IO进线程网络编程

模块一、I/O

1. 标准IO：（也叫缓冲IO）

文件：存储在外部设备上的一组相关数据的有序结合；

文件有一个文件格式头，是用来区分文件的格式的；

Linux下一切皆文件，统一接口，方便对设备的管理；

文件的七中类型：

b：普通文件（ASCII文件或二进制文件）；

d：目录文件；

l：连接文件；

b：块设备文件；

c：字符设备文件；

p：管道文件；

s：套接字文件；

读：将数据从外部设备输入到计算机内存的过程；

写：将数据从计算机内存输出到外部介质的过程；

流：流是计算机数据传输的一种动作、是指一种行为方式，计算机数据的传输就是以流的形式进行传输的。

操作系统与文件：操作系统对数据的管理就是以文件为单位进行管理的，也就是说，操作系统要对外部设备上的数据进行读或写，就必须先找到对应的文件，然后才能对该数据进行操作；

文件指针：

在内存中，每个使用的文件都会开辟一个区域，而文件的信息是保存再一个结构体变量中，即：

typedef struct

{

文件相关信息;

} FILE;

在对文件操作时，就用该类型定义一个文件指针，对文件进行操作：FILE \*fp;

标准IO库：

缓冲文件系统：磁盘与内存文件的交换必须经过缓存区

全缓存：当（缓冲区）到达一定条件（如缓存满），再进行IO的操作；

行缓存：在输入和输出遇到”\n”时，就进行IO操作（如输入输出设备的enter），如果没有遇到换行这不会进行自动将缓冲区进行输入输出；

不缓存：标准IO库不对字符进行缓冲，很多的人机交互就要求不可全缓存，并且标准出错是绝不会全缓存的；

使用setbuf()和setvbuf()，可以更改缓存类型，也可以使用fflush()强制刷新一个数据流（强制输入或输出缓冲区的数据fflush(FILE \*fp)），一般文件是默认全缓存；

标准I/O预定义的三个流，他们可以自动的为进程使用：

|  |  |  |  |
| --- | --- | --- | --- |
| 标准输入 | 0 | STDIN\_FILENO | stdin |
| 标准输出 | 1 | STDOUT\_FILENO | stdout |
| 标准错误输出 | 2 | STDERR\_FILENO | stderr |

注释：这三种类型都是FILE \* 类型

标准IO库：

文件打开I/O：

FILE \*fopen(const char \*path, const char \*mode);

使用方式：

FILE \*fp = fopen(“文件名”, 使用方式);

例如：fp = fopen(“test.c”, “r”);

//打开当前目录文件test.c 对该文件只读操作，成功返回指向文件结构体的指针给fp以供使用，失败返回NULL；

常用命令参数表

|  |  |
| --- | --- |
| **文件使用方式** | **含义** |
| “r” （只读） | 打开文本文件只进行读操作 |
| “w” （只写） | 打开文本文件只进行写操作 |
| “a” （追加） | 向文本文件末尾进行写操作 |
| “rb” （只读） | 对二进制文件进行读操作 |
| “r+” （读写） | 打开文本文件，可读可写 |
| “a+” （读写） | 文本文件可读或在末尾追加 |
| “rb+”（读写） | 对二进制文件可读可写 |

* 特别说明：

1）、使用”r”参数操作文件时，该文件必须要存在，如果该文件不存在，否则就会出错；

2）、使用”w、w+”参数操作文件时，如果该文件不存在，会以该文件名建立新的文件，如果该文件存在，则会删除该文件，重新建立该文件新建文件权限默认是（0666），但是其真正权限受掩码值影响（详情见Linux学习笔记）。

3）、使用”a”参数，文件内部指针自动移到文件末尾，在该文件末尾添加数据。

文件关闭I/O：

int fclose(FILE \*stream);

使用方式：fclose(文件指针);

例：fclose(fp); //将fp所指空间释放，不能在使用fp所指的文件，成功返回0，失败返回EOF（即-1），并设置errno（出错信息编号使用strerror函数可以将错误编号翻译成具体的错误信息），注意流关闭时会强制刷新缓冲区（释放）；

字符读写I/O：

出错处理函数perror();

viod perror(const char \*s);

feof();判断文件是否结束，EOF结束标识符；

fgetc()/fputc() 一次读写一个字符

fputc 函数是将字符写到文件上去

原型：int fputc(int c, FILE \*stream);

使用方式：fputc(ch, fp);//将ch对应的字符写到指针变量fp所指的文件，成功返回输出的字符，失败返回EOF；

fgetc函数是从指定的文件读取一个字符

原型：int getc(FILE \*istream);

使用方式：int ch = fgetc(fp);//从fp指向的文件中读取一个字符，如果读取时以到文件末尾，则返回EOF读取结束。

fgets();

char \*fgets(char \*s, int size, FILE \*stream);

用法：将stream指向的字符串读入size-1个字节大小的长度，读到s所指向的空间；注意该函数遇到\n或缓冲区满会结束读取；

fputs();

int fputs(const char \*s, FILE \*stream);

用法：将s所指的内存单元将字符输出到stream所指流中，遇到NULL截止；

fread函数和fwrite函数

fread函数和fwrite函数处理的是一个数据块

其形式为：

size\_t fread(void \*ptr, size\_t, size, size\_t nmemb, FILE \*stream);

fread(buffer, size, count, fp);

size\_t fwrite(const void \*ptr, size\_t, size\_t nmemb, FILE \*stream);

fwrite(buffer, size, count, fp);

解析：

buffer:是一个指针，指向数据所存放的内存单元

size：表示要读写的内存单元大小（字节数）

count:表示要读写多少个size字节的数据项

fp:文件指针

fread(buffer, size, count, fp);//读，将fp所指文件内容读取count个size大小的字节到buffer中，返回值是成功读取文件的字节块数即count，如果返回0则表示读取数量不够size大小；

fwrite(buffer, size, count, fp);//写，将buffer中的数据写count个size大小的字节到fp所指的文件中，返回值是实际写入数据块的数目；

fseek();文件定位函数：

int fseek(FILE \*stream, long offset, int whence);

stream：定位文件；

offset：偏移多少个字节，可以是负数（表示向前偏移）；

whence：偏移的起始位置：

SEEK\_SET：文件头开始；

SEEK\_CUR：当前位置；

SEEK\_END：文件尾部；

返回值：0或-1；

ftell();//取得当前文件的位置，成功返回当前文件位置，失败返回NULL；

fprintf(文件指针, 格式字符串, 输出列表);

fscanf(文件指针, 格式字符串, 输入列表);

列：

fprintf(fp, ” %d, %3.5d”, i, t);//将i, t的值输出到fp所指的文件

fscanf(fp, ” %d, %d”, &i, &t);//将fp所指文件中的数据依次赋给i, t；

1. 文件I/O：也叫非缓冲I/O

非缓冲：是指每个读或写都调用内核中相应的系统调用，无缓存；

文件I/O描述符：

对于内核而言，所有打开文件都由文件描述符引用，文件描述符是一个非负整数，当打开或创建文件时，内核都会向进程返回一个文件描述符，那么就可以通过这个整数对文件进行访问。

例如：标准输入输出和标准出错文件（描述符分别是0（STDIN\_FILENO）、1（STDOUT\_FILENO）、2（STDERR\_FILENO）），打开文件是顺序编号的从3开始；

文件打开I/O：

int open(const char \*pathname, int flags);

int open(const char \*pathname, int flags, mode\_t mode);

int creat(const char \*pathname, mode\_t mode);

pathname：指向将要打开文件的路径字符串，

mode：被打开文件的权限掩码（8进制表示），

flags：操作方式参数如下，返回值是该文件的描述符（一定是最小的未使用的描述符数值）：

|  |  |  |
| --- | --- | --- |
| flags | 作用 | 备注 |
| O\_RDONLY | 只读方式打开文件 | 互斥 |
| O\_WRONLY | 只写方式打开文件 | 互斥 |
| O\_RDWR | 读写方式打开文件 | 互斥 |
| O\_CREAT | 创建新文件 | 如果不存在，就用第三方参数设置权限 |
| O\_EXCL | 检测文件是否存在 | 存在返回错误信息 |
| O\_NOCTTY | 终端控制参数 | 表示终端不可以作为系统调用的进程终端 |
| O\_TRUNC | 打开删除源文件 | 如果打开的文件已存在，则删除原文件 |
| O\_APPEND | 以追加的方式打开 | 打开文件，并且在末尾追加 |

提示：添加多个权限用”|”表示（O\_RDONLY | O\_CREAT）

open()，可以打开设备文件，但不能创建设备文件，设备文件的创建必须使用mknod()函数；

返回值：成功返回文件描述符，失败返回-1，并设置errono；

create()函数：

int creat(const char \*pathname, mode\_tmode);//指定路劲创建文件并设置权限（tmode）；

close函数：

int close(int fd);//关闭对应文件描述符代表的文件

read()函数：

ssize\_t read(int fd, void \*buf, size\_t count);

fd：文件描述符，被读文件；

count：一次读取的字节数；

返回值：成功返回实际读取的字节数，返回0，表示文件末尾，-1表示错误，并设置errno，另外文件读写时，文件内部指针会随之移动；

write()函数：

ssize\_t write(int fd, const void \*buf, size\_t count);

fd：存放读取的数据的文件的描述符；

count：一次读取的字节数；

buf：被读的字符地址；

返回值：成功读取的字节数，失败返回-1，并设置erron；

write出现常见原因，磁盘满或者超过了一个更定的文件长度限制，对于普通文件，写操作从文件当前位移量开始，如果在打开文件时，指定了O\_APPEND（追加）参数，则每次写操作前，将文件位移量设置在文件结尾处，在一次写成功后，该文件的位移量增加实际写的字节数。

lseek()文件定位函数：对读普通文件有效

off\_t lseek(int fd, off\_t offset, int whence);

fd：文件描述符；

ofset：偏移量，每次读写操作所需移动的字节大小距离，可以为负数；

whence：定位基点取值如下

SEE\_SET：当前位置为文件的开头，新位置为偏移量的大小；

SEEK\_CUR：当前位置为文件指针的位置，新的位置为当前位置加上偏移量；

SEEK\_END：当前位置为文件的结尾，新的位置为文件的大小加上偏移量的大小；

返回值：成功返回文件的当前位移，-1表示出错；

获取文件属性与修改函数：

stat()：

int stat(const char \*filename, struct stat \*buf);

filename：文件名；

buf：将文件状态复制到buf所指的结构体变量中；

返回值：成功返回0，失败返回-1，并设置errno；

fstat()：

int fstat(int fildes, struct stat \*buf);

fildes：文件描述符：

buf：将文件描述符代表的文件状态复制到buf所指的结构体变量中；

lstat()

getpwuid();

struct passwd \*getpwuid(uid\_t uid);

uid：用户的uID

返回值：用户的name；

getgrgid(gid\_t gid);//作用上同

目录操作相关函数：<sys/types.h> <dirent.h>

opendir()：打开目录函数

DIR \* opendir(const char \*name);

返回值：成功返回DIR \* 型目录流；失败返回NULL；

closedir()：关闭目录

int closedir(DIR \*dir);

dir：指向目录流的指针；

返回值：0或-1；

readdir()：读取目录

struct dirent \*readdir(DIR \*dir);

返回值：指向struct dirent 变量的指针结构体如下

struct dirent

{

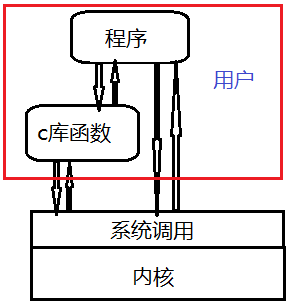
……

har d\_name[256];//文件名

}

chdir()

库调用和系统调用：



系统调用：

是用户空间进程访问内核的接口，提高系统的安全性，使得代码的可移植性提高；

库函数：是为了实现某个功能而封装起来的API集合，提供统一的编程接口，更加便于程序的移植；

静态库和动态库：

库：本质上是一种可执行代码的二进制形式，可以被操作系统载入内存中执行；

静态库：是程序在编译时，被连接到目标文件中，成为目标文件的一部分，程序在执行时，就不在需要调用该静态库，因此数据比较大；但移植性较高；

动态库：在程序编译时不会将动态库连接到目标文件中，而在程序运行时才会载入，因此程序每次执行时都需要再次载入动态库，所以目标文件较小；可移植受不同操作系统库不同而影响；但升级方便，可以实现进程之间的资源共享；

静态库的创建：ar命令

1. gcc –c file.c --->生成file.o；
2. ar crs libfile.a file.o --->libfile.a（静态库文件 lib .a）
3. gcc test.c –L.（指定库路径.表示当前路劲）-lfile（指定库名file） -o test
4. 那么test可执行文件就包含了该静态库的内容了；

动态库（共享库）的创建：

1. gcc –fPIC –Wall –c file.c ---->file.o//-fPIC表示加载的时候不考虑程序的存在位置（路径）；
2. gcc –shared –o libfile.so file.o
3. 使动态库全路径有效：

方式1：将其安装在/lib或/usr/lib下（注意可能会产生垃圾文件）

gcc main.c –lfile –o main.o

方式2：在/etc/ld.so.conf.d/my.conf文件中添加该库的绝对路径；并运行ldconfig（刷新加载），该命令会重建/etc/ld.so.conf.d/my.conf

方式3：设置环境变量（少用）

1. export LD\_LIBRARY\_PATH=绝对路径
2. gcc main.c –L绝对路径 –lfile –o main.o

模块二、进程和线程：

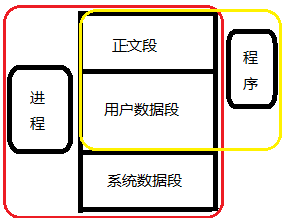
1. 进程：

进程是一个独立的可调度的任务，是一个抽象体，是程序执行一次的过程；

进程和程序的区别：

程序是静态的，是保存在磁盘上的，指令的有序结合；

进程是动态的，是程序执行的过程，包括创建、调度和消亡；进程是程序执行和资源管理的最小单位；



进程不仅是包括程序的指令和数据，而且包括程序计数器值，CPU的所有寄存器值以及存储临时数据的进程堆栈。

系统数据段：包括进程控制块、CPU寄存器值和堆栈；

进程控制块（pcb）：

1）进程的标识：进程号（是唯一的标识一个进程）

PID：进程号（getpid()）

PPID：父进程（getppid()）

2）进程用户；

3）进程状态、优先级；

4）文件描述符表；

进程的数据段：全局变量，常数以及动态开辟的数据空间；

正文段：存放程序代码的代码段，只读；

堆栈段：存放函数返回地址，函数的参数以及程序中的局部变量；

进程的三大类：

1. 交互进程：由shell控制和运行，可以再前台运行，也可以在后台运行（服务进程）；
2. 批处理进程：该类进程不属于某个终端，它被提交到一个队列中进行顺序执行；
3. 守护进程：该类进程在后台运行，一般在Linux启动时开始执行，直到系统关闭；

进程的四种运行状态：

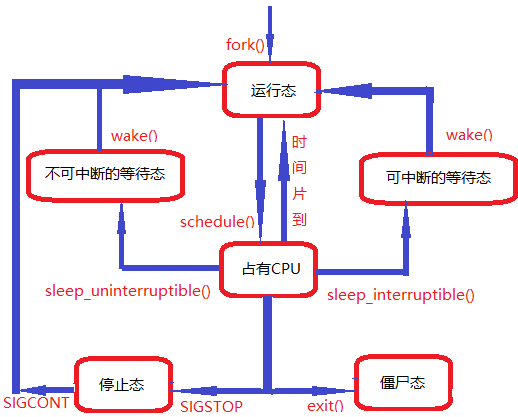
1. 运行态：正在运行或准备运行的进程；
2. 等待态：此时进程等待一个事件的发生，或某种系统资源的到来

可中断：指可以被打断的进程

不可中断：

1. 停止态：被中止的进程；
2. 僵尸态：已经终止的进程，但该进程在内存中占有一定空间；（一般由父进程处理）

进程运行状态图：



进程的模式：

用户模式：

内核模式：

进程的启动：

手动启动：

调度启动命令：（详情节Linux学习笔记）

系统调用函数：

fork()创建进程：<sys/types.h><unistd.h>/vfork();

pid\_t fork(void);

返回值：

0：子进程（表示该函数在子进程空间中执行的结果）；

大于0的整数：父进程（表示该函数在父进程空间中执行的结果）；

-1：出错，并设置erron；

作用：创建子进程，子进程会拷贝父进程的空间；但两片空间是相互独立的，互不影响包括变量等（同名，但是两个不同的地址空间）；

当创建子进程的时候一般是父进程先运行，但两个不能确定谁先运行；

若父进程先结束：

1. 若父进程先结束，子进程变成孤儿进程，被init进程收养；
2. 子进程会变成后台进程；

若子进程先结束：

若父进程没有及时回收，子进程变成僵尸进程；

子进程执行情况：

1. 子进程会从fork()函数的下一条语句开始执行（由操作系统设置入口）
2. 父子进程不能确定那个进程先执行，因为这是两个不同的空间，由操作系统调度；
3. 父进程可以多次调用fork()函数，创建多个子进程（但要主意回收），子进程也可以调用fork()函数，创建孙进程；

exit()/\_exit()进程结束函数：<stdlib.h>/<unistd.h>

void exit(int status);/void \_exit(int status);

结束当前进程并将status返回；

exit()结束进程时会先刷新流缓冲区，而\_exit则不会，故少用；

参数：0表示正常结束，非0表示非正常结束；

返回值：可以用wait接收处理；

exec函数族：<unixstd.h>

作用：

进程调用exec()函数族执行某个程序；

进程当前内容空间被指定的程序替换；

实现让父进程执行不同的程序；

使用步骤：

1. 父进程创建子进程；
2. 子进程调用exec()函数族；
3. 父进程不受影响；（例如我们的shell调用）

exec函数族提供了一种在进程中启动另外一个程序执行的方法。可以根据指定的文件名或路径找到可执行文件（可执行文件可以是二进制文件或Linux下的脚本文件），并用他来取代原调用进程的数据段、代码段和堆栈段。在执行完之后，原调用进程的内用除了进程号外，其他全部都被替换了；

exec函数族的使用：

当进程认为自己不能再为系统和用户做出任何贡献了时就可以调用exec函数，让自己执行新的程序；

如果某个进程相同时执行另外一个程序，他就可以调用fork函数创建子进程，然后在子进程中调用任何一个exec函数。这样看起来就好像通过执行应用程序而产生了一个新进程一样。

函数族：

1. execl()/execlp()：<unistd.h>

int execl(const char \*path, const \*arg, …);//以列表的方式

path：可执行文件的路劲；

arg, …：表示命令行参数的选项（”-l”, “-a”等）argv[0],…，最后一个必须是NULL表示结束；

返回值：成功没有返回值，失败返回-1，设置erron；

如：execl(“/bin/ls”, “ls”, “-a”, “-l”, “/etc”, NULL);

int execlp(const char \*file, const char \*arg, …);

file：可执行文件名，该函数会在PATH环境变量中自动查找符合的file；

arg,…：将第二个以后的 参数当做命令行参数，最后一个NULL；

返回值：上同；

execv()/execvp：

int execv(const char \*path, char \*const argv[]);//将执行方式放在指针数组中

int execvp(const char \*file, char \*const argv[]);//p表示环境变量（自动寻找路劲）

作用同上，只不过将参数放在参数argv[]，以NULL结尾；

excve()/execle();//e 表示环境变量；

int execle(const char \*path, const char \*arg, …, char const envp[]);

int execve(const char \*path, const char \*arg[], char const envp[]);

envp[]：表示环境变量；

system()函数：<stdlib.h>

int system(const char \*command);

作用：该函数会调用fork()函数产生子进程，有子进程来调用/bin/sh-c，父进程会等待子进程执行完毕才继续执行；

command：代表执行的命令；

返回值：

注意调用exec函数族后，本进程的空间将全部覆盖，所以此时的内存中的数据和父进程中的数据完全不同；

进程的回收：

wait()函数(等待子进程中断或结束)：<sys/types.h><sys/wait.h>

pid\_t wait(int \*status);

说明：该函数会暂时阻塞当前父进程的执行，直到有信号或子进程结束，若子进程没有结束，父进程一直阻塞，若有多个子进程，先结束的就先回收；

status：子进程结束状态值（调用函数会设置）可设置为NULL（忽略，表示直接释放子进程PCB），

返回值：成功返回结束子进程的PID，错误返回-1，并设置erron；

进程结束方式和返回值：

子进程通过exit()、\_exit()、return ;返回某个值（0~255）只有第八位有效；（正常结束）

exit：会刷新缓冲区，而\_exit直接销毁整个空间；

父进程调用wait(&status);回收；

pid\_t wait(int \*status);

返回值：子进程pid；

status：

WIFEXITED(status);//判断进程是否正常结束

WEXITSTATUS(status);//获取子进程返回值；

WIFSIGNALED(status);//判断子进程是否被信号结束（非正常结束）；

WTERMSIG(status);//获取子进程的型号类型；

NULL：表示不关心直接回收；

waitpid()函数；

pid\_t waitpid(pid\_t pid, int \*status, int options);

pid：用于指定确定的子进程的进程号（>0），也可以设置为（-1）回收任意子进程，以及设置等待的方式；

status：用来记录子进程结束状态；同上（NULL）

返回值：成功返回子进程的进程号，0表示正在回收的子进程还没有结束，失败返回EOF；

options：指定回收方式，0（等价于wait()）或WNOHANG（表示非阻塞回收）

守护进程（Daemon）：

特点：

1. 在系统启动时运行，系统关闭时结束；
2. 始终在后台运行（后台进程无法从终端输入）；
3. 独立于任何终端（不受终端控制和影响）；
4. 周期性执行某种任务或等待处理特定的事件；

Linux以会话（session）、进程组方式管理进程：

进程组：每个进程都对应一个进程组；

会话：一个或多个进程组的结合，通常打开一个终端，就会创建一个会话，一个终端中运行的所有进程都属于这个会话，当该终端结束时，所有的相关进程都会结束；

守护进程的创建：

1. 创建子进程，退出父进程；

让子进程变成孤儿进程，被init收养，并且子进程在后台运行；（但子进程依然依附于终端，但不受控制）

fork();//创建exit(0);//退出

1. 在子进程中创建新的会话；

setsid();//创建会话，

子进程成为新会话组长；

子进程脱离终端；

pid\_t setsid(void);

返回值：成功返回该会话组的组长，失败返回-1，设置erron

1. 修改当前工作目录为根目录；

chdir(“/”);或chdir(“/tmp”);//两者只是权限不同；

守护进程一直在后台运行，使其工作目录不能被卸载；

重新设定当前工作目录cwd；

int chdir(const char \*path);

path：路径如 ”/tmp”

返回值：成功0，失败-1，设置erron；

1. 修改当前稳定建权限掩码；<sys.types.h><sys/stat.h>

mode\_t umask(mode\_t mask);

umask(0);//设置成0，只影响当前进程；

返回值：总是会成功，不需要；

1. 关闭打开的文件描述符：i = getdtablesize();//最大打开文件数目

close(i--);

关闭从父进程继承的打开的文件，脱离终端（无法再使用（0,1,2标准文件））；

1. 线程：

进程的不足：进程的地址空间是私有的，所以在切换时消耗CPU比较大；

线程：指的是共享地址空间的多个任务（本质也是一个进程）；

所以在Linux中不区分进程和线程；

线程共享的资源：

可执行的指令、静态数据、进程中打开的文件描述符、x信号处理函数、当前工作目录、用户ID和组ID；

线程私有的资源：

线程ID；

pc程序计数器，和相关寄存器；

堆栈（局部变量、返回地址）；

错误号，信号掩码和优先级；

执行状态和属性；

多线程编程：<pthread.h>

1. 创建线程：（NPTL线程库）

int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \* (routine)(void \*), void \*arg);

返回值：创建成功返回0，失败返回错误码；

thread：线程对象，创建的线程id，由该函数返回；

attr：指定线程的属性，NULL表示缺省属性；

routine：线程执行的函数，直接传该类型的函数名；

arg：传递给线程执行的函数参数（可以是NULL）；

线程编译：gcc file.c –lpthread –D\_REENTRANT–o file

-D\_REENTRANT：生成可重入代码，保证数据的完整性；

1. 线程回收：

void pthread\_exit(void \*retval);//线程退出函数

retval：返回线程的值；

int pthread\_join(pthread\_t thread, void \*\*value\_ptr);

//等待线程退出，并接收退出值

thread：等待要回收的线程；

value\_ptr：指针\*value\_ptr指向线程返回的参数；

返回值：成功返回0，失败返回错误码；

非阻塞线程回收函数：

int pthread\_detach(pthread\_t thread);//该函数使当前线程与进程分离（不表示不依赖当前进程），当该线程结束后，交由系统自动回收该线程空间，达到不阻塞的效果；

thread：线程号；

返回值：成功0，失败设置errornumber；

1. 控制线程：

value\_ptr：线程退出时返回的值，可被pthread\_join获取；线程私有资源全部释放；

int pthread\_cancel(pthread\_t thrad);//取消一个进程

thread：取消的进程的进程号；

返回值：成功0，失败错误号；

线程编译 –lpthread –D\_REENTRANT /、-D 表示可重入

线程同步和互斥、线程间通信：

同步：指多个任务（线程）按照一定的顺序完成一件事情；

同步机制：信号量（也叫信号灯，是一个非负整数）决定线程的运行与阻塞；

信号量：也是一类资源，分为二值信号量（0、1）和计数信号量（1~n）；

信号量是一个受保护的变量，只能通过三种操作来访问：

1. 初始化；
2. p操作（申请资源）sem\_wait …… sem – 1；
3. v操作（释放资源）sem\_post …… sem + 1；

p/v操作含义如下：

p(s)：

if (sem > 0){

申请资源的任务继续运行;

sem --;

}else{ 申请资源的任务阻塞 }

v(s)：

if (没有任务在等待该资源){

sem ++;

}else { 唤醒第一个等待的任务，继续运行 }

Posix信号量：

有名信号量；

无名信号量（基于内存的信号量）：

pthread库中常用信号量操作函数：<semaphore.h>

int sem\_init(sem\_t \*sem, int pshared, unsigned int value);//信号量初始化

sem：产生并指向要初始化的信号量；

pshared：0表示线程间，1表示进程间；

value：信号量的初始值；

返回值：成功返回0，失败返回-1，设置erron；

int sem\_wait(sem\_t \*sem);//p操作

int sem\_post(sem\_t \*sem);//v操作

value：信号量初始化值（一般是资源的数量）；

返回值：0表示成功，-1表示失败并设置erron；

sem：指向要操作的信号量对象；

线程间互斥：

临界资源： 一次只允许一个任务（进程、线程）访问的共享资源；

临界区：访问同一个临界资源的代码；

互斥机制：

互斥锁（mutex）：

用来保护临界资源，保证任何时刻，最多只能有一个任务（线程）访问该资源；

线程必须先申请获得互斥锁才能访问临界资源，访问完毕释放该锁。若无法获得锁，线程会一直阻塞；

相关API：<pthread.h>

int pthread\_mutex\_init(pthread\_mutex\_t \*mutex, pthread\_mutex\_t \*attr);//初始化互斥锁

mutex：待初始化的互斥锁名；

attr：初始化后互斥锁的属性；（NULL表示缺省）

返回值：成功返回0，失败返回-1；

int pthread\_mutex\_loc(pthread\_mutex\_t \*mutex);//申请互斥锁（加锁）；

mutex：互斥锁；

返回值：成功0，失败返回错误码

若无法获得锁，会进入阻塞；

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutext);//释放锁；

mutex：指向待解锁的互斥锁；

返回值：成功0，失败返回错误码；

加锁的临界资源使用完毕一定要释放；

1. 进程间通信：
2. 无名管道：（pipe）

只能用于有亲缘关系的进程之间通信；（无名管道存在于内存中，由进程创建，所以只能用于有亲缘关系的进程）；

是半双工的通信模式，具有固定（也就是一个进程只有一个状态）的读端和写端，即对应的读和写描述符；

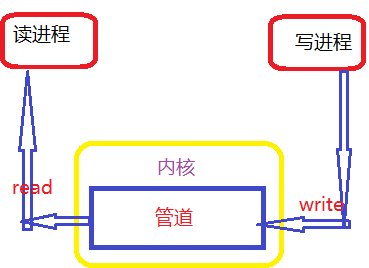
管道可以看成特殊的文件，读和写使用文件IO；

int pipe(int pfd[2]);//创建无名管道<unistd.h>

pfd：保存无名管道的相应两个文件描述符，pfd[0]表示读，pfd[1]用于写管道；（注意：管道中的内容读取后会消失），并且是先进先出；

返回值：成功返回0，失败返回EOF；

无名管道通信图：



读无名管道：

（1）写端存在时：

当管道中无数据时，读操作会阻塞；

有数据时，使用read读取实际的字节数；

当写端关闭，读端才会读到0；

（2）写端不存在时：

有数据，使用read读取实际字节数；

无数据，read直接返回0，不会进入阻塞；

写无名管道：

（1）读端存在：

有空间时，write返回实际写入的字节数，若空间不足，不保证原子操作，多余的数据将进入等待状态（阻塞），直到空间有才继续写操作；

（2）读端不存在：管道断裂；（SIGPIPE信号）

无名管道的大小：（默认64k）

1. 有名管道：（fifo）

有名管道的特点：

对应管道文件（创建与磁盘上存在，但内容存在于内存中，文件中不可见），可以用于任意进程之间的通信；

有名管道的打开可以指定操作方式（读或写）；

注意：管道中的内容是存放在内存中的，所以管道文件大小始终为0；

相关函数：<unistd.h><fcntl.h><sys/types.h>

int mkfifo(const char \*path, mode\_t mode);//创建有名管道文件；

path：创建的有名管道文件路径；

mode：管道文件权限（默认0666）；

返回值：成功返回0，失败返回EOF（已存在时）；

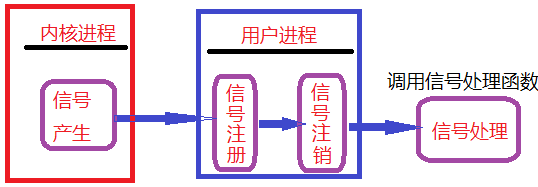
1. 信号通信：

信号是在软件层次上对中断机制的一种模拟，是一种异步通信方式；

信号可以直接进行用户空间进程和内核进程之间的交互，内核进程也可以利用它来通知用户空间进程发生了那些系统事件；

如果该进程当前并未处于执行状态，则该信号就由内核保存起来，直到该进程回复执行在传递给它；如果一个信信号被进程设置为阻塞，则该信号的传递会被延迟，直到其阻塞被取消才被传递给进程；

信号的生存周期图：



进程对信号的处理：

（1）缺省方式：Linux对每种型号都规定了默认操作；

（2）忽略信号：对信号不做任何处理，但有两个信号不能忽略（SIGKILI和SIGSTOP）；

（3）捕捉信号：定义信号处理函数，当信号发生时，执行相应的处理函数；

常用通信信号宏表（可使用kill –l查看）；

|  |  |  |
| --- | --- | --- |
| 信号名 | 含义 | 默认操作 |
| SIGHUP | 通常是在终端的控制进程结束时，通知同一会话内的各个作业与控制与终端断连 | 终止 |
| SIGINT | 通常是用户键入Ctrl+c时发出信号，终端驱动程序发送该信号，送到前台进程中的每个进程 | 终止 |
| SIGQUIT | 与SIGINT类似，Ctrl+\控制 | 终止 |
| SIGILL | 该信号在进程要执行非法指令时（如段错误，）发出； | 终止 |
| SIGFPE | 该信号在发生致命的算术错误时发出，包括溢出的算术错误 | 终止 |

发送信号：<unistd.h><signal.h><sys/types.h>

int kill(pid\_t pid, int sig);

pid：大于0，接收信号的进程号；0代表同组进程，-1代表所有进程；

sig：发送的信号类型（宏）；

返回值：成功返回0，失败返回EOF；

int raise(int sig);//向本进程发送信号；原理上同；

定时器（闹钟函数）：

int alarm(unsigned int seconds);//设定定时器，通常用于超时检测

seconds：定时器的时间（/秒），0表示取消当前定时器；

返回值：成功返回上一个定时器剩余的时间（可以设置多个闹钟）但最终起作用的是最后设置的闹钟，失败返回EOF；

注意：一个进程中只能设置一个定时器，时间到时产生SIGALRM;

int pause(void);//让进程进入阻塞

使进程一直进入阻塞，直到被信号中断；

被信号中断后返回-1，并设置errno为EINTR；

设置信号响应捕捉方式：<unistd.h><signal.h>

特定的信号是与相应的事件联系；

一个进程可以设定对信号的处理方式：主要有两种

1. signal()函数；

void (\* signal(int signum, void (\*handler) (int))) (int);

signum：设置信号类型；

handler：指定的信号处理函数：SIG\_DFL代表缺省方式；SIG\_IGN代表忽略信号；以及自定义信号处理函数指针；

返回值：成功设置之前的信号处理方式，出错返回-1；

（2）使用信号集函数

IPC对象：(ipcs –m)（system(“ipcs -m”);//代码中查看）

1. 共享内存：是一种最为高效的进程间通信方式，，进程可以直接读写内存，不需要任何数据的拷贝；在多个进程间交换信息时，内核会留出一块内存区，可以由需要访问的进程将其映射到自己的私有空间；进程可以直接读写该区域，由于多个进程共享一段内存，所以也需要依靠某种同步机制，如互斥锁、信号量等；

共享内存的实现步骤：

1. 创建/打开共享内存；
2. 映射共享内存，即吧指定的共享内存映射进程的地址空间用于访问；
3. 撤销共享内存映射；
4. 删除共享内存对象；

实现函数：<sys/types.h><sys/ipc.h><sys/shm.h>

int shmget(key\_t key, int size, int shmflg);//创建（打开）共享内存段；

key：IPC\_PRIVATE（0）或ftok的返回值；

size：共享内存区的大小；

shmflg：同open函数的权限位，一般用8进制表示（IPC\_CREAT|0666、IPC\_EXCL）；

返回值：成功返回共享内存段标识符（id）、出错返回-1；

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

pathname：文件路径名；

proj\_id：项目号（内存段的编号），设置不同的整数值（0~255）；

返回值：key的值；

void \*shmat(int shmid, const void \*shmaddr, int shmflg);//映射共享内存到自己的进程空间；

shmid：映射的共享内存区标识符；

shmaddr：将共享内存映射到指定地址（直接设置NULL，则表示由系统自动完成映射）；

shmflg：SHM\_DONLY：只读共享内存；

默认为0：表示可读可写；

返回值：成功映射后的地址；失败返回(char \*)-1；

int shmdt(const void \*shmaddr);//解除映射

shmaddr：共享内存映射后的地址；

返回值：成功为0，失败-1；

int shmctl(int shmid, int cmd, struct shmid\_ds \*buf);//删除共享内存；

shmid：要操作的共享内存标识符；

cmd：IPC\_STAT（获取对象属性）、IPC\_SET（设置对象属性）、IPC\_RMID（删除对象）；

buf：指定IPC\_STAT/IPC\_SET时用以保存/设置属性（直接NULL）；

返回值：成功0，失败-1（errornumber）；

1. 消息队列：（ipcs -q）

消息队列是IPC的一种;

消息队列由唯一的ID来标识；

消息队列就是一个小时的列表（动态队列），用户可以在消息队列中添加消息、读取消息等；

消息队列可以按照类型来发送/接收消息（不同于普通队列的插入删除）；

操作方式：创建打开、读取、添加、控制消息队列；

操作函数：加<sys/msg.h>

int msgget(key\_t key, int flag);//创建（打开）消息队列

key：消息队列关联的key值（ftok）；

flag：消息队列的访问权限（IPC\_CREAT|0666）；

返回值：成功队列ID，出错-1；

int msgsnd(int msqid, const void \*msgp, size\_t size, int flag);//添加消息

msqid：消息队列的ID；

msgp：指向消息的指针；结构如下：

struct msgbuf{

long mtype;//消息类型

char mtext[N];//消息正文

};

size：发送消息正文的大小（sizeof(.mtext)；）；

flag：IPC\_NOWAIT消息没有发送完成，函数也会立即返回；（可以是0阻塞）

返回值：成功0，否-1；

int msgrcv(int msgid, voi \*msgp, size\_t size, long msgtype, int flag);//接收消息，会将队列中的消息删除

msgid：消息队列的ID；

msgp：接收消息的缓冲区；

size：要接收的消息的字节数；

msgtype：0按顺序接收消息队列中的消息；

大于0：只接收消息队列中第一个类型为msgtyp的消息；

小于0：接收消息队列中类型值不小于msgtyp的绝对值且类型值又最小的消息（也是从该大小依次递增接收）；

flag：0表示阻塞，IPC\_NOWAIT表示非阻塞；

返回值：成功返回接收到的消息长度，失败返回-1；、

int msgctl(int msgqid, int cmd, struct msgqid\_ds \*buf);//销毁消息队列

msgqid：消息队列的id

cmd：

IPC\_STAT：读取消息队列的属性，将其保存在buf所指的缓冲区；

IPC\_SET：设置消息队列的属性，有buf参数提供；

IPC\_RMID：从系统中删除消息队列；

buf：（NULL）消息队列缓冲区；

返回值：0或-1；

信号灯集：（也是ipc对象（ipcs -s））

信号灯也叫信号量，是不同进程间或给定进程内部不同线程间同步的机制；

信号灯的种类：

有名信号灯：posix；

无名信号灯：posix基于内存的信号灯；

v信号灯：system（ipc对象）：

System V的信号灯是一个集合，每一个灯都单独计数，而posix是指单个计数信号灯；（v信号灯由内核维护）

计数信号灯、二值信号灯；（表示自愿是否可用）

操作函数：<sys/types.h><sys/ipc.h><sys/sem.h>

int semget(key\_t key, int nsems, int semflag);//创建信号灯集

key：和信号灯集相关联的key值；

nsems：灯集的个数；

semflag：灯集的访问权限，（IPC\_CREAT|0666）；

返回值：成功（灯集的ID），失败-1；

int semop(int semid, struct sembuf \*opsptr, size\_t nops);//pv操作

semid：灯集ID；

opsptr：p/v操作；

struct sembuf{

short sem\_num;//要操作的信号灯编号

short sem\_op; //=1：v操作，释放资源

//=-1：p操作，申请资源

//=0：表示等待，直到信号灯的值变成0

short sem\_flg; //0（阻塞），IPC\_NOWAIT（非阻），SEM\_UNDO （防止死锁，自动释放锁）}

nops：要操作的信号灯个数（1）；

返回值：0，失败-1；

int semctl(int semid, int semnum, int cmd…/\*union semunarg\*/);//信号初始化（要用到后面参数），与删除信号灯（不管后面参数）；

semid：信号灯集ID；

semnum：要修改的信号灯的编号；

cmd：

GETVAL：获取信号灯的值；

SETVAL：设置信号灯的值；

IPC\_RMID：删除信号灯集；

/\*union semunarg\*/：union semun{

int val;//初始值可用的资源

… //不管}

DSP：

1. 套接字：

模块三、Linux网络编程：

1. Internet与TCP/IP协议：

TCP：用来检测网络传输中差错的传输控制协议；

IP：专门负责对不同网络进行互联的互联网协议；

1）、OSI模型的七层结构与TCP/IP协议族：

|  |  |  |
| --- | --- | --- |
| OSI中的层 | 功能 | TCP/IP协议族 |
| （7）应用层 | 文件传输，FTP、E-mail、文件服务、虚拟终端 | TFTP、HTTP（超文本传输）、DNS等 |
| （6）表示层 | 数据格式化、代码转换、数据加密 | 没有协议 |
| （5 ）会话层 | 解除或建立别的节点的联系 | 没有协议 |
| （3）传输层 | 提供端对端的接口（出错处理、流控） | TCP、UDP |
| （4）网络层 | 为数据包分组、选择路由 | IP、ICMP、RIP、等 |
| （2）数据链路 | 传输有地址的帧以及错误检测功能将数据组成可发送、接收的帧 | SLIP、PPP、QRP、MTU等 |
| （1）物理层 | 以二进制数据形式在物理媒体上传输数据（物理信号、接口等） | ISO2110、IEEE802.1、EEE802.2 |

2）、TCP/IP常用协议族：

TCP：传输控制协议；

IP：网际协议；

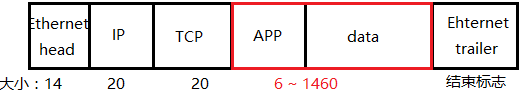
UDP：用户数据报协议；

HTTP：超文本传输协议；

FTP：文件传输协议；

数据的封装：依次加头、倒叙解头（每个头都有原地址和目标地址）

TFP头（应用层）、TCP头（传输层端口）、IP头（网络层）、Ethernet以太网头（网络接口与物理层）

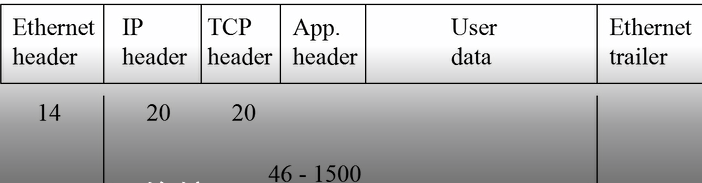


1. TCP协议与UDP协议：

1）、TCP协议：

传输控制协议，特点：是一种面向连接的传输层协议，提供高可靠性（即数据无误、无丢失、无失序、无重复到达的通信）；

分包与粘包：（一次性发送包的大小6-146）



分包：将一个比较大的包分成多分发送；

粘包：将多个小包一起发送；

造成问题：数据错乱；（解决方法：数据包前添加一个信息头，说明大小与类型，以及尾端添加校验码）

2）、UDP协议：

用户数据传输协议，特点：是不可靠的无连接协议，在数据发送前不需要连接，所以可以进行高效率的数据传输；

两者的异同：

同：都是传输层协议；

异：TCP可靠，有连接；UDP不保证可靠，无连接；

1. TCP/IP网络编程基础：

1）、Socket：是一个接口（即一个函数）；是一种特殊的文件描述符，支持有连接（TCP/IP）和无连接（UDP）；他独立于体系协议的网络编程接口（通用性）；

Socket类型：

流式套接字（SOCK\_STREAM），用于TCP协议；

数据报套接字（SOCK\_DGRAM），用于UDP协议；

原始套接字（SOCK\_RAW），用于底层协议；

2）、IP地址：是Internet中主机的标识；

IPV4：IP地址是32位的（4字节192.168.2.85）；

IPV6：IP地址是128位的；

每个数据报都携带有目的IP和源IP，路由器依靠此信息为数据包选择路由；

IP地址分类：A~E类分类由第一个字节最高位区分是0为A、10为B、110为C（11000000-192）（我们使用的就是C类）……

子网掩码：

网络号：将IP地址与子网掩码相与(192.168.2.88 & 255.255.255.0 = 192.168.2)

主机号：将子网掩码取反相与ip（0.0.0.255 & 192.168.2.88 = 88）

主机IP地址小端序转换成网络字节序：

inet\_aton()：将strptr指向的字符串（点分字符串）转换成32位的网络字节序二进制值，返回转换后的地址；

头文件：# include <arpa/inet.h>

原型：int inet\_aton(const char \*strptr, struct in\_addr \*addrptr);

inet\_addr();

原型：in\_addr\_t inet\_addr(const char \*strptr);//将本机点分4字节IP地址转换成网络字节序（大端序）

inet\_ntoa();

原型：char \*inet\_ntoa(struct in\_addr inaddr);// 将32位（4字节）的网络字节序转换成十进制的点分字符串；大端转点分；

3）、端口号：标记进程（运行程序）相绑定的编号，（即用来将接收到的数据包交给对应端口号的进程）

端口号一般由IANA管理：

众所周知端口：1~1023（通常被系统占用）；

已登记端口：1024~49151（可以使用）；

动态或私有端口：49152~65535（可以使用）；

4）、字节序：不同类型的CPU中，内存存储多字节整数序列有两种方法，称为主机字节序（HBO）；

小端序：低序字节存储在低地址单元，高字节序存储在高字节单元（Intel、AMD就是采用该方式）；

大端序：髙序字节存储在低地址单元，低序字节存储在高地址单元（ARM、Motorola使用该存储方式）；

网络字节序：网络数据传输就是用的是大端序；所以整型数据由应用进程送入socket之前，需要将其转换成网络字节序（大端序），而反之从网端取出转换成小端序；

网络字节序（NBO-Network Byte Order）；

主机字节序（HBO-Host Byte Order）；

字节序转换函数：

主机端口号转网络字节序：

u\_long htonl(u\_long hostlong);//4字节转换

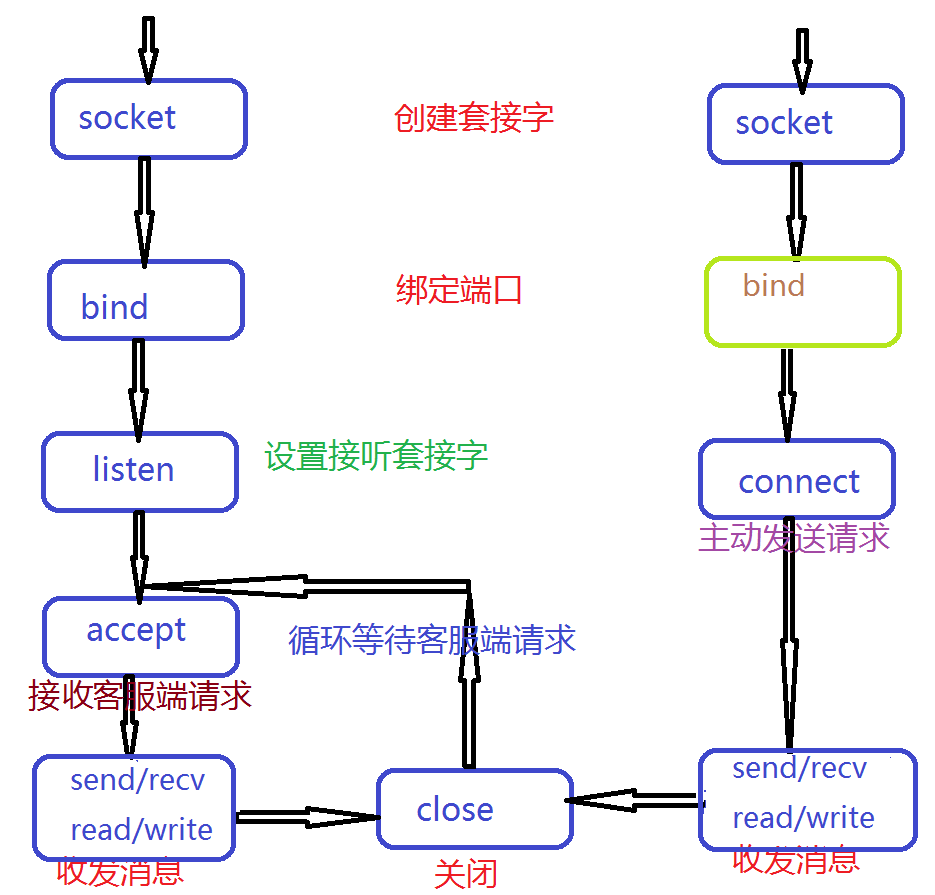
u\_short htons(u\_short short);//将端口号2字节转换为网络字节序

网络转主机：

u\_long ntohl(u\_long hostlong);

u\_short ntohs(u\_short short);

5）、TCP服务端与客服端搭建流程图



网络编程相关API：

编程可能用到的头文件：

# include <stdio.h>

# include <stdlib.h>

# include <unistd.h>

# include <string.h>

# include <sys/types.h>

# inlude <sys/select.h>

# include <sys/socket.h>

# include <netinet/in.h>

# include <arpa/inet.h>

# include <time.h>

1）、socket() 创建套接字：

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

domain：地址族 AF\_INET或（PF\_INET）ipv4协议或Internet协议；

type：套接字类型：

SOCK\_STREAM //流式套接字

SOCK\_DGRAM //数据报套接字

protocol：通常直接置0；

返回值：成功返回套接字的文件描述符，失败返回-1，并设置erron；

2）、bind() 绑定本机地址和端口：

int bind(int sockfd, struct sockaddr \*addr, int addrLen);

sockfd：socket返回的文件描述符

addr：指向sockaddr（通用地址结构）结构体指针，其中包含本机IP和端口号；

addrLen：sizeof(struct sockaddr\_in);

3）、listen() 设置接听套接字：

int listen(int sockfd, int backlog);

sockfd：套接字描述符；

backlog：指定等待连接队列的长度（1024）；

返回值：0或-1；

4）、accept() 接受TCP连接：# include <sys/types.h> # include <sys/socket.h>

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

返回值：已建立好连接的套接字或-1；

sockfd：监听套接字；

addr：对方地址（NULL）；

addrlen：对方地址长度（NULL）；

5）、send()/recv()、rwrite()/read()、recvfrom()/sendmsg() 数据接收：

头文件：# include <sys/socket.h>

ssize\_t recv(int socket, const void \*buffer, size\_t length, int f lags);

返回值：成功返回实际接收的字节数，失败返回-1，0表示对方关闭；

buffer：接收数据缓冲区；

length：发送的字节数；

flags：接收方式（直接置0）；

6）、connect() 请求建立连接（服务端）：

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

返回值：0或-1；

头文件：# include <sys/types.h> # include <sys/socket.h>

sockfd：套接字文件描述符；

serv\_addr：服务器端的地址信息；

addrlen：serv\_addr的长度；

send()、write()、sendto() 数据发送：

ssize\_t send(int socket, const void \*buffer, size\_t length, int flags);

返回值：成功返回实际发送的字节数，失败返回-1，0表示对方关闭；

头文件：# include <sys/socket.h>

buffer：发送缓冲区搜地址；

length：发送的字节数；

flags：发送方式（0）；

close()、shutdown() 关闭套接字：

close：双端关闭；

shutdown：指定关闭某一端

地址相关的数据结构：

通用地址结构：

struct sockaddr

{

u\_short sa\_family; //本机地址族，AF\_INET

char sa\_data[14]; //本机14字节协议地址

};

Internet协议地址结构：

struct sockaddr\_in（struct sockaddr\_un）

{

u\_short sin\_family; //本机地址族，AF\_INET，2字节

u\_short sin\_port; //本机端口，2字节(htons(8888))

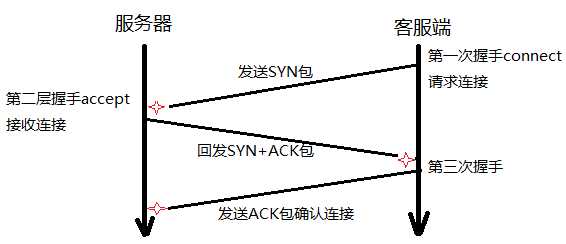
struct in\_addr sin\_addr; //本机IPV4地址，4字节，成员是sin\_addr.s\_addr = inet\_addr(“ip地址”);

char sin\_zero[8]; //8字节，用作填充

};

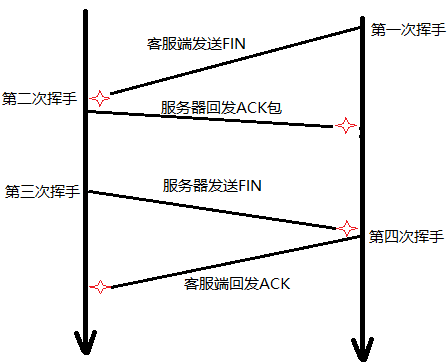
1. TCP的三次握手与四次挥手：

三次握手：



1. 第一次握手：建立连接connect时，客服端向服务器发送SYN包，并进入SYN\_SEND（等待状态）；
2. 第二层握手：服务器接收到SYN包，回发SYN包和ACK包，确认客服端能够接收无误，进入SYN\_RECV状态；
3. 第三次握手：客服端收到SYN+ACK包，向服务器发回ACK包，客服端和服务器进入ESTABLISHED状态，三次握手完毕，进入数据的传输；

四次挥手：



1. 第一次挥手：客服端发送FIN，用来关闭客服端和服务器的数据传输；（服务器读通道关闭）
2. 第二次挥手：服务器接收FIN，回发ACK表示接收到关闭信息；（客服端关闭写通道）
3. 第三次挥手：服务器关闭与客服端之间的连接，边单独发送FIN给客服端；（客服端关闭读通道）
4. 第四次挥手：客服端发送ACK确认关闭；（服务器关闭写通道）

简单说就是：先关读，后关写；

1. IO模型
2. 阻塞IO：

最常用、最简单、效率最低

（UDP套接字上执行的操作永远不会阻塞）

1. 非阻塞IO：

可以防止进程阻塞在IO操作上，但需要多次巡回询问是否有数据可读（消耗CPU）；

实现方式：

建立套接字描述符时，系统内核会将其设置为阻塞IO，可以使用funtl()设置一个套接字的标识：0\_NONBLOCK实现非阻塞；<fcntl.h>

int fcntl(int fd, int cmd, long arg );

(1)int flag = fcntl(sockfd, F\_GETFL, 0);//F\_GETFL用来获取套接字的状态标识；

(2)flag |= O\_NONBLOCK；//改变属性为非阻塞状态；

(3)fcntl(sockfd, F\_SETFL, flag, …);//设置套接字属性，将其设置为flag非阻塞状态；

1. IO多路复用：

允许同时对多个IO进行控制；

基本思想：

1. 先构造一张有关描述符的表（集合）；

fd\_set fds;

FD\_ZERO(&fds);//清空表fds

FD\_SET(fd, &fds);//将文件描述符添加到fds中；

1. 然后调用一个函数select，将该集合递交给内核；
2. 监测文件描述符事件的发生情况；（有事件发生会剔除没有事件发生的文件描述符）
3. 判断事件发生的对应套接字；

FD\_ISSETG(fd, &fds);//判断fd是否还在表中，在则表示fd发生了事件，不在则没有发生；

1. 处理该事件；（具体见NetorkProgram/day3/代码）；

select()函数实现多路复用：/poll();

<ysy/time.h><sys/types.h><unistd.h>/<ysy/poll.h>

int select(int maxfd, fd\_set \*read\_fds, fd\_set \*write\_fds, fd\_set \*except\_fds, struct timeval \*timeout);

int maxfd：所有监控的文件描述符最大值加1；

fd\_set \*read\_fds：所有要读的文件描述符的集合；

fd\_set \*write\_fds：所有的写文件描述符的集合；

fd\_set \*except\_fds：其他要向本服务通知的文件描述符；

返回值：

大于0：有事件发生，此时会剔除没有事件发生的文件描述符；

无返回：表示阻塞，没有事件发生；

小于0：出错；

等于0：超时返回；

struct timeval \*timeout：超时设置

NULL：阻塞模式；

0：非阻塞模式，进入轮训

指定时间值：

struct

在该值时间内，如果没有事件发生，select则超时返回0；

设置文件描述符用到的宏函数：

void FD\_ZERO(fd\_set \*fdset);//清除fdset中的所有文件描述符；

void FD\_SET(int fd, fd\_set \*fdset);//将fd加入到fdset（集合）中；

void FD\_CLR(int fd, fd\_set \*fdset);//将fd从fdset集合中清除；

int FD\_ISSET(int fd, fd\_set \*fdset);//判断fd是否在fdset集合中；

1. 信号驱动IO：一种异步通信模型；
2. 并发服务器：

循环服务器，一次只能处理一个客服端的请求服务，只能当本次服务完毕才能为下一个客服端提供服务；

1. 并发服务器：采用多进程或多线程实现同一时段能够为多个客服端提供服务；

并发服务器创建流程：

1. 接听客服请求；
2. 循环接收客服请求，并且每接收一个请求就fork/pthread\_create一个进程/线程与客服端通信；
3. 主进程关闭本进程开辟的通信套接字，并且为子进程（子进程要有主动退出标识，不然会执行到服务以外的代码）回收空间（非阻塞方式waitepid/或利用signal（非阻塞）接收子进程返回信号再调用waite回收（该方式只能收到实时信号，有可能导致某些非实时信号没有回收））；

/线程不能关闭子线程的通信套接字，但要负责回收子线程的空间（一般调用pthread\_detach将线程脱离本进程（不代表不受本进程控制），由系统自动在线程退出时回收子线程空间）；

1. 网络信息检索和套接字属性设置（函数在使用之前调用设置）

int getsockopt(int sockfd, int level, int optname, void \*optval, socklen\_t \*optlen);//得到

int setsockopt(int sockfd, int level, int optname, const void \*optval, socklen\_t optlen);//设置

level：指定控制套接字的层次

1. SOL\_SOCKET：通用型；
2. OPPROTO\_IP：IP层
3. IPPROTO\_TCP：TCP层

optname：指定控制的方式（层次的选项名称）；

optval：得到/设置套接字选项所对应类型设置（数据类型的具体数值）；

optlen：类型（optval）的长度；

层次和选项表：



网络超时设置定时器，捕捉SIGALRM信号；

alarm();/handler(int signo);

sigaction(int signum, const struct sigaction \*act, struct sigaction \*oldact);//获取\设置型号处理方式

signum：捕捉到的信号；

act：现在的处理方式，NULL（表示后面的oldact有效，将要获取当前处理方式）

oldact：原来的处理方式，（NULL）表示设置前面的act有效，将要设置；

设置流程如下：

void handler(int signo);

struct sigaction act;

1. sigaction(SIGALRM, NULL, act); //得到原处理方式
2. act.sa\_handler = handler; //void (\* sa\_handler)(int);
3. act.sa\_flags &= ~SA\_RESTART;打断阻塞自动重启(可选)
4. sigaction(SIGALRM, &act, NULL);//设置新处理方式
5. 广播和组播

广播：使用UDP协议（数据包套接字），用于同网段发送，最大接收地址254；（要设置网络属性为允许广播，getsockopt）；

需要指定接收方广播地址和端口；

组播：（接收方）

创建用户数据包套接字；

加入多播组：

struct IP\_mreq{

struct in\_addr imr\_multiaddr;//组播地址

struct in\_addr imr\_interface;//本机地址

} mreq;

mreq.imr\_multiaddr.s\_addr = inet\_addr(“244.0.0.1”);//组播地址

mreq.imr\_interface.addr = htonl(INADDR\_ANY);//表示自己可用IP；0.0.0.0

setsockopt(sockfd, IPPROTO\_IP, IP\_ADD\_MEMBERSHIP, &mreq, sizeof(mreq));//加入多播组

绑定组的IP和端口；

1. Unix套接字：

本地协议：PF\_UNIX（或PF\_LOCAL）；