**信号量：**

**创建有名信号量：**

创建或者打开一个信号量，需要使用sem\_open()函数，函数原形如下：

　　sem\_t sem\_open(const char \* name, int oflag, mode\_t mode, unsigned int value)

　　返回值sem\_t 是一个结构，如果函数调用成功，则返回指向这个结构的指针，里面装着当前信号量的资源数。

　　参数name，就是信号量的名字，两个不同的进程通过同一个名字来进行信号量的传递。

　　参数oflag,当他是O\_CREAT时，如果name给出的信号量不存在，那么创建，此时必须给出mode和vaule。

　　参数mode，很好理解，用来指定信号量的权限。数字 4 、2 和 1表示读、写、执行权限，即 r=4，w=2，x=1。a,b,c各为一个数字，分别代表User、Group、及Other的权限。

chmod 777 file (等价于 chmod u=rwx,g=rwx,o=rwx file 或 chmod a=rwx file)

　　参数vaule，则是信号量的初始值。

**关闭有名信号量:**

　　关闭有名信号量所使用的函数是sem\_close(sem\_t \*sem)

　　这个函数只有一个参数，意义也非常明显，就是指信号量的名字。

**信号量操作：**

在使用信号量时，有两个非常重要的操作

　　P操作：使用的函数是sem\_wait(sem\_t \*sem)

　　如果信号量的值大于零，sem\_wait函数将信号量减一，并且立即返回。如果信号量的值小于零，那么该进程会被阻塞在原地。

　　V操作：使用的函数是sem\_post(sem\_t \*sem)

　　当一个进程使用完某个信号量时，他应该调用sem\_post函数来告诉系统收回资源。

　　sem\_post函数和sem\_wait函数的功能刚好相反，他会把指定的信号量加一

**删除有名信号量：**

　　当使用完有名信号后，需要调用函数sem\_unlink来释放资源。

　　函数原形：int sem\_unlink(const char \*name)

**消息队列**

消息队列的实现操作

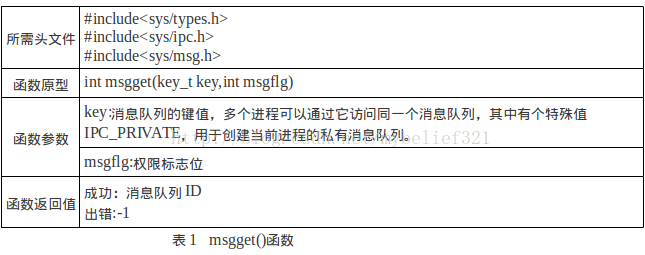
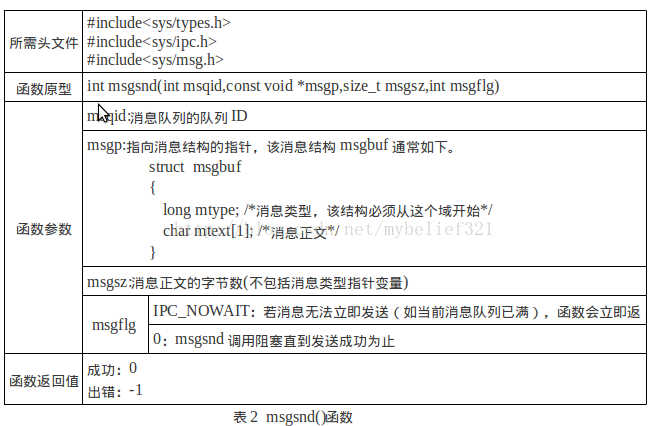
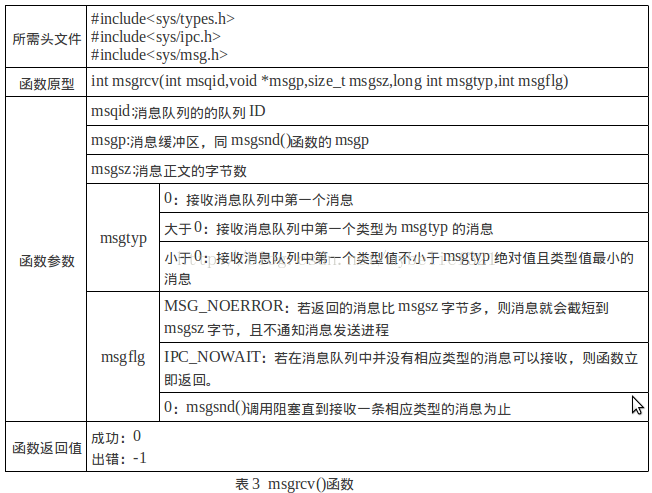
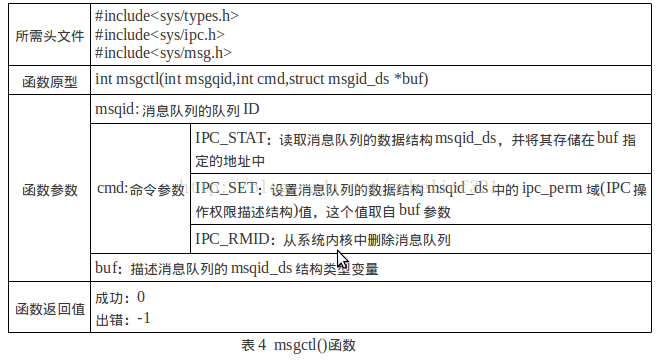
① 创建或打开消息队列。使用的函数是msgget()，这里创建的消息队列的数量会受到系统消息队列数量的限制。

② 添加消息。使用的函数是msgsnd()，它把消息添加到已打开的消息队列末尾。

③ 读取消息。使用的函数是msgrcv()，它把消息从消息队列中取走，与FIFO不同的是，这里可以取走指定的某一条消息。

④ 控制消息队列。使用的函数是msgctl()，它可以完成多项功能。

函数格式

fgets()函数读取到它所遇到的第一个换行符的后面，或者读取比字符串的最大长度少一个的字符，或者读取到文件结尾。然后fgets()函数向末尾添加一个空字符以构成一个字符串。如果在达到字符最大数目之前读完一行，它将在字符串的空字符之前添加一个换行符以标识一行结束。

ftok函数，先不去了解它的作用来先说说为什么要用它，共享内存，消息队列，信号量它们三个都是找一个中间介质，来进行通信的，这种介质多的是。就是怎么区分出来，就像唯一一个身份证来区分人一样。你随便来一个就行，就是因为这。只要唯一就行，就想起来了文件的设备编号和节点，它是唯一的，但是直接用它来作识别好像不太好，不过可以用它来产生一个号。ftok()就出场了。ftok函数具体形式如下：

key\_t ftok(const char \*pathname, int proj\_id);

其中参数fname是指定的文件名，这个文件必须是存在的而且可以访问的。id是子序号，它是一个8bit的整数。即范围是0~255。当函数执行成功，则会返回key\_t键值，否则返回-1。在一般的UNIX中，通常是将文件的索引节点取出，然后在前面加上子序号就得到key\_t的值。

有关该函数的三个常见问题：

1.pathname是目录还是文件的具体路径，是否可以随便设置

2.pathname指定的目录或文件的权限是否有要求

3.proj\_id是否可以随便设定，有什么限制条件

解答：

1、ftok根据路径名，提取文件信息，再根据这些文件信息及project ID合成key，该路径可以随便设置。

2、该路径是必须存在的，ftok只是根据文件inode在系统内的唯一性来取一个数值，和文件的权限无关。

3、proj\_id是可以根据自己的约定，随意设置。这个数字,有的称之为project ID; 在UNIX系统上,它的取值是1到255;

**共享内存**

与信号量一样，在Linux中也提供了一组函数接口用于使用共享内存，而且使用共享共存的接口还与信号量的非常相似，而且比使用信号量的接口来得简单。它们声明在头文件 sys/shm.h 中。

1、shmget()函数

该函数用来创建共享内存，它的原型为：

int shmget(key\_t key, size\_t size, int shmflg);

1. 第一个参数，与信号量的semget函数一样，程序需要提供一个参数key（非0整数），它有效地为共享内存段命名，shmget()函数成功时返回一个与key相关的共享内存标识符（非负整数），用于后续的共享内存函数。调用失败返回-1.不相关的进程可以通过该函数的返回值访问同一共享内存，它代表程序可能要使用的某个资源，程序对所有共享内存的访问都是间接的，程序先通过调用shmget()函数并提供一个键，再由系统生成一个相应的共享内存标识符（shmget()函数的返回值），只有shmget()函数才直接使用信号量键，所有其他的信号量函数使用由semget函数返回的信号量标识符。
2. 第二个参数，size以字节为单位指定需要共享的内存容量
3. 第三个参数，shmflg是权限标志，它的作用与open函数的mode参数一样，如果要想在key标识的共享内存不存在时，创建它的话，可以与IPC\_CREAT做或操作。共享内存的权限标志与文件的读写权限一样，举例来说，0644,它表示允许一个进程创建的共享内存被内存创建者所拥有的进程向共享内存读取和写入数据，同时其他用户创建的进程只能读取共享内存。

2、shmat()函数 -- at：attach

第一次创建完共享内存时，它还不能被任何进程访问，shmat()函数的作用就是用来启动对该共享内存的访问，并把共享内存连接到当前进程的地址空间。它的原型如下：

void \*shmat(int shm\_id, const void \*shm\_addr, int shmflg);

1. 第一个参数，shm\_id是由shmget()函数返回的共享内存标识。
2. 第二个参数，shm\_addr指定共享内存连接到当前进程中的地址位置，通常为空，表示让系统来选择共享内存的地址。
3. 第三个参数，shm\_flg是一组标志位，通常为0。
4. 调用成功时返回一个指向共享内存第一个字节的指针，如果调用失败返回-1.

3、shmdt()函数 -- dt：detach

该函数用于将共享内存从当前进程中分离。注意，将共享内存分离并不是删除它，只是使该共享内存对当前进程不再可用。它的原型如下：

int shmdt(const void \*shmaddr);

参数shmaddr是shmat()函数返回的地址指针，调用成功时返回0，失败时返回-1.

4、shmctl()函数 -- ctl：control

与信号量的semctl()函数一样，用来控制共享内存，它的原型如下：

int shmctl(int shm\_id, int command, struct shmid\_ds \*buf);

第一个参数，shm\_id是shmget()函数返回的共享内存标识符。

第二个参数，command是要采取的操作，它可以取下面的三个值 ：

* IPC\_STAT：把shmid\_ds结构中的数据设置为共享内存的当前关联值，即用共享内存的当前关联值覆盖shmid\_ds的值。
* IPC\_SET：如果进程有足够的权限，就把共享内存的当前关联值设置为shmid\_ds结构中给出的值
* IPC\_RMID：删除共享内存段
* 第三个参数，buf是一个结构指针，它指向共享内存模式和访问权限的结构

**信号机制**

SIGABRT 进程停止运行 6

SIGALRM 警告钟

SIGFPE 算述运算例外

SIGHUP 系统挂断

SIGILL 非法指令

SIGINT 终端中断 2

SIGKILL 停止进程（此信号不能被忽略或捕获）

SIGPIPE 向没有读的管道写入数据

SIGSEGV 无效内存段访问

SIGQOUT 终端退出 3

SIGTERM 终止

SIGUSR1 用户定义信号1

SIGUSR2 用户定义信号2

SIGCHLD 子进程已经停止或退出

SIGCONT 如果被停止则继续执行

SIGSTOP 停止执行

SIGTSTP 终端停止信号

SIGTOUT 后台进程请求进行写操作

SIGTTIN 后台进程请求进行读操作

typedef void (\*sighandler\_t)(int);

sighandler\_t signal(int signum, sighandler\_t handler);

signal函数

作用1：站在应用程序的角度，注册一个信号处理函数

作用2：忽略信号，设置信号默认处理 信号的安装和回复

参数

--signal是一个带signum和handler两个参数的函数，准备捕捉或屏蔽的信号由参数signum给出，接收到指定信号时将要调用的函数有handler给出

--handler这个函数必须有一个int类型的参数（即接收到的信号代码），它本身的类型是void

--handler也可以是下面两个特殊值：① SIG\_IGN 屏蔽该信号 ② SIG\_DFL 恢复默认行为

Alarm函数：

引用头文件：#include <unistd.h>;

函数标准式：unsigned int alarm(unsigned int seconds);

功能与作用：alarm()函数的主要功能是设置信号传送闹钟，即用来设置信号SIGALRM在经过参数seconds秒数后发送给目前的进程。如果未设置信号SIGALARM的处理函数，那么alarm()默认处理终止进程。

函数返回值：如果在seconds秒内再次调用了alarm函数设置了新的闹钟，则后面定时器的设置将覆盖前面的设置，即之前设置的秒数被新的闹钟时间取代；当参数seconds为0时，之前设置的定时器闹钟将被取消，并将剩下的时间返回。

**管道通信**

#include<unistd.h>

int pipe(int fd[2]);

功能： 创建一个简单的管道，若成功则为数组fd分配两个文件描述符，其中fd[0] 用于读取管道，fd[1]用于写入管道。

返回：成功返回0，失败返回-1；

**创建命名管道**

用如下两个函数中的其中一个，可以创建命名管道。

#include <sys/types.h>

#include <sys/stat.h>

int mkfifo(const char \*filename, mode\_t mode);

int mknod(const char \*filename, mode\_t mode | S\_IFIFO, (dev\_t)0);

filname是指文件名，而mode是指定文件的读写权限。mknod是比较老的函数，而使用mkfifo函数更加简单和规范，所以建议用mkfifo。

**打开命名管道**

和打开其他文件一样，可以用open来打开。通常有四种方法：

open(const char \*path, O\_RDONLY);//1

open(const char \*path, O\_RDONLY | O\_NONBLOCK);//2

open(const char \*path, O\_WRONLY);//3

open(const char \*path, O\_WRONLY | O\_NONBLOCK);//4

要注意:

* 就是程序不能以O\_RDWR(读写)模式打开FIFO文件进行读写操作，而其行为也未明确定义，因为如一个管道以读/写方式打开，进程就会读回自己的输出，同时我们通常使用FIFO只是为了单向的数据传递。
* 就是传递给open调用的是FIFO的路径名，而不是正常的文件。（如：const char \*fifo\_name = "/tmp/my\_fifo"; ）
* 第二个参数中的选项O\_NONBLOCK，选项O\_NONBLOCK表示非阻塞，加上这个选项后，表示open调用是非阻塞的，如果没有这个选项，则表示open调用是阻塞的。

**阻塞问题**

* 对于以只读方式（O\_RDONLY）打开的FIFO文件，如果open调用是阻塞的（即第二个参数为O\_RDONLY），除非有一个进程以写方式打开同一个FIFO，否则它不会返回；如果open调用是非阻塞的的（即第二个参数为O\_RDONLY | O\_NONBLOCK），则即使没有其他进程以写方式打开同一个FIFO文件，open调用将成功并立即返回。
* 对于以只写方式（O\_WRONLY）打开的FIFO文件，如果open调用是阻塞的（即第二个参数为O\_WRONLY），open调用将被阻塞，直到有一个进程以只读方式打开同一个FIFO文件为止；如果open调用是非阻塞的（即第二个参数为O\_WRONLY | O\_NONBLOCK），open总会立即返回，但如果没有其他进程以只读方式打开同一个FIFO文件，open调用将返回-1，并且FIFO也不会被打开。

**Scoket通信**

**socket()函数**

int **socket**(int protofamily, int type, int protocol);//返回sockfd

     sockfd是描述符。

  socket函数对应于普通文件的打开操作。普通文件的打开操作返回一个文件描述字，而**socket()**用于创建一个socket描述符（socket descriptor），它唯一标识一个socket。这个socket描述字跟文件描述字一样，后续的操作都有用到它，把它作为参数，通过它来进行一些读写操作。正如可以给fopen的传入不同参数值，以打开不同的文件。创建socket的时候，也可以指定不同的参数创建不同的socket描述符，socket函数的三个参数分别为：

* protofamily：即协议域，又称为协议族（family）。常用的协议族有，AF\_INET(IPV4)、AF\_INET6(IPV6)、AF\_LOCAL（或称AF\_UNIX，Unix域socket）、AF\_ROUTE等等。协议族决定了socket的地址类型，在通信中必须采用对应的地址，如AF\_INET决定了要用ipv4地址（32位的）与端口号（16位的）的组合、AF\_UNIX决定了要用一个绝对路径名作为地址。
* type：指定socket类型。常用的socket类型有，SOCK\_STREAM、SOCK\_DGRAM、SOCK\_RAW、SOCK\_PACKET、SOCK\_SEQPACKET等等（socket的类型有哪些？）。
* protocol：故名思意，就是指定协议。常用的协议有，IPPROTO\_TCP、IPPTOTO\_UDP、IPPROTO\_SCTP、IPPROTO\_TIPC等，它们分别对应TCP传输协议、UDP传输协议、STCP传输协议、TIPC传输协议（这个协议我将会单独开篇讨论！）。

**注意**：并不是上面的type和protocol可以随意组合的，如SOCK\_STREAM不可以跟IPPROTO\_UDP组合。当protocol为0时，会自动选择type类型对应的默认协议。

当我们调用**socket**创建一个socket时，返回的socket描述字它存在于协议族（address family，AF\_XXX）空间中，但没有一个具体的地址。如果想要给它赋值一个地址，就必须调用bind()函数，否则就当调用connect()、listen()时系统会自动随机分配一个端口。

**bind()函数**

正如上面所说bind()函数把一个地址族中的特定地址赋给socket。例如对应AF\_INET、AF\_INET6就是把一个ipv4或ipv6地址和端口号组合赋给socket。

int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

函数的三个参数分别为：

* sockfd：即socket描述字，它是通过socket()函数创建了，唯一标识一个socket。bind()函数就是将给这个描述字绑定一个名字。
* addr：一个const struct sockaddr \*指针，指向要绑定给sockfd的协议地址。这个地址结构根据地址创建socket时的地址协议族的不同而不同
* addrlen：对应的是地址的长度。

通常服务器在启动的时候都会绑定一个众所周知的地址（如ip地址+端口号），用于提供服务，客户就可以通过它来接连服务器；而客户端就不用指定，有系统自动分配一个端口号和自身的ip地址组合。这就是为什么通常服务器端在listen之前会调用bind()，而客户端就不会调用，而是在connect()时由系统随机生成一个。

**listen()、connect()函数**

如果作为一个服务器，在调用socket()、bind()之后就会调用listen()来监听这个socket，如果客户端这时调用connect()发出连接请求，服务器端就会接收到这个请求。

int listen(int sockfd, int backlog);

int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

listen函数的第一个参数即为要监听的socket描述字，第二个参数为相应socket可以排队的最大连接个数。socket()函数创建的socket默认是一个主动类型的，listen函数将socket变为被动类型的，等待客户的连接请求。

connect函数的第一个参数即为客户端的socket描述字，第二参数为服务器的socket地址，第三个参数为socket地址的长度。客户端通过调用connect函数来建立与TCP服务器的连接。

**accept()函数**

TCP服务器端依次调用socket()、bind()、listen()之后，就会监听指定的socket地址了。TCP客户端依次调用socket()、connect()之后就向TCP服务器发送了一个连接请求。TCP服务器监听到这个请求之后，就会调用accept()函数取接收请求，这样连接就建立好了。之后就可以开始网络I/O操作了，即类同于普通文件的读写I/O操作。

int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen); //返回连接connect\_fd

参数sockfd

参数sockfd就是上面解释中的监听套接字，这个套接字用来监听一个端口，当有一个客户与服务器连接时，它使用这个一个端口号，而此时这个端口号正与这个套接字关联。当然客户不知道套接字这些细节，它只知道一个地址和一个端口号。

参数addr

这是一个结果参数，它用来接受一个返回值，这返回值指定客户端的地址，当然这个地址是通过某个地址结构来描述的，用户应该知道这一个什么样的地址结构。如果对客户的地址不感兴趣，那么可以把这个值设置为NULL。

参数len

如同大家所认为的，它也是结果的参数，用来接受上述addr的结构的大小的，它指明addr结构所占有的字节个数。同样的，它也可以被设置为NULL。

如果accept成功返回，则服务器与客户已经正确建立连接了，此时服务器通过accept返回的套接字来完成与客户的通信。

**read()、write()等函数**

至此服务器与客户已经建立好连接了。可以调用网络I/O进行读写操作了，即实现了网咯中不同进程之间的通信！网络I/O操作有下面几组：

* read()/write()
* recv()/send()
* readv()/writev()
* recvmsg()/sendmsg()
* recvfrom()/sendto()

read函数是负责从fd中读取内容.当读成功时，read返回实际所读的字节数，如果返回的值是0表示已经读到文件的结束了，小于0表示出现了错误。如果错误为EINTR说明读是由中断引起的，如果是ECONNREST表示网络连接出了问题。

write函数将buf中的nbytes字节内容写入文件描述符fd.成功时返回写的字节数。失败时返回-1，并设置errno变量。 在网络程序中，当我们向套接字文件描述符写时有俩种可能。1)write的返回值大于0，表示写了部分或者是全部的数据。2)返回的值小于0，此时出现了错误。我们要根据错误类型来处理。如果错误为EINTR表示在写的时候出现了中断错误。如果为EPIPE表示网络连接出现了问题(对方已经关闭了连接)。

### close()函数

在服务器与客户端建立连接之后，会进行一些读写操作，完成了读写操作就要关闭相应的socket描述字，好比操作完打开的文件要调用fclose关闭打开的文件。

#include <unistd.h>

int close(int fd);

close一个TCP socket的缺省行为时把该socket标记为以关闭，然后立即返回到调用进程。该描述字不能再由调用进程使用，也就是说不能再作为read或write的第一个参数。

注意：close操作只是使相应socket描述字的引用计数-1，只有当引用计数为0的时候，才会触发TCP客户端向服务器发送终止连接请求。