文章编号: 1002-0640(2010) 02-0112-03

Linux线程机制研究

雷铭哲,张 勇
(北方自动控制技术研究所,太原 030006)

摘 要: 阐述了线程技术在 linux操作系统中的发展过程,指出了早期 linux线程机制的不足及在发展过程中的改进策略。简单分析了一般线程机制,详细分析了 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的创建和分配, Linux线程实现方法。

关键词: 线程,内核, linux
中图分类号: TP316 文献标识码: A

Research on Linux Threads Mechanism

LEI Ming-zhe, ZHANG Yong
(*North Automatic Control Technique Research Institute*, *Taiyuan* 030006, *China*)

Abstract This paper analyzes the characteristics of threads mi chanism, points out the deficiencies of original threads mechanism and its reason, then I detailed analysis about threads mechanism in the linux-2.6 kernel, including linux necessary of threads technology, some structs, orgniztaions that are used to supports the threads mechanism.

Key words threads, kernel, linux

引 言

线程技术早在 20世纪 60年代就被提出,但真正应用多线程到操作系统中还是在 20世纪 80年代中期。现在,多线程技术已经被许多操作系统所支持,包括 Windows NT/2000和 Linux。

在 Linux 2.2内核中,进程是通过系统调用 fork创建的,新的进程是原来进程的子进程。需要说明的是,在 Linux 2.2.x中,不存在真正意义上的线程, Linux中常用的线程 Pthread实际上是通过进程来模拟的。也就是说, Linux中的线程也是通过 fork创建的,是"轻"进程。 Linux 2.2缺省只允许 4 096个进程。线程同时运行,而高端系统同时要服务上千的用户,所以这显然是一个问题。它一度是阻碍 Linux进入企业级市场的一大因素。

Linux 2.4内核消除了这个限制,并且允许在系统运行中动态调整进程数上限。因此,进程数现在只受制于物理内存的多少。在高端服务器上,即使只安装了 512 MB内存,现在也能轻而易举地同时支持 1.6万个进程。

在 Linux 2.6中改进的线程模是基于一个 1∶1的线程模型(一个内核线程对应一个用户线程),包括内核内在的对新 NPTL(Native Posix Threading Library)的支持。

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,但它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。一个线程可以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、阻塞和运行三种基本状态。

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地支持 SMP以及减小(进程/线程)上下文切换开销。具体地说主要有以下优点:

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

收稿日期: 2008-09-17 修回日期: 2009-01-18
作者简介: 雷铭哲 (1977-),男,湖北咸宁人,硕士研究生,研究方向: 军用嵌入式系统软件。

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。对不同进程来说,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以考虑分为多个线程,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- ha ndles来描述和引用进程所辖线程。在 - pthread- handles中的前两项,Linu x Threads定义了两个全局的系统线程: - pthread- initi al- thread和 - pthread- manager- thread,并用 - pthread - main- thre ad 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,- pthre ad- 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。经过一系列 pthread- create()之后形成的 - pthread- han dles数组将如图 1所示。

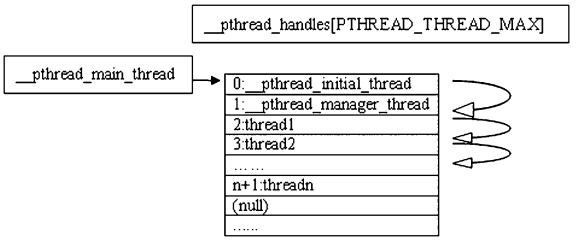

图 1 - ptluead- handles数组结构

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。

2.2 管理线程

Linux Threads所采用的是线程-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[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 - pthr ead- 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 - re ader,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)

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(LW P)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 LW P)以及核外分配和管理的线程 id, id= 2* PTHREAD- THREADS- M AX+ 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-

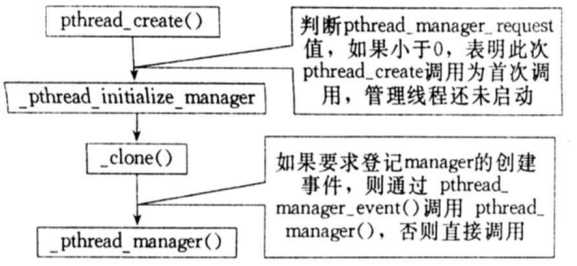

图 2 创建管理线程的流程

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 2 s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- child ren()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。

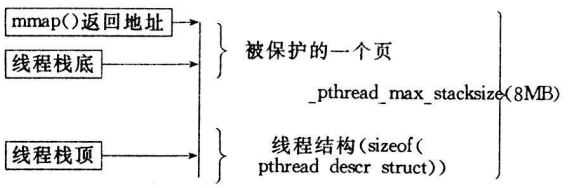

图 3 栈结构示意图

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 Linux Threads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 Linux线程实现方法

目前线程有用户态线程和核心态线程两种方法实现。

3.1 用户态线程

用户态线程,允许多线程的程序运行时不需要特定的内核支持。如果一个进程中的某一个线程调用了一个阻塞的系统调用,则该进程就会被阻塞,该进程中的其他所有线程也同时被阻塞。因此,Unix使用了异步 I/O机制。这种机制主要的缺点在于,在一个进程中的多个线程调度中无法发挥多处理器的优势(如上述的阻塞情况)。

用户态线程优点如下:

* 某些线程操作的系统消耗大大减少。比如,对属于同一个进程的线程之间进行调度切换时,不需要调用系统调用,因此将减少额外的消耗,一个进程往往可以启动上千个线程。

* 用户态线程的实现方式可以被定制或修改,以适应特殊应用的要求。它对于多媒体实时过程等尤其有用。另外,用户态线程可以比核心态线程实现方法默认情况支持更多的线程。

3.2 核心态线程

核心态线程的实现方法允许不同进程中的线程按照同一相对优先调度方法进行调度,这样有利于发挥多处理器的并发优势。

目前,线程主要的实现方法是用户态线程。有几个研究项目已经实现了一些核心态线程的形式,其中比较著名的是 MACH分布式操作系统。

4 结束语

虽然 linux中线程实现机制日趋成熟,并且被广泛应用,但仍然存在一些不足,比如说,由于计算

(下转第 118页)

过程耗时均在 20 s以上[5]。图 6中和右分别为用 sobel算子和 canny算子对图 2进行的边缘检测结果。Sobel算子对于灰度变化较小的部分没有检测出来 (如摄影者远处的建筑物等),canny算子检测出的图像细节又太多,并且有些细节检测失真。由图 6左可以看出,本算法能将灰度变化较小的部分有效地检测出来,而且检测结果比较准确,效率较高,但在图中可以看出在边缘的某些部分出现“点聚集”现象,这样由于传统蚁群算法中部分蚂蚁在搜索中会陷入局部最优解,要想消除这种现象就必须对传统蚁群算法进行改进,这也是笔者下一步研究的重点。各种图像边缘检测算法的时间性能比较如表 1所示。

3 结束语

蚁群算法是一种较新的应用于组合优化问题的启发式搜索算法,本文将蚁群算法应用于图像边缘检测从而提出了一种新的边缘检测方法。大量的实验结果证明该方法能有效地检测出图像的边缘,通过调节算法中蚂蚁数目、蚂蚁行走步数和记忆路径长度等参数的值可对不同尺寸、不同复杂程度的图像进行有效的处理,通过对蚁群算法的改进可以消除“边缘点聚集”现象。因为在本算法中蚂蚁的初始位置设在图像的边缘附近,大大降低了计算量,故本算法执行效率较高,与基于模糊聚类的边缘检测相比,耗时较少。因此,基于蚁群算法的边缘检测方法是一种有效的图像边缘检测方法。然而,蚁群算法是一种新的模拟进化算法,还没有像 GA等算法那样形成系统分析的方法和坚实的数学基础,各种实验参数的确定也没有理论上的指导,目前国际上诸多研究成果还都基于实验分析,还有许多理论问题有待进一步研究。但可以推断的是,随着研究的深入,蚁群算法也将与其他进化算法一样,能够获得越来越多的应用。

表 1 各种图像边缘检测算法时间性能比较

图像边缘检测算法	传统蚁群算法 (蚂蚁初始位置随机分布)	本算法 (蚂蚁初始位置置于边缘附近)	基于模糊聚类蚁群算法[5]	Sobel算子	Canny算子
时间性能	6.8 s	4.31 s	21.4 s	0.16 s	0.23 s

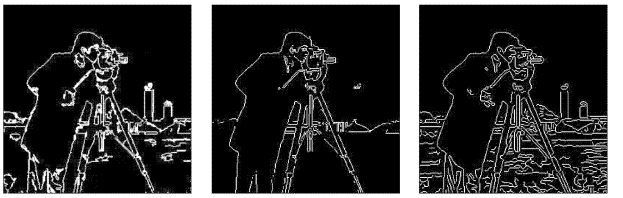

图 6 左为本算法检测结果,中为 sobel边缘检测结果,右为 canny边缘检测结果

参考文献:

[1] 章毓晋. 图像处理和分析 [M]. 北京: 清华大学出版社, 1999.

[2] Dorigo M, Gianni D C, Thomas S. Ant Algorithms [J]. Future Generation Computer Systems, 2000, 16(3): 5-7.

[3] Dorigo M, Maniezzo V, Colorni A. The Ant System: Optimization by a Colony of Cooperating Agents [J]. IEEE Transactions on Systems, Man, and Cybernetics-Part B, 1996, 26(1): 1-13.

[4] Dorigo M, Gambardella L M. Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem [J]. IEEE Transactions on Evolutionary Computation, 1997, 1(1): 53-66.

[5] 韩彦芳,施鹏飞. 基于蚁群算法的图像分割方法 [J]. 计算机工程与应用, 2004, 18(6): 5-7.

[6] 郭子龙,王孙安. 免疫进化模糊聚类算法在边缘检测中的应用 [J]. 西安交通大学学报, 2004, 38(3): 274-277.

[7] 殷润民,平 洁. 一种基于灰度差统计的边缘检测方法 [J]. 计算机工程, 2006, 32(8): 201-203.

(上接第 114页)

线程本地数据的方法是基于堆栈地址的位置的,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,因此会导致很多的上下文切换的开销,这可能会妨碍系统的可伸缩性和性能。由于线程的管理方式,以及每个线程都使用了一个不同的进程 ID,因此 Linux Threads与其他与 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] 李善平. Linux内核 2.4源代码分析大全 [M].北京: 机械工业出版社, 2002.

[2] 范永开,杨爱林. linux 应用开发技术详解 [M].北京: 人民邮电出版社, 2006.

[3] 张 威. linux信号机制解析 [J].电脑知识与应用, 2006, 28(10): 41-42.

[4] 邹治锋,张曦煌. linux2.6进程调度 [J].嵌入式系统应用, 2006, 24(8): 30-31.