了copy_files()来处理这个进程属性；轻量级进程与父进程是共享该结构的，copy_files()时仅增加files->count计数。这一共享使得任何线程都能访问进程所维护的打开文件，对它们的操作会直接反映到进程中的其他线程。

4.CLONE_SIGHAND

每一个Linux进程都可以自行定义对信号的处理方式，在task_struct中的sig（struct signal_struct）中使用一个struct k_sigaction结构的数组来保存这个配置信息，do_fork()中的copy_sighand()负责复制该信息；轻量级进程不进行复制，而仅仅增加signal_struct::count计数，与父进程共享该结构。也就是说，子进程与父进程的信号处理方式完全相同，而且可以相互更改。

do_fork()中所做的工作很多，在此不详细描述。**对于SMP系统，所有的进程fork出来后，都被分配到与父进程相同的cpu上，一直到该进程被调度时才会进行cpu选择。**

尽管Linux支持轻量级进程，但并不能说它就支持核心级线程，因为Linux的"线程"和"进程"实际上处于一个调度层次，共享一个进程标识符空间，这种限制使得不可能在Linux上实现完全意义上的POSIX线程机制，因此众多的Linux线程库实现尝试都只能尽可能实现POSIX的绝大部分语义，并在功能上尽可能逼近。

三.LinuxThread的线程机制

在开始时，Linux是完全的Unix克隆，在内核中并不支持线程。但是它的确可以通过clone()系统调用将进程作为调度的实体。这个调用创建了调用进程的一个拷贝，这个拷贝与调用进程共享相同的地址空间。LinuxThreads万案使用这个调用来完全在用户空间模拟对线程的支持。不幸的是，这个方案有太多缺点，让Windows总是有一种“一直被追赶从未被超越”的自豪。

LinuxThreads是目前Linux平台上使用最为广泛的线程库，由Xavier Leroy (Xavier.Leroy@inria.fr)负责开发完成，**并已绑定在GLIBC（glibc库）中发行。它所实现的就是基于核心轻量级进程的"一对一"线程模型，一个线程实体对应一个核心轻量级进程，而线程之间的管理在核外函数库中实现。**

NPTL是一种1:1的线程方案，一个线程会与内核的一个调度实体一一对应，线程的创建和回收都由内核负责，这样就可以规避掉LinuxThreads的一切问题。这是一种最简单的合理线程实现方案。但是业界还有另外一个备选方案，就是m:n方案。这种方案中用户线程要多于调度实体。如果NPTL选择以这种方式实现的话，会使得线程上下文切换更快，因为它避免了系统调用。但是m:n的方案是以系统复杂度为代价的。既然Linux的骨子里有些“笨”，复杂性的东西是搞不来的，所以NPTL采用的依然是1:1的线程方案。

NPTL是在Linux 2.6内核开始引入的。一个比较有趣的地方是，**Linux内核本身的多任务调度实体被称为“内核线程”**。而且经常有人会非常兴奋的说，Linux已经跟Windows一样了，是以线程为调度实体的。的确不假，从2.6开始，线程是Linux原生支持的特性了，**但是与Windows还是有很大差别的。**

1.线程描述数据结构及实现限制

LinuxThreads定义了一个struct _pthread_descr_struct数据结构来描述线程，并使用全局数组变量__pthread_handles来描述和引用进程所辖线程。在__pthread_handles中的前两项，LinuxThreads定义了两个全局的系统线程：__pthread_initial_thread和__pthread_manager_thread，并用__pthread_main_thread表征__pthread_manager_thread的父线程（初始为__pthread_initial_thread）。

struct _pthread_descr_struct是一个双环链表结构，__pthread_manager_thread所在的链表仅包括它一个元素，实际上，__pthread_manager_thread是一个特殊线程，LinuxThreads仅使用了其中的errno、p_pid、p_priority等三个域。而__pthread_main_thread所在的链则将进程中所有用户线程串在了一起。经过一系列pthread_create()之后形成的__pthread_handles数组将如下图所示：

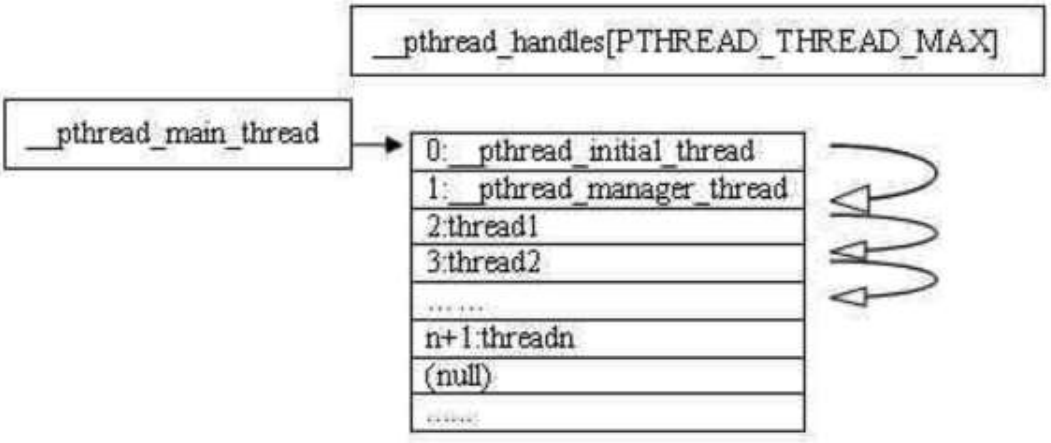

图 2 __pthread_handles 数组结构 CSDN @monsion

新创建的线程将首先在__pthread_handles数组中占据一项，然后通过数据结构中的链指针连入以__pthread_main_thread为首指针的链表中。这个链表的使用在介绍线程的创建和释放的时候将提到。

LinuxThreads遵循POSIX1003.1c标准，其中对线程库的实现进行了一些范围限制，比如进程最大线程数，线程私有数据区大小等等。在LinuxThreads的实现中，基本遵循这些限制，但也进行了一定的改动，改动的趋势是放松或者说扩大这些限制，使编程更加方便。这些限定宏主要集中在sysdeps/unix/sysv/linux/bits/local_lim.h（不同平台使用的文件位置不同）中，包括如下几个：

每进程的私有数据key数，POSIX定义_POSIX_THREAD_KEYS_MAX为128，LinuxThreads使用PTHREAD_KEYS_MAX，1024；私有数据释放时允许执行的操作数，LinuxThreads与POSIX一致，[illegible]
4；每进程的线程数，POSIX定义[illegible]

（PTHREAD_THREADS_MAX）；线程运行栈最小空间大小，POSIX未指定，LinuxThreads使用PTHREAD_STACK_MIN，16384（字节）。

2.管理线程

而且管理线程只能在一个CPU上运行，显然不适合现在的多核CPU。 Linux就这个范儿 第15章 七种武器

"一对一"模型的好处之一是线程的调度由核心完成了，而其他诸如线程取消、线程间的同步等工作，都是在核外线程库中完成的。在LinuxThreads中，专门为每一个进程构造了一个管理线程，负责处理线程相关的管理工作。当进程第一次调用pthread_create()创建一个线程的时候就会创建（__clone()）并启动管理线程。

在一个进程空间内，管理线程与其他线程之间通过一对"管理管道（manager_pipe[2]）"来通讯，该管道在创建管理线程之前创建，在成功启动了管理线程之后，管理管道的读端和写端分别赋给两个全局变量__pthread_manager_reader和__pthread_manager_request，之后，每个用户线程都通过__pthread_manager_request向管理线程发请求，但管理线程本身并没有直接使用__pthread_manager_reader，管道的读端（manager_pipe[0]）是作为__clone()的参数之一传给管理线程的，管理线程的工作主要就是监听管道读端，并对从中取出的请求作出反应。

创建管理线程的流程如下所示：
（全局变量pthread_manager_request初值为-1）

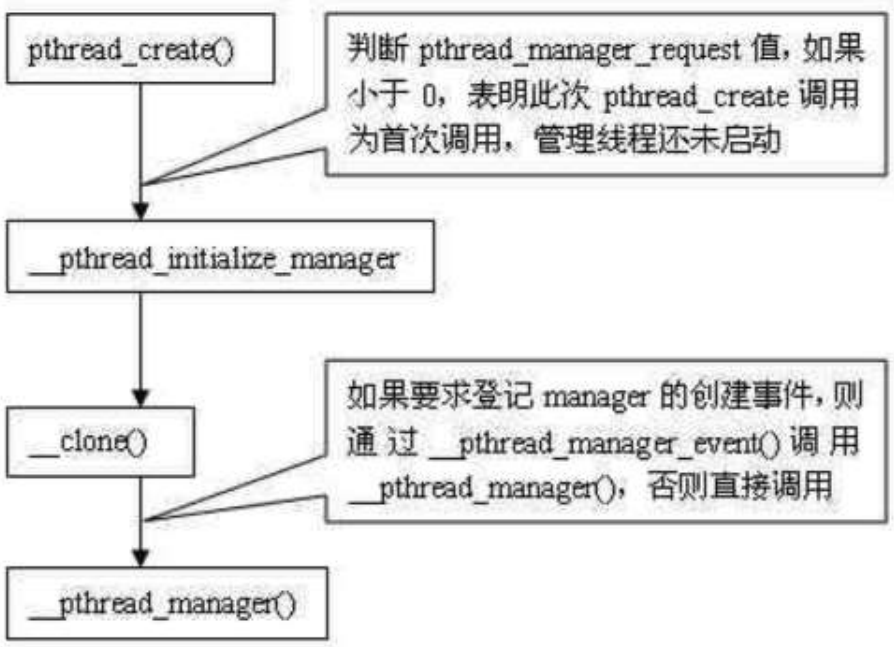

图 3 创建管理线程的流程

初始化结束后，在__pthread_manager_thread中记录了轻量级进程号以及核外分配和管理的线程id，2*PTHREAD_THREADS_MAX+1这个数值不会与任何常规用户线程id冲突。**管理线程作为pthread_create()的调用者线程的子线程运行，而pthread_create()所创建的那个用**

际上是管理线程的子线程。（此处

__pthread_manager()就是管理线程的主循环所在，在进行一系列初始化工作后，进入while(1)循环。在循环中，线程以2秒为timeout查询（__poll()）管理管道的读端。**在处理请求前，检查其父线程（也就是创建manager的主线程）是否已退出，如果已退出就退出整个进程。如果有退出的子线程需要清理，则调用pthread_reap_children()清理。**

然后才是读取管道中的请求，根据请求类型执行相应操作（switch-case）。具体的请求处理，源码中比较清楚，这里就不赘述了。

3.线程栈

在LinuxThreads中，管理线程的栈和用户线程的栈是分离的，管理线程在进程堆中通过malloc()分配一个THREAD_MANAGER_STACK_SIZE字节的区域作为自己的运行栈。

用户线程的栈分配办法随着体系结构的不同而不同，主要根据两个宏定义来区分，一个是NEED_SEPARATE_REGISTER_STACK，这个属性仅在IA64平台上使用；另一个是FLOATING_STACK宏，在i386等少数平台上使用，此时用户线程栈由系统决定具体位置并提供保护。与此同时，用户还可以通过线程属性结构来指定使用用户自定义的栈。因篇幅所限，这里只能分析i386平台所使用的两种栈组织方式：FLOATING_STACK方式和用户自定义方式。

在FLOATING_STACK方式下，LinuxThreads利用mmap()从内核空间中分配8MB空间（i386系统缺省的最大栈空间大小，如果有运行限制（rlimit），则按照运行限制设置），使用mprotect()设置其中第一页为非访问区。该8M空间的功能分配如下图：

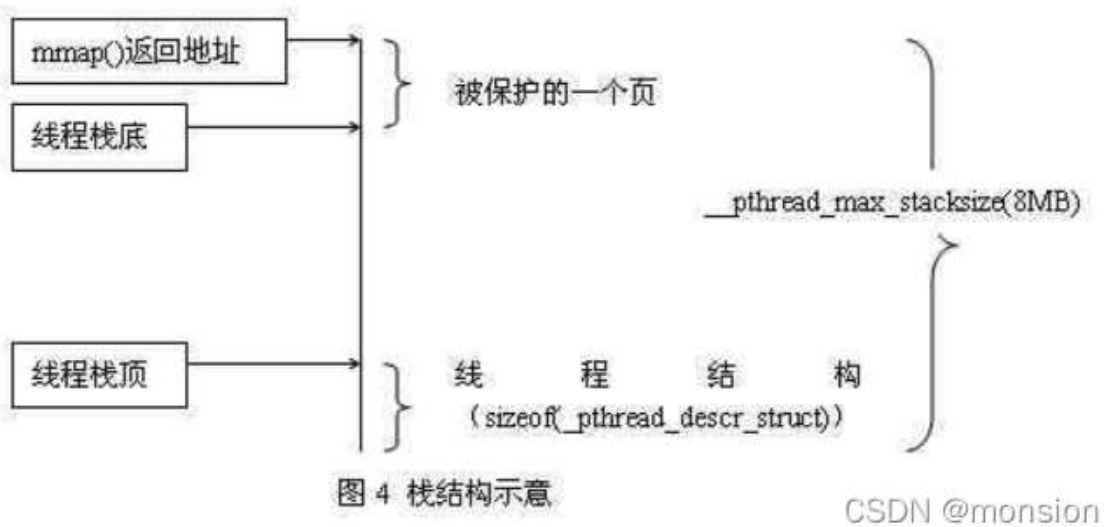

图 4 栈结构示意

低地址被保护的页面用来监测栈溢出。

对于用户指定的栈，在按照指针对界后，设置线程栈顶，并计算出栈底，不做保护，正确性由用户自己保证。

不论哪种组织方式，线程描述结构总是位于栈顶紧邻堆栈的位置。

4.线程id和进程id

每个LinuxThreads线程都同时具有线程id和进程id，其中进程id就是内核所维护的进程号，而线程id则由LinuxThreads分配和维护。

__pthread_initial_thread的线程id为PTHREAD_THREADS_MAX，__pthread_manager_thread的是2*PTHREAD_THREADS_MAX+1，第一个用户线程的线程id为PTHREAD_THREADS_MAX+2，此后第n个用户线程的线程id遵循以下公式：

```
tid=n*PTHREAD_THREADS_MAX+n+1
```

这种分配方式保证了进程中所有的线程（包括已经退出）都不会有相同的线程id，而线程id的类型pthread_t定义为无符号长整型（unsigned long int），也保证了有理由的运行时间内线程id不会重复。

从线程id查找线程数据结构是在pthread_handle()函数中完成的，实际上只是将线程号按PTHREAD_THREADS_MAX取模，得到的就是该线程在__pthread_handles中的索引。

5.线程的创建

在pthread_create()向管理线程发送REQ_CREATE请求之后，管理线程即调用pthread_handle_create()创建新线程。分配栈、设置thread属性后，以pthread_start_thread()为函数入口调用__clone()创建并启动新线程。pthread_start_thread()读取自身的进程id号存入线程描述结构中，并根据其中记录的调度方法配置调度。一切准备就绪后，再调用真正的线程执行函数，并在此函数返回后调用pthread_exit()清理现场。

6.LinuxThreads的不足

由于Linux内核的限制以及实现难度等等原因，LinuxThreads并不是完全POSIX兼容的，在它的发行README中有说明。

1)进程id问题

这个不足是最关键的不足，引起的原因牵涉到LinuxThreads的"一对一"模型。

Linux内核并不支持真正意义上的线程，LinuxThreads是用与普通进程具有同样内核调度视图的轻量级进程来实现线程支持的。这些轻量级进程拥有独立的进程id，在进程调度、信号处理、IO等方面享有与普通进程一样的能力。在源码阅读者看来，就是Linux内核的clone()没有实现对CLONE_PID参数的支持。

在内核do_fork()中对CLONE_PID的处理是这样的：

```
if (clone_flags &
      if (current
```

```
                        goto fork_out;
        }
```

这段代码表明，目前的Linux内核仅在pid为0的时候认可CLONE_PID参数，实际上，仅在SMP初始化，手工创建进程的时候才会使用CLONE_PID参数。

按照POSIX定义，同一进程的所有线程应该共享一个进程id和父进程id，这在目前的"一对一"模型下是无法实现的。

2)信号处理问题

由于异步信号是内核以进程为单位分发的，而LinuxThreads的每个线程对内核来说都是一个进程，且没有实现"线程组"，因此，某些语义不符合POSIX标准，比如没有实现向进程中所有线程发送信号，README对此作了说明。

如果核心不提供实时信号，LinuxThreads将使用SIGUSR1和SIGUSR2作为内部使用的restart和cancel信号，这样应用程序就不能使用这两个原本为用户保留的信号了。**在Linux kernel 2.1.60以后的版本都支持扩展的实时信号（从_SIGRTMIN到_SIGRTMAX），因此不存在这个问题。**

某些信号的缺省动作难以在现行体系上实现，比如SIGSTOP和SIGCONT，LinuxThreads只能将一个线程挂起，而无法挂起整个进程。

3)线程总数问题

LinuxThreads将每个进程的线程最大数目定义为1024，但实际上这个数值还受到整个系统的总进程数限制，这又是由于线程其实是核心进程。

在kernel 2.4.x中，采用一套全新的总进程数计算方法，使得总进程数基本上仅受限于物理内存的大小，计算公式在kernel/fork.c的fork_init()函数中：

```
        max_threads = mempages / (THREAD_SIZE/PAGE_SIZE) / 8
```

在i386上，THREAD_SIZE=2*PAGE_SIZE，PAGE_SIZE=2^12（4KB），mempages=物理内存大小/PAGE_SIZE，对于256M的内存的机器，mempages=256*2^20/2^12=256*2^8，此时最大线程数为4096。

但为了保证每个用户（除了root）的进程总数不至于占用一半以上物理内存，fork_init()中继续指定：

```
    init_task.rlim[RLIMIT_NPROC].rlim_cur = max_threads/2;
    init_task.rlim[RLIMIT_NPROC].rlim_max = max_threads/2;
```

这些进程数目的检查都在do_fork()中进行，因此，对于LinuxThreads来说，线程总数同时受这三个因素的限制。

4)管理线程问题

管理线程容易成为瓶颈，这是这种结构的通病；同时，管理线程又负责用户线程的清理工作，因此，尽管管理线程已经屏蔽了大部分的信号，但一旦管理线程死亡，用户线程就不得不手工清理了，而且用户线程并不知道管理线程的状态，之后的线程创建等请求将无人处理。

5)同步问题

LinuxThreads中的线程同步很大程度上是建立在信号基础上的，这种通过内核复杂的信号处理机制的同步方式，效率一直是个问题。

6）其他POSIX兼容性问题

Linux中很多系统调用，按照语义都是与进程相关的，比如nice、setuid、setrlimit等，在目前的LinuxThreads中，这些调用都仅仅影响调用者线程。

7）实时性问题

线程的引入有一定的实时性考虑，但LinuxThreads暂时不支持，比如调度选项，目前还没有实现。不仅LinuxThreads如此，标准的Linux在实时性上考虑都很少。

四.其他的线程实现机制

LinuxThreads的问题，特别是兼容性上的问题，严重阻碍了Linux上的跨平台应用（如Apache）采用多线程设计，从而使得Linux上的线程应用一直保持在比较低的水平。在Linux社区中，已经有很多人在为改进线程性能而努力，其中既包括用户级线程库，也包括核心级和用户级配合改进的线程库。目前最为人看好的有两个项目，**一个是RedHat公司牵头研发的NPTL（Native Posix Thread Library），另一个则是IBM投资开发的NGPT（Next Generation Posix Threading）**，二者都是围绕完全兼容POSIX 1003.1c，同时在核内和核外做工作以而实现多对多线程模型。这两种模型都在一定程度上弥补了LinuxThreads的缺点，且都是重起炉灶全新设计的。

1.NPTL

NPTL的设计目标归纳可归纳为以下几点：

- POSIX兼容性
- SMP结构的利用
- 低启动开销
- 低链接开销（即不使用线程的程序不应当受线程库的影响）
- 与LinuxThreads应用的二进制兼容性
- 软硬件的可扩展能力
- 多体系结构支持

- NUMA支持
- 与C++集成

在技术实现上，NPTL仍然采用1:1的线程模型，并配合glibc和最新的Linux Kernel2.5.x开发版在信号处理、线程同步、存储管理等多方面进行了优化。和LinuxThreads不同，NPTL没有使用管理线程，核心线程的管理直接放在核内进行，这也带了性能的优化。

主要是因为核心的问题，NPTL仍然不是100%POSIX兼容的，但就性能而言相对LinuxThreads已经有很大程度上的改进了。

2.NGPT

IBM的开放源码项目NGPT在2003年1月10日推出了稳定的2.2.0版，但相关的文档工作还差很多。就目前所知，NGPT是基于GNU Pth（GNU Portable Threads）项目而实现的M:N模型，而GNU Pth是一个经典的用户级线程库实现。

按照2003年3月NGPT官方网站上的通知，NGPT考虑到NPTL日益广泛地为人所接受，为避免不同的线程库版本引起的混乱，今后将不再进行进一步开发，而今进行支持性的维护工作。**也就是说，NGPT已经放弃与NPTL竞争下一代Linux POSIX线程库标准。**

3.其他高效线程机制

此处不能不提到Scheduler Activations。这个1991年在ACM上发表的多线程内核结构影响了很多多线程内核的设计，其中包括Mach3.0、NetBSD和商业版本Digital Unix（现在叫Compaq True64 Unix）。它的实质是在使用用户级线程调度的同时，尽可能地减少用户级对核心的系统调用请求，而后者往往是运行开销的重要来源。采用这种结构的线程机制，实际上是结合了用户级线程的灵活高效和核心级线程的实用性，因此，包括Linux、FreeBSD在内的多个开放源码操作系统设计社区都在进行相关研究，力图在本系统中实现Scheduler Activations。

很久很久之前的文章了，综合了多篇文章整理而成：

Linux 线程实现机制分析 Linux 线程模型的比较：LinuxThreads 和 NPTL - xiaohuazi - 博客园

Linux线程实现机制分析 - 豆丁网

Linux 线程实现机制分析　10-08

LinuxThreads所采用的是线程-进程"一对