linux线程-应用编程-专题讲座

written by 王保明

# 线程概念

## 1线程概念

|  |
| --- |
| **什么是线程**   * 在一个程序里的一个执行路线就叫做线程（thread）。更准确的定义是：线程是“一个进程内部的控制序列” * 一切进程至少都有一个执行线程 |
| **进程与线程**   * 进程是资源竞争的基本单位 * 线程是程序执行的最小单位 * 线程共享进程数据，但也拥有自己的一部分数据   + 线程ID   + 程序计数器   + 寄存器组   + 栈   + errno * 一个进程内部的线程可以共享资源   + 代码段   + 数据段   + 打开文件和信号 |
| **fork和创建新线程的区别**   * 当一个进程执行一个fork调用的时候，会创建出进程的一个新拷贝，新进程将拥有它自己的变量和它自己的PID。这个新进程的运行时间是独立的，它在执行时几乎完全独立于创建它的进程 * 在进程里面创建一个新线程的时候，新的执行线程会拥有自己的堆栈（因此也就有自己的局部变量），但要与它的创建者共享全局变量、文件描述符、信号处理器和当前的工作目录状态 |
| **线程的优点**   * 创建一个新线程的代价要比创建一个新进程小得多 * 与进程之间的切换相比，线程之间的切换需要操作系统做的工作要少很多 * 线程占用的资源要比进程少很多 * 能充分利用多处理器的可并行数量 * 在等待慢速I/O操作结束的同时，程序可执行其他的计算任务 * 计算密集型应用，为了能在多处理器系统上运行，将计算分解到多个线程中实现 * I/O密集型应用，为了提高性能，将I/O操作重叠。线程可以同时等待不同的I/O操作。 |
| **线程缺点**   * 性能损失   + 一个很少被外部事件阻塞的计算密集型线程往往无法与共它线程共享同一个处理器。如果计算密集型线程的数量比可用的处理器多，那么可能会有较大的性能损失，这里的性能损失指的是增加了额外的同步和调度开销，而可用的资源不变。 * 健壮性降低   + 编写多线程需要更全面更深入的考虑，在一个多线程程序里，因时间分配上的细微偏差或者因共享了不该共享的变量而造成不良影响的可能性是很大的，换句话说线程之间是缺乏保护的。 * 缺乏访问控制   + 进程是访问控制的基本粒度，在一个线程中调用某些OS函数会对整个进程造成影响。 * 编程难度提高   + 编写与调试一个多线程程序比单线程程序困难得多 |
| **线程调度竞争范围**   * 操作系统提供了各种模型，用来调度应用程序创建的线程。这些模型之间的主要不同是：在竞争系统资源（特别是CPU时间）时，线程调度竞争范围(thread-scheduling contention scope）不一样 * 进程竞争范围（process contention scope）：各个线程在同一进程竞争“被调度的CPU时间”（但不直接和其他进程中的线程竞争）。 * 系统竞争范围（system contention scope）：线程直接和“系统范围”内的其他线程竞争。 |

## 2线程模型

|  |
| --- |
| **N:1用户线程模型**   * “线程实现”建立在“进程控制”机制之上，由用户空间的程序库来管理。OS内核完全不知道线程信息。这些线程称为用户空间线程。 * 这些线程都工作在“进程竞争范围”     **N:1用户线程模型**   * 在N:1线程模型中，内核不干涉线程的任何生命活动，也不干涉同一进程中的线程环境切换。 * 在N:1线程模型中，一个进程中的多个线程只能调度到一个CPU，这种约束限制了可用的并行总量。   第二个缺点是如果某个线程执行了一个“阻塞式”操作（如read），那么，进程中的所有线程都会阻塞，直至那个操作结束。为此，一些线程的实现是为这些阻塞式函数提供包装器，用非阻塞版本替换这些系统调用，以消除这种限制。 |
|  |
| **1:1核心线程模型**   * 在1:1核心线程模型中，应用程序创建的每一个线程都由一个核心线程直接管理。 * OS内核将每一个核心线程都调到系统CPU上，因此，所有线程都工作在“系统竞争范围”。 * 这种线程的创建与调度由内核完成，因为这种线程的系统开销比较大（但一般来说，比进程开销小） |
| **N:M混合线程模型**   * N:M混合线程模型提供了两级控制，将用户线程映射为系统的可调度体以实现并行，这个可调度体称为轻量级进程（LWP:lightweight process），LWP再一一映射到核心线程 |
| **总结：**  （1）用户级线程  用户级线程主要解决的是上下文切换的问题，它的调度算法和调度过程全部由用户自行选择决定，在运行时不需要特定的内核支持。在这里，操作系统往往会提供一个用户空间的线程库，该线程库提供了线程的创建、调度和撤销等功能，而内核仍然仅对进程进行管理。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函数，那么该进程包括该进程中的其他所有线程也同时被阻塞。这种用户级线程的主要缺点是在一个进程中的多个线程的调度中无法发挥多处理器的优势。  （2）轻量级进程  轻量级进程是内核支持的用户线程，是内核线程的一种抽象对象。每个线程拥有一个或多个轻量级线程，而每个轻量级线程分别被绑定在一个内核线程上。  （3）内核线程  这种线程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，这样就可以发挥多处理器的并发优势。  现在大多数系统都采用用户级线程与核心级线程并存的方法。一个用户级线程可以对应一个或几个核心级线程，也就是“一对一”或“多对一”模型。这样既可满足多处理机系统的需要，也可以最大限度地减少调度开销。  使用线程机制大大加快上下文切换速度而且节省很多资源。但是因为在用户态和内核态均要实现调度管理，所以会增加实现的复杂度和引起优先级翻转的可能性。一个多线程程序的同步设计与调试也会增加程序实现的难度 |
|  |
|  |

# 2线程API

## 线程基本api

|  |
| --- |
| **POSIX线程库**   * 与线程有关的函数构成了一个完整的系列，绝大多数函数的名字都是以“pthread\_”打头的 * 要使用这些函数库，要通过引入头文<pthread.h> * 链接这些线程函数库时要使用编译器命令的“-lpthread”选项 |
| **pthread\_create函数**   * 功能：创建一个新的线程 * 原型 * int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*start\_routine)(void\*), void \*arg); * 参数 * thread:返回线程ID * attr:设置线程的属性，attr为NULL表示使用默认属性 * start\_routine:是个函数地址，线程启动后要执行的函数 * arg:传给线程启动函数的参数 * 返回值：成功返回0；失败返回错误码 |
| **错误检查**   * 传统的一些函数是，成功返回0，失败返回-1，并且对全局变量errno赋值以指示错误。 * pthreads函数出错时不会设置全局变量errno（而大部分其他POSIX函数会这样做）。而是将错误代码通过返回值返回 * pthreads同样也提供了线程内的errno变量，以支持其它使用errno的代码。对于pthreads函数的错误，建议通过返回值业判定，因为读取返回值要比读取线程内的errno变量的开销更小 |
| **pthread\_exit函数**   * 功能：线程终止 * 原型 * void pthread\_exit(void \*value\_ptr); * 参数 * value\_ptr:value\_ptr不要指向一个局部变量。 * 返回值：无返回值，跟进程一样，线程结束的时候无法返回到它的调用者（自身） |
| **pthread\_join函数**   * 功能：等待线程结束 * 原型   + int pthread\_join(pthread\_t thread, void \*\*value\_ptr); * 参数   + thread:线程ID   + value\_ptr:它指向一个指针，后者指向线程的返回值 * 返回值：成功返回0；失败返回错误码 |
| **pthread\_self函数**   * 功能：返回线程ID * 原型 * pthread\_t pthread\_self(void); * 返回值：成功返回0 |
| **pthread\_cancel函数**   * 功能：取消一个执行中的线程 * 原型 * int pthread\_cancel(pthread\_t thread); * 参数 * thread:线程ID * 返回值：成功返回0；失败返回错误码 |
| **pthread\_detach函数**   * 功能：将一个线程分离 * 原型   + int thread); * 参数   + thread:线程ID * 返回值：成功返回0；失败返回错误码 |

## 线程客户端测试框架

## 多线程服务器框架

# 3线程属性

|  |
| --- |
| **线程属性**   * 初始化与销毁属性   + int pthread\_attr\_init(pthread\_attr\_t \*attr);   + int pthread\_attr\_destroy(pthread\_attr\_t \*attr); * 获取与设置分离属性   + int pthread\_attr\_getdetachstate(const pthread\_attr\_t \*attr, int \*detachstate);   + int pthread\_attr\_setdetachstate(pthread\_attr\_t \*attr, int detachstate); * 获取与设置栈大小   + int pthread\_attr\_setstacksize(pthread\_attr\_t \*attr, size\_t stacksize);   + int pthread\_attr\_getstacksize(pthread\_attr\_t \*attr, size\_t \*stacksize); * 获取与设置栈溢出保护区大小   + int pthread\_attr\_setguardsize(pthread\_attr\_t \*attr, size\_t guardsize);   + int pthread\_attr\_getguardsize(pthread\_attr\_t \*attr, size\_t \*guardsize); * 获取与设置线程竞争范围   + int pthread\_attr\_getscope(const pthread\_attr\_t \*attr,int \*contentionscope);   + int pthread\_attr\_setscope(pthread\_attr\_t \*attr, int contentionscope); * 获取与设置调度策略   + int pthread\_attr\_getschedpolicy(const pthread\_attr\_t \*attr, int \*policy);   + int pthread\_attr\_setschedpolicy(pthread\_attr\_t \*attr, int policy); * 获取与设置继承的调度策略   + int pthread\_attr\_getinheritsched(const pthread\_attr\_t \*attr, int \*inheritsched);   + int pthread\_attr\_setinheritsched(pthread\_attr\_t \*attr, int inheritsched); * 获取与设置调度参数   + int pthread\_attr\_getschedparam(const pthread\_attr\_t \*attr, struct sched\_param \*param);   + int pthread\_attr\_setschedparam(pthread\_attr\_t \*attr, const struct sched\_param \*param); |
| **并发级别**   * 获取与设置并发级别   + int pthread\_setconcurrency(int new\_level);   + int pthread\_getconcurrency(void); * 仅在N:M线程模型中有效，设置并发级别，给内核一个提提示：表示提供给定级别数量的核心线程来映射用户线程是高效的。 |

其他说明

|  |
| --- |
| • 绑定属性  Linux中采用“一对一”的线程机制，也就是一个用户线程对应一个内核线程。绑定属性就是指一个用户线程固定地分配给一个内核线程，因为CPU时间片的调度是面向内核线程（也就是轻量级进程）的，因此具有绑定属性的线程可以保证在需要的时候总有一个内核线程与之对应。而与之对应的非绑定属性就是指用户线程和内核线程的关系不是始终固定的，而是由系统来控制分配的。  • 分离属性  分离属性是用来决定一个线程以什么样的方式来终止自己。在非分离情况下，当一个线程结束时，它所占用的系统资源并没有被释放，也就是没有真正的终止。只有当pthread\_join()函数返回时，创建的线程才能释放自己占用的系统资源。而在分离属性情况下，一个线程结束时立即释放它所占有的系统资源。这里要注意的一点是，如果设置一个线程的分离属性，而这个线程运行又非常快，那么它很可能在pthread\_create()函数返回之前就终止了，它终止以后就可能将线程号和系统资源移交给其他的线程使用。 |

# 4线程互斥和同步

## 基本概念

|  |
| --- |
| 多线程同步问题：  线程共享进程的资源和地址空间  任何线程对系统资源的操作都会给其他线程带来影响  多线程同步方法：  互斥锁  信号量  条件变量 |

## posix互斥锁

|  |
| --- |
| posix互斥锁   * 互斥锁是用一种简单的加锁方法来控制对共享资源的原子操作。这个互斥锁只有两种状态，也就是上锁和解锁，可以把互斥锁看作某种意义上的全局变量。在同一时刻只能有一个线程掌握某个互斥锁，拥有上锁状态的线程能够对共享资源进行操作。若其他线程希望上锁一个已经被上锁的互斥锁，则该线程就会挂起，直到上锁的线程释放掉互斥锁为止。可以说，这把互斥锁保证让每个线程对共享资源按顺序进行原子操作。 |
| posix互斥锁api   * 互斥锁机制主要包括下面的基本函数。 * 互斥锁初始化：pthread\_mutex\_init() * 互斥锁上锁：pthread\_mutex\_lock() * 互斥锁判断上锁：pthread\_mutex\_trylock() * 互斥锁接锁：pthread\_mutex\_unlock() * 消除互斥锁：pthread\_mutex\_destroy() * 其中，互斥锁可以分为快速互斥锁、递归互斥锁和检错互斥锁。这三种锁的区别主要在于其他未占有互斥锁的线程在希望得到互斥锁时是否需要阻塞等待。快速锁是指调用线程会阻塞直至拥有互斥锁的线程解锁为止。递归互斥锁能够成功地返回，并且增加调用线程在互斥上加锁的次数，而检错互斥锁则为快速互斥锁的非阻塞版本，它会立即返回并返回一个错误信息。默认属性为快速互斥锁。 |
| 该函数用于C函数的多线程编程中，互斥锁的初始化。  头文件：#include <pthread.h>  函数原型：int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex,const pthread\_mutexattr\_t \*restrict attr);  pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;  pthread\_mutex\_init()函数是以动态方式创建互斥锁的，参数attr指定了新建互斥锁的属性。如果参数attr为空，则使用默认的互斥锁属性，默认属性为快速互斥锁 。互斥锁的属性在创建锁的时候指定，在LinuxThreads实现中仅有一个锁类型属性，不同的锁类型在试图对一个已经被锁定的互斥锁加锁时表现不同。  pthread\_mutexattr\_init()函数成功完成之后会返回零，其他任何返回值都表示出现了错误。  函数成功执行后，互斥锁被初始化为未锁住态。 |
| 互斥锁属性  使用互斥锁（互斥）可以使线程按顺序执行。通常，互斥锁通过确保一次只有一个线程执行代码的临界段来同步多个线程。互斥锁还可以保护单线程代码。  要更改缺省的互斥锁属性，可以对属性对象进行声明和初始化。通常，互斥锁属性会设置在应用程序开头的某个位置，以便可以快速查找和轻松修改 |
| 销毁互斥锁对象  pthread\_mutexattr\_destroy(3C)可用来取消分配用于维护 pthread\_mutexattr\_init() 所创建的属性对象的存储空间。  pthread\_mutexattr\_destroy 语法  int pthread\_mutexattr\_destroy(pthread\_mutexattr\_t \*mattr)  #include <pthread.h>  pthread\_mutexattr\_t mattr;  int ret;/\* destroy an attribute \*/  ret = pthread\_mutexattr\_destroy(&mattr);  pthread\_mutexattr\_destroy 返回值  pthread\_mutexattr\_destroy() 成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。  EINVAL 描述: 由 mattr 指定的值无效。 |

## posix条件变量

|  |
| --- |
| 基本概念   * 与互斥锁不同，条件变量是用来等待而不是用来上锁的。条件变量用来自动阻塞一个线程，直到某特殊情况 发生为止。通常条件变量和互斥锁同时使用。 * 条件变量使我们可以睡眠等待某种条件出现。条件变量是利用线程间共享的全局变量进行同步的一种机制， 主要包括两个动作：一个线程等待"条件变量的条件成立"而挂起；另一个线程使"条件成立"（给出条件成立信号）。 * 条 件的检测是在互斥锁的保护下进行的。如果一个条件为假，一个线程自动阻塞，并释放等待状态改变的互斥锁。如果另一个线程改变了条件，它发信号给关联的条件 变量，唤醒一个或多个等待它的线程，重新获得互斥锁，重新评价条件。如果两进程共享可读写的内存，条件变量可以被用来实现这两进程间的线程同步。 * api列表 * pthread\_cond\_init * pthread\_cond\_destroy * pthread\_cond\_wait * pthread\_cond\_signal * pthread\_cond\_broadcast |
| **条件变量使用规范**   * 等待条件代码   pthread\_mutex\_lock(&mutex);  while (条件为假）  pthread\_cond\_wait(cond, mutex);  修改条件  pthread\_mutex\_unlock(&mutex);   * 给条件发送信号代码   pthread\_mutex\_lock(&mutex);  设置条件为真  pthread\_cond\_signal(cond);  pthread\_mutex\_unlock(&mutex); |
| * 使用条件变量之前要先进行初始化。可以在单个语句中生成和初始化一个条件变量如： * pthread\_cond\_t my\_condition=PTHREAD\_COND\_INITIALIZER;（用于进程间线程的通信）。   也可以利用函数pthread\_cond\_init动态初始化  C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\SNAGHTML120212b.PNG |
| pthread\_cond\_destroy函数可以用来摧毁所指定的条件变量，同时将会释放所给它分配的资源。调用该函数的进程也并不要求等待 在参数所指定的条件变量上。 |
| 第一个参数\*cond是指向一个条件变量的指针。第二个参数\*mutex则是对相关的互斥锁的指针。函数pthread\_cond\_timedwait函数类型与函数pthread\_cond\_wait,区别在于，如果达到或是超过所引用的参数\*abstime,它将结束并返回错误ETIME.pthread\_cond\_timedwait函数的参数\*abstime指向一个timespec结构。该结构如下：  typedef struct timespec{         time\_t tv\_sec;         long tv\_nsex;  }timespec\_t; |
| 参数\*cond是对类型为pthread\_cond\_t 的一个条件变量的指针。当调用pthread\_cond\_signal时一个在相同条件变量上阻塞的线程将被解锁。如果同时有多个线程阻塞，则由调度策略确定接收通知的 线程。如果调用pthread\_cond\_broadcast,则将通知阻塞在这个条件变量上的所有线程。一旦被唤醒，线程仍然会要求互斥锁。如果当前没有线程等待通知，则上面两种调用实际上成为一个空操 作。如果参数\*cond指向非法地址，则返回值EINVAL。 |
|  |

## 综合案例1多线程互斥

## 综合案例2线程同步