文件指针位置被谁改变？

- open 时
  - 如果设定了 O_APPEND，设置为末尾，并且以后在每次写文件时，指针位置都被自动原子的移动到文件尾
  - 如果未设置 O_APPEND，文件指针设置为文件开头位置（绝对偏移为 0 的位置）
- close() 时
  - close() 调用与文件指针移动无关
- read()
  - 成功时，文件指针位置向文件末尾方向移动实际读取的字节数
- write()
  - 成功时，文件指针位置向文件末尾方向移动实际写的字节数
- lseek()
  - 根据需要显式的移动文件读写指针的位置

文件描述符的复制

- 使用 fcntl(F_DUPFD)
  - 得到的描述符不小于传入的 arg 参数值（结果是大于或者等于 arg 参数的可用的最小描述符）
- 使用 dup()/dup2()
  - dup() 得到的描述符是顺序分配的可用的最小的描述符，跟 open() 调用得到的描述符的分配机制是相同的
  - dup2() 得到的是指定的描述符
- 通过 fork() 实现描述符的复制
  - 在子进程中，得到的与父进程值相等的描述符

共享的问题

- File descriptor flags 在复制的文件描述符之间不共享，也就是说对某个描述符修改了该标志，另一个描述符不受影响
- File status flags 在复制的文件描述符之间共享，其本质是复制的文件描述符指向同一个文件描述信息 (open file description)
- 文件的读写位置指针（ File offset， position) 在复制的文件描述符之间是共享的

stat()/fstat()/lstat() 规律

- stat() 也就是不是以“ *f* ” 或者“ *l* ” 开头的函数，使用的时文件名作为第一个参数，如果 stat() 传入的第一个参数是一个符号连接，那么 stat() 最终处理的是符号连接最终指向的文件
  - stat() 会跟随符号连接
- fstat() ，也就是以“ *f* ” 开头的函数，使用的是文件描述符作为第一个参数
- lstat() ，也就是以“ *l* ” 开头的函数，处理的是符号连接本身
  - lstat() 不跟随符号连接
- 符合该规律的还有如下调用
  - chmod()/fchmod()
  - chown()/fchown()/lchown()
  - truncate()/ftruncate()
  - chdir()/fchdir()

字符串处理是基本功

- 字符串连接(把两个字符串接起来)
  - strcat()
- 字符串拷贝
  - strcpy()/strncpy()/memcpy()
  - strdup()可以用于字符串复制
- 字符串分割
  - strtok(),
  - strsep()
  - 还可以使用状态机编程思想处理字符串)
- 字符串内查找子串
  - strstr()
  - 最好了解一下正则表达式
  - 如何想办法提高效率？
- 字符串解析，把需要的内容提取出来，如何提高效率？
- 字符串格式化
  - snprintf()

文件系统相关的命令

- 使用 fdisk 对磁盘进行分区
- 使用 mkfs 创建文件系统 (类似于 Windows/Dos 下的格式化 format)
  - mkfs 的参数 -t 指定文件系统类型
- e2label 可以对文件系统设置 label
- dumpe2fs 可以查看文件系统信息
- debugfs 可以不通过内核来对文件系统操作
- fsck 可以对磁盘进行检查

系统的文件组织

- 系统存储文件时，文件名和文件的实体(inode)是分开存储的
- 文件名存储在所在目录的目录文件中(没错：目录也是一个文件)，通常给目录分配一个块，如果一个块不够用，则继续分配一个块，所以目录的大小是块大小的整数倍。
- 文件的实体(实质上就是inode)存在其他的地方，存储在文件系统中，比如EXT2文件系统，在用户空间一般来说，无法通过inode编号访问到一个文件，只能通过文件名找到对应的inode，然后访问该inode，获取或者修改文件内容和文件属性。
- 文件名和inode之间的映射关系就是一个硬连接，存储在目录文件中
- 可以有多个文件名指向同一个inode这种情况，那么这些文件名之间互为硬连接
- 如果有兴趣的话，可以去看EXT2文件系统的组织和VFS，其中通过文件名找对应的inode的过程是使用dcache来实现的。

权限？

- 中文的“权限”是有歧义的
  - 可以表示为许可权 (permission)
  - 可以表示为优先权 (privilege)
  - 可以表示为权利 (right)
- 在我们进行文件许可权检查的时候，权限的意思是 permission，最好翻译为“许可权”

系统如何做权限检查？

- 每个进程都有real uid, real gid, effective uid, effective gid, 添加组ID
- 添加组ID怎么来？
  - 某个用户可以属于多个组，其中有一个是默认组，用户所属的默认组在/etc/passwd中设置
  - 在/etc/group文件里设置的是用户所属的其他组
  - 该用户启动一个程序的时候，其他组就变成了进程的添加组
- 进程的实际用户ID从哪儿来？谁启动程序，谁就是进程的实际用户ID
- 进程的实际组ID从哪儿来？谁启动程序，该用户所属的组ID就是进程的实际组ID
- 通常来说，进程的有效用户ID等于进程的实际用户ID，进程的有效组ID等于进程的实际组ID，进程的添加组ID从用户所属的组中取得。
- 但是，如果程序本身是set-user-id的，那么该程序启动为进程之后，就将进程的有效用户ID变更为程序文件的所有者ID
- 如果程序本身是set-group-id的，那么该程序启动为进程之后，就将进程的有效组ID变更为程序文件的组ID
- 当进程欲操作一个文件时，按照如下的规则去检查permission
  - Step 1，检查进程的有效用户ID是否等于被操作文件的owner id，如果等于的话，进程就拥有owner的permission(rwx------)，后续的检查不做了
  - Step 2，如果进程的有效用户ID不等于被操作文件的owner id，那么检查进程的有效组ID，如果进程的有效组ID等于文件的group id，那么，进程将拥有文件group的permission
  - Step 3，如果不符合step 1，也不符合step 2，那么依次检查进程的添加组ID，如果某个添加组(一个进程可以有0个或多个添加组ID)ID等于文件的group，则进程拥有文件group的permission
  - 最后，上述都不满足时，进程将拥有文件的other的permission

ACL 是更好的权限机制

- UNIX 的权限机制比较初级，比较高级的权限机制是 ACL(access control list)
- sudo 就使用类似 ACL 的机制

chmod() 执行的条件

- 如果进程的有效用户 ID 等于文件的 owner，才可以做 chmod()，否则就不可以做 chmod()
- 如果被操作文件的 group 不等于进程的有效组 ID，也不等于进程的任何一个添加组 ID，并且进程是非特权的，那么试图去对文件设置 set-group-id 标志，则该标志会被屏蔽掉（也就是说在这种情况下该标志不允许设置）。

chown() 执行的条件

- 只有特权进程才可以改变文件的 owner，通常来说时 root 用户进程改变文件的 owner。
- 文件的 owner 所启动的进程 (确切的应该是进程的有效用户 ID 等于文件的 owner 时) 可以改变文件的组所有者关系，将文件的组所有者在 owner 所属的组之间改变。
- 假设 x 用户属于以下组：
  - stuff
  - manager
  - hr
- 那么如果某个一个 x 用户所拥有的文件 (即该文件的 owner=x)，可以将该文件的组在 stuff/manager/hr 之间变换。
- 特权进程可以改变文件的组为任意的组 ID。
- 如果非特权用户改变了文件的 owner 或者 group，那么文件原有的 set-uid-bit 和 set-gid-bit 会被清除掉。

与进程有关的限制问题

- Resource limit，可以通过 getrlimit()/setrlimit() 取得和设置，还可以通过一些命令来进行调整，比如 ulimit 命令。
- sysconf() 可以取得限制值
- /proc 目录下某些参数也可能会影响到某些限制，如果需要提高限制值，可能需要调整 /proc 参数
- 上述参数之间具体是什么关系，没有明确的说明，对于程序员来说，具体问题具体分析。
- 有些限制值可以通过 sysctl 命令来进行调整， sysctl 是系统优化时需要使用的命令。
- 在有些情况下，甚至需要重新编译内核来调整上限。

Shell 如何启动一个程序？

- Shell 会等待用户输入
- 如果用户在 shell 下输入某些命令，则 Shell 进行解析，解析的结果有：
  - 几个命令？（ $ ls -l | grep “abc” > a.file ）
  - 命令的参数有哪些？
  - 是否有重定向
- Shell 会调用 fork() 产生需要数量的子进程（几个命令就产生几个子进程）
- Shell 会调用 pipe() 函数产生命令数减一个管道，并且使用 dup2() 建立子进程之间的输出和输入重定向，重定向的结果构成一个 pipeline。
  - ls 进程的标准输出，重定向给管道的写端
  - grep 进程的标准输入会重定向给管道的读端
  - grep 进程的标准输出会重定向到文件 a.file
- Shell 会给第一个命令的进程创建一个新的进程组，将后续的命令的进程加入到该进程组。
- 在每个子进程内调用 exec() 去启动相应程序，在调用 exec 时将该命令的参数传递给 exec
- 在命令执行期间，Shell 将调用 wait() 函数等待进程执行完毕，一旦进程执行完毕，Shell 将调用 tcsetpgrp() 函数将自己提到前台，shell 变为前台进程。

进程优先级

- 静态优先级（使用 top 命令时， NI 列显示的值），可以通过 nice/renice 来设置或者调整，可以通过 setpriority() 函数来设置进程的静态优先级。
- 动态优先级（使用 top 命令时， PR 列显示的值），系统调度时根据静态优先级自动调整

如何收尸

- 常规办法，父进程主动调用 wait()/waitpid()，父进程会阻塞
- 轮询方式，父进程循环调用 waitpid()， options 设为 WNOHANG，如果没有子进程退出，则会得到状态，如果有子进程退出，也会得到状态。
- 信号触发方式，父进程注册一个 SIGCHLD 信号的处理函数，当子进程终止时，会有一个信号发送给父进程，此时父进程注册的信号处理函数被自动调用，在该函数内，可以调用 wait() 进行收尸。
  - 该办法不是很稳妥，当有多个子进程同时终止时，很可能只做一次 wait()，造成对其他的子进程不做 wait()，形成僵尸进程。
- 转嫁方式，父进程调用一次 fork，产生的子进程再次调用 fork()，在孙子进程内执行相应的工作，儿子进程调用完 fork() 之后自我终止，造成孙子进程被 init 进程收养，孙子进程的收尸最终由 init 完成。
  - 这种方式比较稳妥，因为 init 进程设计上就确保能够为终止的子进程收尸。
- 自动收尸，在父进程中将 SIGCHLD 信号的处理方式通过 sigaction() 调用设为忽略 (SIG_IGN)，当子进程终止时，自动进行收尸，不通知父进程。
  - 这种方式依赖于实现，不建议使用。

ID 变换的问题 (我们的理解，不见的完全正确)

- 对于 root 用户 (特权用户) 来说，可以将 real uid, effective uid, saved set-user-id 设置为任意有效值
  - 有效值：应该理解为系统内存在的用户，如果设置为不存在的用户 id ，则无效
- 对于非特权用户来说，如何修改 effective user ID?
  - 可以调用 setuid()/setreuid()/seteuid() 修改 euid ，将 real uid 、 effective uid 、 saved set-user-id 设置为 euid
  - 如果非特权进程的程序文件带有 set-uid-bit 标志，则该进程启动时， euid 被设置为程序文件的 owner id
- 对于非特权用户来说，如何修改 real user ID?
  - Posix ： 未定义该功能
  - Linux: 可以调用 setreuid() 来修改 real UID ，可以将 real UID 修改为 real UID 或者 Effective UID
  - 不可以调用 setuid() ，对于非特权用户来说， setuid() 只能设置 euid

进程组的作用

- 进程组跟系统调度有关系，系统调度的是进程组而不是单个进程，此时进程组就是 job(作业)
- 进程组跟信号的分发有关系，在终端上产生的信号，会发送给前台进程组里边的所有进程
- 进程组跟终端上的输入 / 输出有关系，只有前台进程组中的进程才可以从终端读或者向终端写，后台进程如果试图从终端读或者向终端写，会被 SIGTTIN 和 SIGTTOU 信号暂停。
- 在 kill(1) 或者 kill(2) 时，可以给某个进程组发送信号，该进程组内的所有进程都将接收到信号
- 在 waitpid(2) 调用时，可以对某个特定的进程组做 wait()

Session 的作用

- 相对于进程组而言， session 的作用不是很重要， session 像容器一样，包含一个或者多个进程组
- Session 通常开始于用户登录，终止于用户登出
- Session 与终端相关联
  - 如果 session 有一个控制终端，则 session 必然有一个前台进程组
  - 如果 session 没有控制终端，则肯定没有前台进程组
- 如果在进程组之间移动进程，需要满足什么条件？
  - 两个进程组必须属于同一个 session
- 如果一个进程调用 setsid() 创建新的 session 成功，必须满足什么条件？
  - 调用 setsid() 的进程不是进程组的组长，确切的说法是：调用 setsid() 的进程的 pid ，不等于任何进程的进程组 ID

/dev/tty 和 /dev/tty0

- /dev/tty 映射到当前终端，可以是虚终端，也可以是伪终端
- /dev/tty0 也映射到当前终端，只能是虚终端，不能是伪终端

内核如何处理信号

- 什么时机处理？
  - 当进程运行期间因为中断或者异常切换到内核态时，会进行相应的处理(中断处理、异常处理)，完成处理之后，一般来说，需要返回用户空间去继续执行被挂起的main函数，但是在返回用户空间之前，去做信号的检查。
- 如何检查信号？
  - 按照某种顺序，依次检查pending signals，如果某个信号处在pending状态，则检查该信号是否被block，
    - 如果该信号被block，则放弃当前信号的检查，去检查下一个信号
    - 如果该信号没有被block，则根据信号的处理方式进行相应的处理（默认、忽略、调用用户函数）
- 如何捕获信号
  - 如果信号的处理方式是调用用户函数的话，那么内核会返回到用户空间去调用用户函数，该函数与main函数使用不同的堆栈，但是共用代码段、数据段
  - 用户函数执行完毕，会自动的调用sigreturn回到内核空间，在内核空间去处理下一个信号。
  - 如果在内核空间，所有的信号都处理完毕，则再次返回到用户空间去恢复main继续执行

信号处理函数调用的过程

- 不是我们定义一个用户处理函数就直接调用的，在调用我们定义的用户处理函数之前，还需要做些准备
- 为什么做这些准备呢？
  - 如果在调用某个信号的处理函数期间，该信号再次到来，会导致信号处理函数再次被调用，甚至于多次被调用。
- 解决方式，有两种，来自于不同的实现
  - 第一种方式， System V 采用这种方式，在信号处理函数执行期间，临时将该信号的处理方式恢复为默认，也就意味着在此期间，信号又到来的话，会执行默认的动作。
  - 第二种方式， BSD 采用这种方式，在信号处理函数执行期间，临时将该信号阻塞，执行完之后，解除阻塞。
  - Linux(目前版本) 的默认是第二种方式，但是好像可以设置。

Value & mask

- value，说明这个参数被作为整体来看待
- mask，说明这个参数被按位来看待

信号继承的问题

- 在 fork 之后的情况
  - 信号的处理方式 (dispositions) 是继承的
  - 信号屏蔽码 (signal mask) 是继承的
  - 未决信号集 (pending signals) 被清空
- 在 exec 之后的情况
  - 信号处理方式 (dispositions) 不继承
  - 信号屏蔽码 (signal mask) 继承
  - 未决信号集 (pending signals) 继承

- 管道、有名管道传输的流数据
  - 流数据特点
    - 有方向
    - 无记录边界
    - 耦合度高
- 消息队列传递的是消息（记录块）
  - 消息特点
    - 方向性不强
    - 有记录边界
    - 耦合度低
- 管道和消息队列是顺序访问的，读一次之后就没有了，共享内存是随机访问的，读一次之后还有，还可以重复读，可以从任意的地方读或者写。

匿名和有名

- 匿名只能在具有亲缘关系的进程之间使用
  - 相关参数通过 fork 时继承
- 有名可以在无亲缘关系的进程之间使用
  - 相关参数可以通过名字访问，该名字我们通常称为 IPC name，可能是文件系统的路径名（如有名管道），也可能是有其他约定和规则的某个标识（如 /ipc_name ）

IPC name

- 匿名管道没有 IPC name
- 有名管道的 IPC name 是文件系统的路径名
- Posix message queue 的 IPC name 是 /somename，以”/”开头，后边跟一个字符串，字符串后边不能再跟”/”，它不是一个文件系统的路径名

16 进制工具

- xxd
- hexdump
- od

创建消息队列失败的可能情况

- 系统范围的 (system-wide) 消息队列数达到上限了，而我这个进程又不是特权进程
  - /proc/sys/fs/mqueue/queues_max
- 进程的实际用户 ID 所拥有的全部进程的消息队列开销达到 resource limit(RLIMIT_MSGQUEUE)

线程里边返回的问题

- 如果 main() 函数返回，则导致进程终止，相当于调用 exit() ，进程内所有的线程都将终止
- 如果线程的 start_routine() 返回，则导致该线程终止，相当于调用了 pthread_exit() ，函数的返回值是线程的退出码。
- 在多线程编程时，如果 main() 函数可能提前结束，则对 main() 不进行返回，调用 pthread_exit() ，将 main() 函数当成一个线程终止，进程并不终止，因此其他的线程不受影响。

信号的问题

- 新创建线程的信号采用如下的处理方式：
  - 新创建线程的信号屏蔽码 (Blocked signals)，将从创建线程继承
  - 新创建线程的未决信号集 (Pending signals)，将被置为空集
  - 信号的处理方式 (signal disposition) 如何处理？
    - 一个进程内的所有线程，共用进程的信号处理方式，也就是说，如果一个线程修改了信号的处理方式 (捕获、忽略、默认)，那么进程的信号处理方式就被修改了。
    - 不能使用信号去终止某个线程
    - 如果线程不希望受到信号影响的话，需要阻塞信号

多线程信号处理的原则

- 最好设计一个线程去专门处理信号，其他线程则阻塞信号，信号处理线程可以捕获信号，然后进行相应的处理。
- 线程屏蔽（阻塞）信号集，通常来说不要去给每个线程单独去设置，最好是在主线程设置，然后被新创建的线程继承。
- 信号会投递给哪个线程？
  - 投递给进程的信号，比如通过 kill(1)/kill(2)，那么信号将投递给没有阻塞信号的所有线程，如果线程不希望被信号影响，则需要阻塞该信号。
  - 通过 pthread_kill() 发送的信号，将投递给目标线程，如果目标线程没有阻塞该信号，进程也没有捕获该信号，如果该信号的默认动作是终止进程，则会造成进程终止。
  - 某个线程因为某种原因自己产生的信号，比如被 0 除会产生 SIGFPE 信号，该信号将投递给本线程，如果线程没有阻塞该信号，进程也没有捕获该信号，如果该信号的默认动作是终止进程，则会造成进程终止。

线程退出

- 返回值可以被调用 pthread_join() 的线程得到
- 取消清理函数 (cancellation cleanup handlers) 将被按照注册的相反顺序调用。
  - 只会调用那些 push 进去但是没有被 pop 出来的
- 线程的私有数据会被调用相应的析构函数清理
  - 线程私有数据的析构发生在清理函数调用之后
  - 析构函数的调用顺序是不确定的
- 线程终止不会释放进程级的资源
  - 包括但不限于锁、文件描述符
- 线程终止时，不会调用进程级的清理函数
  - 进程级的清理函数通过 atexit() 设置

在多线程的环境下做 fork

- 在多线程环境下，应该不去做 fork ，因为 fork 会导致很多问题。
  - fork() 之后，子进程会继承父进程所有的线程
  - 如果父进程的某个线程做过加锁操作，在该锁也会被子进程的线程继承，如果子进程的线程不解锁，则锁会一直会锁定，父进程的其他线程不能获得锁。
- 有一个特殊情况，如果 fork 之后就做 exec ，则做 fork 调用是没有问题的。

线程函数的命名规范

- 前缀，表示操作对象
  - pthread_，表示操作的是线程本身
  - pthread_attr_，表示操作的是线程的属性
  - pthread_mutex_，表示操作的是线程的 mutex
  - ……
- 后缀，表示动作
  - create
  - init
  - destroy
  - ……

mutex 类型

- PTHREAD_MUTEX_NORMAL
  - 递归加锁 – 死锁
  - 对其他线程加锁的锁解锁 – 未定义
  - 对于未处于加锁状态的锁进行解锁 - 未定义
- PTHREAD_MUTEX_ERRORCHECK
  - 递归加锁 – 报错
  - 对其他线程加锁的锁解锁 – 报错
  - 对于未处于加锁状态的锁进行解锁 - 报错
- PTHREAD_MUTEX_RECURSIVE
  - 递归加锁 – 成功(允许)
  - 对其他线程加锁的锁解锁 – 报错
  - 对于未处于加锁状态的锁进行解锁 - 报错
- PTHREAD_MUTEX_DEFAULT
  - 递归加锁 – 未定义
  - 对其他线程加锁的锁解锁 – 未定义
  - 对于未处于加锁状态的锁进行解锁 - 未定义

网络相关的协议

- RFC
- ITU 协议，电信相关
  - H.323
  - H.264
  - V.90
- IEEE，链路层以下的
- ISO，超集，拿来主义

- 分层
  - 每层的作用
  - 理解面向对象设计的思想
- 从上到下穿透协议栈
  - 数据封装的过程
- 从下到上的过程
  - 数据解包的过程，分用 Demultiplexing
  - 数据过滤的过程
- 了解数据包如何从源主机到目的主机的
  - 直接相连的情况
  - 通过路由器连接的情况
- 了解数据包如何从发送程序到接收程序的过程

- IP 协议
  - IP 协议的作用
  - IP 协议的特点
- UDP 协议
  - UDP 协议的特点
  - UDP 协议的用途
- TCP 协议
  - TCP 协议的特点
  - 连接建立过程中三次握手的过程
  - 连接断开时四次握手的过程
  - 可靠性如何保障
  - 状态变迁图
- UDP vs TCP

End to end

- Endpoint to endpoint
- Socket to socket
- Process to process
- Process → socket? bind()

TCP 的语境问题

- 当和 IP 一起提 TCP 时，此时 TCP 是广义的概念，代表狭义的 TCP 和 UDP
- 当和 UDP 一起提的时候，此时 TCP 是狭义的传输控制协议的概念。

特殊的 IP 地址

- 全部比特为 0 的地址 (0.0.0.0)
  - 不能作为目的 IP 地址
  - 但是 bind 调用时，可以传递该地址，做为绑定所有接口的参数
- 全部比特为 1 的地址（ 255.255.255.255 ）
  - 表示绝对广播地址，在不知道子网参数的时候使用，比如 DHCP、 ARP
- 子网号确定的情况下
  - 主机号全部为 1 的地址，叫子网广播地址，在子网参数确定的情况下，使用该地址进行广播通讯
  - 主机号全部为 0 的地址，代表一个子网，通常在配置防火墙等网络参数时候使用。

私有 IP 地址

- 私有 IP 地址
  - A 类：10.0.0.0~10.255.255.255
  - B 类：172.16.0.0~172.31.255.255
  - C 类：192.168.0.0~192.168.255.255
- 私有 IP 地址的数据不能在 Internet 上路由
- 私有网络的数据包要想通过 Internet 传输，需要做 NAT(网络地址转换)
  - 数据包从一个私有网络离开时 (A->B)，边界的 NAT 设备将该数据包的目的地址不变，源地址由 A 变为 NAT 设备的外出接口 IP 地址，结果是 NAT->B
  - 当 B 收到数据包时，如果需要返回，这种情况下 B->NAT，则数据包到达私有网络时，NAT 设备将数据包的目的 IP 地址，替换为私有网络内的主机 IP 地址，比如给 A 的，替换为 A，最终 A 收到的为 B->A
  - NAT 设备需要记录状态，A->B，在数据包返回时，需要翻译为 B->A
  - B 如果也是一个私有网络的地址的时候，如何处理？
- NAT 设备记录转换状态的方式不同，NAT 的类型也不同
- 做 P2P 类应用时，需要处理 NAT 穿越的问题。

任务：搞清楚网络配置的问题

- IP 地址如何配置？配置文件在哪儿？
  - 静态
  - 动态
- 路由如何配置？
  - 给出命令行。
- DNS 解析如何配置？
  - 相关的配置文件有哪些？
- 有兴趣的话，研究一下防火墙怎么配置？
  - iptables

ARP 缓存机制

- 链路层会维护 ARP 缓存，通过如下的命令行可以查询：
  - $ arp -a
- 用户空间可以通过 arp 命令来操作 ARP 缓存，
  - arp -a
  - arp -d 删除
  - arp -s 设定静态的 ARP
- 缓存是有超时机制的，当超时时间到时，则清掉超时的缓存条目
- 缓存自动更新机制，当上层 (IP 层) 有数据包要发送时，会查询 ARP 缓存
  - 如果在缓存中找到对应的条目，并且该条目未超时，则直接使用 MAC 地址
  - 如果在缓存中未找到对应的条目，则链路层自动发起一个 ARP 查询请求，等待对方的 ARP 应答，得到应答后，更新 ARP 缓存。

Socket 相关的术语

- Socket
  - 大陆：套接字、套接口
  - 港台：插口
- Port
  - 大陆：端口 (号)
  - 港台：埠 (号)

地址表示方法

- IPv4，采用十进制点分法表示
  - 192.168.1.108
  - 有些时候，也采用十六进制表示，这种方法不常见
  - 在程序里，使用二进制表示
- IPv6，采用十六进制，以冒号分割的方式，也有时候采用点分割，点分割的方式不常见
  - fe80::20d:60ff:fecb:86e5
  - fe80:0000:0000:0000:020d:60ff:fecb:86e5
  - 有两种表示方式，长方式和短方式
    - 长方式，为 0 的位置也要写出来
    - 短方式，连续为 0 的位置，通过两个连续的冒号表示

地址转换函数

- 只能处理 IPv4 地址的
  - inet_aton
  - inet_ntoa
  - inet_addr
    - 这个函数有问题，不能正确处理 255.255.255.255
  - ……
- 既可以转 IPv4，也可以转 IPv6 的
  - inet_ntop
  - inet_pton

Address family & protocol family

- Address family 都是以“ AF_” 前缀开头的
- Protocol family 都是以“ PF_” 前缀开头的
- 从设计上来讲， Protocol family 支持多个 address family ，但是实现上将二者实现为一对一的，也就是说 PF_XXX=AF_XXX
- AF_INET(PF_INET) ，对应的是 IPv4
- AF_INET6(PF_INET6) ，对应的是 IPv6
- AF_UNIX/AF_LOCAL(PF_UNIX/PF_LOCAL) ，对应的是 UNIX domain
- 使用的一般规则：
  - 当传递给 socket() 函数的 domain 参数时用 PF_XXX
  - 当传递给地址相关的处理函数时，用 AF_XXX

端口号

- 周知端口，由 /etc/services 文件定义，可以通过如下函数访问：
  - getservbyname()
  - getservbyport()
- 特权端口 (1-1023)，不含 1024，只能由 euid=0 的用户打开
- 端口号的使用规则：
  - 尽可能的避免使用周知端口
    - 如果使用了周知端口，可能会造成一些问题，比如 blind attack
    - 避免 blink attack，可以在协议中设置一个 magic value
  - 如果需要使用多个端口号，最好在一个连续的区间使用 (该区间够用即可)，方便系统的部署，因为很多时候，网络的防火墙是要限制端口号的，如果端口号随意使用，防火墙需要全部开放端口，不安全，如果端口号在某个段内使用，防火墙可以只打开该段。

字节序的问题

- 在编程序的时候，不需要程序员去关注 CPU 体系结构来判断字节序，系统提供的头文件都定义好了字节序相关的参数
  - BYTEORDER
  - __BYTE_ORDER，如果程序中需要检查字节序时，使用该宏定义
  - 如果针对不同字节序编码时，可以在程序内判断 __BYTE_ORDER 的定义：
    - __BIG_ENDIAN，表示大端字节序
    - __LITTLE_ENDIAN，表示小端字节序
- 所以，不要随便拷贝头文件来使用

字节序转换

- 对于长度为 8 bits 的数据类型，不需要转换
- 对于长度是 16 bits 的数据类型，需要转换
  - htons()
  - ntohs()
- 对于长度是 32 bits 的数据类型，需要转换
  - htonl()
  - ntohl()
- 对于长度是 64 bits 的数据类型，需要转换
  - 系统一般不提供转换的函数，需要自己手动的转换，或者自己定义协议进行约定
  - 有些系统提供 htonll() 和 ntohll()

传输数据处理理

- 整型数(short/long/long long)，采用字节序转换函数，但是64位类型没有提供转换函数
- float/double，如何处理？
  - 转换为整数，在发送时乘以系数，在接收时除以相应的系数,系数为10的整数倍。
- 负值如何处理？
  - 将符号位和值(转为绝对值)单独传输
- 在协议设计时，避免使用char/int/long类型，用stdint.h定义的类型
  - int8_t/uint8_t
  - int16_t/uint16_t
  - int32_t/uint32_t
  - int64_t/uint64_t
- 传输结构体，在协议设计时，主动进行对齐(推荐方式)，或者设置#progma pack(1)
  - 对齐的问题
  - 填充的问题
  - 位域(尽可能不用位域，如果必须用的话，对不同的字节序分别定义，类似IP头、 TCP头的定义)

Socket 编程里边的一致问题

- 以下代码中的地址参数应该是一致的：
  - 创建 socket 时候，调用 socket() 的 domain 参数
  - 调用 connect()/bind()/accept()/sendto()/recvfrom() 的地址参数
  - 地址参数中的 family 成员

UDP socket 做 connect() 的问题

- UDP 可以做 connect()，但是不是发起与远端主机的连接，只是通知本地的传输层，
  - 以后默认发送数据报的地址是 connect 时的地址
    - 因此发送可以使用 send()，而不用 sendto()
  - 以后只从 connect 的地址接收数据报，从其他地址来的数据报不接收。
- UDP 类型的 socket 可以调用 connect 多次，去改变默认的关联地址
- 如果不想跟任何地址关联，将地址结构中的 sa_family 参数设为 AF_UNSPEC。

TCP socket 调用 connect

- TCP 连接 (面向连接的 socket) 可以成功调用 connect 一次，如果连接成功之后被断开，关掉当前的 socket ，重新创建新的，再去进行 connect
- TCP 的 socket 在调用 connect() 时，会发起三次握手，那意味着 connect 调用可能会阻塞。
- connect() 阻塞的问题对于程序员来说，处理比较困难。

backlog

- 传统上， backlog 翻译为”积压值”
- backlog 参数非常重要
  - 如果设置过小，可能会造成 DoS
  - 如果设置过大，客户端的 connect 请求等待时间太长

Socket 的层 (level)

- SOL_SOCKET(通用选项)
- IPPROTO_IP(IP 层选项)
- IPPROTO_ICMPV6(ICMPv6 选项)
- IPPROTO_IPV6(IPv6 选项)
- IPPROTO_TCP(TCP 选项)

Socket options

- setsockopt()/getsockopt() 参数有三种情况：
  - 开关： 0 表示关闭，非 0 表示打开
    - SO_BROADCAST
    - SO_REUSEADDR/SO_REUSEPORT
  - 值：将 option 设置为该值大小
    - SO_RCVBUF/SO_SNDBUF
    - SO_RCVLOWAT/SO_SNDLOWAT
  - 结构体，不同的 option 可能定义不同的结构体，其成员的意义分别说明
    - SO_LINGER
    - SO_RCVTIMEO/SO_SNDTIMEO

UNIX domain 编程

- 接口是 socket
  - domain(protocol faimly)=PF_UNIX/PF_LOCAL
  - address family = AF_UNIX/AF_LOCAL
  - 地址结构： struct sockaddr_un
  - 类型：流式 (SOCK_STREAM)，类似于 TCP/ 数据报 (SOCK_DGRAM)，类似于 UDP
- 内部的实现机制类似于管道，相当于一个全双工的管道。
- 优势：本地通讯速度快
  - 如果本地通讯使用网络 127.0.0.1，数据包要从应用程序传输给传输层，再传给网络层，网络层发现数据包的目的地址是本机，则传给传输层，再传给应用层
  - 使用 UNIX domain 方式通讯，相当于两个进程之间建立管道，数据类似于直接拷贝。
- 应用：
  - X-window，本地通讯使用 unixdomain
  - MySQL，本地数据库连接，使用 unixdomain
  - Syslog，本地通讯使用 unixdomain

- I/O多路复用的三个调用
  - select(),来源于BSD
  - poll()，来源于System V
  - epoll()， Linux特有的
- 相同点
  - 作用是一样的，通知该函数我们希望的事情，该函数会告诉我们结果
- 不同点
  - 参数不相同
    - select()使用的是fdset
    - poll()用的是pollfd，一个结构体
    - epoll()用的是struct epoll_event，对于用户进程来说，数据结构不是很重要。
  - 性能，select()/poll()是O(n)， epoll()是O(1)
    - select()/poll()返回的结果没有独立出来，用户进程需要遍历查找已经就绪的来进行处理
    - epoll()将已经就绪的通过独立的集合返回，这样用户进程可以直接访问该结果集，命中率是100%
  - epoll()能够自动的清除已经关闭的描述符，select()/poll()要由程序来清除已经关闭的描述符。
  - select()最麻烦，poll()较为简单，epoll()最简单
  - select()所管理的描述符数较少，poll()较大，epoll()很大。
  - select()检查的状态比较少，poll()支持的状态检查数更多，epoll()支持两种触发方式(ET/LT)

- 如何管理多个连接？
  - 使用线性表（数组变种），以描述符作为下标来索引数组
- 每个连接都需要管理什么？
  - 需要管理两个缓冲区，一个发送缓冲区，一个接收缓冲区
- 缓冲区如何管理？
  - 缓冲区需要动态分配
  - 缓冲区需要有一个地址指针
  - 缓冲区需要有一个 size
  - 定义一个 payload_length

socket 状态的问题

- 对于处于 LISTEN 状态的 socket 来说
  - 可以做 accept()
  - 可以 close()，不可以做 shutdown()
  - 不可以 read()，也不可以 write()
- 对于 listen 状态的 socket 来说，可读表示可以做 accept() 而不阻塞
- 对于 ESTABLISHED 状态的 socket 来说
  - 可以 read()
  - 可以 write()
  - 可以 close()/shutdown()
  - 不可以做 accept()