目　录

一、标准 IO

1、常用概念

（1）当打开一个文件，会产生一个流，后续的 IO 操作均是围绕流进行的。

（2）当打开一个流时，fopen 会返回一个指向 FILE 对象的指针，FILE 类型包括实际 IO 的文件描述符、缓冲区地址、缓冲区长度（4K）、当前缓冲区中的字符数及出错状态。

（3）操作系统开始运行打开三个流，可供进程使用：标准输入、标准输出、标准错误。

（4）标准 IO 存在缓冲区，缓冲区的作用：

可以整合系统调用，减少调用系统调用（read 和 write）的次数，提高效率。（每次调用 read 和 write 函数时，系统都会有用户态切换至内核态，执行完 IO 操作在切换至用户态，频繁使用系统调用可能降低程序运行效率）。

（5）缓冲区的类型

A.全缓冲：缓冲区填满才进行实际的 IO 操作，如磁盘

B.行缓冲：当输入输出遇到换行符时，执行 IO 操作

C.无缓冲：不对字符进行缓冲，如标准错误

（6）刷新缓冲区方式：

A.全缓冲的缓冲区被填满时刷新缓冲区

B.行缓冲遇到\n

C.fflush 函数 （fflush(NULL) 刷新所有打开的流）

D.进程不已调用系统调用结束进程时（return 0）

2、常用函数

（1）打开流

函数原型：FILE *fopen(const char *pathname, const char *mode)

参数：　　pathname 需要打开的文件的路径

mode:

r　以只读方式打开

r+　读写方式打开　（以上文件流指针在文件开头）

w　只写

w+　读写（以上两种方式为文件有截短为零，没有就创建，流在开头）

a　追加（写在结尾，没有文件就创建）

a+　追加同时可读（写在结尾，读在开头，没有就创建）

返回值：

成功返回文件流指针

失败返回 NULL，同时设置 errno(可以打印 strerror(errno)判断什么类型错误，也可以使用 perror("fopen()")打印错误信息)

（2）关闭流

函数原型：int fclose(FILE *stream);

参数：　　stream　指向文件的文件流指针

返回值：

成功返回 0

失败返回 EOF 同时设置 errno

（3）读写

A.按字符读写

读：int getc(FILE *stream);

int fgetc(FILE *stream);

int getchar(void);

getc 可以被实现成宏，fgetc 只能是函数，getchar 相当于 getc(stdin)

参数：stream　文件流指针

返回值:以上三个函数成功返回读到的字符,失败或者读到文件结尾返回 EOF,

可以通过 ferror()和 feof()判断时出错还是读到文件结尾

int feof(FILE *stream);　（到结尾返回真，没到返回假）

int ferror(FILE *stream);（两个函数均是成功返回真，失败返回假）

写:

int putc(int c, FILE *stream);

int fputc(int c, FILE *stream);

int putchar(int c);

putchar(c)相当于 putc(c, stdout)

参数:

c 需要写入的字符

stream 需要写入的流

返回值：成功返回字符，失败返回 EOF

B.按行读写

读：char *gets(char *s);

参数：s 需要写入的地址,gets 是终端获取一个字符串，不检查缓冲区溢出

返回值：成功返回字符串地址，失败返回 NULL

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

参数：　s 需要写入的地址

size 需要写入的字符串大小（字节数）

stream 获取字符串的来源

成功返回字符串,失败或者到文件结尾返回 NULL,可以通过 ferror(fp)或者 feof(fp)判断是出错还是读到文件结尾。

写：int puts(const char *s);

向终端输出一串字符，返回输出的字符串字节数或者出错 EOF

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

参数：s 需要输出的字符串的首地址

stream 需要输出的文件

返回输出的字符串字节数或者出错 EOF

C.按二进制读写

读：size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

将 stream 指向的文件读出 nmemb 个 size 大小的数据到 ptr 指向的空间中

参数:

ptr 需要读入的数据的地址

size 每一个数据的大小，字节数

nmemb 需要读多少项数据

stream 从哪个文件内读出来

返回值: 成功返回读取的数据的个数（项数），失败或出错返回 0(需要使用 ferror 或 feof 判断是出错还是到达文件结尾)

写：size_t fwrite(const void *ptr, size_t size, size_t nmemb,FILE *stream);

将 ptr 指向的空间的 nmemb 个 size 大小的数据写入 stream 指向的文件中

参数:

Ptr 需要读出的数据的地址

Size 每个数据的大小，字节数

Nmemb 需要写入多少项数据

Stream 写入哪个文件

返回值：成功写入读取的数据的个数（项数），失败或出错返回 0

D.格式化读写:

int fscanf(FILE *stream, const char *format, ...);

int fprintf(FILE *stream, const char *format, ...);

可以格式化读写文件

（4）文件定位

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

参数：

stream：文件流指针

offset：偏移量

whence：偏移方式 SEEK_SET SEEK_END SEEK_CUR(头/尾/当前位置)

返回值：成功返回 0，失败非零

long ftell(FILE *stream);

计算当前偏移量,读出 stream 文件的偏移量返回

void rewind(FILE *stream);

相当于 fseek(fp, 0, SEEK_SET)；将文件偏移量设置到文件开头

计算文件长短（多少字符）fseek(fp, 0, SEEK_END); len = ftell(fp);(系统调用：len = 了 seek(ds, 0, SEEK_END);)

以上三个函数失败设置 errno

3、临时文件

FILE *tmpfile(void);

FILE *fp = tmpfile(); 失败返回 NULL

程序结束或者关闭文件后自动删除临时文件

二、文件 IO（系统调用 IO）

1、常见概念

（1）对于内核来说，所有文件都是通过文件描述符来访问

（2）当打开或者创建一个文件时，内核向进程返回一个文件描述符

（3）文件描述符的特点：

A.非负整形数（0-1023）

B.文件打开标志

C.当前可用最小

0，1，2 > < 2>

2、IO 操作

（1）打开

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

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

参数：pathname 需要打开的文件的路径名

flags

O_RDONLY/O_WRONLY/O_RDWR 只读/只写/可读可写

O_CREAT 创建

O_TRUNC 截断（清空）

O_APPEND 附加

mode 如果有 O_CREAT，必须有权限 （一般 0666）^umask0002

返回值：成功返回文件描述符，失败返回-1 设置 errno

（2）关闭

int close(int fd);

参数：fd 文件描述符

返回值：成功返回 0，失败返回-1 设置 errno

（3）读

ssize_t read(int fd, void *buf, size_t count);

将 fd 的文件读 count 个字节读入 buf 内

成功返回读到的字节数，文件结尾返回 0， 失败返回-1 设置 errno

（4）写

ssize_t write(int fd, const void *buf, size_t count);

将 buf 中的 count 个字节写入 fd 文件内

成功返回写入的字节数，失败返回-1 设置 errno

（5）文件定位

off_t lseek(int fd, off_t offset, int whence);

成功返回**当前偏移量**，失败返回-1 设置 errno

（6）getline

ssize_t getline(char **lineptr, size_t *n, FILE *stream);

从文件 stream 中完整获取一行的的内存，返回读到的字节数；

如果开始 lineptr 是 NULL,n 是 0，函数会申请空间存放这一行的内容，n 不是 0,够用就不申请

3、文件共享

内核使用三种数据结构表示打开的文件，它们之间的关系决定了在文件共享方面一个进程对另一个进程可能产生的影响。

(1) 每个进程在进程表中都有一个记录项，记录项中包含有一张打开文件描述符表，可将其视为一个矢量，每个描述符占用一项。与每个文件描述符相关联的是：

(a) 文件描述符标志（close_on_exec，参见图3-1和3.14节）。

(b) 指向一个文件表项的指针。

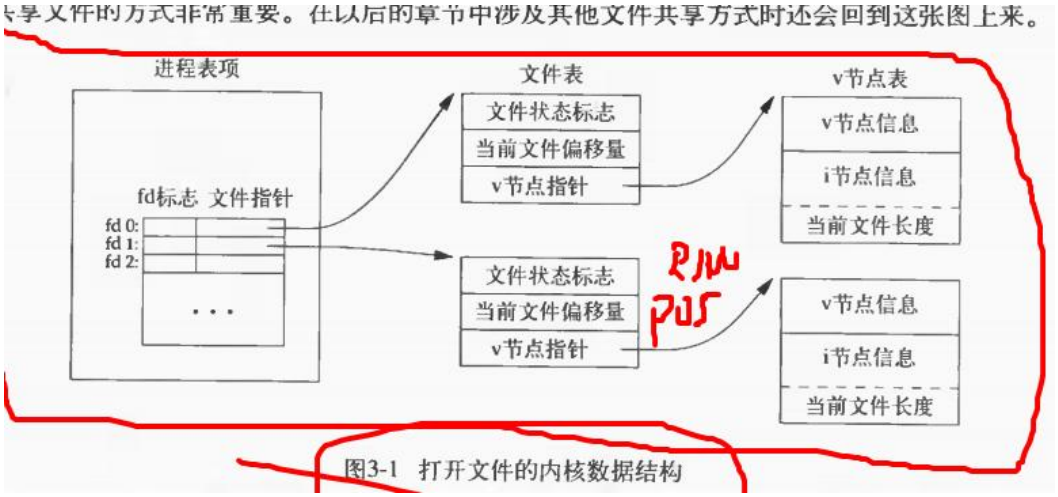
(2) 内核为所有打开文件维持一张文件表。每个文件表项包含：

(a) 文件状态标志（读、写、添写、同步和非阻塞等，关于这些标志的更多信息参见3.14节）。

(b) 当前文件偏移量。

(c) 指向该文件v节点表项的指针。

(3) 每个打开文件（或设备）都有一个v节点（v-node）结构。v节点包含了文件类型和对此文件进行各种操作的函数的指针。对于大多数文件，v节点还包含了该文件的i节点（i-node，索引节点）。这些信息是在打开文件时从磁盘上读入内存的，所以所有关于文件的信息都是快速可供使用的。例如，i节点包含了文件的所有者、文件长度、文件所在的设备、指向文件实际数据块在磁盘上所在位置的指针等等（4.14节较详细地说明了典型UNIX系统文件系统，并将更多地介绍i节点）。



int dup(int oldfd);

当前可用最小作为 oldfd 的复制

int dup2(int oldfd, int newfd);

newfd 作为 oldfd 的复制，将 newfd 重定向到 oldfd

dup 函数当前最小可用作为 oldfd 的复制

dup2（2， 1）newfd 1>&2

原子操作：一个操作，不能中断，不能分割

面试题：文件 IO 和标准 IO 之间的联系和区别

联系：

文件 IO 又称系统调用 IO，标准 IO 是在系统调用 IO 的基础上封装的，旨在减少对系统调用函数的调用次数，提高系统效率，而且不同系统都是对系统调用的使用，对用户的接口不变。标准 IO 可以跨平台使用，移植性较好

区别：

（1）标准 IO 使用文件流访问操作文件；文件 IO 使用文件描述符访问操作文件

（2）标准 IO 有缓冲区，文件 IO 无缓冲区

（3）标准 IO 可以跨平台使用，文件 IO 针对某个系统使用

三、文件和目录

1、文件操作

int stat(const char *pathname, struct stat *statbuf);

int fstat(int fd, struct stat *statbuf);

int lstat(const char *pathname, struct stat *statbuf);

函数区别：

stat：通过文件路径获取属性，面对符号链接文件时获取的是所指向的目标文件的属性

fstat：通过文件描述符获取文件属性

lstat：面对符号链接文件时获取的是符号链接文件的属性

stat 函数：

参数： pathname 需要操作的文件路径

statbuf 回填文件相关信息的地址

返回值：成功返回 0，失败返回-1 并设置 errno

struct stat 类型的结构体的数据：

```
struct stat {
        dev_t     st_dev;      /* ID of device containing file */
        ino_t     st_ino;      /* Inode number */
        mode_t    st_mode;     /* File type and mode */
        nlink_t   st_nlink;    /* Number of hard links */
        uid_t     st_uid;      /* User ID of owner */
        gid_t     st_gid;      /* Group ID of owner */
        dev_t     st_rdev;     /* Device ID (if special file) */
        off_t     st_size;     /* Total size, in bytes */
        blksize_t st_blksize;  /* Block size for filesystem I/O */
        blkcnt_t  st_blocks;   /* Number of 512B blocks allocated */
```

```
        struct timespec st_atim;    /* Time of last access */
        struct timespec st_mtim;    /* Time of last modification */
        struct timespec st_ctim;    /* Time of last status change */
        #define st_atime st_atim.tv_sec      /* Backward compatibility */
        #define st_mtime st_mtim.tv_sec
        #define st_ctime st_ctim.tv_sec
};
```

（1）文件类型

(st_mode & S_IFMT)		
S_IFSOCK	140000	s 套接字文件
S_IFLNK	0120000	l 符号链接文件
S_IFREG	0100000	-普通文件
S_IFBLK	0060000	b 块设备文件
S_IFDIR	0040000	d 目录文件
S_IFCHR	0020000	c 字符设备文件
S_IFIFO	0010000	p 管道文件

也可以通过下面方式判断文件类型:

S_ISREG(m)	is it a regular file?
S_ISDIR(m)	directory?
S_ISCHR(m)	character device?
S_ISBLK(m)	block device?
S_ISFIFO(m)	FIFO (named pipe)?
S_ISLNK(m)	symbolic link?
S_ISSOCK(m)	socket?

（2）文件权限

(st_mode & X) X 代表下面的位图定义

文件拥有者:

S_IRWXU	00700	rwx
S_IRUSR	00400	r
S_IWUSR	00200	w
S_IXUSR	00100	x

文件所属组:

S_IRWXG	00070	rwx
S_IRGRP	00040	r
S_IWGRP	00020	w
S_IXGRP	00010	x

其他用户:

S_IRWXO	00007	rwx
S_IROTH	00004	r
S_IWOTH	00002	w
S_IXOTH	00001	x

（3）硬连接数

nlink_t st_nlink

（4）拥有者（st_uit）所属组（st_gid）

A.uid------>uname(/etc/passwd)

struct passwd *getpwuid(uid_t uid);

struct passwd 类型的结构体里的 char *pw_name; /* username */

b.gid------>gname(/etc/group)

struct group *getgrgid(gid_t gid);

struct group 类型的结构体里的 char *gr_name; /* group name */

（5）字节个数

字节个数(st_size) !=磁盘空间大小(st_blocks) （512B 的个数）

（6）文件时间

st_atime 文件最后访问时间

st_mtime 文件最后修改时间

st_ctime 文件创建时间

Localtime 转换成标准时间--->strftime 格式化时间

2、时间处理相关函数

```
time_t time(time_t *tloc);
struct tm *gmtime(const time_t *timep);
struct tm *localtime(const time_t *timep);
time_t mktime(struct tm *tm);
size_t strftime(char *s, size_t max, const char *format,const struct tm *tm);
```

time 函数获取系统内核时间（时间戳）（从 1970-01-01 00:00:00 到现在的秒数）又称日历时间，函数成功返回系统时间，如果 tloc 不是 NULL，时间也会回填到这块空间;

localtime 将日历时间转换成 struct tm 类型的结构体时间(本地时间年月日时分秒星期)

gmtime 将日历时间转换成 struct tm 类型的结构体时间（国际标准时间年月日时分秒星期)

mktime 可以根据本地时间 struct tm 类型的年月日转换成 time 类型时间(秒数（日历时间）)

strftime 可以格式化时间放到 s 指向的空间内,根据 struct tm 的时间和 format 参数内的格式（%Y %m %d %H %M %S %年月日时分秒星期（还有很多格式））

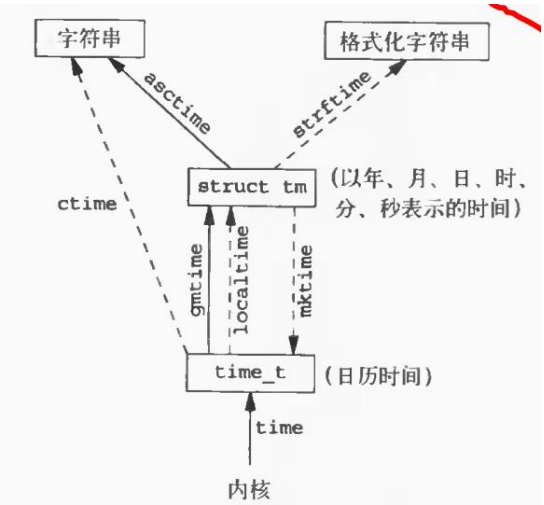


图6-1 各个时间函数之间的关系

3、目录操作

DIR *opendir(const char *name);

打开 name 目录，成功返回目录操作指针，失败返回 NULL 并设置 errno

struct dirent *readdir(DIR *dirp);

读目录，返回读到的目录下的文件或者目录的信息结构体指针 struct dirent

```
struct dirent {
                ino_t          d_ino;       /* Inode number */
                off_t          d_off;       /* Not an offset; see below */
                unsigned short d_reclen;    /* Length of this record */
                unsigned char  d_type;      /* Type of file; not supported
                                               by all filesystem types */
                char           d_name[256]; /* Null-terminated filename */
            };
```

int closedir(DIR *dirp);

关闭目录，失败或者到达目录结尾返回-1 并设置 errno

4、匹配（可操作目录的函数）

```
int glob(const char *pattern, int flags,
        int (*errfunc) (const char *epath, int eerrno),
        glob_t *pglob);
void globfree(glob_t *pglob);
```

功能：Linux 文件系统中路径名称的模式匹配，即查找文件系统中指定模式的路径。

参数：

pattern：匹配模型。如/*是匹配根文件下的所有文件（不包括隐藏文件，要找的隐藏文件附加匹配新匹配）

Flag： 选择匹配模式。

errfunc:：查看错误信息用，一般置为 NULL

pglob：存放匹配出的结果（glob_t 类型）

```
typedef struct {
        size_t   gl_pathc;    /* Count of paths matched so far  */
        char     **gl_pathv;  /* List of matched pathnames.  */
        size_t   gl_offs;     /* Slots to reserve in gl_pathv.  */
    } glob_t;
```

（后期可以用来存储多个字符串，**gl_pathv）

5、解析命令行参数

```
int getopt(int argc, char * const argv[],const char *optstring);
extern char *optarg;
extern int optind, opterr, optopt;
int getopt_long(int argc, char * const argv[],const char *optstring,
        const struct option *longopts, int *longindex);
```

optring:

"-"开头，识别非选项参数

"c"识别选项-c

"h:"带参数选项 optarg 指向参数

"w::"可选择参数

返回值

选项字符

-1 没有选项了

'?'未识别选项

1 非选项参数

getopt 函数只能处理短选项，而 getopt_long 函数两者都可以，可以说 getopt_long 已经包含了 getopt_long 的功能。

参数以及返回值介绍（以上三个函数都适用）：

1、argc 和 argv 和 main 函数的两个参数一致。

2、optstring: 表示短选项字符串。

形式如“a:b::cd:“，分别表示程序支持的命令行短选项有-a、-b、-c、-d，冒号含义如下:

(1)只有一个字符，不带冒号——只表示选项， 如-c

(2)一个字符，后接一个冒号——表示选项后面带一个参数，如-a 100

(3)一个字符，后接两个冒号——表示选项后面带一个可选参数，即参数可有可无， 如果带参数，则选项与参数直接不能有空格，形式应该如-b200

3、longopts：表示长选项结构体。结构如下：

```
struct option
{
const char *name;
int has_arg;
int *flag;
int val;
};
```

(1)name:表示选项的名称,比如 daemon,dir,out 等。

(2)has_arg:表示选项后面是否携带参数。该参数有三个不同值，如下:

a: no_argument(或者是 0)时 ——参数后面不跟参数值，eg: --version,--help

b: required_argument(或者是 1)时 ——参数输入格式为：--参数 值 或者 --参数=值。eg:--dir=/home

c: optional_argument(或者是 2)时 ——参数输入格式只能为：--参数=值

(3)flag:这个参数有两个意思，空或者非空。

a:如果参数为空 NULL，那么当选中某个长选项的时候，getopt_long 将返回 val 值。

eg，可执行程序 --help，getopt_long 的返回值为 h.

b:如果参数不为空，那么当选中某个长选项的时候，getopt_long 将返回 0，并且将 flag 指针参数指向 val 值。

eg: 可执行程序 --http-proxy=127.0.0.1:80 那么 getopt_long 返回值为 0,并且 lopt 值为 1。

(4)val：表示指定函数找到该选项时的返回值，或者当 flag 非空时指定 flag 指向的数据的值 val。

4、longindex:longindex 非空,它指向的变量将记录当前找到参数符合 longopts 里的第几个元素的描述，即是 longopts 的下标值。

5、全局变量:

（1）optarg：表示当前选项对应的参数值。

（2）optind：表示的是下一个将被处理到的参数在 argv 中的下标值。

（3）opterr：如果 opterr = 0，在 getopt、getopt_long、getopt_long_only 遇到错误将不会输出错误信息到标准输出流。opterr 在非 0 时，向屏幕输出错误。

（4）optopt：表示没有被未标识的选项。

6、返回值:

（1）如果短选项找到，那么将返回短选项对应的字符。

（2）如果长选项找到，如果 flag 为 NULL，返回 val。如果 flag 不

为空，返回 0

（3）如果遇到一个选项没有在短字符、长字符里面。或者在长字符里面存在二义性的，返回“？”

（4）如果解析完所有字符没有找到（一般是输入命令参数格式错误，eg： 连斜杠都没有加的选项），返回“-1”

（5）如果选项需要参数，忘了添加参数。返回值取决于 optstring，如果其第一个字符是“：”，则返回“：”，否则返回“？”。

注意：

（1）longopts 的最后一个元素必须是全 0 填充，否则会报段错误

（2）短选项中每个选项都是唯一的。而长选项如果简写，也需要保持唯一性。

6、密码校验过程

密码存储位置：/etc/shadow

（1）获取输入的密码

char *getpass(const char *prompt);

getpass()函数用于从控制台输入一行字符串，关闭了回显（输入时不显示输入的字符串），适用于用密码的输入。

参数 prompt 为提示字符串地址。

getpass()函数返回值：输入字符串地址。

（2）获取登录用户的密码

struct spwd *getspnam(const char *name);

name 为用户名字符串

struct spwd 结构体里的 sp_pwdp 就是加密后的密码

（3）使用同样的加密方式对输入的带校验密码加密

char *crypt(const char *key, const char *salt);

可以对 key 指向的密码字符串采用 salt 的加密算法加密，如果 salt 传入的是加密后的密码，该函数会解析其加密算法，使用同样加密算法对 key 加

密后返回

（4）校验

使用 strcmp 函数对加密后的输入密码和系统用户的加密密码比较，相同校验成功密码正确，不等输入密码错误

四、进程环境

1、进程终止的 8 种方式

正常终止：

（1）main 函数返回

（2）调用 exit

（3）调用_exit 或_Exit

（4）最后一个线程从启动例程返回

（5）最后一个线程调用 pthread_exit

异常终止：

（1）调用 abort

（2）接到一个信号并终止

（3）最后一个线程对取消请求做出响应

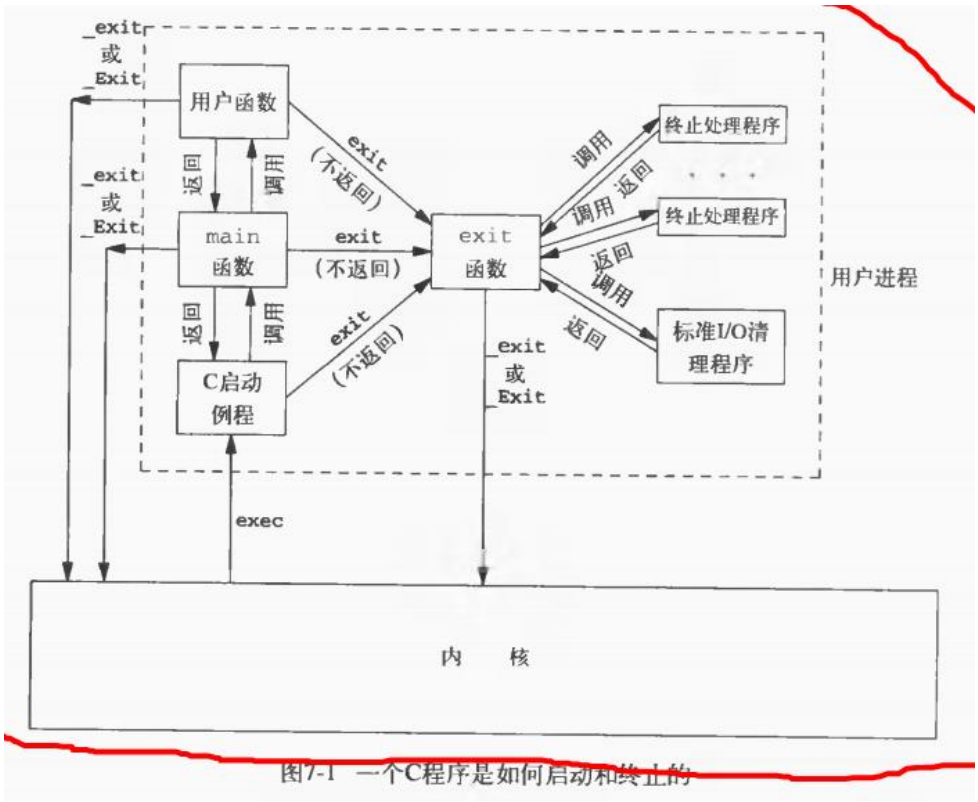


图7-1 一个C程序是如何启动和终止的

内核使程序执行的唯一方法时调用一个 exec 函数,这个 C 启动例程调用 main 函数，从而调用其他用户函数。在程序的任何一个位置调用_exit 或_Exit（两个均为系统调用）立即终止当前进程，进入内核，而 exit 函数（标准库函数）先执行一些清理处理（包括调用执行各终止处理程序、关闭所有标准 IO 流等），然后进入内核。

void exit(int status);

void _exit(int status);

void _Exit(int status);

三个函数的参数都是终止状态（或退出状态），一般进程正常结束默认返回 0

int atexit(void (*function)(void));

可以用来等级登记注册终止处理函数（调用 exit，结束进程之间的处理），一个进程可以登记 32 个终止处理函数，多次登记会处理多次，登记不同的终止处理函数会是压栈处理，即先登记的最后调用。

2、命令行参数

int main(int argc, char *argv[]) //char **argv

C 语言启动例程调用 main 函数，给 main 函数传参传入命令行参数

3、环境表与环境变量

存储进程所需要的环境变量的字符串指针数组，每个环境变量字符串需要一个指针存储起始地址，多个环境变量字符串需要一个指针数组存储他们的起始地址，需要使用全局变量 environ 访问，environ 记录了环境表中环境字符串的地址

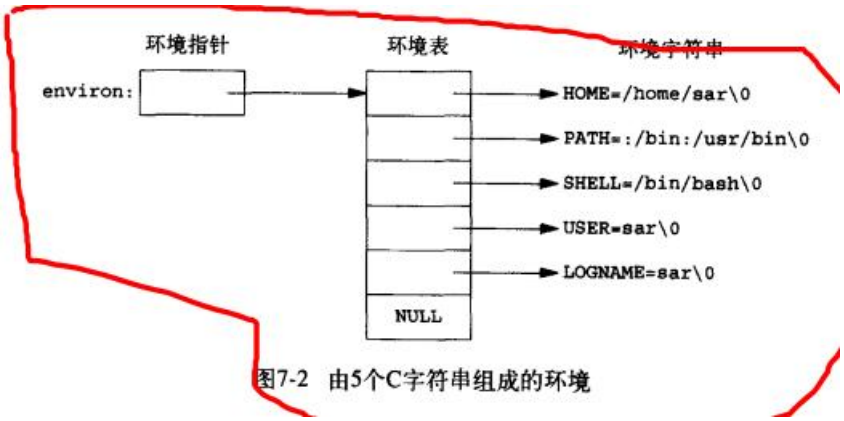


图7-2 由5个C字符串组成的环境

char *getenv(const char *name);

int setenv(const char *name, const char *value, int overwrite);

getenv 可以获得名字为 name 的环境变量的值并返回起始地址

setenv 可以设置修改环境变量的值，如果 name 环境变量存在 overwrite 为 0，则先把原来的环境值删除再修改成 value，如果非零，不设置新的也不会出错；

4、C 语言程序内存空间分配

命令行参数和环境变量

栈　　往下生长，系统自动分配和回收空间，存放的局部变量、函数形参、函数返回值等

堆区　　往上生长，可以动态开辟，使用完后需要释放空间

bss 段　　未初始化的全局变量和静态局部变量，初始值都是 0，生命周期至进程结束，全局变量作用域整个进程，静态局部变量作用域为定义的函数内或程序块内

data 段　　已经初始化的全局变量和局部静态变量生命周期至进程结束，全局

变量作用域整个进程，静态局部变量作用域为定义的函数内或程序块内

代码区　　存放代码

5、动态内存管理函数

void *malloc(size_t size);

申请 size 字节大小的存储空间,成功返回这块空间起始地址,失败返回 NULL，空间未初始化，随机值

void *calloc(size_t nmemb, size_t size);

申请 nmemb 个 size 字节大小的存储空间，成功返回这块空间起始地址，失败返回 NULL，空间初始化为 0

void *realloc(void *ptr, size_t size);

从 ptr 指向的起始地址重新开辟 size 字节大小的空间，成功返回这块空间起始地址，失败返回 NULL，ptr 指针一定是 calloc 和 malloc 的返回值，size 为 0 时相当于 malloc

void free(void *ptr);

可以释放 ptr 指向的空间，但是这块空间是使用 malloc(), calloc(), realloc()函数申请的,ptr 为 NULL 时什么都不不做

6、共享库（动态库）

（1）制作方式

动态库:

gcc -fPIC -shared -o libxxxx.so xxx.c

gcc -o a.out main.c -lxxxx -Lpath

ldd a.out 查看所有链接库的地址

vim ~/.bashrc 将动态库所在路径加入

LD_LIBRARY_PATH=$(LD_LIBRARY_PATH):

export LD_LIBRARY_PATH

或写入/etc/ld.so.conf

source ~/.bashrc

静态库：

gcc -c -o xxx.o xxx.c

ar -cr libxxx.a xxx.o

gcc -o a.out main.c -lxxx -Lpath

（2）区别与联系

A. windows 系统下静态库扩展名为.lib，动态库扩展名为.dll

Linux 系统下静态库扩展名为.a，动态库扩展名为.so

B. 无论是静态库还是动态库，都是在链接时被加载

C. 静态库多次使用都需要加载，占用内存空间大，但是移植性较好，程序编译完成可以独立运行，不在依赖库文件

动态库使用的时候记录库的地址，在内存中只加载一份；程序运行时跳到相应位置执行，占用空间小，但是执行过程中依赖函数库，库的更新方便；

7、跳转函数（setjmp 和 longjmp）

Goto 语句不能跨函数跳转，setjmp 和 longjmp 可以。

int setjmp(jmp_buf env);

void longjmp(jmp_buf env, int val);

需要借用全局变量 env 判断是否需要跳转，直接调用 setjmp(env)返回 0，从 longjmp 调用的情况返回非零值，longjmp(env, val),val（写 setjmp 的返回值）是非零值，存在多次同跳转可以通过 setjmp 的返回值区分跳转哪或者想跳到哪

五、进程控制

1、查看进程

pid_t getpid(void);　返回调用进程的 ID

pid_t getppid(void); 返回调用进程的父进程的进程 ID

2、创建进程

pid_t fork(void);

（1）创建一个新进程，成功返回两次，子进程返回值为 0，父进程返回新的子进程的进程 ID，失败返回-1 并设置 errno

（2）进程基本运行环境有哪些

环境表、终端、共享库、缓存区、虚拟地址空间、工作目录、资源限制

（3）新创建的子进程会继承父进程哪些属性

环境、终端、缓存区、共享库、进程表项、存储映射、资源限制、根目录、信号屏蔽字、文件屏蔽字、共享存储段

（4）子进程和父进程区别：

fork 返回值不同

进程 ID 不同

（5）fork 的应用场景

1）一个父进程希望复制自己，使父子进程同时执行不同代码段

2）进程替换，使一个进程执行不同的程序

（6）子进程和父进程或者多个进程之间谁先执行，取决系统的调度算法；

（7）pid_t vfork(void);

功能返回值同 fork

和 vfork 几点不同：

1、Fork 函数创建进程后，进程间谁先执行看系统调度算法；vfork 函数创建进程后，会阻塞父进程，子进程先运行，在子进程调用 exec 函数或者 exit 之后父进程开始执行

2、Fork 函数创建进程后，子进程和父进程各自有自己的数据空间，不会影响；vfork 函数创建进程后，子进程在父进程的地址空间内运行共用虚拟地址空间

3、Fork 函数创建进程后，子进程和父进程各自有自己的缓存区，不会影响；vfork 函数创建进程后，子进程在父进程的共用缓冲区

3、终止进程

void exit(int status);

void _exit(int status);

void _Exit(int status);

三个函数的参数都是终止状态（或退出状态），终止进程会通知父进程如何终止的

4、进程收尸

pid_t wait(int *wstatus);

pid_t waitpid(pid_t pid, int *wstatus, int options);

都是返回要被收尸的进程的进程 ID

wstatus 是一个指针，指向记录进程终止状态的空间，如果不关心，可以写 NULL;

waitpid 可以等待特定的进程结束

Pid<-1　等待其组 ID 等于 pid 的绝对值的任一子进程

Pid=-1　等待任意子进程，此时类似 wait 函数

Pid=0　等待其组 ID 和调用进程组 ID 相等的任一子进程

Pid>0　等待其进程 ID 等于 pid 的子进程

options 可以进一步控制 waitpid 的操作

5、进程替换

int execl(const char *path, const char *arg, ...

/* (char　*) NULL */);

int execlp(const char *file, const char *arg, ...

/* (char　*) NULL */);

int execle(const char *path, const char *arg, ...

/*, (char *) NULL, char * const envp[] */);

int execv(const char *path, char *const argv[]);

int execvp(const char *file, char *const argv[]);

int execvpe(const char *file, char *const argv[],

char *const envp[]);

第一个参数是文件名，这个文件要默认在环境变量 PATH 内，arg 为参数和参数选项，最后一个为 NULL;*argv[]是参数的指针数组

调用 exec 函数后，该进程执行的程序完全替换成新程序，进程 ID 不变，只是用新程序替换当前的正文、数据空间、堆栈等

六、进程关系

会话承载进程组，进程组承载进程，进程承载线程；

1、进程组

int setpgrp(void); 返回调用进程的进程组 ID

进程组是一个或多个进程的集合，通常他们与同一作业相关，每个进程组有唯一的进程 ID。

每个进程组都有一个组长进程，组长进程标识是进程组 ID 等于其进程 ID。

组长进程可以创建一个进程组，创建该组中的进程，只要进程组中有一个进程存在，进程组就存在，这与进程组长是否终止无关。

进程可以通过调用 setpgid 加入现有的组或者创建一个新进程组。

2、会话

是一个或多个进程组的集合

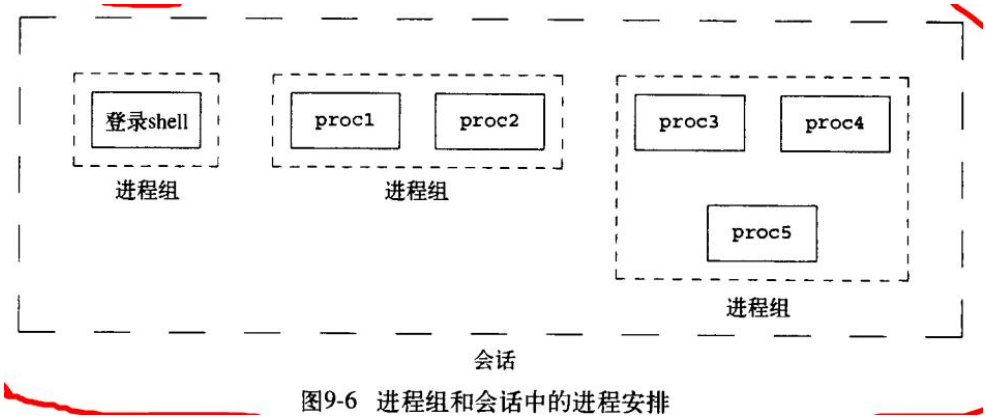


图9-6 进程组和会话中的进程安排

pid_t setsid(void);成功返回进程组 ID，失败返回-1

进程调用 setsid 函数建立一个新会话；

调用此函数的进程不是进程组组长，此函数就会创建一个新会话，会发生三件事：

（1）该进程变成新会话首进程（就是创建该会话的进程），该进程是新会话中的唯一进程

（2）该进程称为新进程组的组长进程

（3）无控制终端

如果调用进程是一个进程组组长，则函数出错返回。（为保证这种情况，通常是先调用 fork，然后终止父进程，子进程继续，子进程一定不是进程组组长，子进程会进程父进程的进程 ID,但是 ID 是新分配的，两者不可能相等）

七、守护进程

PID=GID=SID（进程 ID、进程组 ID、会话 ID）且不占用控制终端；

在系统后台运行，生命周期较长；

查看守护进程命令 ps -axj　ps -aux 查看所有进程

1、创建守护进程的过程

（1）调用 umask()函数设置文件屏蔽字为 0(umask(0))

（2）调用 fork，然后终止父进程(exit)

（3）调用 setsid 创建一个新会话(setsid())

（4）将当前工作目录更改为根目录(chdir("/"))

（5）关闭不再需要的文件描述符

（6）将 0 1 2 重定向到/dev/null (dup2)

官方函数：int daemon(int nochdir, int noclose);

Nochdir:如果是 0，将切换工作目录到根（/）

Noclose:如果是 0，把　0 1 2 重定向到/dev/null

2、出错记录（日志）

void openlog(const char *ident, int option, int facility);

Ident：身份信息

Option：可选选项

Facility：配置说明

void syslog(int priority, const char *format, ...);

Priority：日志优先级（LOG_ERR, LOG_DEBUG.....）

void closelog(void);

openlog 和系统日志建立链接；syslog 格式日志信息，等待写入（/var/log/syslog）；

closelog 和系统日志断开链接；

3、单实例守护进程

同时只能运行一个的进程（同一时刻仅一个进程可以运行）

可以通过文件锁或记录锁提供的互斥机制实现。（只加锁，不解锁，否者就不是单实例）

系统调用：int flock(int fd, int operation);

Fd:文件描述符

Operation：

LOCK_SH：共享锁

LOCK_EX：互斥锁

LOCK_UN：关锁

标准库函数：int lockf(int fd, int cmd, off_t len);

Fd:文件描述符

Cmd:

F_LOCK:互斥锁

F_TLOCK:如果没锁就加速，有锁不阻塞进程，返回

F_ULOCK：解锁

F_TEST：测试锁

Len:可以选择对文件加锁的位置，0 是整个文件

int ftruncate(int fd, off_t length);可以将文件截断到指定位置

通常守护进程使用了锁文件，该文件一般放在/var/run/,名字一般叫 name.pid

八、信号

1、信号的概念

信号是软件中断，是处理异步事件的一种机制。

异步：什么时候来不知道，来了有什么结果也不知道

2、信号分类

标准信号（1-31）和实时信号（34-64），查看信号 kill -l　　man 7 signal

3、标准信号的默认行为

终止进程（Ctrl c　　3　　SIGINT）

终止进程并产生 dump 文件（Ctrl /　2　　SIGQUIT）

忽略

停止进程执行（Ctrl z　　19　SIGSTOP）

继续进程执行

4、信号的行为注册（signal、sigaction(!!!)）

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);

signum 是信号编号，　handler 是为信号注册的行为（信号处理程序的入口地址）

返回值为之前的信号行为

常用：int sigaction(int signum, const struct sigaction *act,struct sigaction *oldact);

signum 是信号编号，act 为需要注册的新的行为的结构体地址，oldact 需要回填的旧的行为的结构体地址，sa_flags 是 SA_NOCLDWAIT 时不需要收尸，子进程终止后自动回收资源，不会产生僵尸进程，sa_flags 是 SA_SIGINFO 是可以使用第二个信号处理程序函数；sa_handler 可以是 SIG_DFL(恢复信号)，SIG_IGN(忽略信号)，信号处理函数

```
struct sigaction {
        void        (*sa_handler)(int);    //新的行为的函数入口地址
        void        (*sa_sigaction)(int, siginfo_t *, void *); //不常用
        sigset_t    sa_mask; //可以设置信号屏蔽字
        int         sa_flags; //特殊选项
        void        (*sa_restorer)(void);
};
```

5、信号的特点

（1）信号都有默认行为

（2）打断阻塞的系统调用

（3）同一个信号多次响应不会重复，会出现信号丢失（位图）

（标准信号多次响应不会重复（位图），实时信号会排队响应（队列）

给进程 ID 为 pid 的进程发送实时信号 x(34-64) kill -x pid)

（4）信号响应是嵌套的

（5）信号处理函数不能使用 longjmp 跳转（调用信号处理函数结束设置 mask = 0）

（6）信号处理函数中可以调用可重入函数

函数不可重入的原因：

（1）这些函数使用了静态数据结构

（2）这些函数调用了 malloc 或 free 函数

（3）标准 IO 函数（不可重入的方式使用全局数据结构）

6、信号产生

（1）Ctrl+c --->SIGINT Ctrl+\ --->SIGQUIT Ctrl+z --->SIGSTOP

（2）程序出现段错误

（3）常用函数（kill　　alarm）

int kill(pid_t pid, int sig);

Pid > 0 将 sig 信号发送给进程 ID 为 pid 的进程

Pid = 0 将 sig 信号发送给和调用进程统一进程组的所有进程

Pid < 0 将 sig 信号发送给进程组 ID 等于|pid|的进程

Pid = -1 将 sig 信号发送给系统上有权限让发送进程向他们发送信号的所有进程

unsigned int alarm(unsigned int seconds);

经过 seconds 秒产生 SIGALRM 信号，如果之前使用过 alarm 函数且计时未到有调用 alarm 函数，余留值会作为本次函数返回值返回，并且闹钟时钟被新值代替

int pause(void);

可以使得调用进程挂起直到捕捉到一个信号；而且只有执行一个信号处理程序从其返回时才会返回-1 并设置 errno 为 EINTR

7、信号集

多个信号的集合，数据类型 sigset_t 类型

常用函数：

int sigemptyset(sigset_t *set); //设置信号集 set 为空

int sigfillset(sigset_t *set); //设置信号集 set 为满

int sigaddset(sigset_t *set, int signum); //将信号 signum 添加到 set 信号集中

int sigdelset(sigset_t *set, int signum);//将信号 signum 从 set 信号集中删除

以上 4 个函数成功返回 0，失败返回-1 并且设置 errno 值

int sigismember(const sigset_t *set, int signum);

该函数可以测试 set 信号集中是否有 signum 信号量，有返回 1，没有返回 0，出

错返回-1 并且设置 errno 值

8、设置信号屏蔽字（mask 位图）

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);

How:

SIG_BLOCK:原本信号集和 set 取并集

SIG_UNBLOCK：原本信号集和 set 取交集

SIG_SETMASK：设置新的信号集 set

Set：新的信号集

Oldset：回填的旧的信号集

int sigsuspend(const sigset_t *mask); --->信号驱动程序运行，临时改变信号屏蔽字

相当于：

sigprocmask(SIG_MASK, &set, &oldset)//设置信号屏蔽字

pause(2)//等待信号到来

sigprocmask(SIG_MASK, &oldset, NULL) //恢复信号屏蔽字

该进程信号屏蔽字设置为 mask 指向的值，进程被挂起，直到捕捉到一个信号进程开始运行。

9、Sleep 和 alarm

以上两个函数都会用到 SIGARLM 信号，不要一起用

10、settimer

int setitimer(int which,const struct itimerval *new_value,struct itimerval *old_value);

参数 which 指定定时方式：

ITIMER_REAL 计时实际时间，到达时间，产生 SIGALRM 信号

ITIMER_VIRTUAL 该计时器根据进程消耗的用户模式 CPU 时间进行倒计时。（该度量包括进程中所有线程消耗的 CPU 时间。）在每次到期时，都会生成 SIGVTALRM 信号。

ITIMER_PROF 根据进程消耗的总 CPU 时间（即用户和系统）进行倒计时.

new_value 为传入参数 表示定时的时长

old_value 为传出参数 表示上一次定时剩余的时间，例如第一次定时 10s，但是过了 3s 后，再次用 setitimer 函数定时，此时第二次的计时会将第一次计时覆盖，而上一次定时的剩余时间则为 7s,不想要可以 NULL。

```
struct itimerval {
        struct timeval it_interval; /* Interval for periodic timer */
        struct timeval it_value;    /* Time until next expiration */
};
struct timeval {
        time_t      tv_sec;         /* seconds */
        suseconds_t tv_usec;        /* microseconds */
};
```

方式：先从 it_value.tv_sec 和 tv_usec 开始倒计时，计时到装载 it_interval 的时间重新倒计时

11、流量控制

漏桶和令牌桶

九、进程间通信

1、管道

（1）分类

匿名管道（pipe）：用于父进程和子进程间通信（常用）pipe(2)

命名管道（FIFO）：同一系统下的任意两个进程间通信 mkfifo(2)

（2）使用方式

int pipe(int pipefd[2]);

pipefd[2]：函数成功回填两个文件描述符 pipefd[0]读端，pipefd[1]写端

失败返回-1 并设置 errno

（3）特点：

子进程复制父进程的进程表项；

（1）一个管道提供两个文件描述符，一个读端一个写端，读在读端，写在写端

（2）Read 读空管道，read 会阻塞；write 写满管道,write 会阻塞

（3）管道写端关闭，read 函数去读将读到文件结束标志，返回 0

（4）管道读端关闭，write 函数去写，将产生异常信号 SIGPIPE

（5）不能对管道使用 lseek 函数

2、XSI IPC

Key_t ------>ftok(3)生成 key

Xxxget(2)----->获取实例

Xxxop(2)------>操作（读/写）

Xxxctl(2)------>获取结构/删除实例

消息队列 msgget(2)/msgop(2)(msgsnd(2)/msgrcv(2))/msgctl(2)

共享内存 shmget(2)/shmop(2)(shmat(2)/shmdt(2))/shmctl(2)

信号量机制 segget(2)/semop(2)/semctl(2)

消息队列

有亲缘关系的进程、无亲缘关系的进程之间的数据传输与数据交换

共享存储段

多个进程之间资源共享（无同步和互斥机制），速度最快

信号量机制

进程之间资源共享和资源竞争，例如访问“临界代码区”，有同步和互斥机制

（1）消息队列

1）key_t ftok(const char *pathname, int proj_id); //产生唯一的 key 值

Pathname：必须存在的文件路径，proj_id 和 key 值有关的 id（有亲缘关系的进程之间不用 key，只需要 msgget 第一个参数为 IPC_PRIVATE）

2）int msgget(key_t key, int msgflg); //创建或访问一个消息队列

Key:无亲缘关系的进程之间需要 key 值，亲缘关系的进程 IPC_PRIVATE

Msgflg:和创建文件类似， IPC_CREAT | IPC_EXCL | 0600， 不存在就创建，存在函数失败返回-1 并设置 errno 为 EEXIT,此时需要重新获取消息队列 id,msgget(key,0)成功返回消息队列 ID

3）收发数据

消息结构体：struct msgbuf{ long mtype;//类型 char mtext[]; }

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

功能：把一条消息添加到消息队列中

参数说明：

msgid:消息队列标识码

msgp:是一个指针，指针指向准备发送的消息结构体

msgsz:发送消息长度，不包括消息类型的 long int 型

msgflg=IPC_NOWAIT 表示队列满不等待，返回错误，一般写 0

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,int msgflg);

功能：从一个消息队列接收消息

参数说明：

msgid:消息队列的标识（区别其他消息队列）

msgp:回填信息的结构

msgsz: 发送消息长度，不包括消息类型的 mtype

Msgtyp:指定想要哪一种消息（一般写 0）

msgtyp == 0 返回队列中的第一个消息。

msgtyp > 0 返回队列中消息类型为 type 的第一个消息。

msgtyp < 0 返回队列中消息类型值小于或等于 type 绝对值的消息，如果这种消息有若干个，则取类型值最小的消息。

msgtyp 值非 0 用于以非先进先出次序读消息。

msgflg:（一般不指定，0）

IPC_NOWAIT：立即返回，如果没有请求类型的消息在队列中。

MSG_EXCEPT：与 msgtyp 大于 0 用于在队列中与从 msgtyp 不同消息类型读取所述第一消息。

MSG_NOERROR：如果大于 msgsz 字节数进行截断。

4）int msgctl(int msqid, int cmd, struct msqid_ds *buf);

功能：控制消息队列的函数

参数说明：

msgid:消息队列标识码

cmd:要采取的动作(IPC_RMID:删除消息队列)

Buf：不需要回填 NULL

返回值：失败返回-1，成功返回 0

（2）共享内存

共享内存可以说是最有用的进程间通信方式，也是最快的 IPC 形式。两个不同进程 A、B 共享内存的意思是，**同一块物理内存被映射到进程 A、B 各自的进程地址空间。**进程 A 可以即时看到进程 B 对共享内存中数据的更新，反之亦然。由于多个进程共享同一块内存区域，必然需要某种同步机制，互斥锁和信号量都可以。采用共享内存通信的一个显而易见的好处是效率高，因为进程可以直接读写内存，而不需要任何数据的拷贝。因此，采用共享内存的通信方式效率是非常高的。

共享内存的缺点：**共享内存没有进行同步与互斥机制**

1）int shmget(key_t key, size_t size, int shmflg);

功能：创建共享内存

参数：

key:共享内存 key 值

size：共享内存的大小(必须是页的整数倍，1 页=4kb=4096 字节)

shmflg:和创建文件是 mode 的使用一样

返回值：成功返回共享内存的标识码，失败返回-1

2）void *shmat(int shmid, const void *shmaddr, int shmflg);

功能：进程与共享内存建立连接

参数：

shmid:共享内存标识符

shmaddr:指定内存的地址

返回值：成功返回指向内向内存的指针，失败返回（void*）-1

3）int shmdt(const void *shmaddr);

功能：将共享内存段与当前进程脱离

参数： shmaddr:由 shmat 所返回的指针

返回值：成功返回 0，失败返回-1

4）int shmctl(int shmid, int cmd, struct shmid_ds *buf);

功能：删除（移除）共享段

参数： shmid:共享内存标识码

cmd:将要采取的动作(IPC_RMID：删除共享内存段)

buf:指向一个保存着共享内存的模式状态和访问权限的数据结构，不关心 NULL

返回值：成功返回 0，失败返回-1

（3）信号量机制

信号量机制是一种功能较强的机制，可以用来解决互斥与同步问题，它只能被两个标准的原语 wait(S)和 signal(S)来访问，也可以记为“P 操作”和“V 操作”。

临界资源：两个进程看到的一份公共资源称为“临界资源”，也就是说这些资源一次只允许一个进程使用，各个进程中访问“临界资源“的代码称为”临界区“。

互斥：在临界区中，每个进程只能独占式、排他式的访问临界资源。

同步：在互斥的基础上，让进程按照顺序公平的访问资源

信号量本质上是一个计数器，它是用来描述资源数目的。P(申请--)、V 释放++

1）int semget(key_t key, int nsems, int semflg);

功能：用来创建和访问一个信号量集
原型
```
int semget(key_t key, int nsems, int semflg);
```
参数
key：信号集的名字
nsems:信号集中信号量的个数
semflg：由九个权限标志构成，它们的用法和创建文件时使用的mode模式标志是一样的
返回值：成功返回一个非负整数，即该信号集的标识码；失败返回-1

2）int semop(int semid, struct sembuf *sops, size_t nsops);

功能：用来创建和访问一个信号量集
原型
```
int semop(int semid, struct sembuf *sops, unsigned nsops);
```
参数
semid:是该信号量的标识码，也就是semget函数的返回值
sops:是个指向一个结构数值的指针
nsops:信号量的个数
返回值：成功返回0；失败返回-1

```
struct sembuf {
    short sem_num;
    short sem_op;
    short sem_flg;
};
```
sem_num是信号量的编号。
sem_op是信号量一次PV操作时加减的数值，一般只会用到两个值：
一个是"-1"，也就是P操作，等待信号量变得可用；
另一个是"+1"，也就是我们的V操作，发出信号量已经变得可用
sem_flag的两个取值是IPC_NOWAIT或SEM_UNDO

3）int semctl(int semid, int semnum, int cmd, ...);

功能：用于控制信号量集
原型
```
int semctl(int semid, int semnum, int cmd, ...);
```
参数
semid:由semget返回的信号集标识码
semnum:信号集中信号量的序号
cmd:将要采取的动作（有三个可取值）
最后一个参数根据命令不同而不同
返回值：成功返回0；失败返回-1

（4）内存映射

（先映射，然后子进程可以继承父进程的虚拟地址空间）

void *mmap(void *addr, size_t length, int prot, int flags,int fd, off_t offset);

UNIX 网络编程第二卷进程间通信对 mmap 函数进行了说明。该函数主要用途有三个：

1、将一个普通文件映射到内存中，通常在需要对文件进行频繁读写时使用，这样用内存读写取代 I/O 读写，以获得较高的性能；

2、将特殊文件进行匿名内存映射，可以为关联进程提供共享内存空间；

3、为无关联的进程提供共享内存空间，一般也是将一个普通文件映射到内存中。

参数

start：指向欲映射的内存起始地址，通常设为 NULL，代表让系统自动选定地址，映射成功后返回该地址。

length：代表将文件中多大的部分映射到内存。

prot：映射区域的保护方式。可以为以下几种方式的组合：

PROT_EXEC 映射区域可被执行

PROT_READ 映射区域可被读取

PROT_WRITE 映射区域可被写入

PROT_NONE 映射区域不能存取

flags：影响映射区域的各种特性。在调用 mmap()时必须要指定 MAP_SHARED 或 MAP_PRIVATE。

MAP_SHARED 对映射区域的写入数据会复制回文件内，而且允许其他映射该文件的进程共享。

MAP_PRIVATE 对映射区域的写入操作会产生一个映射文件的复制，即私人的“写入时复制”（copy on write）对此区域作的任何修改都不会写回原来的文件内容。

MAP_ANONYMOUS 建立匿名映射。此时会忽略参数 fd，不涉及文件，而且映射区域无法和其他进程共享。

MAP_DENYWRITE 只允许对映射区域的写入操作，其他对文件直接写入的操作将会被拒绝。

MAP_LOCKED 将映射区域锁定住，这表示该区域不会被置换（swap）。

fd：要映射到内存中的文件描述符。如果使用匿名内存映射时，即 flags 中设置了 MAP_ANONYMOUS，fd 设为-1。

参数 offset：文件映射的偏移量，通常设置为 0，代表从文件最前方开始对应，offset 必须是分页大小的整数倍。

返回值：若映射成功则返回映射区的内存起始地址，否则返回 MAP_FAILED(－1)，错误原因存于 errno 中。

```
int munmap(void *addr, size_t length);
```

addr：要取消映射的内存区域的起始地址

length：要取消映射的内存区域的大小。

返回值：成功执行时返回 0。失败时返回-1.

十、线程

1、线程概念

线程是系统任务调度的基本单位，每个进程可以有一个或多个线程，这些线程处理自己的栈区独立，其他均共享。

2、线程标识

```
pthread_t pthread_self(void); //获得调用线程的线程标识
int pthread_equal(pthread_t t1, pthread_t t2); //比较线程标识是否一致
```

3、创建线程

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

参数：

thread　　需要回填的线程标识

Attr　　可指定线程属性，默认属性 NULL

start_routine　　线程的任务函数

Arg　　线程的任务函数的参数

返回值：成功返回 0，失败返回 errno 的值

4、线程终止

（1）调用 pthread_exit

（2）线程所在的进程终止

（3）线程从启动例程返回

void pthread_exit(void *retval);

retval　线程终止状态

5、线程收尸

int pthread_join(pthread_t thread, void **retval);

参数：

Thread　　线程标识

Retval　　回填线程标识为 thread 的线程的任务函数的返回值

返回值：成功返回 0，失败返回-1

6、线程同步

（1）互斥量（pthread_mutex_t）（注意死锁）

int pthread_mutex_init(pthread_mutex_t *restrict mutex,

const pthread_mutexattr_t *restrict attr);

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;(可以用这个代替初始化)

int pthread_mutex_lock(pthread_mutex_t *mutex);　//加锁

int pthread_mutex_trylock(pthread_mutex_t *mutex); //尝试加锁

int pthread_mutex_unlock(pthread_mutex_t *mutex); //解锁

（2）条件变量

（多线程竞争条件有变化情况下）

int pthread_cond_init(pthread_cond_t *restrict cond,

const pthread_condattr_t *restrict attr);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;（常用）

int pthread_cond_broadcast(pthread_cond_t *cond); //**发送广播**，进程内线程都能接收到（竞争条件变化之后就要发广播，告诉其他线程准备抢任务）

int pthread_cond_wait(pthread_cond_t *restrict cond,

pthread_mutex_t *restrict mutex); //等待唤醒（接收到广播后被唤醒）

7、死锁

常见死锁场景和解决方法

加锁不解锁----------》加锁的时限

多次加锁----------》检测死锁的状态

多个资源锁的加锁顺序---------》加锁顺序

产生死锁的四个必要条件

产生死锁必须同时满足以下四个条件，只要其中任一条件不成立，死锁就不会发生。

互斥条件：进程要求对所分配的资源（如打印机）进行排他性控制，即在一段时间内某 资源仅为一个进程所占有。此时若有其他进程请求该资源，则请求进程只能等待。

不剥夺条件：进程所获得的资源在未使用完毕之前，不能被其他进程强行夺走，即只能 由获得该资源的进程自己来释放（只能是主动释放)。

请求和保持条件：进程已经保持了至少一个资源，但又提出了新的资源请求，而

该资源 已被其他进程占有，此时请求进程被阻塞，但对自己已获得的资源保持不放。

循环等待条件：存在一种进程资源的循环等待链，链中每一个进程已获得的资源同时被 链中下一个进程所请求。即存在一个处于等待状态的进程集合{Pl, P2, ..., pn}，其中 Pi 等 待的资源被 P(i+1)占有（i=0, 1, ..., n-1），Pn 等待的资源被 P0 占有

8、线程取消

void pthread_cleanup_push(void (*routine)(void *), void *arg);

当遇到如下三种情况时候被安装的 routine 函数会被执行（可以安装多个，但执行顺序和安装顺序相反）：

1.线程调用 pthread_exit 函数时

2.响应取消请求时

3.调用 pthread_cleanup_pop(非零)时

注意：如果是调用 return 从线程返回时不会执行安装的线程清理函数

执行线程清理函数

void pthread_cleanup_pop(int execute);

如果 execute 是非 0 值，则最近安装的线程清理函数被执行

如果 execute 是 0，则取消最近安装的线程清理函数，不执行

注意：pthread_cleanup_push 和 pthread_cleanup_pop 一定要成对使用

十一、高级 IO

1、非阻塞 IO

默认情况下 read、write 等函数有可能会阻塞（比如管道中没数据 read 就会阻塞，管道满了 write 就会阻塞），当然也可以把 IO 设置为非阻塞的，即不能读或者写的时候出错返回而不是阻塞等待

（1）在 open 的时候加 O_NONBLOCK 标志

（2）用 fcntl 对已经打开的文件进行设置

2、IO 多路转接

（1）select

当一个进程需要同时监视多个文件描述符时，可以用多线程（需要线程间同步）或者多进程（进程间同步），当然更为简洁有效的办法是 IO 多路转接

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);

select 函数共有 5 个参数，其中参数和返回值：

maxfd：监视对象文件描述符数量。

readset：将所有关注“是否存在待读取数据”的文件描述符注册到 fd_set 变量，并传递其地址值。

writeset：将所有关注“是否可传输无阻塞数据”的文件描述符注册到 fd_set 变量，并传递其地址值。

exceptset：将所有关注“是否发生异常”的文件描述符注册到 fd_set 变量，并传递其地址值。

timeout：调用 select 后，为防止陷入无限阻塞状态，传递超时信息。

返回值：错误返回-1，超时返回 0。当关注的事件返回时，返回大于 0 的值，该值是发生事件的文件描述符数。

select 函数用来验证 3 种监视项的变化情况。根据监视项声明 3 个 fd_set 变量，分别向其注册文件描述符信息，并把变量的地址传递到函数的第二到第四个参数。

但是，在调用 select 函数前需要决定 2 件事：

“文件描述符的监视范围是？”

“如何设定 select 函数的超时时间？”

（2）poll

int poll(struct pollfd *fds, nfds_t nfds, int timeout);

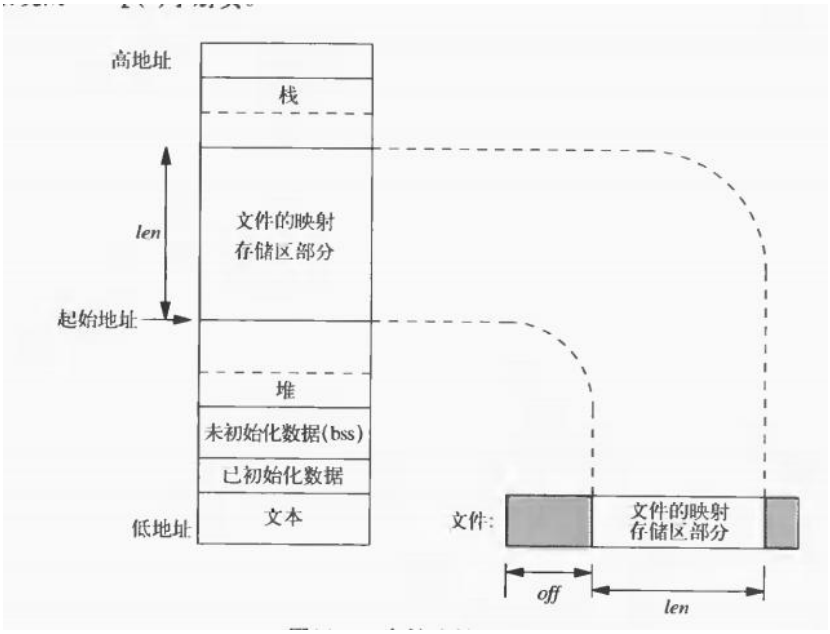
fds 是要监视的文件描述符及对应事件的数组

nfds 是数组的元素个数

timeout 是超时时间（如果为-1 则永不超时，为 0 的话不阻塞）

3、存储映射

mmap 可以将磁盘文件映射到内存中。



void *mmap(void *addr, size_t length, int prot, int flags,int fd, off_t offset);

UNIX 网络编程第二卷进程间通信对 mmap 函数进行了说明。该函数主要用途有三个：

2、将一个普通文**件映射到内存中**，通常在需要对文件进行频繁读写时使用，这样用内存读写取代 I/O 读写，以获得较高的性能；

2、将特殊文件进行匿名内存映射，可以为**关联进程提**供共享内存空间；

3、为**无关联的进程提供共享内存空间**，一般也是将一个普通文件映射到内存中。

参数

start：指向欲映射的内存起始地址，通常设为 NULL，代表让系统自动选定地址，映射成功后返回该地址。

length：代表将文件中多大的部分映射到内存。Length*1024B

prot：映射区域的保护方式。可以为以下几种方式的组合：

PROT_EXEC 映射区域可被执行

PROT_READ 映射区域可被读取

PROT_WRITE 映射区域可被写入

PROT_NONE 映射区域不能存取

flags：影响映射区域的各种特性。在调用 mmap()时必须要指定 MAP_SHARED 或 MAP_PRIVATE。

MAP_SHARED 允许其他映射该文件的进程共享。

MAP_PRIVATE 即私人的“写入时复制”（copy on write）

MAP_ANONYMOUS 建立匿名映射。此时会忽略参数 fd，不涉及文件，而且映射区域无法和其他进程共享。

MAP_DENYWRITE 只允许对映射区域的写入操作，其他对文件直接写入的操作将会被拒绝。

MAP_LOCKED 将映射区域锁定住，这表示该区域不会被置换（swap）。

fd：要映射到内存中的文件描述符。如果使用匿名内存映射时，即 flags 中设置了 MAP_ANONYMOUS，fd 设为-1。

参数 offset：文件映射的偏移量，通常设置为 0，代表从文件最前方开始对应，offset 必须是分页大小的整数倍。

返回值：若映射成功则返回映射区的内存起始地址，否则返回 MAP_FAILED(－1)，错误原因存于 errno 中。

int munmap(void *addr, size_t length);

addr：要取消映射的内存区域的起始地址

length：要取消映射的内存区域的大小。

返回值：成功执行时返回 0。失败时返回-1.

实例（将文件映射到内存中使用 strstr 找字符串）

十二、socket

1、地址分类与划分

地址----》IP + 端口

Ipv4 32bit -------->4byte-----> Ipv6 128bit

网络号 主机号

A 类地址 1 3 大型网络

0~127 内网（10.0.0.0~10.255.255.255）

B 类地址 2 2

128~191 内网（172.16~172.31）

C 类地址 3 1

192~223 内网（192.168）

D 类地址 多播

224~239 组播地址

E 类地址 保留

0.0.0.0---->本地所有网卡地址

127.0.0.1 ---->环回测试地址

255.255.255.255 广播地址

点分十进制"192.168.1.10"---->uint32_t inet_aton(3) / inet_ntoa(3)

（字符串转网址，网址转字符串）

int inet_aton(const char *cp, struct in_addr *inp);

将形如“192.168.1.1"类型的点分十进制 ip 转换成二进制，并存放在 struct in_addr 中

192 = 0xc0 168 = 0xa8 1 = 0x01 因此转换后的值为 0xc0a811

char *inet_ntoa(struct in_addr in);网址转字符串

```
typedef uint32_t in_addr_t;
    struct in_addr {in_addr_t s_addr;};
```

2、端口

运行服务的标识

0~65535（0~1024 周知端口（用于军工等））

3、字节序

大端格式：高字节数据存储在低地址段，低字节数据存储在高地址段

小端格式：高字节数据存储在高地址段，低字节数据存储在低地址段

uint16_t htons(uint16_t hostshort);　本地字节序转网络字节序

uint16_t ntohs(uint16_t netshort);　网络字节序转本地字节序

4、网络模型

（1）OSI

OSI七层网络模型	TCP/IP四层概念模型	对应网络协议
应用层（Application）	应用层	HTTP、TFTP, FTP, NFS, WAIS、SMTP
表示层（Presentation）		Telnet, Rlogin, SNMP, Gopher
会话层（Session）		SMTP, DNS
传输层（Transport）	传输层	TCP, UDP
网络层（Network）	网络层	IP, ICMP, ARP, RARP, AKP, UUCP
数据链路层（Data Link）	数据链路层	FDDI, Ethernet, Arpanet, PDN, SLIP, PPP
物理层（Physical）		IEEE 802.1A, IEEE 802.2到IEEE 802.11

第一层：物理层

在 OSI 参考模型中，物理层是参考模型的最低层，也是 OSI 模型的第一层。物理层的主要功能是：**利用传输介质为数据链路层提供物理连接，实现比特**

流的透明传输。物理层的作用是实现相邻计算机节点之间比特流的透明传送，尽可能屏蔽掉具体传输介质和物理设备的差异，使其上面的数据链路层不必考虑网络的具体传输介质是什么。

第二层：数据链路层

数据链路层（Data Link Layer）是 OSI 模型的第二层，负责建立和管理节点间的链路。在计算机网络中由于各种干扰的存在，导致物理链路是不可靠的。因此这一层的主要功能是：**在物理层提供的比特流的基础上，通过差错控制、流量控制方法，使有差错的物理线路变为无差错的数据链路，即提供可靠的通过物理介质传输数据的方法。**

第三层：网络层

网络层（Network Layer）是 OSI 模型的第三层，它是 OSI 参考模型中最复杂的一层，也是通信子网的最高一层，它在下两层的基础上向资源子网提供服务。其主要功能是：**在数据链路层提供的两个相邻端点之间的数据帧的传送功能上，进一步管理网络中的数据通信，控制数据链路层与传输层之间的信息转发，建立、维持和终止网络的连接，将数据设法从源端经过若干个中间节点传送到目的端（点到点），**从而向传输层提供最基本的端到端的数据传输服务。具体地说，数据链路层的数据在这一层被转换为数据包，然后通过路径选择、分段组合、顺序、进/出路由等控制，将信息从一个网络设备传送到另一个网络设备。数据链路层和网络层的区别为：数据链路层的目的是解决同一网络内节点之间的通信，而网络层主要解决不同子网间的通信。

第四层：传输层

OSI 下 3 层的任务是数据通信，上 3 层的任务是数据处理。而传输层

（Transport Layer）是 OSI 模型的第 4 层。该层提供建立、维护和拆除传输连接的功能，起到承上启下的作用。该层的主要功能是：向用户提供可靠的端到端的差错和流量控制，保证报文的正确传输，同时向高层屏蔽下层数据通信的细节，即向用户透明地传送报文。

第五层：会话层

会话层是 OSI 模型的第 5 层，是用户应用程序和网络之间的接口，该层的主要功能是：**组织和协调两个会话进程之间的通信 ，并对数据交换进行管理**。当建立会话时，用户必须提供他们想要连接的远程地址。而这些地址与 MAC 地址或网络层的逻辑地址不同，它们是为用户专门设计的，更便于用户记忆。域名就是一种网络上使用的远程地址。会话层的具体功能如下：

会话管理：允许用户在两个实体设备之间建立、维持和终止会话，并支持它们之间的数据交换。

会话流量控制：提供会话流量控制和交叉会话功能。

寻址：使用远程地址建立会话连接。

出错控制：从逻辑上讲会话层主要负责数据交换的建立、保持和终止，但实际的工作却是接收来自传输层的数据，并负责纠正错误。

第六层：表示层

表示层是 OSI 模型的第六层，它对来自应用层的命令和数据进行解释，对各种语法赋予相应的含义，并按照一定的格式传送给会话层。该层的主要功能是：**处理用户信息的表示问题，如编码、数据格式转换和加密解密等。**表示层的具体功能如下：

数据格式处理：协商和建立数据交换的格式，解决各应用程序之间在数据格式表

示上的差异。

数据的编码：处理字符集和数字的转换。

压缩和解压缩：为了减少数据的传输量，这一层还负责数据的压缩与恢复。

数据的加密和解密：可以提高网络的安全性。

第七层：应用层

应用层是 OSI 参考模型的最高层，它是计算机用户，以及各种应用程序和网络之间的接口，该层的主要功能是：**直接向用户提供服务，完成用户希望在网络上完成的各种工作。**它在其他 6 层工作的基础上，负责完成网络中应用程序与网络操作系统之间的联系，建立与结束使用者之间的联系，并完成网络用户提出的各种网络服务及应用所需的监督、管理和服务等各种协议。此外该层还负责协调各个应用程序间的工作。应用层的具体功能如下：

总结：

应用层：产生网络流量的程序

表示层：传输之前是否进行加密或者压缩处理

会话层：查看会话，查木马　netstat-n

传输层：可靠传输、流量控制、不可靠传输

网络层：负责选择最佳路径、规划 ip 地址

数据链路层：帧的开始和结束、透明传输、差错校验

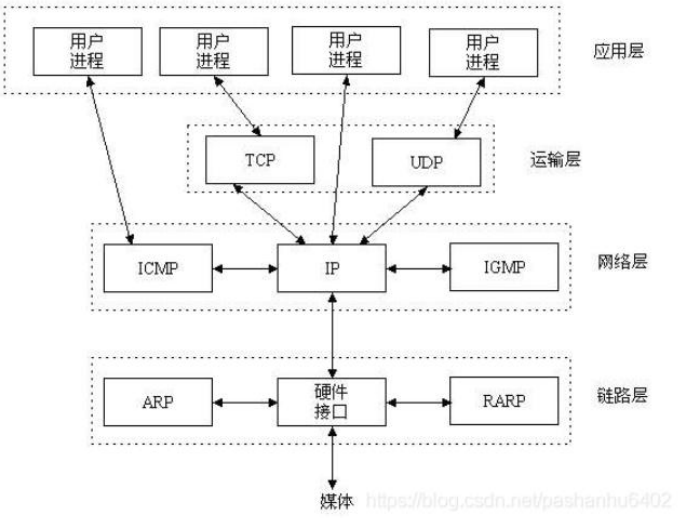
物理层：接口标准、电器标准、如何更快传输数据

（2）TCP/IP

TCP/IP（Transmission Control Protocol/Internet Protocol）即传输控制协议/网间协

议，是一个工业标准的协议集，它是为广域网（WANs）设计的。

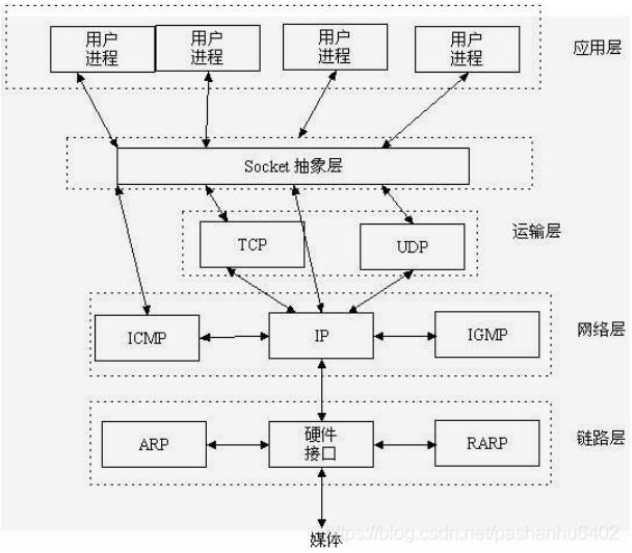
UDP（User Data Protocol，用户数据报协议）是与 TCP 相对应的协议。它是属于 TCP/IP 协议族中的一种。



通过源 IP 地址、目标 IP 地址、协议号、源端口号以及目标端口号这五个元素识别一个通信

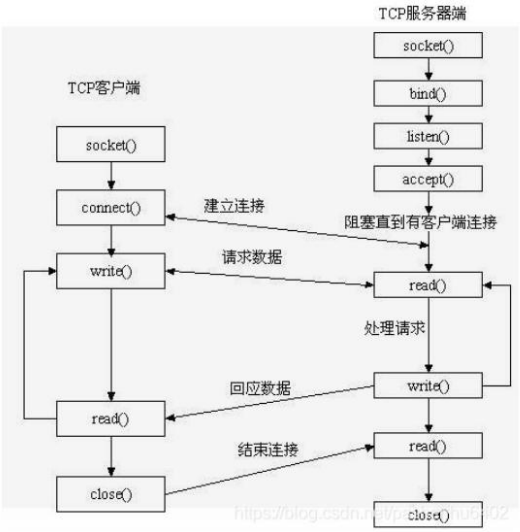
5、套接字

（1）简介



Socket 是应用层与 TCP/IP 协议族通信的中间软件抽象层，它是一组接口。在设计模式中，Socket 其实就是一个门面模式，它把复杂的 TCP/IP 协议族隐藏在 Socket 接口后面，对用户来说，一组简单的接口就是全部，让 Socket 去组织数据，以符合指定的协议。

（2）TCP 流程



先从服务器端说起。服务器端先初始化 Socket，然后与端口绑定(bind)，对端口进行监听(listen)，调用 accept 阻塞，等待客户端连接。在这时如果有个客户端初始化一个 Socket，然后连接服务器(connect)，如果连接成功，这时客户端与服务器端的连接就建立了。客户端发送数据请求，服务器端接收请求并处理请求，然后把回应数据发送给客户端，客户端读取数据，最后关闭连接，一次交互结束。

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

socket()用于创建一个 socket 描述符（socket descriptor），它唯一标识一个 socket。

参数：

domain：即协议域，又称为协议族（family）。常用的协议族有，AF_INET（IPV4）、AF_INET6、AF_LOCAL（或称 AF_UNIX，Unix 域 socket）、AF_ROUTE 等等。

type：指定 socket 类型。常用的 socket 类型有，SOCK_STREAM（TCP 常用）、SOCK_DGRAM(UDP 常用)、SOCK_RAW、SOCK_PACKET、SOCK_SEQPACKET 等等

protocol：就是指定协议。通常 0，默认协议常用的协议有，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

B.int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

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。

参数：

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，如 ipv4 对应的是：

```
struct sockaddr_in {
    sa_family_t    sin_family;
    in_port_t      sin_port;
    struct in_addr sin_addr;
};
```

```
struct in_addr {
        uint32_t            s_addr;
};
```

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(不用 bind)，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。

C.int listen(int sockfd, int backlog);

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

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。

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。

D.int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

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()之后就想 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。

accept 函数的第一个参数为服务器的 socket 描述字，第二个参数为指向 struct sockaddr *的指针，用于返回客户端的协议地址，第三个参数为协议地址的

长度。如果 accpet 成功，**那么其返回值是由内核自动生成的一个全新的描述字，代表与返回客户的 TCP 连接。**

注意：accept 的第一个参数为服务器的 socket 描述字，是服务器开始调用 socket()函数生成的，称为监听 socket 描述字；**而 accept 函数返回的是已连接的 socket 描述字**。一个服务器通常通常仅仅只创建一个监听 socket 描述字，它在该服务器的生命周期内一直存在。内核为每个由服务器进程接受的客户连接创建了一个已连接 socket 描述字，当服务器完成了对某个客户的服务，相应的已连接 socket 描述字就被关闭。

E.读写

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
                                const struct sockaddr *dest_addr, socklen_t addrlen);
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                                  struct sockaddr *src_addr, socklen_t *addrlen);
```

Sockfd：套接字描述符

Buf：存储空间

Len：空间大小

Flags：一般为 0

dest_addr/src_addr:地址

Addrlen：地址长度

（3）UDP 组播

客户端加入多播组：

```
setsockopt(sd, IPPROTO_IP, IP_MULTICAST_IF, struct ip_mreqn, sizeof())
```

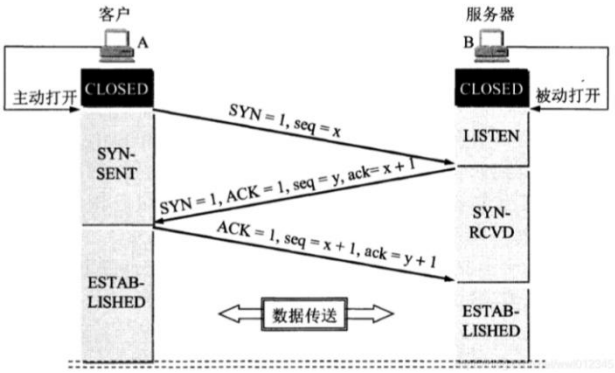
服务端设置多播选项：

setsockopt(sd, IPPROTO_IP, IP_ADD_MEMBERSHIP, struct ip_mreqn,)

（4）UDP 广播

setsockopt(sd, SOL_SOCKET, SO_BROADCAST, val=1,);

（5）TCP 三次握手

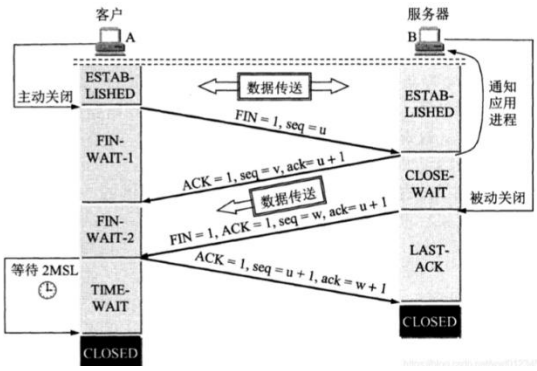


1）client 向 server 发送 syn 序列号，假定为 y，client 处于 SYN_SENT

2）server 接收到 client 的 syn 之后，向 client 端发送 ack=y+1,syn=x，server 处于 SYN_RECV

3）client 接收到后，发送 server 应答数据包 ack=x+1，处于 ESTABLISHED

（6）TCP 四次挥手

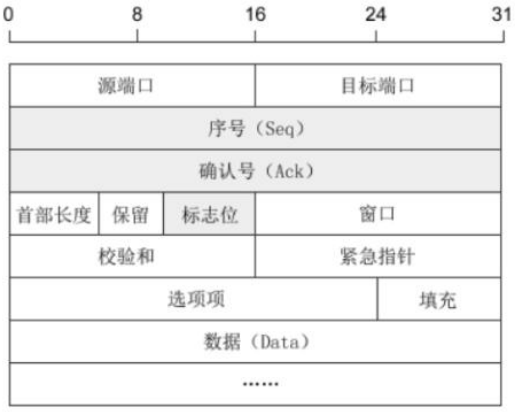


1）client 向 server 发送 fin 断开序列号，假定为 z

2）server 收到后立即向 client 发送 ack=z+1

3）server 向 client 发送 fin=m

4）client 收到后向 server 发送 ack=m+1



序号：Seq（Sequence Number）序号占 32 位，用来标识从计算机 A 发送到计算机 B 的数据包的序号，计算机发送数据时对此进行标记。

②确认号：Ack（Acknowledge Number）确认号占 32 位，客户端和服务器端都可以发送，Ack = Seq + 1。

③标志位：每个标志位占用 1Bit，共有 6 个，分别为 URG、ACK、PSH、RST、SYN、FIN，具体含义如下：

URG：紧急指针（urgent pointer）有效。

ACK：确认序号有效。

PSH：接收方应该尽快将这个报文交给应用层。

RST：重置连接。

SYN：建立一个新连接。

FIN：断开一个连接。

（7）TCP 和 UDP 区别

各自特点：

（1）无连接的不可靠的报式传输；

（2）udp 实时性高，传输高效；

（3）首部 8 个字节，开销小；

（4）可以一对一、一对多、多对一、多对多，组播和广播

tcp:

（1）基于连接的

（2）可靠的

（3）流式传输

（4）点对点

区别总结:

（1）tcp 是基于有连接的流式套接字； udp 是基于无连接的报式套接字

（2）tcp 的报头 20 个字节，udp 的报头 8 字节

（3）tcp 的传输比 udp 传输更加可靠

（4）udp 的传输效率比 tcp 高，udp 适合传输音频视频，tcp 适合传输文件

（5）tcp 只能是点对点传输；udp 可以实现一对一、一对多、多对一、多对多，可以实现组播和广播