第8章 多线程编程

8.1 Linux线程概述
8.2 Linux线程编程
8.3 线程的并发访问

8.1 线程概述

- 线程：在一个程序中的多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一个控制序列。
- 线程(thread)技术早在60年代就被提出，但真正的应用多线程到操作系统中去，是在80年代中期，solaris是这方面的佼佼者。
- Linux线程技术的发展：

Linux线程技术的发展

- 在Linux2.2内核中，并不存在真正意义上的线程。当时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“轻”进程，并且线程的个数也很有限，最多只能有4096个进程/线程同时运行。
- Linux2.4内核消除了这个线程个数的限制，并且允许在系统运行中动态地调整进程数上限。
- 在Linux 内核2.6之前的版本中，进程是最主要的处理调度单元，并没支持内核线程机制。Linux系统在1996年第一次获得线程的支持，当时所使用的函数库被称为LinuxThread。

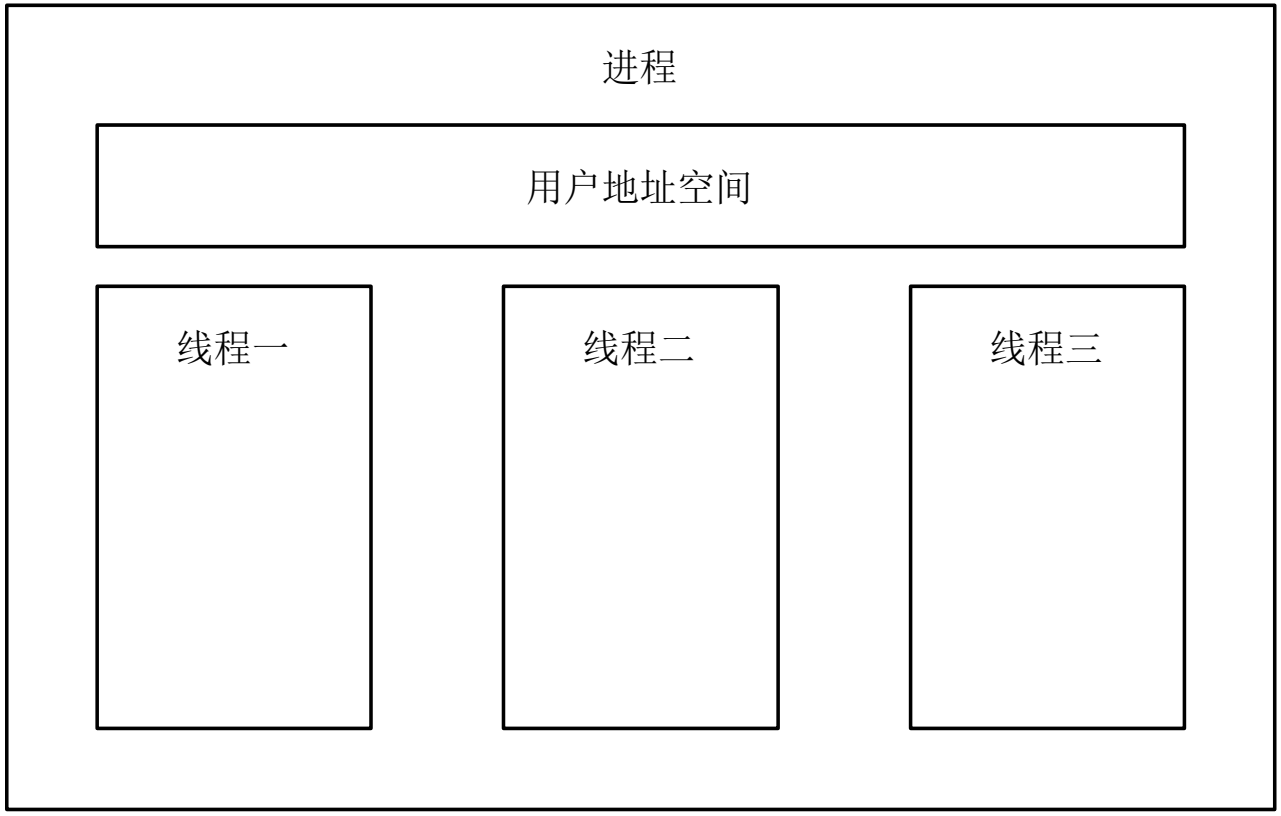
Linux线程技术的发展 （2）

- 为了改善LinuxThread问题，出现根据新内核机制重新编写线程库的问题。许多项目在研究如何改善Linux对线程的支持，其中两个最有竞争力的有由IBM主导的新一代POSIX线程库（Next Generation POSIX Threads，简称为NGPT）和由Red Hat主导的本地化POSIX线程库 (Native POSIX Thread Library，简称为NPTL)。
- NGPT项目在2002年启动，但为了避免出现有多个Linux线程标准，所以在2003年停止该项目。与此同时NPTL问世，最早在Red Hat Linux9中被支持，现在已经成为GNU C函数库的一部分，同时也成为Linux线程的标准。

线程与进程

- 前面已经提到，进程是系统中程序执行和资源分配的基本单位。每个进程都拥有自己的数据段、代码段和堆栈段，这就造成了进程在进行切换等操作时都需要有比较复杂的上下文切换等动作。为了进一步减少处理机的空转时间，支持多处理器以及减少上下文切换开销，进程在演化中出现了另一个概念——线程。它是进程内独立的一条运行路线，处理器调度的最小单元，也可以称为轻量级进程。线程可以对进程的内存空间和资源进行访问，并与同一进程中的其他线程共享。因此，线程的上下文切换的开销比创建进程小很多。
- 同进程一样，线程也将相关的执行状态和存储变量放在线程控制表内。一个进程可以有多个线程，也就是有多个线程控制表及堆栈寄存器，但却共享一个用户地址空间。要注意的是，由于线程共享了进程的资源和地址空间，因此，任何线程对系统资源的操作都会给其他线程带来影响。由此可知，多线程中的同步非常重要。

线程概述（2）



线程拥有自己的栈（有自己的局部变量），但与其他线程共享全局变量、文件描述符、信号处理函数和当前的目录状态。

线程的优点

- 线程和进程相比，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
- 运行于一个进程中的多线程，他们之间使用相同的地址空间，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的时间，据统计，一个进程的开销大约是一个线程开销的30倍左右。

线程的优点

- 多线程之间方便的通信机制。对不同进程来说，他们具有独立的数据空间，要进行数据的传递只能通过进程间通信的方式，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使用，不仅快捷而且方便。
- 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。
- 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，这样的程序会利于理解和修改。

线程的缺点

- 编写多线程程序需要非常仔细的设计(同步和互斥问题)
- 多线程程序的调试要比单线程程序困难的多，因为线程之间的交互很难控制

8.2 线程基本编程

- Linux系统下，多线程遵循POSIX线程接口，称为pthread，编写Linux下的多线程程序，需要使用头文件pthread.h,连接时需要使用库libpthread.a

Linux 线程的基本函数

- 常用线程函数
  - pthread_create 创建一个线程
  - pthread_exit 线程自行退出
  - pthread_join 其它线程等待某一个线程退出
  - pthread_cancel 其它线程强行杀死某一个线程

- pthread线程库的使用
  - glib库内置了线程库.
  - 在源码中使用头文件 pthread.h
  - 用gcc链接时加上 –lpthread 选项,链接线程库

pthread_create 创建一个线程

- 线程的创建
  - int pthread_create(pthread_t *thread,pthread_attr_t *attr, void *(*start_routine)(void *),void *arg);
    - pthread_create创建一个线程,thread是用来表明创建线程的ID,attr指出线程创建时候的属性,我们用NULL来表明使用缺省属性.start_routine函数指针是线程创建成功后开始执行的函数,arg是这个函数的唯一一个参数.表明传递给start_routine的参数。
    - 成功返回值为0，出错返回错误码。
  - 一个进程中的每个线程都由一个线程ID（thread ID）标识，其数据类型是pthread_t（常常是unsigned int）。如果新的线程创建成功，其ID将通过thread指针返回。

pthread_exit 退出一个线程

- 线程的退出有两种方式,一种线程函数运行结束,比如到函数结尾或用return退出.线程自然结束.这是最常用的方式.
- 调用pthread_exit退出.
  - void pthread_exit(void *retval);
    - 退出当前线程,并且设线程的返回值为retval
    - 返回值可以用pthread_join取得
- 需要注意一点：线程函数中不要使用exit()退出,会使整个进程终止。

pthread_join 等待线程退出

- int pthread_join(pthread_t *thread,void **thread_return);
  - 这个函数类似于waitpid,pthread_join是当前线程在等待另一个线程结束,只不过前者是在等待一个进程退出,后者在等待一个线程编号为thread的线程退出.
  - 当一个线程调用pthread_exit时,如果其它线程或进程使用了pthead_join在等待这个线程结束,那线程ID和退出状态将一直保留到pthread_join执行时.
  - pthread_exit()设定的返回值,或者return的返回值可以被pthread_join的thread_return捕获

pthread_cancel 杀死一个线程

- pthread_cancel(pthread_t thread) ;
  - 当前线程将杀死线程ID为thread的线程
  - pthread_exit()是当前线程自已退出,而pthread_cancel是其它线程杀死别的线程

关于线程的生存周期

- 程序的主进程默认为主线程.
- 一个子线程的生存周期由pthread_create 创建后开始,
- 一直到线程函数执行完毕,或者执行到pthread_exit处.线程的生命到此结束.
- 其它线程可以用pthread_cancel强制杀死另一线程.
- 主线程退出,(比如在主函数里调用exit),它的子线程无论是否执行完毕,都会随主线程退出而一同消失.
  - 所以在一般在主函数里,必须要判断的一下子线程是否真正退出,如是方可以把主线程退出.否则很可能造成程序结果不正确
  - 判断线程是否结束最简单的方法是在主线程的退出用pthread_join来等待子线程退出即可

举例：thread.c、thread1.c(多个线程)

修改线程的属性

- 绝大部分情况下,创建线程使用了默认参数，即将pthread_create函数的第二个参数设为NULL。
- 在特殊情况才需要设置线程属性pthread_attr_t,
  - 属性对象主要包括是否绑定、是否分离、堆栈地址、堆栈大小、优先级。默认的属性为非绑定、非分离、缺省1M的堆栈、与父进程同样级别的优先级。
- 初始化线程属性
  - 首先调用pthread_attr_init(&attr)初始化属性结构指针，这个函数必须在pthread_create函数之前调用。
  - 然后调用相应的属性设置函数
  - 最后调用pthread_attr_destroy(&attr)对分配的属性结构指针进行清理回收。

修改线程的属性

- 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。

修改线程的属性

- 设置线程绑定特性
  - pthread_attr_setscope
  - 设置线程绑定状态的函数为pthread_attr_setscope，它有两个参数，第一个是指向属性结构的指针，第二个是绑定类型，它有两个取值：PTHREAD_SCOPE_SYSTEM（绑定的）和PTHREAD_SCOPE_PROCESS（非绑定的）。
- 设置线程分离属性
  - pthread_attr_setdetachstate()
  - 它有两个参数，第一个是指向属性结构的指针，第二个是分离属性，它有两个取值：PTHREAD_CREATE_DETACHED（分离）、PTHREAD_CREATE_JOINABLE（非分离）

修改线程的属性

- 设置线程的优先级
  - 它存放在结构sched_param
  - 用函数pthread_attr_getschedparam和函数pthread_attr_setschedparam进行存放，一般说来，我们总是先取优先级，对取得的值修改后再存放回去

  举例：thread_attr.c

8.3 线程的并发访问

- 线程安全函数
  - 单线程程序只有一个控制流。不需要考虑一个资源（比如静态或全局变量如何处理）被同时访问或并发访问，但是多线程程序必须考虑并发访问一个资源。为了保证资源的完整性，为多线程程序写的代码必须是线程安全的。
    - 线程安全的(Thread-Safe)：如果一个函数在同一时刻可以被多个线程安全地调用，就称该函数是线程安全的。线程安全函数解决多个线程调用函数时访问共享资源的冲突问题。

线程安全函数

- 如果你的程序所在的进程中有多个线程在同时运行，而这些线程可能同时运行一段代码或同时访问一个对象，如果每次运行完这段代码或访问完这个对象之后，所得到的结果和单线程运行的结果一样，而其他变量的值也和预期的保持一致，那么就认为是线程安全的。
- 也就是说当多个线程同时运行同一段代码，不会造成资源的冲突，不会产生错误的结果就是线程安全的。
- 在C程序中，局部变量是在栈上分配的。因此，任何未使用静态数据或其他共享资源的函数都是线程安全的。
- 线程不安全的代码在多线程环境中必须作同步处理，否则会造成不可不可预期的后果。

多线程编程

- 多线程编程的主要问题
  - 是对共享数据的保护.即在多个线程同时访问同一个数据时,保证数据读写安全.
  - 线程安全函数除了尽量不使用静态或全局变量,另一个主要手段是加锁
- pthread 线程一般通过以下两种机制来完成数据的保护
  - 线程互斥锁 (pthread mutex)
  - Posix无名信号量

互斥锁线程控制

- 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。
- 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()
  - 互斥锁解锁：pthread_mutex_unlock()
  - 消除互斥锁：pthread_mutex_destroy()

线程互斥锁

- 互斥锁的创建

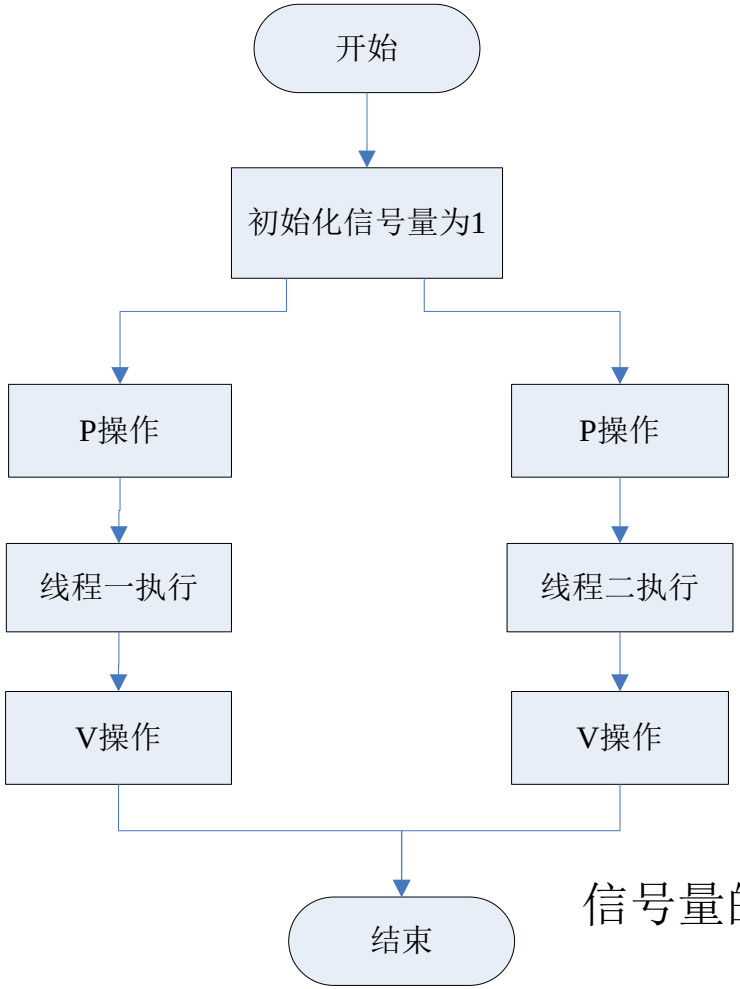
  int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)
- 互斥锁的销毁
  - int pthread_mutex_destroy(pthread_mutex_t *mutex)
- 锁操作
  - 锁操作主要包括加锁pthread_mutex_lock()、解锁pthread_mutex_unlock()和测试加锁pthread_mutex_trylock()三个，不论哪种类型的锁，都不可能被两个不同的线程同时得到，而必须等待解锁。
  - 在同一进程中的线程，如果加锁后没有解锁，则任何其他线程都无法再获得锁
- 这个锁机制同时也不是异步信号安全的，也就是说，不应该在信号处理过程中使用互斥锁，否则容易造成死锁。

举例：thread_mutex.c

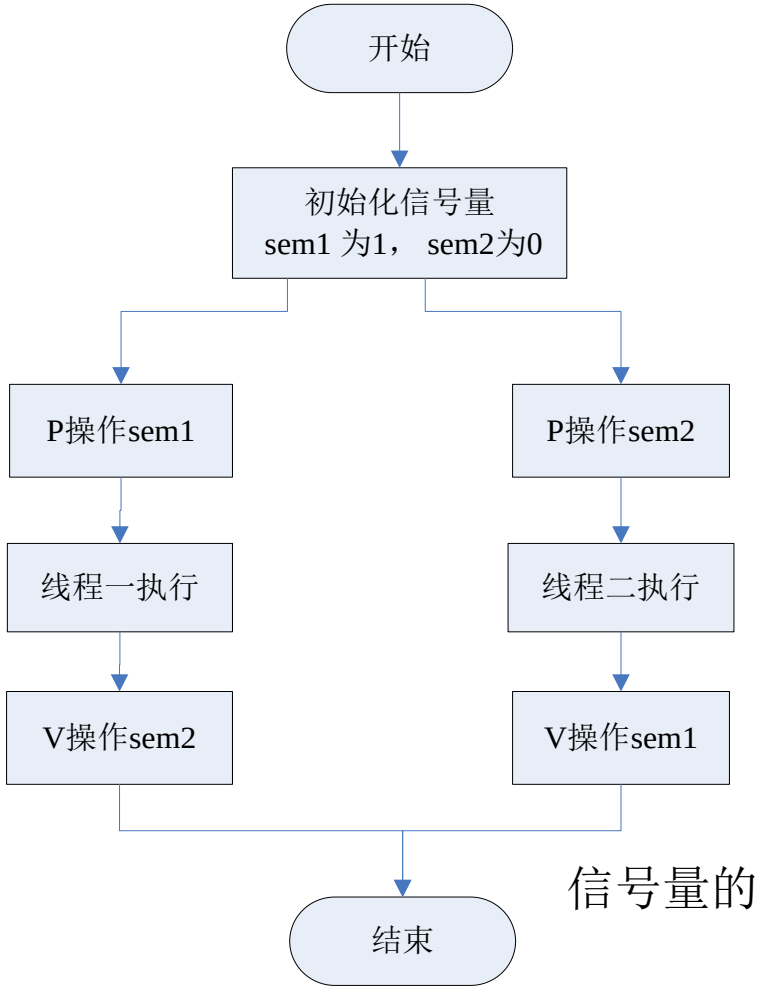
信号量线程控制

- 在前面已经讲到，信号量也就是操作系统中所用到的PV原子操作，它广泛用于进程或线程间的同步与互斥。信号量本质上是一个非负的整数计数器，它被用来控制对公共资源的访问。
- PV原子操作是对整数计数器信号量sem的操作。一次P操作使sem减1，而一次V操作使sem加1。进程（或线程）根据信号量的值来判断是否对公共资源具有访问权限。当信号量sem的值大于零时，该进程（或线程）具有公共资源的访问权限；相反，当信号量sem的值小于等于零时，该进程（或线程）就将阻塞直到信号量sem的值大于0为止。
- PV原子操作主要用于进程或线程间的同步和互斥这两种典型情况。若用于互斥，几个进程（或线程）往往只设置一个信号量sem。

信号量线程控制



信号量的互斥



信号量的同步

信号量线程控制

- 头文件 <semaphore.h>
- 创建信号量
  - int sem_init(sem_t *sem, int pshared, unsigned int value)
  - value为信号量的初值，pshared表示是否为多进程共享而不仅仅是用于一个进程。
  - LinuxThreads没有实现多进程共享信号量，因此所有非0值的pshared输入都将使sem_init()返回-1，且置errno为ENOSYS。
- 销毁信号量
  - sem_destroy(sem_t *sem)
- 获取信号量值
  - sem_getvalue(sem_t *sem)

信号量线程控制

- 申请一个信号资源
  - sem_wait(sem_t *sem)
  - 被用来阻塞当前线程直到信号量sem的值大于0，解除阻塞后将sem的值减一，表明公共资源经使用后减少。
- 释放一个信号资源
  - sem_post(sem_t *sem)
  - 用来增加信号量的值。当有线程阻塞在这个信号量上时，调用这个函数会使其中的一个线程不在阻塞
- 测试一个资源是否可用
  - sem_trywait (sem_t *sem)是

举例：thread_sem.c