最新随笔

1. socket泄露的问题
2. gdb 调试多线程
3. MMAP和DIRECT IO区别
4. 三年回首：C基础
5. 定时器管理：nginx的红黑树和libevent的堆
6. strsep和strtok_r替代strtok
7. 缓存穿透和缓存失效
8. mmap为什么比read/write快(兼论buffercache和pagecache)
9. B+Tree和MySQL索引分析
10. c++拷贝构造和编译优化

我的标签

linux(9)
c/c++(8)
data structure(7)
tcp/ip(7)
operating system(4)
databse(4)
Optimization(3)
interview(3)
algorithm(1)
network programming(1)
更多

积分与排名

积分 - 154272
排名 - 1704

阅读排行榜

1. WEB服务器、应用程序服务器、HTTP服务器区别(56682)
2. shell中exec解析(52635)
3. Linux进程调度原理(38794)
4. Linux内存管理原理(37561)
5. python中的编码问题：以ascii和unicode为主线(35704)
6. mysql事务和锁InnoDB(31425)
7. JavaScript和JQuery好书推荐(23743)
8. 正确解读free -m(22086)
9. c语言libcurl库的异步用法(20013)
10. VIM插件攻略(19683)

评论排行榜

1. mysql事务和锁InnoDB(9)
2. 80X86寄存器详解(5)
3. HTTP报文(4)
4. 公司大了怎么办(4)
5. WEB服务器、应用程序服务器、HTTP服务器区别(3)
6. shell中exec解析(3)
7. Linux内存管理原理(3)
8. 可重入性与线程安全(2)
9. Linux进程调度原理(2)
10. 自白书(2)

linux线程的实现

首先从OS设计原理上阐明三种线程：内核线程、轻量级进程、用户线程

内核线程

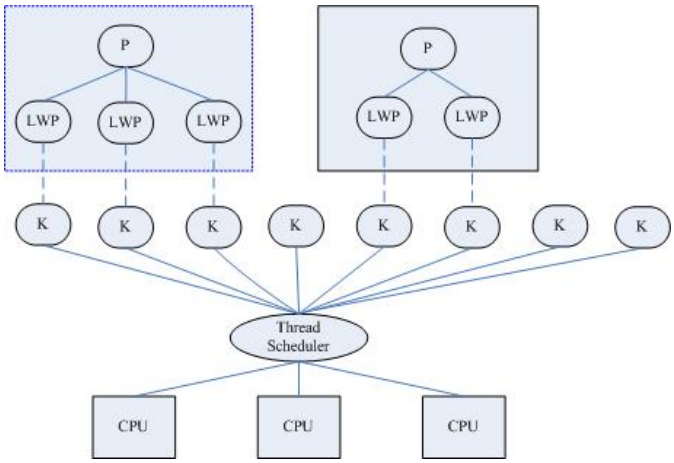
内核线程就是内核的分身，一个分身可以处理一件特定事情。这在处理异步事件如异步IO时特别有用。内核线程的使用是廉价的，唯一使用的资源就是内核栈和上下文切换时保存寄存器的空间。支持多线程的内核叫做多线程内核(Multi-Threads kernel)。

轻量级进程

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。每一个进程有一个或多个LWPs，每个LWP由一个内核线程支持。这种模型实际上就是恐龙书上所提到的一对一线程模型。在这种实现的操作系统中，LWP就是用户线程。

由于每个LWP都与一个特定的内核线程关联，因此每个LWP都是一个独立的线程调度单元。即使有一个LWP在系统调用中阻塞，也不会影响整个进程的执行。

轻量级进程具有局限性。首先，大多数LWP的操作，如建立、析构以及同步，都需要进行系统调用。系统调用的代价相对较高：需要在user mode和kernel mode中切换。其次，每个LWP都需要有一个内核线程支持，因此LWP要消耗内核资源（内核线程的栈空间）。因此一个系统不能支持大量的LWP。

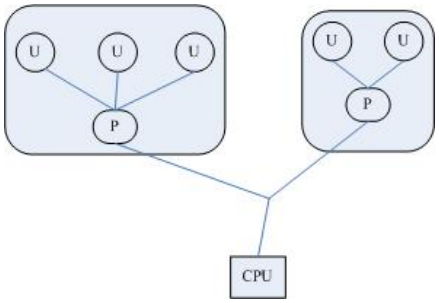


注：

LWP 的术语是借自于 SVR4/MP 和 Solaris 2.x 。有些系统将 LWP 称为虚拟处理器。而将之称为轻量级进程的原因可能是：在内核线程的支持下， LWP 是独立的调度单元，就像普通的进程一样。所以 LWP 的最大特点还是每个 LWP 都有一个内核线程支持。

用户线程

LWP虽然本质上属于用户线程，但LWP线程库是建立在内核之上的，LWP的许多操作都要进行系统调用，因此效率不高。而这里的用户线程指的是完全建立在用户空间的线程库，用户线程的建立，同步，销毁，调度完全在用户空间完成，不需要内核的帮助。因此这种线程的操作是极其快速的且低消耗的。

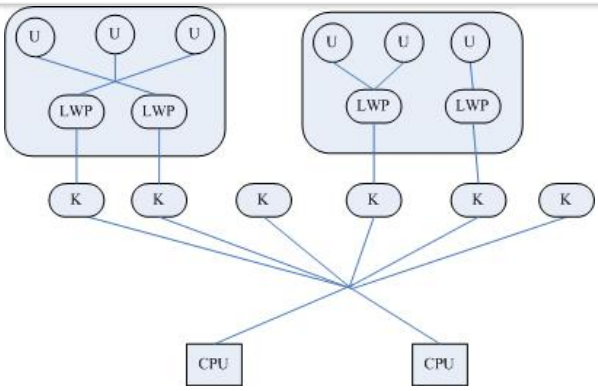


上图是最初的一个用户线程模型，从中可以看出，进程中包含线程，用户线程在用户空间中实现，内核并没有直接对用户线程进程调度，内核的调度对象和传统进程一样，还是进程本身，内核并不知道用户线程的存在。用户线程之间的调度由在用户空间实现的线程库实现。

这种模型对应着恐龙书中提到的多对一线程模型，其缺点是一个用户线程如果阻塞在系统调用中，则整个进程都将会阻塞。

加强版的用户线程——用户线程+LWP

这种模型对应着恐龙书中多对多模型。用户线程库还是完全建立在用户空间中，因此用户线程的操作还是很廉价，因此可以建立任意多需要的用户线程。操作系统提供了 LWP 作为用户线程和内核线程之间的桥梁。 LWP 还是和前面提到的一样，具有内核线程支持，是内核的调度单元，并且用户线程的系统调用要通过 LWP ，因此进程中某个用户线程的阻塞不会影响整个进程的执行。用户线程库将建立的用户线程关联



很多文献中都认为轻量级进程就是线程，实际上这种说法并不完全正确，从前面的分析中可以看到，只有在用户线程完全由轻量级进程构成时，才可以说轻量级进程就是线程。

LinuxThreads

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（我们常用的pthread库）中实现。 一直以来, linux内核并没有线程的概念. 每一个执行实体都是一个task_struct结构, 通常称之为进程.

进程是一个执行单元, 维护着执行相关的动态资源. 同时, 它又引用着程序所需的静态资源.通过系统调用clone创建子进程时, 可以有选择性地让子进程共享父进程所引用的资源. 这样的子进程通常称为轻量级进程，如上文所述，又叫内核线程。

[插曲]说下fork和vfork的区别

fork时，子进程是父进程的一个拷贝。子进程从父进程那得到了数据段和堆栈段，但不是与父进程共享而是单独分配内存。然而这里的非共享最初状态是共享的，linux下使用了写时复制技术，刚开始共享父进程的数据段，在写数据段的时候才进行复制，以fork为例，最终共享的资源就是task_struct、系统空间堆栈（copy_thread）、页面表等。

vfork时，因为实现为子进程先执行，所以是不拷贝（没必要）父进的虚存空间，也就是用户空间堆栈，clone(clone_vfork|clone_vm|sigchld,0)，指明的参数是不会拷贝，注定共享的。因为现在的fork都是写实拷贝，所以vfork的优势便不明显了—只是不用向vfork那样拷贝页表。另外，内核都是优先让子进程先执行，考虑到调度问题，fork不保证；但vfork可保证这点 2.4内核 do_dork是fork/vfork/clone系统调用的共同代码，其核心流程如下：

1) 默认对所有资源进行共享暂不复制；

2) 如果flags未指定共享（相应位为0），则进行深层次复制。包括，file,fs,sighand,mm。

以CLONE_FILES为例，对fork而言为1，也就是必须复制两份files，这样父子进程才有独立上下文（各自独立lseek不影响，但是文件指针肯定还是指向一个）；但是对于vfork，这个标志为1，也就是父子进程共享文件上下文（注意这里不是共享文件指针，连上下文都共享！也就是子进程lseek会改变父进程读写位置），这岂不乱套（类似还有vfork的CLONE_VM标志）！别担心，do_fork中会保证vfork时候，自进程先执行完！

特殊说下，mm资源即使是非共享的，即CLONE_VM=1（fork如此），也不马上复制，而是复制页面表后，把也表项设置为写保护，这样无论谁写，届时都会再复制一份出来，这才完成的资源的独立—对fork而言。

3) 复制系统堆栈（区别于用户空间VM）

用户态线程由pthread库实现，使用pthread以后, 在用户看来, 每一个task_struct就对应一个线程, 而一组线程以及它们所共同引用的一组资源就是一个进程. 但是, 一组线程并不仅仅是引用同一组资源就够了, 它们还必须被视为一个整体.

POSIX线程实现基于如下要求:
1, 查看进程列表的时候, 相关的一组task_struct应当被展现为列表中的一个节点;
2, 发送给这个"进程"的信号(对应kill系统调用), 将被对应的这一组task_struct所共享, 并且被其中的任意一个"线程"处理;
3, 发送给某个"线程"的信号(对应pthread_kill), 将只被对应的一个task_struct接收, 并且由它自己来处理;
4, 当"进程"被停止或继续时(对应SIGSTOP/SIGCONT信号), 对应的这一组task_struct状态将改变;
5, 当"进程"收到一个致命信号(比如由于段错误收到SIGSEGV信号), 对应的这一组task_struct将全部退出;
6, 以上可能不全;

在linux 2.6以前, pthread线程库对应的实现是一个名叫linuxthreads的lib. linuxthreads利用前面提到的轻量级进程来实现线程, 但是对于POSIX提出的那些要求, linuxthreads 除了第5点以外, 都没有实现(实际上是无能为力):
1, 如果运行了A程序, A程序创建了10个线程, 那么在shell下执行ps命令时将看到11个A进程, 而不是1个(注意, 也不是10个, 下面会解释);
2, 不管是kill还是pthread_kill, 信号只能被一个对应的线程所接收;
3, SIGSTOP/SIGCONT信号只对一个线程起作用;

还好linuxthreads实现了第5点, 我认为这一点是最重要的. 如果某个线程"挂"了, 整个进程还在若无其事地运行着, 可能会出现很多的不一致状态. 进程将不是一个整体, 而线程也不能称为线程. 或许这也是为什么

接下来要说说, 为什么A程序创建了10个线程, 但是ps时却会出现11个A进程了. 因为linuxthreads自动创建了一个管理线程. 上面提到的"第5点"就是靠管理线程来实现的.当程序开始运行时, 并没有管理线程存在(因为尽管程序已经链接了pthread库, 但是未必会使用多线程). 程序第一次调用pthread_create时, linuxthreads发现管理线程不存在, 于是创建这个管理线程. 这个管理线程是进程中的第一个线程(主线程)的儿子.

然后在pthread_create中, 会通过pipe向管理线程发送一个命令, 告诉它创建线程. 即是说, 除主线程外, 所有的线程都是由管理线程来创建的, 管理线程是它们的父亲.于是, 当任何一个子线程退出时, 管理线程将收到SIGUSER1信号(这是在通过clone创建子线程时指定的). 管理线程在对应的sig_handler中会判断子线程是否正常退出, 如果不是, 则杀死所有线程, 然后自杀.

那么, 主线程怎么办呢? 主线程是管理线程的父亲, 其退出时并不会给管理线程发信号. 于是, 在管理线程的主循环中通过getppid检查父进程的ID号, 如果ID号是1, 说明父亲已经退出, 并把自己托管给了init进程(1号进程). 这时候, 管理线程也会杀掉所有子线程, 然后自杀. 那么, 如果主线程是调用pthread_exit主动退出的呢? 按照posix的标准，这种情况下其他子线程是应该继续运行的. 于是, 在linuxthreads中, 主线程调用pthread_exit以后并不会真正退出, 而是会在pthread_exit函数中阻塞等待所有子线程都退出了, pthread_exit才会让主线程退出. (在这个等等过程中, 主线程一直处于睡眠状态.)

可见, 线程的创建与销毁都是通过管理线程来完成的, 于是管理线程就成了linuxthreads的一个性能瓶颈. 创建与销毁需要一次进程间通信, 一次上下文切换之后才能被管理线程执行, 并且多个请求会被管理线程串行地执行.

NPTL(Native POSIX Threading Library)

到了linux 2.6, glibc中有了一种新的pthread线程库NPTL. NPTL实现了前面提到的POSIX的全部5点要求. 但是, 实际上, 与其说是NPTL实现了, 不如说是linux内核实现了.

在linux 2.6中, 内核有了线程组的概念, task_struct结构中增加了一个tgid(thread group id)字段. **如果这个task是一个"主线程", 则它的tgid等于pid, 否则tgid等于进程的pid(即主线程的pid),此外，每个线程有自己的pid**。在clone系统调用中, 传递CLONE_THREAD参数就可以把新进程的tgid设置为父进程的tgid(否则新进程的tgid会设为其自身的pid).类似的XXid在task_struct中还有两个：task->signal->pgid保存进程组的打头进程的pid、task->signal->session保存会话打头进程的pid。通过这两个id来关联进程组和会话。

有了tgid, 内核或相关的shell程序就知道某个tast_struct是代表一个进程还是代表一个线程, 也就知道在什么时候该展现它们, 什么时候不该展现(比如在ps的时候, 线程就不要展现了).**而getpid(获取进程ID)系统调用返回的也是tast_struct中的tgid, 而tast_struct中的pid则由gettid系统调用来返回**.在执行ps命令的时候不展现子线程，也是有一些问题的。比如程序a.out运行时，创建了一个线程。假设主线程的pid是10001、子线程是10002（它们的tgid都是10001）。这时如果你kill 10002，是可以把10001和10002这两个线程一起杀死的，尽管执行ps命令的时候根本看不到10002这个进程。如果你不知道linux线程背后的故事，肯定会觉得遇到灵异事件了。

为了应付"发送给进程的信号"和"发送给线程的信号", **task_struct里面维护了两套signal_pending, 一套是线程组共享的, 一套是线程独有的**.通过kill发送的信号被放在线程组共享的signal_pending中, 可以由任意一个线程来处理; 通过pthread_kill发送的信号(pthread_kill是pthread库的接口, 对应的系统调用中tkill)被放在线程独有的signal_pending中, 只能由本线程来处理.

当线程停止/继续, 或者是收到一个致命信号时, 内核会将处理动作施加到整个线程组中.

NGPT(Next Generation POSIX Threads)

上面提到的两种线程库使用的都是内核级线程(每个线程都对应内核中的一个调度实体), 这种模型称为1:1模型(1个线程对应1个内核级线程);而NGPT则打算实现M:N模型(M个线程对应N个内核级线程), 也就是说若干个线程可能是在同一个执行实体上实现的.

线程库需要在一个内核提供的执行实体上抽象出若干个执行实体, 并实现它们之间的调度. 这样被抽象出来的执行实体称为用户级线程.大体上, 这可以通过为每个用户级线程分配一个栈, 然后通过longjmp的方式进行上下文切换. (百度一下"setjmp/longjmp", 你就知道.)

但是实际上要处理的细节问题非常之多. 目前的NGPT好像并没有实现所有预期的功能, 并且暂时也不准备去实现.

用户级线程的切换显然要比内核级线程的切换快一些, 前者可能只是一个简单的长跳转, 而后者则需要保存/装载寄存器, 进入然后退出内核态. (进程切换则还需要切换地址空间等).而用户级线程则不能享受多处理器, 因为多个用户级线程对应到一个内核级线程上, 一个内核级线程在同一时刻只能运行在一个处理器上.不过, M:N的线程模型毕竟提供了这样一种手段, 可以让不需要并行执行的线程运行在一个内核级线程对应的若干个用户级线程上, 可以节省它们的切换开销.

据说一些类UNIX系统(如Solaris)已经实现了比较成熟的M:N线程模型, 其性能比起linux的线程还是有着一定的优势.

参考文献：linux内核设计与实现 3.4 线程在linux中的实现

linux内核源代码情景分析 4.3

APUE 8.4 VFORK函数

linux内核源代码情景分析 4.3 系统调用fork vfork clone

Linux 线程实现机制分析 http://www.ibm.com/developerworks/cn/linux/kernel/l-thread/

关于进程、线程和轻量级进程的一些笔记 http://www.cnitblog.com/tarius.wu/articles/2277.html