第19卷 第2期 小型微型计算机系统 Vol. 19 No. 2
1998年2月 MINI- MICRO SYSTEMS Feb. 1998

多线程技术

李新明 李 艺
(国防科工委指挥技术学院四系 北京 101407)

摘 要 多线程技术已经在许多商用操作系统中实现,并即将在国产操作系统 COSIX V2.X系列中实现。由于多个线程共享同一个进程的地址空间,使得线程创建,线程之间的切换及通信的开销大大降低,因而很适合于多 CPU、多任务等并行环境下的处理。本文结合 SunSoft Solaris 2.X系列,从用户级多线程库和操作系统核心两个层次上,对多线程机制的实现进行了剖析。

关键词 进程 线程 用户线程 核心线程 LWP
分类号 TP316

1 引言

传统的操作系统中,资源分配和 CPU 调度的单位是进程,即一个程序的一次执行。进程在任何时候只有一个执行现场,即称为单线程结构。这种单线程结构的进程已不能很好地适应计算机的发展。首先,计算机硬件向多处理机、网络方向发展,这就要求操作系统能适应这种发展,合理地使用各处理机和网络上的其它处理机,许多工作可以分配到不同的处理机上同时运行。这一点上,传统的单线程进程不能有效地实现;其次,应用程序要求并发执行。如数据库中,可以同时有多个用户在交互执行,同时对几个文件或网络操作。又如窗口系统中,同时处理多个子窗口的请求等,这就要求操作系统提供一些机制,使得用户能按需求,在一个程序中设计出多个线程同时运行,而这又不是多个进程组合在一起能完成的,因为他们之间共享大部分资源。

基于上述原因,操作系统在系统结构上有了新的发展,与传统操作系统中的单线程结构相对应,提出了多线程结构的概念。很多著名的操作系统都已经采用了多线程结构,如 Solaris 2.X, Mach 2.6, OSF/1, Windows NT等。目前,对多线程结构还没有国际共认的标准,许多计算机公司和国际组织都有自己的标准,如 Solaris Thread接口规范, OS/2 Thread接口规范, Window NT Thread接口规范, POSIX Pthread标准和 DEC Pthread标准等。所有这些规范都对操作系统的多线程结构给出了一系列的外部接口,虽然这些标准在具体的接口格式或功能上有所区别,但主要的内容是一致的。本文结合 Solaris2.X,说明了多线程的概念,并对有关多线程的调度、同步、信号等问题作了分析。在有关多线程技术的研究中,多线程调度和同步

1996-10-29收稿 李新明,1965年生,副教授,主要从事操作系统,实时系统,分布式领域的研究。李艺,副教授,1964年生,主要从事操作系统实时系统的研究。

及资源共享与保护等仍将会是令人感兴趣的课题.

2 多线程概述

在 UNIX进程里,能够被分割成独立处理的单元叫线程(Thread),它是进程内的一连串指令的执行,一个进程里可以有多个线程,它们之间共享全部的进程地址空间和进程资源.

用户可以按需要编写拥有多个线程的程序,从理论上说,用户进程可以创建任意多个用户线程. 这些线程可以在不同的处理机上同时执行. 但从资源分配、处理机调度、线程切换、效率等诸多问题出发,并不是每创建一个线程,就作为一个独立的单位,交给核心管理. 在多线程结构的操作系统中,多线程结构大都分成两个层次来实现,一层是用户层,在用户级多线程库中实现,一层是核心层,在操作系统内核中实现. 为了区别,处于不同层次中的线程分别叫做用户线程和核心线程. 用户线程只是一个代表对应线程的数据结构,它只占用用户空间的资源,是用户级的对象,对核心是透明的. 而核心线程才是线程执行的实体,是核心调度的单位. 用户线程和核心线程并不是一一对应的,它们之间要通过一个新的对象来联系,在 Solaris操作系统中,这个新的对象叫 LWP(LightWeight Process),即轻进程. LWP如同用户线程的虚拟处理机,与用户线程一样共享进程中的所有资源. LWP总是按一定的策略,从用户线程中选择一个执行,而 LWP对核心是可见的,是核心级的对象,它与一个核心线程一一对应,核心按一定的策略,选择一个核心线程运行. 因此,一个线程要经过核心和 LWP的两级调度后才能真正占有 CPU,开始运行.

LWP的引入使得用户级的多线程能浮在核心之上实现. 每个进程可以创建几千个线程,而不需要占用核心资源,只占用用户空间的资源. 线程由于共享地址空间,当 LWP在一个进程的不同用户线程之间进行切换时,时间开销远远低于两个核心线程之间的切换时间,因为线程的用户空间是共享的,仅仅是一种数据结构的转换,并且这种切换完全在用户级的线程库中完成,对核心是透明的. 线程间的同步也可以独立于核心的同步对象,在用户空间中独立实现,而不用陷入内核,在核心对象上实现同步. 用户线程对核心来说是不可见的,核心能看见的只是 LWP.

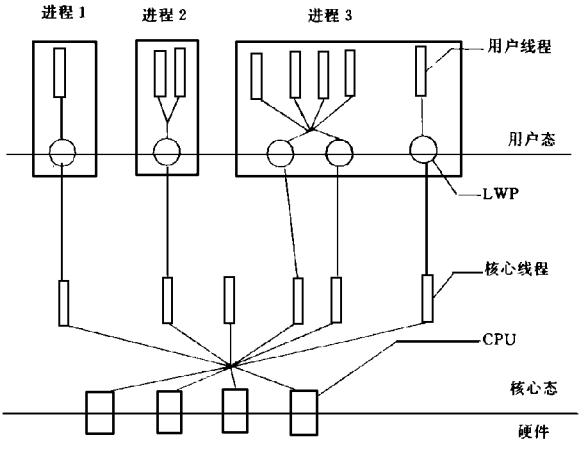

图 1 多线程结构系统示例

因此,多线程结构有两层模式,第一层为线程接口,提供给用户,使用户能编写多线程的应用程序. 这一层由动态链接库利用轻进程 LWP来实现,线程库在进程的 LWP池上调度线程,线程之间的同步也在线程库中实现. 第二层为 LWP接口,提供给线程库,用于管理 LWP,这一层由核心通过核心线程实现,LWP通过这些接口访问核心资源. 典型的多线程结构的系统如图 1所示:

如上所示,进程 1为传统的单线程结构的进程,进程 2和进程 3都是多线程结构的进程. 每个

用户进程有一个对应的 LWP池. 用户级的线程在用户级调度,被线程库指派到本进程的任意 LWP中. 用户线程之间的切换在用户级线程库中进行,不用进入内核. 用户线程也能与某一个 LWP捆绑在一起,占有对应的 LWP,在此线程退出之前,对应的 LWP不调度其它的用户线程运行. 每个 LWP与一个核心级的线程相对应. 除此之外,还有一些系统核心线程,完成系统功能,不与任何 LWP相关联,核心按照 LWP的优先级,调度核心级线程到某一个 CPU上运行,也可以把某个核心线程与某个 CPU 捆绑在一起,使它不运行别的核心线程. 用户级线程之间的同步在线程库中实现,不用进入内核, 核心线程之间的同步在内核中实现.

3 用户级多线程结构

3.1 数据结构

用户级多线程结构通过线程库来实现. 线程库利用 LWP实现对用户多线程的管理. 用户线程在线程库只是一个简单的数据结构和一个堆栈,核心不知道它. 线程库维护的主要的数据结构包括用户线程结构、用户态堆栈、各种队列和 LWP池.

① 用户线程结构. 线程库是通过用户线程结构来标识和管理用户线程的. 主要有以下内容:

· 用户线程号 (只在本进程内有效,对其它进程是不可见的)
· 信号屏蔽码
· 线程调度优先级
· PC和 ESP等寄存器的值 (线程的上下文关联)
· 线程状态
· 线程队列指针
· 线程堆栈指针 (可以由应用程序传递,或由线程库在匿名区中分配.)
· 线程局部存储区指针 (由线程独有,不被所有线程共享的一段区域.)

其中的线程状态可以是下列五种之一:

· ACTIVE 运行态,有一个 LWP正在运行它;
· RUNNABLE 就绪态,等待一个 LWP来调度它;
· SLEEPING 睡眠态,睡眠在进程内部的同步变量上,等待被唤醒;
· STOPPED 暂停态,等待被继续;
· ZOMBLE 僵死态,线程退出前的一个临时状态.

② 各种队列. 根据用户线程状态的不同,线程库主要维护以下一些队列:

· 按线程优先级组织的一组调度队列. 当线程处于就绪态,且没有与某个 LWP相捆绑时,放在对应的就绪队列中;

· 按不同同步变量组织的一组睡眠队列,当线程被某一个同步变量所阻塞时,被放在对应的同步变量的睡眠队列中;

· 一个运行队列,所有处于 ACTIVE状态的线程位于此队列中;

· 一个死亡队列,当被分离的线程退出时,放在此队列中,由线程库中的特殊线程 reaper 定期清理此队列中的线程;

· 一个僵死队列,当未被分离的线程退出时,放在此队列中,由执行联接的线程对它进行释放堆栈等退出前的处理.

③ LWP池 LWP池中是本进程所拥有的全部 LWP. 每个 LWP有两种存在形式,一种是正调度某一个线程运行,此时它与一个核心线程连在一起,由核心线程调度它执行;另一种是处于空闲等待状态,此时没有就绪的用户线程可以运行,在 LWP池中睡眠,等在 idle同步变量上. 当睡眠时间超过某个时间后,表示此 LWP是多余的,终止此 LWP并从 LWP池中删除. 线程库将根据本进程线程的个数等自动调节 LWP池的大小,保证不会因为 LWP太少,就绪的线程没有 LWP去运行它,而使进程死锁,也不会分配过多的 LWP而造成核心资源的浪费.

3.2 用户线程调度

用户线程调度在 LWP池中实现. 每个线程都有一个从 0到无穷大的一个调度优先级,LWP总是从最高优先级的就绪队列中调度一个用户线程运行. 当某一个线程就绪时,它插入到对应的就绪队列中,并唤醒 LWP池中的一个空闲 LWP,去选择一个最高优先级的线程运行. 如果 LWP池空,则在就绪队列中等待.

线程调度允许抢占,如果就绪线程的优先级比活动队列中的线程的任一个的优先级高,则要实施抢占. 从活动队列中找到一个优先级最低的线程,向所对应的 LWP发信号,LWP接收到信号后,重新调度优先级最高的线程,则原来的线程被抢占.

当正运行的线程被某一个同步变量阻塞,或退出,或被暂停时,要释放对应的 LWP,则该 LWP从就绪队列中调度新的线程去运行,如果没有可运行的线程,则等待在 idle变量上,放入 LWP池中睡眠.

3.3 用户线程同步

用户线程同步机制有以下几种:

· 互斥锁: 一次只有一个线程能获得锁,常用于临界区的互斥执行;

· 条件变量: 用于使一个线程等待某一个条件为真,必须与互斥锁一起使用;

· 信号灯: 信号灯是一个非负的整数计数器,当一个线程进入时其值减少,当一个线程退出时其值增加;

· 多读者,单写者锁: 允许多个线程同时读某个共享对象,但当一个线程要写该对象时,要阻塞读线程.

同步机制可用于同一进程的各个线程之间,此时同步变量在进程的常规内存空间中分配,对核心是透明的,每个同步类型都支持几种实现方法,程序员在变量初始化时选择实现方法.

同步机制也可用于不同进程的各个线程之间,在变量初始化时标志为共享,在共享的映射文件中分配. 当线程被阻塞时,核心对其处理,挂在核心的睡眠队列中,此时线程同执行系统调用一样,与 LWP捆绑在一起,LWP不能去执行其他的线程.

3.4 信号处理

进程中所有的线程共享信号的处理动作,即一个信号不论被哪个线程处理,其动作都是相同的. 但每个线程有自己的信号屏蔽码,以便在进入临界区时,进行信号屏蔽. 线程自己产生的信号,如异常所产生的信号,由产生者处理,而其他信号可由进程内任何一个线程处理.

线程号只在进程内部有效,其它进程不能指定向某一个线程发信号,而只能向一个进程发信号,但同一个进程内部可以向指定的线程发信号,由指定的线程来处理.

用户线程抢占调度和 LWP池大小的自动调节是通过两个新信号 SIGLWP和 SIGWAITING来实现的. 当就绪线程的优先级高于某一个正在运行线程的优先级时,向线程库发

SIGLWP信号,线程库在收到此信号后,将使指定的 LWP实施线程切换,进行抢占。当核心发现一个进程的所有 LWP均处于等待状态时,向线程库发送 SIGWAITING信号,线程库在接收到该信号后,认为是由于缺少 LWP而使进程处于死锁状态,因此,线程库将创建新的 LWP,保证进程有空闲的 LWP可以运行。

3.5 线程接口

线程库提供了一系列接口函数,用户可以用这些函数设计自己的多线程结构的应用程序。各个多线程结构的操作系统遵守各自的标准,提供的接口函数不完全相同,Solaris提供的多线程库接口主要包括以下内容:

① 与进程控制相关的

· thread_create() 创建一个线程,可以通过参数同时创建 LWP或与 LWP相捆绑;

· thread_exit() 终止一个线程,传统的 exit()将终止进程中的所有线程;

· thread_wait() 等待线程终止,可以通过参数设置来标识等待指定的线程还是进程中的任一个线程;

· thread_stop() 暂停当前线程的运行;

· thread_continue() 继续一个线程的执行;

· thread_get_id() 获得线程的标识号;

② 与同步控制相关的

a.互斥锁相关的

· mutex_init() 对互斥锁进行初始化;

· mutex_enter() 请求互斥锁,如果不成功,则等待;

· mutex_exit() 释放互斥锁,并唤醒等待的线程;

· mutex_tryenter() 请求互斥锁,如果不成功,则出错返回;

b.条件变量相关的

· cv_init() 对条件变量进行初始化;

· cv_wait() 请求条件变量,如果不成功,则等待;

· cv_signal() 释放条件变量,并唤醒一个调用 cv_wait()时被阻塞的线程;

· cv_broadcast() 释放条件变量,并唤醒所有调用 cv_wait()时被阻寒的线程;

c.信号量相关的

· sema_init() 对信号量进行初始化;

· sema_v()增加信号量的值,并唤醒等待信号量值增加的线程;

· sema_p() 减少信号量的值,如果不够减,则等待信号量值增加;

· sema_tryp() 减少信号量的值,如果不够减,则出错返回;

d.读写锁(即多读者,单写者锁)相关的。这部分是 Solaris所独有的,其它标准都没有提供这个功能。这对于有多个线程要读某个临界区中的内容,而修改其内容的线程很少时很有用。使得可以同时有多个线程读该区的内容。

· rw_init() 对读写锁进行初始化;

· rw_enter() 请求读写锁,由参数决定是读还是写,如果不成功,则等待;

· rw_tryenter() 请求读写锁,如果不成功,则返回;

· rw_ exit() 释放读写锁,并唤醒等待锁的线程;

· rw_ downgrade() 将写锁自动转换为读锁;

· rw_ tryupgrade() 将读锁自动转换为写锁,如果不成功,则出错返回。

③ 与信号相关的

· thread_ sigsetmask()设置线程的信号屏蔽码;

· thread_ kill() 向指定线程发信号;

④ 与 LWP池相关的。这部分是 Solaris所独有的,其它标准中没有提供。它可以使得用户参与管理 LWP池的大小,使达到最高的效率。

· thread_ setconcurrency()设置进程的实际并发度,将影响 LWP池的大小;

· thread_ getconcurrency()获得进程的实际并发度;

⑤ 其它

· thread_ priority() 设置指定线程的优先级;使得同一进程中的各个线程有各自的优先级,按紧急程度进行调度,而有些操作系统如 MACH中没有对应的功能,同一进程中的所有线程按顺序调度,除了提供上述新的接口函数外,原有的一些接口的语义有所扩展,这些接口函数如 waitid(), fork(), sigsuspend等。

4 多线程结构的操作系统核心

4.1 数据结构

传统内核下,进程的数据分为两部分,即 proc结构和 user结构。在支持多线程结构的内核中,内核所管理的数据结构主要有四部分:

① proc结构

proc结构中主要包括与本进程相关的一些信息,如:

· 指向核心线程的 kernel_ thread结构的指针;

· 指向本进程地址空间的指针;

· 用户安全信息;

· 信号动作数组;

· 原 user结构中遗留的信息,比传统的 user结构小得多。

② LWP结构

LWP结构包括与本 LWP相关的一些信息,如:

· 存放用户级处理器寄存器的 PCB;

· 系统调用的参数;

· 资源使用信息;

· 统计信息;

· 指向 kernel_ thread结构和 user结构的指针。

③ kernel_ thread结构

kernel_ thread结构主要包括与本核心线程相关的信息,如:

· 核心线程的处理器寄存器的值;

· 调度类;

· 在核心管理的各个队列中的链指针;
· 核心栈指针;
· 到 proc结构、LWP结构和 CPU结构的指针;

④ CPU结构 CPU结构包括与 CPU相关的一些信息,如:
· 到 kernel thread结构的指针;
· 对应的空闲线程的指针;
· 当前调度和中断处理的信息;
· 与体系结构相关的一些信息。

上述结构通过指针互相链接在一起。此外,核心管理着如下一些队列:
· 每个进程所对应的所有核心线程构成的队列;
· 全系统所有核心线程构成的队列;
· 所有就绪核心线程按调度优先级构成一组就绪队列;
· 与同步变量相对应的一组睡眠队列;

4.2 内核线程的调度

内核的调度单位是内核线程。内核线程分成三种类型,一种是在进程中执行的 LWP;第二种是执行内核功能的特殊线程,完成调页、换进换出或为流服务的后台服务等功能;最后一种是空闲线程,当 CPU没有就绪线程可运行时,执行空闲线程,每个 CPU有一个对应的空闲线程,由 CPU结构的指针指向。

系统一般支持分时类、系统类和实时类三种调度类型,每一调度类型有各自的调度策略。系统采用全局优先级模型,优先级从 0到 159。

调度时,内核线程处于三种状态:阻塞态、就绪态和运行态。被阻塞的线程都挂在某一个同步对象的睡眠队列上,当该对象被释放时,睡眠队列中最高优先级的线程将被唤醒,进入就绪态。如果本线程的优先级高于相关联的 CPU上正在运行的线程的优先级,则要进行"抢占"。

抢占分成用户级抢占和核心级抢占两种。用户级抢占采用"lazy"方式,直到线程从核心态返回到用户态时才实施。当有用户级抢占时,在 CPU结构中设置一个标志位,当线程从陷入或系统调用返回用户态时,检查此标志,调用调度函数进行切换。核心级抢占立即进行,当有核心级抢占时,在 CPU结构中设置一个标志,同一个 CPU的抢占可立即调用调度函数来进行,不同 CPU的抢占要通过 CPU之间发送中断来实现。

4.3 同步机制

核心提供的核心内部使用的同步对象与在用户级库中为多线程结构的应用程序提供的同步对象是很类似的。提供两套类似的同步机制,可以使得用户线程在用户级库中进行同步互斥控制,而不用占用核心的资源,由核心来管理,以提高系统效率。

4.4 中断处理

传统的操作系统中,中断是在当前进程的关联下处理的。中断处理时,提高 CPU的中断优先级,以禁止同级和低级的中断,同时中断处理时,不允许抢占。这种方法对提供完全可抢占的内核是不合适的。Solaris操作系统用内核线程来处理中断。系统中存在一个线程级,一般与时钟中断级相同,当一个中断的中断级低于线程级时,用中断线程来处理中断。

系统为每个 CPU中的中断级低于线程级的潜在中断预先创建一个中断线程,这些线程已部分初始化,当中断发生时,对中断线程做少量处理后,将 CPU切换到中断线程上,这种切

换比通常的线程切换的开销要小得多,此时被中断线程与中断线程处于一种钉住(pinned)状态,两者没有完全分开。被中断线程直到中断线程返回或被阻塞,一直处于钉住状态,不能恢复执行。当中断线程返回时,恢复被它钉住的线程的执行。当中断线程被某一个同步对象阻塞时,要对中断线程进行处理,使它能与其他线程一样,在任何 CPU上运行,然后返回到被钉线程,此时,中断线程和被中断线程完全脱离关系。因此,为中断创建一个完整线程的开销只要当中断线程被阻塞时才真正需要。

时钟中断线程是全系统共用一个,而不是每个 CPU一个,从而保证时间的一致性。

5 小结

多线程结构具有许多优点,在多线程结构的应用程序中,当一个线程等待 I/O完成或大量计算的最终结果时,另一个线程可以继续其他处理,使得进程总处于运行态,随时进行响应,从而提高系统的响应效率;对多 CPU机器,每一个线程可以在不同的 CPU上运行,多线程结构是应用程序使用硬件并行性的最佳解决方案;由于多线程共享同一个进程空间,这比利用多个独立的进程,通过共享内存来访问共同的数据结构所占用的系统资源要小得多,而且线程之间的通信也比进程之间的通信要简单得多;设计程序时,每个过程用一个线程来实现,使得程序的结构更清晰。多线程结构的操作系统核心的实现,使得操作系统的核心是完全可抢占的,而这正是实时系统所要求的。多线程结构是为了适应计算机硬件向多 CPU和网络方向发展以及应用程序的并发性要求而产生的,目前主要的商业操作系统均采用多线程的思想,如 Solaris 2.X, Mach 2.6, SCO UNIX, Windows NT等,相应的多线程结构的国际标准也正在制定。在未来的操作系统中,基于微内核结构,采用多线程技术的操作系统将是主流。我们国产的操作系统 COSIX V2.0在国家"九五"计划期间一个主要任务就是向多线程结构发展。

参 考 文 献

〔1〕 D. Stein, D. Shar. Implementing lightweight threads. USA. Summer 92 USENIX. 1992, 6
〔2〕 Sandeep Khanna, Michael Sebree. Realtime scheduling in SunOS 5.0. USA Winter 92 USENIX. 1992, 12
〔3〕 J. R. Eykholt, S. R. Kleman. Beyond multiprocessing... Multithreading the SunOS Kernel. USA. Sun Microsystems, Inc. 1992
〔4〕 M. L. Powell, S. R. Kleiman. SunOS Multi-thread Architecture. USA Sun Microsystems, Inc. 1992
〔5〕 POSIX1003.1c *IEEE Draft standard*. New York. 1994, 10

MULTI_THREAD MECHANISM

LI Xinming LI Yi
(*Institute of Command and Technology, Beijing* 101407)

Abstract Multi_thread mechanism has been implemented in many commercial operating system, and it will be implemented in COSIX V2.X which is under development and is financed by Chinese Government. Because all threads in one process shared the same addres space, the overhead of creation of threads, the context switch and communication between threads is much low, This made the multi_thread mechanism very suitable for applications running on multi_CPU and multi_task environment. Based on Solaris 2.X, which is the product of SunSoft coorperation, this paper disscussed the implementation of multi_thread meachanism supported by user_level multi_thread library and operating system kernel.

Key words Process Thread User thread Kernel thread LWP
Class number TP316