Linux2.5线程机制研究

丁 成[1] 孙玉芳[2]
(中国科学技术大学 合肥230026)[1] (中科院软件研究所 北京100080)[2]

Research of Thread Support in Linux2.5

DING Cheng[1] SUN Yu-Fang[2]
(University Science and Technology of China,Hefei 230026)[1] (Institute of Software,Chinese Academy of Sciences,Beijing 100080)[2]

Abstract In this paper, we discuss the shortage of thread support in linux2.4 and analyze the recently progress in this area,introduce two pthreads libraries:NGPT,NPTL.

Keywords Linux, Thread, Posix threads, NGPT, NPTL

1 线程和PThreads标准

有效的并行计算依赖于有效的创建和管理并行性。并行性在多任务操作系统如Unix中以进程的形式来体现,但是创建一个进程的开销很大,所以轻量级进程即线程的概念被提出来。20世纪90年代初期,线程被广泛讨论并在UNIX系统中流行起来。对线程公认的描述为线程是进程的一个独立的执行序列,一个进程的线程间共享全局数据(比如全局变量,文件描述符等)而线程拥有独立的堆栈,局部变量和程序计数器。

线程主要的优点在于:在多处理器环境中,线程的使用可以有效地利用处理器的能力;在单处理器的环境中,线程模型简化了对异步事件的编程方式;在图形界面环境中,多线程的使用可以改善对用户事件的响应速度。

现在主流的线程模型有2个,1个是WINNT Threads,1个是POSIX Threads,简称PThreads即IEEE 1003.1c标准。1003.1c标准提出之前,各主要UNIX厂商和UNIX变种设计了多种不同的线程库API,PThreads的主要目的是提供一个可移植的、在各个平台都可用的线程API。PThreads标准自从提出后被各大Unix/类Unix系统广泛支持,它详细描述了线程相关的函数调用接口和它们的语义,包括如下几个方面:线程管理:包括创建、合并(join)、终止和取消(cancel);线程同步;线程局部变量TLS;线程调度;信号处理。

虽然Linux作为一个基本的设计思想,做到了尽可能地符合Posix标准,但核心Linux开发人员对PThreads的设计有不同的看法,不愿在核心中对PThreads的实现做更多的支持,所以它在对PThreads的支持上一直不够完善。

2 Linux对线程的支持的不足之处

在2.2/2.4系列中作为Linux标准的线程库是LinuxThreads,目前它属于glibc的一部分被维护。但是它存在两个主要的问题,第一是对Posix标准实现不完全。例如

·signal处理不符合Posix标准的语义;

·需要创建一个额外的管理员线程;

·ps会列出所有线程,而不是列出进程;

·core dump不会列出所有线程的上下文;

·getpid()对同一进程的不同线程返回不同的pid号;

·线程不能共享uid和gid。

LinuxThreads在语义上有很多地方和Posix标准的的语义有所不同,所以造成了其他平台的PThreads应用程序不能方便地移植到Linux上。

第二个主要问题是当创建大量线程时会出现性能问题。Linux限制了任务数最大为8192个,而且Linux在任务创建、调度等代码中存在着线性查找的算法,使得在任务数超过一定数目时,性能下降到不可接受的程度。这两个问题既影响了Linux的兼容性,也影响了它的可伸缩性,所以这是Linux作为一个企业级的操作系统平台必须要解决的问题。

3 Linux线程的基础:clone()系统调用

Linux内核中进程和线程的区别并不是那么严格,它使用更多的术语是任务(task)。子任务可以是一个和父任务完全独立的任务,也就是进程,也可能是一个和父任务共享资源的子任务,从语义上来说就是一个核心线程。下面的讨论将沿用Linux的术语。

clone()系统调用是LinuxThreads库实现的基础,也是Linux系统中其他大多数线程库的基础(除了M:1模型的线程库),包括现在正在发展的两个新的线程库NGPT, NPTL都使用它来创建线程。(M:1模型见5.1节)clone系统调用的定义如下:

```
int clone( int( * fn)( void * arg), void * child_stack,
int flags, void * arg)
```

clone系统调用是Linux特有的,它和fork()一样用来创建一个新的任务,不同的地方在于,它允许子任务和父任务共享某些运行资源,例如文件描述符表,内存空间等。它的参数定义如下:子任务被创建后,执行函数fn(arg),由于子任务和父任务可能共享内存,因此要为子任务提供新的堆栈空间,这个空间由child_stack指定。

参数flags低位指定当子任务死亡后,发给父任务的信号。flags高位指定子任务和父任务共享哪些部分。在Linux2.4.0中flags包括:

CLONE_VM:共享内存空间;

CLONE_FS:共享文件系统信息,如根文件系统、当前工

丁 成 硕士生,主要研究方向为Linux操作系统。孙玉芳 博士生导师。

作目录等；

CLONE_FILES：共享文件描述符表；

CLONE_SIGHAND：共享信号处理函数；

CLONE_PID：共享 pid。

Linux 下的线程库正是使用 clone（CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND）来创建一个新线程的，当然为了实现 PThreads 接口，线程库还需要大量的代码。

4 核心的改进

目前(2002-11)Linux 的最新稳定版本是2.4.20，最新的开发版本是2.5.40。2.5系列对线程的支持做了重大的改进。核心的改进主要在几个方面：1)大线程/进程数的支持和性能优化；2)扩展 clone 系统调用；3)futex 的使用；4)PThreads 兼容的语义，主要改正了对信号的处理；5)TLS 支持。

4.1 大线程/进程数的支持和性能优化

核心现在可以处理几乎任意多的任务，任务 ID(pid)的可用空间可以达到400万之多。代码针对大量线程的情况做了优化，申请和释放一个 pid 的算法现在是 O(1)复杂度的。以前在某些条件下分配一个 pid 是非常费时的工作，因为需要做线形性遍历，现在使用了一个位图数组 pid_map_array 来保存 pid 的分配情况，查找可用的 pid 可以使用汇编代码来有效地实现。为了节省空间，pid_map_array 本身是动态扩展的，最初只使用一个内存页。同时对 pid，tgid，pgid，sid 各设置了1个 hash 表，可以对这四种类型的 id 分别进行查找。这四个 hash 表使用一个2维数组 pid_hash [4][4096]来表示，通过 hash 表，针对 pid 的查找可以很快完成。大量相关的代码在利用了这一机制后，性能得到了很大的提高，(例如 sys_exit，sys_clone)测试显示，在相同的硬件条件下，以前开始并终止10万个线程需要15分钟，目前的算法只需要2秒钟。

proc 文件系统以前只支持最多64k 个任务，而且所有的任务包括进程和线程都显示在 proc 目录下，这样限制了系统可以创建的线程数，当存在大量线程的时候，proc 文件系统的性能也让人无法接受，新核心的 proc 文件系统现在支持超过64k 个进程，同时只显示进程而不再显示线程。另一个使 Linux 不能支持大任务数的原因在于 TLS 的限制，这方面的内容见4.5节。

4.2 对 clone 系统调用进行了扩展

clone 系统调用在新版本中被扩展了，使得对核心线程的创建有了更多的控制选项。对新线程的创建也进行了优化，主要是 O(n)的 get_pid()函数，被 O(1)的 alloc_pidmap()函数所代替，另外增加了多个标志位。其中和线程有关的有：CLONE_DETACHED，CLONE_SYSVSEM，CLONE_SETTLS，CLONE_SETTID，CLONE_CLEARTID。

如果 CLONE_SETTID 标志被置位，核心会将新线程的 id 保存到特定的地址（WORD 型），如果 CLONE_CREARTID 被置位，则在线程被终止时，清除那个特定的内存位置，这使线程库可以决定何时能够安全地重用原先作为线程堆栈的内存空间。

CLONE_DETACHED：如果此标志被置位，则线程退出时不给父进程发送 child-exit 信号(SIGCHLD)，并且上述 WORD 型内存地址同时还被扩展作为一个 futex 使用。这样可以简化 pthread_join()的实现。当子线程退出时，核心会对这个 futex 做一个 FUTEX_WAKE 操作，来唤醒等待此线程退出的线程(关于 Futex 见4.3节)。

CLONE_SETTLS 帮助实现信号安全的线程寄存器装载。因为信号是个异步事件，任何时候都可能发生，很可能当任务一被创建就会收到信号，而信号的处理需要用到当前任务的 TLS，所以，当 clone 调用的时候，或者信号被禁止，或者新任务在 clone()系统调用返回的时候它的 TLS 就已经被正确地设置好了。CLONE_TLS 实现了这一要求，如果 CLONE_TLS 被置位，则 clone()系统调用的第4个参数是调用者提供的新建线程的 TLS，而在 clone()系统调用中，核心就可以正确地设置好新线程的 TLS。

4.3 Futex

多线程程序通常都需要在线程间使用相应的同步机制，由于同步机制的使用频率很高，因此它的效率对多线程程序效率的影响很大。一个新的同步机制 futex 应运而生，自2.5.7开始出现在 Linux 内核中。futex 是 Fast Userspace Mutex 的缩写。它是一种快速的互斥量，是在设计 NGPT 线程库的过程中发展起来的。用户空间的互斥量的基本思想是：对互斥量的操作主要在用户态的库函数完成，只有在线程因加锁失败需要睡眠或被唤醒这些情况下才调用系统调用。由于系统调用的开销通常需要上百条汇编指令以上，因此相对于互斥量的测试并设置(test and set)操作，开销是很大的，减少这部分的开销，将使得用户空间的互斥量在同步操作很多的情况下会得到比较好的性能。具体实现上，futex 并不很复杂，基本上，futex 就是一个在共享内存中的结构，由于在共享内存中，同时此共享内存页被锁定在内存中，使得多个线程可以同时使用简单的“测试并设置”指令来访问它。对不同的线程，同一个 futex 可能对应不同的虚地址，但是 futex 所在的共享内存页的指针 struct page* 和页内偏移量是不变的，所以，linux 使用这两个值的组合来作为 futex 的 id。

futex 的系统调用界面从最初出现以来，经过了较大的变化，目前的系统调用如下：

```
sys_futex(ulong address, int op, int val, struct timespec *
    utime);
```

其中参数 op 为 FUTEX_WAIT 或 FUTEX_WAKE，语义上类似于互量斥的 down，up 操作。参数 address 是 futex 的句柄。val 对不同的操作有不同的含义。utime 是超时时间，只对 FUTEX_WAIT 有效。

FUTEX_WAIT 简化了的流程如下：

读 futex，如果值不等于 val，返回 EWOULDBLOCK，否则睡眠 utime 时间，如果时间到，返回 ETIMEOUT，如果睡眠的时候收到信号，返回 EINTR

FUTEX_WAKE 简化了的流程如下：

从队列中唤醒 val 个等待的线程，返回唤醒的线程数

从根本上来说，futex 提供了一个基础服务，它的系统调用接口经过了认真的设计。在它的基础上，可以在不修改内核的条件下设计其它的同步机制，如信号量、读写锁、spin 锁、Pthreads 互斥量等等。

在 futex 基础上实现一个互斥量的代码如下：

```
mutex_lock(int * mutex){
    //最高位为1，表示锁被占用，
    //0-30位表示等待锁的线程数
    if( atomic_bit_test_set( mutex,31)==0)return;
    atomic_increment(mutex );
    while(1){
        if(atomic_bit_test_set(mutex,31)== 0){
            atomic_decrement( mutex );
            return;
```

```
        }
        v= * mutex;
        sys_futex(mutex,FUTEX_WAIT,v,NULL);
    }
}
mutex_unlock( int * mutex){
    //如果0-30位都为0,表示没有等待线程
    if( atomic_add_zero( 0x80000000, mutex ))return;
    sys_futex(mutex, FUTEX_WAKE, 1, NULL);
}
```

注：atomic_bit_test_set，atomic_add_zero，atomic_decrement 是原子操作，在i386体系中可以用 LOCK 前缀实现。

可见在没有同步冲突的情况下，mutex_lock/mutex_unlock 都不需要进入系统调用。futex 目前被广泛地用于实现同步机制，包括 PThreads 同步机制。

4.4 posix 兼容性

针对 Posix 兼容性的改进比较零碎，但是为了符合 Posix 标准，又是非常重要的。

(1)符合 Posix 标准的信号处理模型。按照 POSIX Threads 标准对多线程进程的信号的处理进行了改写，发送到进程的信号被投递到进程的一个可用的线程，而不是所有线程，致命信号则终止整个进程。

(2)针对 pid 的改动和 task_struct 结构的变化。Linux 将进程称为任务，在内核中对应的数据结构是 struct task_struct，自 Linux2.4起，和 CPU 密切相关的部分被组合到一起构成一个 thread_struct 结构，每个 task_struct 结构都包含一个 thread_struct 结构。具体地说，thread_struct 结构包含 esp0,eip,esp,fs,gs，硬件调试寄存器，任务的浮点信息，任务的 vm86信息，IO 权限表。

2.4内核在 task_struct 中添加了2个成员，一个是 tgid，(tgid 应该是 thread group id 的意思)，同一个进程的所有线程有不同的 pid，而有相同的 tgid，即进程的第一个线程的 pid。这样从含义上来说，tgid 更应该被命名为 pid(process id)，而 pid 应该被命名为 tid(thread id)，但是为了保持和以前版本的兼容性，不引起混乱，Linux 并没有这样命名。

另一个是 thread_group，是个链表头，通过这个链表可以找到同一个进程的所有线程。类似于遍历所有进程的宏 for_each_task，Linux 为遍历一个进程的所有线程定义了一个宏 next_thread，其使用方法和 for_each_task 略有不同。

在2.4系列内核中，系统调用 sys_getpid()返回的不是 pid，而是 tgid，这样对同一个进程的线程 sys_getpid()返回相同的值，这就解决了以前版本中同一个进程的线程返回不同的进程号的问题。在2.5.4内核，为了编写线程库的要求，又在内核中添加了一个新的系统调用 sys_gettid()用来返回一个线程的线程号，即返回的是 task_struct 的 pid 成员。在调用系统 clone 调用的时候，指定 CLONE_THREAD 标志位，核心就创建一个线程，和创建进程不同的地方，就是新创建的任务的 tgid 被设置为和父任务的 tgid 一致，并将它加入到父任务的 thread_group 中。

(3)exec 系统调用，新创建的进程拥有老线程的 pid，所有老进程的线程都被终止。

(4)资源报告现在针对整个进程而不是第一个线程。

(5)proc 中现在只显示进程，或者说只显示线程组的组长。

(6)核心保持线程组的组长不退出，直到组内的所有线程退出，这确保 proc 文件系统和信号处理能够正确实现。

(7)通过为 clone 系统调用添加了一个标志 CLONE_SYSVSEM，改正了关于 SystemVs SEM_UNDO 操作的问题，现在 SEM_UNDO 在整个进程退出时才进行，而不是在创建此信号量的线程退出时进行。

4.5 线程局部变量(TLS)s

TLS 的实现是以前版本的 Linux 在 IA32体系结构下最多只能支持8192个任务的原因。因为使用 GS 作为线程寄存器，使用局部描述符表 LDT 来保存任务的 TLS 的存储段的描述符，而一个 LDT 只能容纳8192个描述符，所以使得 Linux 最大任务数有了上限。现在通过添加一个新的系统调用 sys_set_thread_area()，在 IA32体系结构上支持了任意多的 TLS。这个系统调用允许分配多个 GDT 表项，用来存取 TLS。每个线程可以分配并设置自己的 GDT 表项，整个核心中，对每个 CPU 有一个全局描述符表 GDT，表中第6个表项被固定用来保存当前任务的 TLS 存储段的描述符，由调度程序在调度一个任务的同时设置为任务相应的 TLS。这样通过共用一个表项解决了8192个数的限制。

值得一提的是 gcc2.3对 TLS 也有了新的支持。在不使用 C/C++的新关键字_thread 的时候，程序员需要使用 POSIX 函数调用来使用 TLS：

```
int pthread_key_create (pthread_key_t * , void ( * )
    (void * ))
int pthread_setspecific(pthread_key_t, const void * )
    void * pthread_getspecific( pthread_key_t)
```

这样使用是很不方便，而且容易出错。对 C/C++语言，一个新的关键字_thread 被设计用来简化对线程局部变量的使用。这样，我们可以使用一个简单的声明：

```
_thread int i;
```

来声明一个线程局部变量，(注意，只有全局变量和静态变量允许使用_thread 修饰，因为局部变量本来就是线程私有的)目前版本的 gcc 已经支持_thread 关键字。

为了实现_thread，gcc，ELF 文件格式，ELF 装载程序，都作了相应的修改。这些不属于本文的讨论范围，请参阅文[4]。

5 线程库

5.1 3种实现方式

对程序员来说，他们真正直接面对的还是一个线程函数库。作为线程库，最本质的区别在于线程和核心线程的对应关系。

线程的实现通常需要由函数库和操作系统核心合作完成，如果所有实现都在核心中做，也就是每个线程对应一个核心线程，称做1:1模型。如果所有实现都在函数库做，也就是多个线程对应一个核心进程称做 M:1模型。它的优点是不需要操作系统的支持，线程的创建同步等操作开销小，缺点是在多处理器的条件下性能不会提高。(关于阻塞型的系统调用，如果一个线程调用了一个阻塞型的系统调用，在 M:1模型下，整个进程都会被堵塞，但是设计巧妙的 M:1的线程库都通过在内部使用异步 I/O 型系统调用来模拟阻塞型系统调用，解决了这个问题)。最复杂的是 M 个线程对应 N 个核心线程，称做 M:N 模型，它兼具1:1和 M:1两种模型的优点，但是缺点是实现比较复杂。同一个 API，比如 PThreads 可以用上述任何一种方式实现，而程序员使用的时候只需要连接到相应的函数库即可。

5.2 下一代 Linux 线程库 NGPT 和 NPTL

目前 Linux 在线程方面主要有2个项目 NGPT(Next

(下转第165页)

务恢复函数。

```
index=end_of_stable_log;
flag=0;
while(index<= system_log_length)
{
    status=check(system_log[index])
    if(status==BAD&&flag==0)
    {
        begin_audit= index;
        end_audit =index;
        flag=1;
    }
    else if(status=BAD&&flag==1)
      end_audit=index;
    index ++;
}
recover(needed_audit_segment, begin_audit, end_audit)
```

直接的逻辑异常数据，由于与应用程序的逻辑结构以及用户的交互输入有关，对这类异常数据的检测比较困难，一旦发现这类不一致状态，可以通过对日志进行分析，追溯产生不一致产生的根源，然后将与此相关的事务进行撤销或者重做处理，消除间接异常数据的影响。

结论 校验码技术是一种检测数据状态不一致性的手段，读时比较可以避免异常数据的传播。直接物理异常数据的静态检测与恢复，能及时修复部分异常数据，但采用静态的方式，进行检测恢复处理时，需要等待活动事务结束，这对只有短事务的应用系统而言是很有效的方法，但对存在长事务的应用系统而言，势必会影响系统的性能，这也是今后工作中需要进一步研究解决的问题。

参 考 文 献

1 Lehman T J, Shekita E J. An Evaluation of Starburst's Memory Resident Storage Component [J]. IEEE Transactions on knowledge and data engineering, 1992, 4(6): 555～566

2 Sullivan M, Stonebraker M. Using Write Protected Data Structures to Improve Software Fault Tolerance in Highly Available Database Management Systems [A]. In: Proc. of the 17th VLDB conf. [C], 1991. 171～180

3 Jagadish H V, Lieuwen D. Dali: A High Performance Main Memory Storage Manager [A]. In: Proc. of the 20th VLDB Conf. [C], 1994. 48～59

4 Delis A, Kanitkar V, Kollios G. Database Architectures [M]. New York, NY, Feb. 1999

5 Garcia-Molina H, Ullman J D, Widom J. 数据库系统实现[M]. 机械工业出版社, 2001

6 Liu Yun-Sheng, et al. Data Organization and Management of Real-Time Main Memory Database [J]. Computer Research & Development, 1998, 35(5): 469～473

7 王意洁，胡守仁. 面向对象数据库中的故障恢复[J]. 计算机科学, 1999, 26(2): 35～38

8 张振华. 业务对象中恢复机制研究[D]: [西安电子科技大学2002年硕士论文]

（上接第160页）

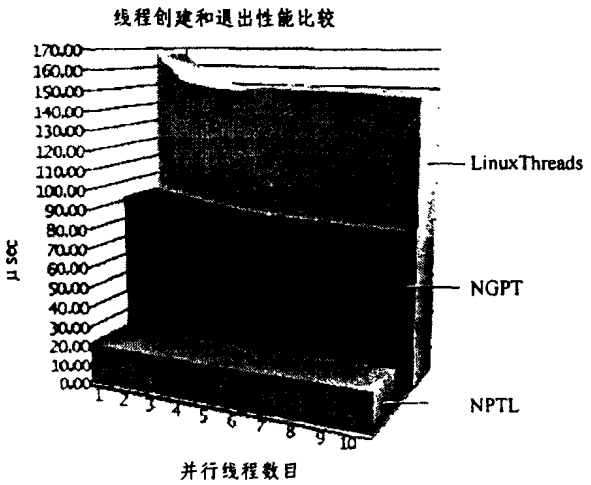

图1

Generation POSIX Threading）和 NPTL（Native POSIX Thread Library）。它们采用不同的设计思想来增强 Linux 对线程的支持。NGPT 使用 M:N 模型，开发者认为 M:N 模型能完全地使用多 CPU 的能力，同时又使开销尽可能少。通过将线程调度主要放在用户态完成，使核心态的任务切换变得很少。当然，由于需要使用核心态和用户态两个调度器，使 NGPT 实现起来很复杂，信号处理也很复杂，当一个线程被阻塞时需要避免阻塞其他线程（它对 Linux 核心做的修改非常小）。NPTL 仍然使用1:1模型，但是在核心做了很大的改变。开发者认为以下事实使1:1模型非常有吸引力：1个调度器，信号处理完全由核心完成，不存在阻塞问题。本文所述对核心的修改一部分是同时针对 NGPT 和 NPTL 的，另外很大部分则是专门针对 NPTL 的。

就测试的结果来看，NGPT 目前性能比 NPTL 稍逊一筹。这可能是由于 NGPT 没有完全利用内核最新的进展所造成的，也有可能是因为 M:N 模型的复杂性造成了它的性能稍差。由于 NGPT 相对于 NPTL 有自身的优点，而且更加成熟，我们下一步的计划是对 NGPT 做深入的研究，尽可能地发挥出它的潜力。

图1中纵坐标是 NPTL、NGPT 和 LinuxThreads 做10000次线程创建和退出操作，平均每一次需要的时间。横坐标是并行的线程数。

无论如何 NGPT，NPTL 两者相对于 LinuxThreads 都有着非常大的性能提升，同时它们都很好地实现了 PThreads 标准。两者最后谁会成为下一代 Linux 标准线程库还很难说。可以预计，当它们最终替代了 LinuxThreads 以后，Linux 作为一个企业级的操作系统必然又迈出了坚实的一步。

参 考 文 献

1 Lewis Bil, Berg D J. Threads Primer - a Guide to Multithreaded Programming. SunSoft Press, 1996

2 Richter J. Windows NT 高级编程技术. 清华大学出版社, 1994

3 Franke H, Russell R. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In: Proc. of The Ottawa Linux Symposium, 2002

4 Drepper U. ELF Handing For Thread-Local Storage. http: // people. redhat. com/drepper/tls. pdf