# Linux 线程实现技术研究

范 小 鸥
(吉林建筑工程学院计算机科学与工程学院,长春 130118)

**摘要:** 本文简要分析了一般线程机制,详细阐述了 Linux 线程思想及在内核中的实现,包括 linux 线程描述数据结构、管理线程机制和策略、线程栈结构、线程 id 和进程 id 的创建及分配,Linux 线程实现方法.

**关键词:** 线程; 内核; Linux
**中图分类号:** TP 391 **文献标志码:** A **文章编号:** 1009 - 0185(2012)03 - 0082 - 03

## Research on Realization of Linux Threads

FAN Xiao - ou
(*School of Computer Science and Engineering, Jilin Institute of Architecture and Civil Engineering, Changchun, China* 130118)

**Abstract:** This paper analyzes the characteristics of threads mechanism, then detailed analysis about threads mechanism in the Linux 2.6 kernel, including Linux necessary of threads technology, some struts, organizations that are used to supports the threads mechanism.

**Keywords:** threads; kernel; Linux

线程技术早在20世纪60年代已提出,但真正将多线程应用到操作系统中还是在20世纪80年代中期[1-2].现在,多线程技术已被许多操作系统所支持,包括 Windows NT/2000 和 Linux.

## 1 线程简析

线程是进程中的一个实体,是被系统独立调度和分派的基本单位,线程自己不拥有系统资源,只拥有一点在运行中必不可少的资源,但其可与同属一个进程的其他线程共享进程所拥有的全部资源.一个线程可创建和撤消另一个线程,同一进程中的多个线程之间可以并发执行.由于线程之间的相互制约,致使线程在运行中呈现出间断性.线程也有就绪、阻塞和运行3种基本状态[3].

在操作系统设计上,从进程演化出线程,目的是更好地支持 SMP 以及减小(进/线程)上下文切换开销.线程与进程相比主要有以下优点[4]:

(1) 线程和进程相比,它是一种非常"节俭"的多任务操作方式. 在 linux 系统下,启动一个新的进程必须分配给它独立的地址空间,建立众多的数据表来维护它的代码段、堆栈段和数据段,这是一种"昂贵"的多任务工作方式.而运行于一个进程中的多个线程,它们彼此之间使用相同的地址空间,共享大部分数据,启动一个线程所花费的空间远远小于启动一个进程所花费的空间,而且,线程间彼此切换所需的时间也远远小于进程间切换所需要的时间;

(2) 线程间方便的通信机制. 对不同进程,它们具有独立的数据空间,要进行数据的传递只能通过通信的方式进行,这种方式不仅费时,而且很不方便.线程则不然,由于同一进程下的线程之间共享数据空间,所以一个线程的数据可以直接为其他线程所用,这不仅快捷,而且方便.当然,数据的共享也带来其他一些问题,有的变量不能同时被两个线程所修改;有的子程序中声明为 static 的数据;更有可能给多线程程序带来

收稿日期: 2012 - 03 - 19.
作者简介: 范小鸥(1964 ~),女,吉林省长春市人,高级实验师.

灾难性的打击,这些正是编写多线程程序时最需要注意的地方;

(3) 提高应用程序响应. 这对图形界面的程序尤其有意义,当一个操作耗时很长时,整个系统都会等待这个操作,此时程序不会响应键盘、鼠标、菜单的操作,而使用多线程技术,将耗时长的操作(time consuming)置于一个新的线程,可以避免这种尴尬的情况;

(4) 使多 CPU 系统更加有效. 操作系统会保证当线程数不大于 CPU 数目时,不同的线程运行于不同的 CPU 上;

(5) 改善程序结构. 一个既长又复杂的进程,可考虑分为多个线程,成为几个独立或半独立的运行部分,有利于程序的可读性和可维护性.

## 2 LINUX 线程的思想及特点

### 2.1 线程描述数据结构

线程机制 Linux Threads 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,Linux Threads 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,Linux Threads 仅使用了其中的 errno,p_pid,p_priority 等 3 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.

### 2.2 管理线程

Linux Threads 所采用的是线程一进程"一对一"模型(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),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. "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 Linux Threads 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)"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为:

初始化结束后,在_pthread_manager_thread 中,记录了轻量级进程号(轻量级线程(LWP) 是一种由内核支持的用户线程. 它是基于内核线程的高级抽象,因此只有先支持内核线程,才能有 LWP),以及核外分配和管理的线程 id,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() 所创建的那个用户线程则是由管理线程来调用 clone() 创建,因此实际上是管理线程的子线程(此处子线程的概念应该当作子进程来理解). _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 2s 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).

### 2.3 线程栈

在 Linux Threads 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.

2.4 线程 id 和进程 id

每个 Linux Threads 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.

## 3 结语

虽然 linux 中线程实现机制日趋成熟且广泛应用,但仍存在不足. 比如说,由于计算线程本地数据的方法是基于堆栈地址的位置,因此,对于这些数据的访问速度都很慢. 由于 Linux Threads 是围绕一个管理线程来设计的,因此会导致较多上下文切换的开销,这可能妨碍系统的可伸缩性能. 由于线程的管理方式及每个线程都使用了一个不同的进程 ID,因此 Linux Threads 和其他与 Posix 相关的线程库并不兼容. 总之,随着 linux 内核的发展,其中所使用的线程机制也将不断改进与完善.

## 参 考 文 献

[1] 李建军,陈鸿星,张红新,张 威. Linux 线程库的实现机制[J]. 计算机与现代化,2005(4):98 -100.

[2] 郑燕飞,余海燕. Linux 的多线程机制探讨与实践[J]. 计算机应用,2001(1):83 -85.

[3] 金惠芳,陶利民,张基温. Linux 下多线程技术分析及应用[J]. 计算机系统应用,2003(9):30 -31.

[4] 周 丽,焦程波,兰巨龙. LINUX 系统下多线程与多进程性能分析[J]. 微计算机信息,2005(17):123 -124.