**Linux基础理论学习**

—— 李宣廷

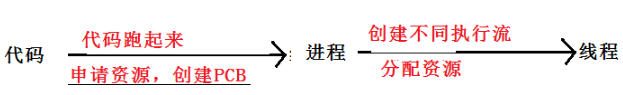
1. **线程学习**
2. 进程概念

程序从硬盘加载到内存开始运行时，进程就产生了。也就是操作系统开始为这个程序创建PCB，分配系统资源，比如分配一块虚拟地址空间，一个页表，一块物理内存。当这个进程内部有多个执行流时，现在我们可以简单理解父进程用vfork（）创建了多个子进程时，这些子进程也需要资源。如果能分得资源就会独自分的资源，但如果分不了就共享。共享一块地址空间。共享一个页表和一块物理内存。

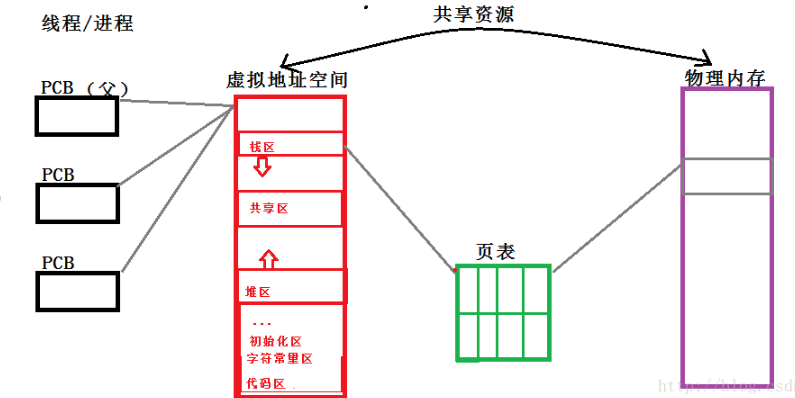
1. 线程概念

线程是一个进程内部的控制序列，是一个执行流。一个进程至少有一个线程。

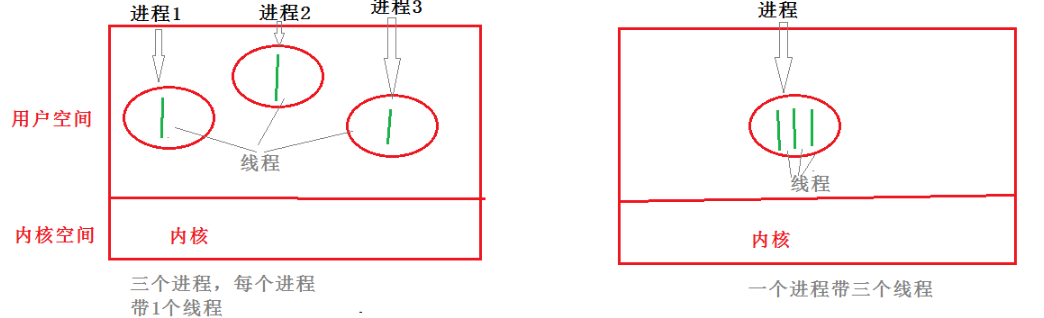
进程是承担分配系统资源的实体，线程共享进程所获得资源。



下图中有3个线程，他们是在进程里的3个执行控制流，共享一块虚拟地址，一块页表，所以它们的屋里内存也是一样的。但是可以执行各自的任务，并有自己生命特征。



下图分别为3个线程的不同存在方式，左图为三个进程，每个进程带1个线程；右图为一个进程带3个线程。



1. 线程的特点

**·线程是资源竞争的基本单位**

操作系统有很多资源。进程与进程之间要竞争操作系统资源，当一个进程申请得到一大堆资源。而这些资源又会分配给线程。一个进程内部有多个线程，去竞争进程所获得的资源。所以说线程是资源竞争的基本单位。

**·线程是程序执行的最小单位**

当用户让进程去执行某个任务时，进程又会将任务细化。进程内部有很多线程，让这些线程去执行

**·线程共享进程数据，但也拥有自己独立的一部分数据: 线程ID ，一组寄存器，栈，errno值，信号。**

其中最重要的数据是栈和寄存器。私有栈是为了保存临时变量，便于函数调用等操作。私有寄存器是为了方便线程切换，保存上下文。

**·进程的多个线程共享**

同地址空间，因此Test Segment,Data Segment都是共享的，如果定义个函数雇各线程中都可以调用，如果定义一个全局变量，在各线程中都可以访问到，除此之外，各线程还共享以下进程资源和环境：

1）文件描述符表

2）每种信号的处理方式(SIG\_IGN、SIG\_DFL或者自定义的信号处理函数）,也就是Hdanle表

3）当前工作目录

1. 进程和线程的联系

Linux下没有真正意义上的线程，它是由进程模拟的。进程是分配系统资源的一个实体，而线程是CPU调度的基本单位。在单个程序中同时运行多个线程完成不同的工作，称为多线程。多进程相对稳定，多线程相对不稳定。

进程有独立的地址空间，而同一进程里的所有线程共享进程的地址空间。

1. 线程的优缺点

**·优点**

1. 创建一个新线程的代价要比创建一个新进程小得多，释放成本也更低。
2. 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。
3. 线程占用的资源要比进程少很多。
4. 能充分利用多处理器的可并行数量。
5. 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。
6. 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。

**·缺点**

1. 性能损失
2. 健壮性降低
3. 缺乏访问控制
4. 编程难度提高
5. 线程控制

**POSIX线程库**

与线程有关的函数构成了一个完整的系列，绝⼤大多数函数的名字都是以“pthread\_”打头的，要使用这些函数库，要通过引入头文件， 链接这些线程函数库时要使用编译器命令的“-lpthread”选项。

**创建线程**

NAME

pthread\_create - create a new thread

SYNOPSIS

#include <pthread.h>

Int pthread\_cread ( pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*start routine) (void \*), void \*arg);

Compile and link with –pthread.

\*thread为线程ID；\*attr代表线程属性，置为NULL；void \*(\*start routine)决定新线程即将执行代码，返回void\*；\*arg为传递函数参数。

**错误检查**

返回值：成功返回0，失败返回错误码。

Pthreads函数出错时不会设置全局变量errno，而是将错误代码通过返回值返回，pthread提供了线程内的errno变量，支持其他使用errno代码。

**线程终止**

**只终止某个线程而不终止整个进程**

从线程函数return，这种方法对主线程不适用。从main函数retun相当于调用exit。

线程可以调用pthread\_exit终止自己。

**pthread\_exit函数**

功能：线程终止

#include <pthread.h>

void pthread\_exit(void \*retval);

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 exit或者return返回的指针所指向的内存单元必须是全局的或者是用malloc分配的,不能在线程函数的栈上分配,因为当其它线程得到这个返回指针时线程函数已经退出了。

调用phread\_exit（）,他会等待所有其他对等线程（就是同一个进程中的除自身外其他线程）终止，然后再终止主线程和整个进程。

如果某个对等线程调用linux的exit函数，则该函数终止进程以及所有与该进程相关的线程。

**pthread\_cancel函数**

功能：取消一个执行中的线程

#include <pthread.h>

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

**获取当前线程id的函数**

#include <pthread.h>

pthread\_t pthread\_self(void);

**线程等待** 线程等待的原因：已经退出的线程，其空间没有被释放，仍然在进程的地址空间内。创建新的线程不会复用刚才退出线程的地址空间。

功能：等待线程结束

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：成功返回0；失败返回错误代码

**线程分离**

在默认情况下，新线程默认joinable。线程退出后，需要对其进行pthread\_join操作来回收资源，否则无法释放资源，造成内存泄露。

如果不关心线程的返回值，join就是一种负担。线程提供一种自动回收的方法，叫做线程分离。当线程分离，退出后，自动释放线程资源。

**线程分离接口**

·线程组内其他线程对目标线程进行分离

#include <pthread.h>

int pthread\_detach(pthread\_t thread);

·线程自我分离

#include <pthread.h>

int pthread\_detach(pthread\_t self( ) );

返回值：都是成功返回0，失败返回-1

线程被分离后，就不能再进行pthread\_join()操作，否则会出错。因为分离后的线程资源自动就被回收了。

1. **信号量**
2. 信号量的概念

信号量是一种特殊的变量，访问具有原子性。

只允许对他进行两个操作：

1. 等待信号量

当信号量值为0时，程序等待；当信号量值大于0时，信号量减1，程序继续运行。

1. 发送信号量

将信号量加1。

1. 信号量的作用

使用信号量来解决进程或线程间共享资源引发的同步问题。信号量又称为信号灯，它是用来协调不同进程间的数据对象的，最主要的应用是共享内存方式的进程间通信。实质上，信号量是一个计数器，它同来记录对某个资源（如共享内存）的存取状况，信号量是一个特殊的变量，并且只有两个操作可以改变其值：等待（wait）与信号（signal）。因linux中wait和signal已经有特殊意义，所以：

用于等待的P（信号量变量）；

用于信号的V（信号量变量）；

**P操作：**负责把当前进程由运行状态换为阻塞状态，直到另一个进程唤醒它。

操作为：申请一个空闲资源（把信号量减1），若成功，则退出；若失败，则该进程被阻塞；

**V操作：**负责把当前被阻塞的进程唤醒，它有一个参数表，存放着等待被唤醒的进程信息。

操作为：释放一个被占用的资源（把信号量加1），如果发现有被阻塞的进程，则选择一个唤醒。

PS: 查看共享信息的内存命令是ipcs[-m|-s|-q]（全部是ipcs -a）。

**创建信号量**

int semget(key­\_t key, int nsems, int semflg);

key: 自定义一个整数，这个参数类似open函数的第一个参数，由调用者指定一个 “文件名”。由ftok（）生成。

nsems: 要初始化多少个信号量，通常设为1。

semflg: 这个参数也和open函数类似。支持权限和IPC\_CREAT以及IPC\_EXCL

IPC\_CREAT表示要创建一个信号量，但是如果信号量已经存在了也不会报错。

IPC\_EXCL和IPC\_CREAT一起使用时，如果信号量已经存在就会报EEXIST。

通常可以将semflg设为IPC\_CREAT\IPC\_EXCL\0666等

**信号量删除和初始化**

int semctl(int semid, int semnum, int cmd, …);

参数：

semid: 标识要操作那个信号量集

semnum： 初始化的是信号量集中的哪个信号量

cmd：设置为SETVAL表示需要进行初始化；设置为IPC\_RMID表示要删除的信号量集

…: 可变参数，一般为union semun的一个实例，注意删除的时候不用给出。

如果需要使用第四个参数时，第四个参数的类型为union，并且需要像以下一样定义出：

union semun{

int val; //使用的值

struct semid\_ds \*buf; //IPC\_STAT、IPC\_SET使用缓存区

unsigned short \*array; //GETTALL、SETALL使用数组

struct seminfo \*\_buf; //IPC\_INFO（Linux特有）使用缓存区

};

因为结构没有在头文件中声明，所以使用时要自己定义。

**信号量的操作**

int semop(int semid, struct sembuf \*sops, unsigned nsops); //这个操作是原子性的

参数：

semid：标识是哪个信号量集

nsops：标识是信号量集的哪个信号量

sops：在介绍这个参数时先对struct sembuf这个结构体进行介绍

信号量在system V版本中是以信号量集形式出现的，在内部就是以数组形式对信号进行保存的，也就是说这块这个sops其实是一个结构体数组，每个结构体中间保存了不同的信号量信息。

struct sembuf

{

unsigned short sem\_num; //表示是哪个信号量

short sem\_op; //对信号量的操作（P操作，这个值就是-1；V操作，这个值就是1）

short sem\_flg; //一般为0

}

注意：sem\_flg:信号操作标志，可能的选择有两种

IPC\_NOWAIT //对信号的操作不能满足时，semop()不会阻塞，并立即返回，同时设定错误信息。

SEM\_UNDO //程序结束时（不论正常不正常），保证信号值会被重设为semop()调用前的值。这样做的目的在于避免程序在异常情况下结束时未将锁定的资源解锁，造成该资源永远锁定。

1. **Linux下编程用到的锁**

线程之间的锁有：互斥锁、条件锁、自旋锁、读写锁、递归锁。一般而言，锁的功能越强大，性能就会越低。

1. 互斥锁

互斥锁用于控制多个线程对他们之间共享资源互斥访问的一个信号量。为了避免多个线程在某一时刻同时操作一个共享资源。例如线程池中有多个空闲线程和一个任务队列。任何一个线程都要使用互斥锁互斥访问任务队列，以避免多个线程同时访问任务队列以发生错乱。

在某一时刻，只有一个线程可以获取互斥锁，在释放互斥锁之前其他线程都不能获取该互斥锁。如果其他线程想要获取这个互斥锁，那么这个线程只能以阻塞方式进行等待。

头文件：<pthread.h>

类型： pthread\_mutex\_t

函数：

//动态方式创建锁，相当于new动态创建一个对象

pthread\_mutex\_init(pthread\_mutex \* mutex, const pthread\_mutexattr\_t \* mute)

//释放互斥锁，相当于delete

pthread\_mutex\_destory(pthread\_mutex\_t \*mutex)

//以静态方式创建锁

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

//以阻塞方式运行的。如果之前mutex被加锁了，那么程序会阻塞在这里

pthread\_mutex\_lock(pthread\_mutex\_t \*mutex)

pthread\_mutex\_unlock(pthread\_mutex \*mutex)

//会尝试对mutex加锁。如果mutex之前已经被锁定，返回非0；如果mutex没有被锁定，则函数返回并锁定mutex

//该函数是以非阻塞方式运行了。也就是说如果mutex之前已经被锁定，函数会返回非0，程序继续往下执行。

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);

1. 条件锁

条件锁就是所谓的条件变量，某一个线程因为某个条件未满足时可以使用条件变量使该程序处于阻塞状态，当条件满足时以“信号量”的方式唤醒一个因为该条件而被阻塞的线程。最为常见就是在线程池中，起初没有任务时任务队列为空，此时线程池中的线程因为“任务队列为空”这个条件处于阻塞状态。一旦有任务进来，就会以信号量的方式唤醒一个线程来处理这个任务。这个过程中就使用到了条件变量pthread\_cond\_t

头文件：<pthread.h>

类型：pthread\_cond\_t

//对条件变量进行动态初始化，相当于new创建对象

pthread\_cond\_init(pthread\_cond\_t \* condtion, const phtread\_condattr\_t \* condattr);

//释放动态申请的条件变量，相当于delete释放对象

pthread\_cond\_destory(pthread\_cond\_t \* condition);

//静态初始化

pthread\_cond\_t condition = PTHREAD\_COND\_INITIALIZER;

//该函数以阻塞方式执行。如果某个线程中的程序执行了该函数，那么这个线程就会以阻塞方式等待，直到收到pthread\_cond\_signal或者pthread\_cond\_broadcast函数发来的信号而被唤醒。

pthread\_cond\_wait(pthread\_cond\_t \* cond, pthread\_mutex\_t \* mutex);

注意：pthread\_cond\_wait函数的语义相当于：首先解锁互斥锁，然后以阻塞方式等待条件变量的信号，收到信号后又会对互斥锁加锁。为了防止“虚假唤醒”，该函数一般放在while循环体中。例如

pthread\_mutex\_lock(mutex);//加互斥锁

while(条件不成立)//当前线程中条件变量不成立

{

pthread\_cond\_wait(cond, mutex);//解锁，其他线程使条件成立发送信号，加锁。

}

...//对进程之间的共享资源进行操作

pthread\_mutex\_unlock(mutex);//释放互斥锁

//在另外一个线程中改变线程，条件满足发送信号。唤醒一个等待的线程（可能有多个线程处于阻塞状态），唤醒哪个线程由具体的线程调度策略决定

pthread\_cond\_signal(pthread\_cond\_t \* cond);

//以广播形式唤醒所有因为该条件变量而阻塞的所有线程，唤醒哪个线程由具体的线程调度策略决定

pthread\_cond\_broadcast(pthread\_cond\_t \* cond);

//以阻塞方式等待，如果时间time到了条件还没有满足还是会结束

pthread\_cond\_timedwait(pthread\_cond\_t \* cond, pthread\_mutex\_t \* mutex, struct timespec \* time);

1. 自旋锁

假设有一台两个处理器core1和core2计算机，现在在这台计算机上运行的程序中有两个线程：T1和T2分别在处理器core1和core2上运行，两个线程之间共享着一个资源。

互斥锁是一种sleep-waiting的锁。假设线程T1获取互斥锁并且正在core1上运行时，此时线程T2也想要获取互斥锁（pthread\_mutex\_lock），但是由于T1正在使用互斥锁使得T2被阻塞。当T2处于阻塞状态时，T2被放入到等待队列中去，处理器core2会去处理其他任务而不必一直等待（忙等）。也就是说处理器不会因为线程阻塞而空闲着，它去处理其他事务去了。

自旋锁是一种busy-waiting的锁。也就是说，如果T1正在使用自旋锁，而T2也去申请这个自旋锁，此时T2肯定得不到这个自旋锁。与互斥锁相反的是，此时运行T2的处理器core2会一直不断地循环检查锁是否可用（自旋锁请求），直到获取到这个自旋锁为止。从“自旋锁”的名字也可以看出来，如果一个线程想要获取一个被使用的自旋锁，那么它会一致占用CPU请求这个自旋锁使得CPU不能去做其他的事情，直到获取这个锁为止，这就是“自旋”的含义。

当发生阻塞时，互斥锁可以让CPU去处理其他的任务；而自旋锁让CPU一直不断循环请求获取这个锁。通过两个含义的对比可以我们知道“自旋锁”是比较耗费CPU的。

头文件：<linux\spinlock.h>

自旋锁的类型：spinlock\_t

相关函数：初始化：spin\_lock\_init(spinlock\_t \*x);

//只有在获得锁的情况下才返回，否则一直“自旋”

spin\_lock(x);

//如立即获得锁则返回真，否则立即返回假

spin\_trylock(x);

//释放锁：

spin\_unlock(x);

//该宏用于判断自旋锁x是否已经被某执行单元保持（即被锁），如果是，返回真，否则返回假

spin\_is\_locked(x)

注意：自旋锁适合于短时间的的轻量级的加锁机制

1. 读写锁

计算机中某些数据被多个进程共享，对数据库的操作有两种：一种是读操作，就是从数据库中读取数据不会修改数据库中内容；另一种就是写操作，写操作会修改数据库中存放的数据。因此可以得到我们允许在数据库上同时执行多个“读”操作，但是某一时刻只能在数据库上有一个“写”操作来更新数据。这就是一个简单的读者-写者模型。

1. **网络编程**
2. TCP/IP协议概述

协议protocol: 通信双方必须遵循的规矩，由osi规定。

osi参考模型: （应用层-表示层-会话层-传输层-网络层-数据链路层-物理层）

tcp/ip模型4层：

应用层：http超文本传输协议、ftp文件传输协议、telnet远程登录、ssh安全外壳协议、stmp简单邮件发送、pop3收邮件

传输层：tcp传输控制协议、udp用户数据包协议

网络层：ip网际互联协议、icmp网络控制消息协议、igmp网络组管理协议

网络接口层：arp地址转换协议、rarp反向地址转换协议、mpls多协议标签交换

tcp协议：传输控制协议，面向连接的协议，能保证传输安全可靠，速度慢（3次握手）

udp协议：用户数据包协议，非面向连接，速度快，不可靠

通常是ip地址后面跟上端口号：ip用来定位主机，port区别应用（进程）  
http的端口号80 ssh🡪22 tenet🡪23 tfp🡪21 用户自己定义的通常要大于1024

TCP/IP协议族的每一层的作用：

网络接口层：负责将二进制流转换为数据帧，并运行数据帧的发送和接收。数据帧是独立的网络信息传输单元。

网络层：负责将数据帧封装成IP数据包，并运行必要的路由算法。

传输层：负责端对端之间的通信会话连接和建立。传输协议的选择根据数据传输方式而定。

应用层：负责应用程序的网络访问，这里通过端口号来识别各个不同的进程。

TCP/IP协议族的每一层协议的相关注解：

ARP：（地址转换协议）用于获得同一物理网络中的硬件主机地址。

MPLS：（多协议标签转换）很有发展前景的下一代网络协议。

IP：（国际互联协议）负责在主机和网络之间寻址和路由数据包

ICMP：（网络控制消息协议）用于发送报告有关数据包的传送错误的协议

IGMP：（网络组管理协议）被IP主机用来向本地多路广播路由器报告主机组成员的协议。

TCP：（传输控制协议）为应用程序提供可靠的通信连接。适合于一次传输大批数据的情况，并适用于要求得到相应的应用程序。

UDP：（用户数据包协议）提供了无连接通信，且不对传输包进行可靠的保证。适合于一次传输少量数据。

1. 网络相关概念

**“套接字”**是操作系统内核中的一个数据结构，它是网络节点间进行通信的门户，是网络进程的ID。网络通信，归根结底还是进程间的通信（不同计算机上进程间通信）。两个进程通信时，首先要确定各自所在网络节点的网络地址，然后确定端口号。在一台计算机中，一个端口号一次只能分配给一个进程，端口号和进程之间是一一对应关系。所以，使用端口号和网络地址的组合可以唯一的确定整个网络中的一个网络进程。（例如：网络中的某台计算机的IP为192.168.13.192 某一个应用进程的端口号为1500，则此时192.168.13.192 1500就构成了一个套接字）

**端口号**：http:80， ftp：21， ssh：22， telnet：23

用户自定义的通常是大于1024的整型值。

**socket概念**

Linux中的网络编程是通过socket接口来进行的。socket是一种特殊的I/O接口，也是一种文件描述符，在进程之间的通信机制。通过它不仅能实现本地机器上的进程之间的通信，而且通过网络能够在不同机器上的进程之间进行通信。

每一个socket都用一个半相关描述{协议、本地地址、本地端口}来表示；一个完整的套接字则用一个相关描述{协议、本地地址、本地端口、远程地址、远程端口}来表示。socket也有一个类似于打开文件的函数调用，该函数返回一个整型的socket描述符，随后的连接建立、数据传输等操作都是通过socket来实现的。

**socket类型**

* 1. 流式socket（SOCK\_STRESM）用于TCP通信

流式套接字提供可靠的、面向连接的通信流；它可以使用于TCP协议，从而保证了数据传输的正确性和顺序性。

* 1. 数据报socket（SOCK\_DGRAM）用于UDP通信

数据报套接字定义了一种无连接的服务，数据通过相互独立的报文进行传输，是无序的，并且不保证是可靠、无差错的。它使用数据报协议UDP。

3）原始socket（SOCK\_RAW）用于新的网络协议实现的测试等

原始套接字允许对底层协议如IP或ICMP进行直接访问，它功能强大但使用较为不便，主要用于一些协议的开发。

**socket信息数据结构**

struct sockaddr

{

unsigned short sa\_family; /\*地址族\*/

char sa\_data[14]; /\*14字节的协议地址，包含该socket的IP地址和端口号。\*/

};

struct sockaddr\_in

{

short int sa\_family; /\*地址族\*/

unsigned short int sin\_port; /\*端口号\*/

struct in\_addr sin\_addr; /\*IP地址\*/

unsigned char sin\_zero[8]; /\*填充0 以保持与struct sockaddr同样大小\*/

};

struct in\_addr

{

unsigned long int s\_addr; /\* 32位IPv4地址，网络字节序 \*/

};

头文件<netinet/in.h>

sa\_family:AF\_INET IPv4协议 AF\_INET6 IPv6协议

**数据存储优先顺序转换**

某个系统中所采用的字节序为主机字节序，则它的大小端模式不能确定。而端口号和IP地址都是以网络字节序存储的，为大端模式。要把主机字节序和网络字节序相互对应起来，需要对这两个字节存储优先顺序进行相互转化。这里用四个函数：htons（），ntohs（），htonl（）和ntohl（）．　这四个地址分别实现网络字节序和主机字节序的转化，这里的h代表host，n代表network，s代表short，l代表long。通常16位的IP端口号用s代表，而IP地址用l表示。从字面意思可以明白每个函数作用。

**地址格式转化**

通常用户在表达地址时采用的是点分十进制表示数值（或者是为冒号分开的十进制Ipv6地址），而在通常使用的socket编程中使用的则是32位的网络字节序二进制值，这就需要将这两个数值进行转换。这里在Ipv4中用到的函数有inet\_aton()、inet\_addr()和inet\_ntoa(),而IPV4和IPV6兼容的函数有inet\_pton（）和inet\_ntop()。

IPv4的函数原型：

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

int inet\_aton(const char \*straddr, struct in\_addr \*addrptr);

char \*inet\_ntoa(struct in\_addr inaddr);

in\_addr\_t inet\_addr(const char \*straddr);

函数inet\_aton()：将点分十进制数的IP地址转换成为网络字节序的32位二进制数值。返回值：成功，则返回1，不成功返回0.

参数straddr：存放输入的点分十进制数IP地址字符串。

参数addrptr：传出参数，保存网络字节序的32位二进制数值。

函数inet\_ntoa()：将网络字节序的32位二进制数值转换为点分十进制的IP地址。

函数inet\_addr()：功能与inet\_aton相同，但是结果传递的方式不同。inet\_addr()若成功则返回32位二进制的网络字节序地址。

IPv4和IPv6的函数原型：

#include <arpa/inet.h>

int inet\_pton(int family, const char \*src, void \*dst);

const char \*inet\_ntop(int family, const void \*src, char \*dst, socklen\_t len);

函数inet\_pton跟inet\_aton实现的功能类似，只是多了family参数，该参数指定为AF\_INET，表示是IPv4协议，如果是AF\_INET6，表示IPv6协议。

函数inet\_ntop跟inet\_ntoa类似，其中len表示表示转换之后的长度（字符串的长度）

Example：

#include <stdio.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

int main()

{

char ip[] = "192.168.0.101";

struct in\_addr myaddr;

/\* inet\_aton \*/

int iRet = inet\_aton(ip, &myaddr);

printf("%x\n", myaddr.s\_addr);

/\* inet\_addr \*/

printf("%x\n", inet\_addr(ip));

/\* inet\_pton \*/

iRet = inet\_pton(AF\_INET, ip, &myaddr);

printf("%x\n", myaddr.s\_addr);

myaddr.s\_addr = 0xac100ac4;

/\* inet\_ntoa \*/

printf("%s\n", inet\_ntoa(myaddr));

/\* inet\_ntop \*/

inet\_ntop(AF\_INET, &myaddr, ip, 16);

puts(ip);

return 0;

}　

**名字地址转化**

主机名与域名的区别：主机名通常在局域网里面使用，通过/etc/hosts文件，主机名可以解析到对应的ip；域名通常是在internet上使用。

gethostbyname()是将主机名转化为IP地址，gethostbyaddr()则是将IP地址转化为主机名。

函数原型：

#include <netdb.h>

struct hostent\* gethostbyname(const char\* hostname);

struct hostent\* gethostbyaddr(const char\* addr, size\_t len, int family);

结构体：

struct hostent

{

char \*h\_name; /\*正式主机名\*/

char \*\*h\_aliases; /\*主机别名\*/

int h\_addrtype; /\*主机IP地址类型 IPv4为AF\_INET\*/

int h\_length; /\*主机IP地址字节长度，对于IPv4是4字节，即32位\*/

char \*\*h\_addr\_list; /\*主机的IP地址列表\*/

}

#define h\_addr h\_addr\_list[0] /\*保存的是ip地址\*/

函数gethostbyname()：用于将域名（www.baidu.com）或主机名转换为IP地址。参数hostname指向存放域名或主机名的字符串。

函数gethostbyaddr()：用于将IP地址转换为域名或主机名。参数addr是一个IP地址，此时这个ip地址不是普通的字符串，而是要通过函数inet\_aton()转换。len为IP地址的长度，AF\_INET为4。family可用AF\_INET：Ipv4或AF\_INET6：Ipv6。

Example1：将百度的www.baidu.com 转换为ip地址

#include <netdb.h>

#include <sys/socket.h>

#include <stdio.h>

int main(int argc, char \*\*argv)

{

char \*ptr, \*\*pptr;

struct hostent \*hptr;

char str[32] = {'\0'};

/\* 取得命令后第一个参数，即要解析的域名或主机名 \*/

ptr = argv[1]; //如www.baidu.com

/\* 调用gethostbyname()。结果存在hptr结构中 \*/

if((hptr = gethostbyname(ptr)) == NULL)

{

printf(" gethostbyname error for host:%s\n", ptr);

return 0;

}

/\* 将主机的规范名打出来 \*/

printf("official hostname:%s\n", hptr->h\_name);

/\* 主机可能有多个别名，将所有别名分别打出来 \*/

for(pptr = hptr->h\_aliases; \*pptr != NULL; pptr++)

printf(" alias:%s\n", \*pptr);

/\* 根据地址类型，将地址打出来 \*/

switch(hptr->h\_addrtype)

{

case AF\_INET:

case AF\_INET6:

pptr = hptr->h\_addr\_list;

/\* 将刚才得到的所有地址都打出来。其中调用了inet\_ntop()函数 \*/

for(; \*pptr!=NULL; pptr++)

printf(" address:%s\n", inet\_ntop(hptr->h\_addrtype, \*pptr, str, sizeof(str)));

printf(" first address: %s\n", inet\_ntop(hptr->h\_addrtype, hptr->h\_addr, str, sizeof(str)));

break;

default:

printf("unknown address type\n");

break;

}

return 0;

}

编译运行

#gcc test.c

#./a.out www.baidu.com

official hostname:www.a.shifen.com

alias:www.baidu.com

address: 220.181.111.148

……

first address: 220.181.111.148

（注：这里需要联网才能访问www.baidu.com。可以尝试用自己的虚拟机的主机名，利用命令hostname可以查看自己的主机名，用hostname –i可以查看主机名对应的ip地址。那么如何修改主机名呢？直接用hostname wangxiao只是暂时有效，重启之后就没有了，想要永久有效，需要修改/etc/sysconfig/network将HOSTNAME修改，当然要重启虚拟机。如果ip地址不对，你可以修改/etc/hosts这个文件，添加你自己的主机名对应的ip地址）

1. socket编程

**使用TCP协议的流程图**

TCP通信的基本步骤如下：

服务端：socket---blind---listen---while（1）{---accept---recv---send---close---}---close

客户端：socket-------------------------------------connect---send---recv---------------close

服务器端：

1. 头文件包含：

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <unistd.h>

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

1. socket函数：生成一个套接口描述符

原型：int socket(int domain, int type, int protocol);

参数：domain{AF\_INET：Ipv4网络协议 AF\_INET6：Ipv6网络协议}

type{tcp：SOCK\_STREAM udp：SOCK\_DGRAM}

protocol指定socket所使用的传输协议编号，通常是0.

返回值：成功返回套接口描述符，失败返回-1.

常用实例：int sfd=socket(AF\_INET,SOCK\_STREAM,0);

if(sfd==-1){perror(“socket”);exit(-1);}

1. blind函数：用来绑定一个端口号和IP地址，使套接口与指定的端口号和IP地址相关联。

原型：int blind(int sockfd,struct sockaddr \*my\_addr, int addrlen);

参数：sockfd为前面socket的返回值。

my\_addr为结构体指针变量

对于不同的socket domain定义了一个通用的数据结构体

struct sockaddr //此结构体不常用

{

unsigned short int sa\_family; //调用socket()时的domain参数，即AF\_INET值

char sa\_data[14]; //最多使用14个字符长度

} //此sockaddr结构会因使用不同的socket domain而有不同结构定义，例如使用AF\_INET domain，其socketaddr结构定义便为

struct sockaddr\_in //常用的结构体

{ unsigned short int sin\_family; //即为sa\_family AF\_INET

uint16\_t sin\_port; //为使用的port编号

struct in\_addr sin\_addr; //为IP 地址

unsigned char sin\_zero[8]; //未使用

};

struct in\_addr { uint32\_t s\_addr; };

addrlen sockaddr的结构体长度。通常是计算sizeof(struct sockaddr);

返回值：成功返回0，失败返回-1

常用实例：struct sockaddr\_in my\_addr; //定义结构体变量

memset(&my\_addr,0,sizeof(struct socketaddr)); //将结构体清空

my\_addr.sin\_family = AF\_INET; //表示采用Ipv4网络协议

my\_addr.sin\_port = htons(8888); //表示端口号为8888，通常是大于1024的一个值。

//htons()用来将参数指定的16位hostshort转换成网络字符顺序

my\_addr.sin\_addr.s\_addr = inet\_addr("192.168.0.101"); // inet\_addr()用来将IP地址字符串转换成网络所使用的二进制数字，如果为INADDR\_ANY，这表示服务器自动填充本机IP地址。

if(bind(sfd, (struct sockaddr\*)&my\_str, sizeof(struct socketaddr)) == -1)

{perror("bind");close(sfd);exit(-1);}

（注：通过将my\_addr.sin\_port置为0，函数会自动为你选择一个未占用的端口来使用。同样，通过将my\_addr.sin\_addr.s\_addr置为INADDR\_ANY，系统会自动填入本机IP地址。）

1. listen函数

使服务器的这个端口和IP处于监听状态，等待网络中某一客户机的连接请求。如果客户端有连接请求，端口就会接受这个连接。

原型：int listen(int sockfd, int backlog);

参数：sockfd为前面socket的返回值。即sfd

backlog指定同时能处理的最大连接请求，通常位10或者5，最大可设至128

返回值：成功则返回0，失败则返回-1

常用实例：if(listen(sfd,10) == -1)

{perror(“listen”);close(sfd);exit(-1);}

1. accept函数

接受远程计算机的连接请求，建立起与客户机之间的通信连接。服务器处于监听状态时，如果某时刻获得客户的连接请求，此时并不是立即处理这个请求，而是将这个请求放在等待队列中，当系统空闲时再处理客户机的连接请求。当accept函数接受一个连接时，会返回一个新的socket标识符，以后的数据传输和读取就要通过这个新的socket编号来处理，原来参数中的socket也可以继续使用，继续监听其它客户机的连接请求。

原型：int accept(int s,struct sockaddr \* addr, int \* addrlen);

参数：s为前面socket的返回值。即sfd

addrà为结构体指针变量，和bind的结构体是同种类型的，系统会把远程主机的信息（远程主机的地址和端口号信息）保存到这个指针所指的结构体中。

addrlen表示结构体的长度，为整型指针

返回值：成功则返回新的socket处理代码new\_fd，失败返回-1

常用实例：struct sockaddr\_in clientaddr;

memset(&clientaddr, 0, sizeof(struct sockaddr));

int addrlen = sizeof(struct sockaddr);

int new\_fd = accept(sfd, (struct sockaddr\*)&clientaddr, &addrlen);

if(new\_fd == -1)

{perror("accept");close(sfd);exit(-1);}

printf("%s %d success connect\n", inet\_ntoa(clientaddr.sin\_addr), ntohs(clientaddr.sin\_port))

1. recv函数

用新的套接字来接收远端主机传来的数据，并把数据存到由参数buf 指向的内存空间。

原型：int recv(int sockfd,void \*buf,int len,unsigned int flags);

参数：sockfdà为前面accept的返回值.即new\_fd，也就是新的套接字。

buf表示缓冲区

len表示缓冲区的长度

flags通常为0

返回值：成功则返回实际接收到的字符数，可能会少于你所指定的接收长度。失败返回-1

常用实例：char buf[512] = {0};

if(recv(new\_fd, buf, sizeof(buf), 0) == -1)

{perror("recv");close(new\_fd);close(sfd);exit(-1);}

puts(buf);

1. send函数

用新的套接字发送数据给指定的远端主机

原型：int send(int s,const void \* msg,int len,unsigned int flags);

参数：为前面accept的返回值.即new\_fd

msg一般为常量字符串

len表示长度

flags通常为0

返回值：成功则返回实际传送出去的字符数，可能会少于你所指定的发送长度。失败返回-1

常用实例：if(send(new\_fd, "hello", 6, 0) == -1)

{perror("send");close(new\_fd);close(sfd);exit(-1);}

1. close函数

当使用完文件后若已不再需要则可使用close()关闭该文件，并且close()会让数据写回磁盘，并释放该文件所占用的资源

原型：int close(int fd);

参数：fdà为前面的sfd,new\_fd

返回值：若文件顺利关闭则返回0，发生错误时返回-1

常用实例：close(new\_fd);

close(sfd);

**客户端**

connect函数：用来请求连接远程服务器，将参数sockfd 的socket 连至参数serv\_addr 指定的服务器IP和端口号上去。

原型：int connect (int sockfd,struct sockaddr \* serv\_addr,int addrlen);

参数：sockfd为前面socket的返回值，即sfd

serv\_addr为结构体指针变量，存储着远程服务器的IP与端口号信息。

addrlen表示结构体变量的长度

返回值：成功则返回0，失败返回-1

常用实例：

struct sockaddr\_in seraddr;//请求连接服务器

memset(&seraddr, 0, sizeof(struct sockaddr));

seraddr.sin\_family = AF\_INET;

seraddr.sin\_port = htons(8888); //服务器的端口号

seraddr.sin\_addr.s\_addr = inet\_addr("192.168.0.101"); //服务器的ip

if(connect(sfd, (struct sockaddr\*)&seraddr, sizeof(struct sockaddr)) == -1)

{perror("connect");close(sfd);exit(-1);}

还可以不写客户端程序，使用telnet远程登录来检测我们的服务器端程序。比如我们的服务器程序在监听8888端口，我们可以用telnet 192.168.0.101 8888来查看服务端的状况。