中国矿业大学计算机学院

2021-2022(2)本科生 Linux 操作系统课程作业

专业班级 信息安全 19-1 班 学生姓名 江一川 学 号 08193041

序号	作业内容	基础理论掌握程度	综合知识应用能力	报告内容	报告格式	完成状况	工作量	学习、工作态度	抄袭现象	其它	综合成绩
1	文件系统	熟练□ 较熟练□ 一般□ 不熟练□	强□ 较强□ 一般□ 差□	完整□ 较完整□ 一般□ 不完整□	规范□ 较规范□ 一般□ 不规范□	好□ 较好□ 一般□ 差□	饱满□ 适中□ 一般□ 欠缺□	好□ 较好□ 一般□ 差□	学号: 姓名:		
2	进程通信	熟练□ 较熟练□ 一般□ 不熟练□	强□ 较强□ 一般□ 差□	完整□ 较完整□ 一般□ 不完整□	规范□ 较规范□ 一般□ 不规范□	好□ 较好□ 一般□ 差□	饱满□ 适中□ 一般□ 欠缺□	好□ 较好□ 一般□ 差□	学号: 姓名:		

任课教师：杨东平

年 月 日

1 文件系统

1.1. 详述 Linux 的节点 inode

1.1.1 什么是 inode

文件储存在硬盘上，硬盘的最小存储单位叫做扇区（Sector）。每个扇区储存 512 字节（相当于 0.5KB）。

操作系统读取硬盘的时候，不会一个个扇区地读取，这样效率太低，而是一次性连续读取多个扇区，即一次性读取一个块（block）。这种由多个扇区组成的块，是文件存取的最小单位。块的大小，最常见的是 4KB，即连续八个 sector 组成一个 block。

文件数据都储存在块中，那么很显然，我们还必须找到一个地方储存文件的元信息，比如文件的创建者、文件的创建日期、文件的大小等等。这种储存文件元信息的区域就叫做 inode，中文译名为“索引节点”。

每一个文件都有对应的 inode，里面包含了与该文件有关的一些信息。

1.1.2 inode 包含的内容

（1）inode 包含文件的元信息，主要有以下内容：

① 以字节为单位表示的文件大小
② 文件拥有者的 User ID
③ 文件的 Group ID
④ 文件的读、写、执行权限
⑤ 链接数，即指向该 inode 的文件名总数
⑥ 文件的时间戳：inode 最近修改时间（ctime）、文件最近访问时间（atime）和文件最近修改时间（mtime）
⑦ 文件数据 block 的位置

（2）可以使用 stat 命令查看某个文件的 inode 信息：

```
┌──(kali㉿kali)-[~/桌面]
└─$ stat cumt
  文件：cumt
  大小：72              块：8          IO 块：4096   普通文件
设备：801h/2049d        Inode：407640      硬链接：1
权限：(0644/-rw-r--r--)  Uid：( 1000/    kali)   Gid：( 1000/    kali)
最近访问：2022-04-18 13:16:29.286745984 +0800
最近更改：2022-04-05 00:09:23.158864281 +0800
最近改动：2022-04-05 00:09:23.158864281 +0800
创建时间：2022-04-05 00:09:23.158864281 +0800
```

1.1.3 inode 的大小

（1）通常情况下，一个 inode 的大小是 128 字节或 256 字节，并且在磁盘中每 1KB 或 2KB 就会设定一个 inode。假设在一块 1GB 的硬盘中，每个 inode 节点的大小为 128 字节且每 1KB 就设置一个 inode，那么 inode 区域的大小就会达到 128MB，占总容量的 12.8%。

（2）可使用命令 df -i 查看每个硬盘分区 inode 总数和已经使用的数量：

```
┌──(kali㉿kali)-[~/桌面]
└─$ df -i
文件系统        Inodes 已用(I) 可用(I) 已用(I)% 挂载点
udev            240606    382  240224       1% /dev
tmpfs           249798    651  249147       1% /run
/dev/sda1      1248480 330421  918059      27% /
tmpfs           249798      1  249797       1% /dev/shm
tmpfs           249798      2  249796       1% /run/lock
tmpfs            49959     72   49887       1% /run/user/1000
tmpfs            49959     58   49901       1% /run/user/132
```

1.1.4 inode 号码

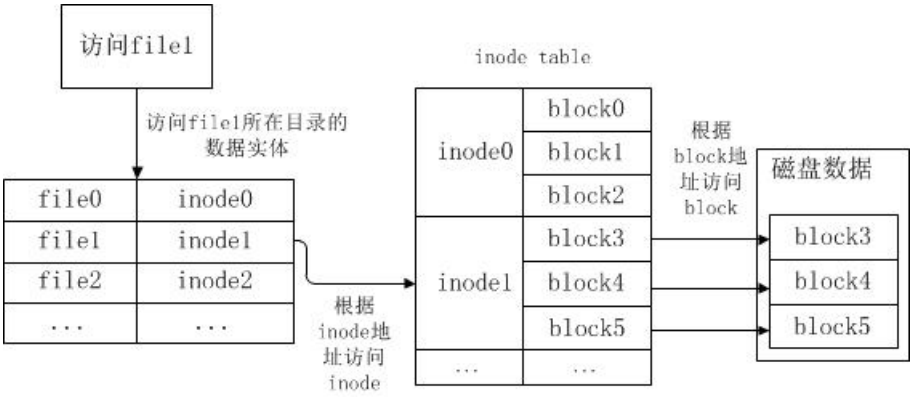
（1）每个 inode 都有一个号码，操作系统使用 inode 号码来识别不同的文件（Unix/Linux 系统不使用文名来识别文件，对于系统来说，文件名只是 inode 号码便于识别的别称）。

（2）当系统给打开某个特定文件时，实际要经历以下三步：

① 系统找到此文件对应的 inode 号码

② 通过 inode 号码获取 inode 信息

③ 根据 inode 信息找到对应的 block 位置，读取数据



（3）使用 ls -i 命令可以查看某个特定文件对应的 inode 号码：

```
┌──(kali㉿kali)-[~/桌面]
└─$ ls -i cumt
407640 cumt
```

1.1.5 inode 的优点

（1）对于有些无法删除的文件可以通过删除 inode 节点来删除。

（2）移动或者重命名文件时只是改变了目录下的文件名到 inode 的映射，并不需要实际对硬盘操作。

（3）删除文件的时候只需要删除 inode，不需要实际清空那块硬盘，只需要在下次写入的时候覆盖即可（这也是为什么删除了数据可以进行数据恢复的原因之一）。

（4）打开一个文件后，只需要通过 inode 来识别文件。

1.2. 详述硬链接与软链接

1.2.1 硬链接及软链接概念

（1）硬链接

① 硬链接的实质是现有文件在目录树中的另一个入口。也就是说，硬链接与原文件是分居于不同或相同目录下的 dentry 而已，它们指向同一个 inode，对应于相同的磁盘数据块，具有相同的访问权限、属性等。简言之，硬链接其实就是给现有的文件起了一个别名。

② 默认情况下，文件名与 inode 号码是一一对应的关系，每个 inode 号码对应一个文件名称，但是 Unix/Linux 系统允许多个文件名指向同一个 inode 号码，可以用不同的文件名访问同样的内容，对文件内容进行修改，会影响到所有文件名，但是删除一个文件名，不会影响另一个文件名的访问，这种情况称为硬链接。

（2）软链接

① 软链接又称为符号链接（symbolic link），简写为“symlink”。与硬链接仅仅是一个目录项不同，软连接实质上本身也是个文件，不过这个文件的内容是另一个文件名的指针。当 Linux 访问软链接时，它会循着指针找出含有实际数据的目标文件。

② 文件 A 和文件 B 的 inode 号码不同，但是文件 A 的内容是文件 B 的路径，读取文件 A 时，系统会自动访问导向文件 B，因此无论打开哪一个文件，最终都是读取文件 B，这时文件 A 就称为文件 B 的软链接，Windows 下称此方式为快捷方式。

1.2.2 硬链接及软链接创建

（1）硬链接

命令：ln [原文件] [硬链接]

（2）软链接

命令：ln -s [原文件] [软链接]

1.2.3 硬链接及软链接的特点

（1）硬链接

① 文件有相同的 indoe 及文 data block。

② 只能对已存的文件进行创建。

③ 不能垮分区创建硬链接。

④ 不能对目录进行创建，只可对文件创建。

⑤ 删除一个硬链接文件并不影响其他有相同 inode 号的文件。

(2) 软链接

① 软连接有自己的文件属性及权限等。

② 可对不存放在的文件或者目录创建软连接。

③ 软链接可垮分区创建。

④ 软接接可对文件或目录创建。

⑤ 创建软链接时，链接计数不会增加。

⑥ 删除软链接并不影响被指向的文件，但若被指向的原文件被删除，则相关软链接被称为死链接，若被指向路径文件被重新创建，死链接可恢复为正常的软链接。

1.2.4 硬链接及软链接的区别

(1) 硬链接是同一个 inode，只是多个名字。软链接是不同的文件，inode 不同。

(2) 硬链接无法跨分区、跨设备建立，软链接可以。

(3) 硬链接无法创建目录硬链接，软链接可以。

(4) 硬链接没有主次之分，相互独立。软链接依赖于原文件，原文件被删除，软链接即不可用。

(5) 硬链接会删除增加会影响链接数，软链接不会，因为 inode 不一样。

(6) 硬链接创建时，原始文件路径是相对于当前路径。软链接创建时，原始文件路径是相对于软链接的路径。

(7) 硬链接的类型与原始文件类型一致，软链接则会显示 symbolic link。

2 进程通信

请查阅资料，阐述进程通信的分类和方法，说明主要的通信函数和功能。

2.1 进程通信概述

2.1.1 进程通信的概念

进程通信是指在进程间传输数据(交换信息)。进程通信根据交换信息量的多少和效率的高低，分为低级通信（只能传递状态和整数值）和高级通信（提高信号通信的效率，传递大量数据，减轻程序编制的复杂度）。

2.1.2 进程通信的应用场景

（1）数据传输：一个进程需要将它的数据发送给另一个进程。
（2）共享数据：多个进程操作共享数据，一个进程对共享数据的修改，别的进程应该立刻看到。
（3）通知事件：一个进程需要向另一个或一组进程发送消息，通知它(它们)发生了某种事件(如进程终止时要通知父进程)。
（4）资源共享：多个进程之间共享同样的资源。为此，需要内核提供锁和同步机制。
（5）进程控制：有些进程希望完全控制另一个进程的执行(如 Debug 进程)，此时控制进程希望能够拦截另一个进程的所有陷入和异常，并能够及时知道它的状态改变。

2.1.3 进程通信的方式

（1）管道（pipe）
（2）信号量（semophore）
（3）消息队列（message queue）
（4）信号（signal）
（5）共享内存（shared memory）
（6）套接字（socket）

2.2 管道

2.2.1 管道概述

（1）管道是由内核管理的一个缓冲区，它被设计成为环形的数据结构。
（2）管道的两端分别连接两个通信的进程，当两个进程都终结的时候，管道也自动消失。
（3）管道包括三种：无名管道，命名管道和流管道。

2.2.2 无名管道

（1）无名管道概述
① 管道也叫无名管道，它是 UNIX 系统 IPC(进程间通信)的最古老形式，所有的 UNIX 系统都支持这种机制。
（2）无名管道特点
① 半双工，数据在同一时刻只能在一个方向上流动。
② 数据只能从管道的一端写入，另一端读出。
③ 写入管道中的数据遵循先进先出的规则。
④ 管道所传送的数据是无格式的，要求管道的读出方与写入方必须事先约定好数据的格式，如多少字节算一个消息等。
⑤ 管道不是普通的文件，不属于某个文件系统，只存在于内存中。

⑥ 管道在内存中对应一个缓冲区，不同的系统其大小不一定相同。

⑦ 从管道读数据是一次性操作，数据一旦被读走，管道就将其抛弃，释放空间以便写更多的数据。

⑧ 管道没有名字，只能在具有公共祖先的进程(父进程与子进程，或者两个兄弟进程，具有亲缘关系)之间使用。

(3) 函数介绍

① 函数原型

```
int pipe(int filedis[2])
```

② 头文件

```
unistd.h
```

③ 参数

- filedis：返回两个文件描述符
  - filedes[0]：用于读出数据，读取时必须关闭写入端，即 close(filedes[1])
  - filedes[1]：用于写入数据，写入时必须关闭读取端，即 close(filedes[0])

④ 返回值

- 成功：返回 0。
- 失败：返回-1，并设置 error。

2.2.3 命名管道

(1) 命名管道概述

① POSIX 标准中的 FIFO 又名有名管道或命名管道。我们知道前面讲述的 POSIX 标准中的无名管道是没有名称的，所以它的最大劣势是只能用于具有亲缘关系的进程间的通信。FIFO 最大的特性就是每个 FIFO 都有一个路径名与之相关联，从而允许无亲缘关系的任意两个进程间通过 FIFO 进行通信。

② FIFO 在文件系统中表现为一个文件，大部分的系统文件调用都可以用在 FIFO 上面，如：read、open、write、close、unlink、stat 等，但 seek 等函数不能对 FIFO 调用。

(2) 命名管道特点

① FIFO 可以在无关的进程之间交换数据，与无名管道不同。

② FIFO 有路径名与之相关联，它以一种特殊设备文件形式存在于文件系统中。

③ 遵循先进先出的原则，即第一个进来的数据会第一个被读走。

(3) 函数介绍

① 函数原型

```
int mkfifo(const char *filename, mode_t mode)
```

② 头文件

```
sys/types.h
sys/stat.h
```

③ 参数

- filename：是有名管道的路径，包含了有名管道文件的名字，如："/tmp/myfifo"
- mode：是对管道的读写权限，是个八进制数，如 0777

④ 返回值

- 成功：返回 0。
- 失败：返回-1，并设置 error。

(4) 打开 FIFO 文件的四种方式

① 只读、阻塞模式：open(pathname, O_RDONLY)

- 当以阻塞、只读模式打开 FIFO 文件时，将会阻塞直到其他进程以写方式打开访问文

件为止。

② 只读、非阻塞模式：open(pathname, O_RDONLY | O_NONBLOCK)

- 当以非阻塞、只读方式(指定 O_NONBLOCK)打开 FIFO 的时候，立即返回。

③ 只写、阻塞模式：open(pathname, O_WRONLY)

- 当以阻塞、只写模式打开 FIFO 文件时，将会阻塞直到其他进程以读方式打开文件。

④ 只写、非阻塞模式：open(pathname, O_WRONLY | O_NONBLOCK)

- 当以只写、非阻塞方式打开时，如果没有进程为读打开 FIFO，则返回-1，其 errno 是 ENXIO。

2.2.4 流管道

（1）流管道概述

① 流管道(s_pipe)是一种基于文件流的管道，例如用户执行 ls -l 或./pipe 这类很常见的操作，其实都是把一些列操作合并到 popen 函数中完成。

（2）popen()函数

① 函数原型

```
FILE *popen(const char *command, const char *type)
```

② 头文件

```
stdio.h
```

③ 参数

- command：指向的是一个以 null 结束符结尾的字符串，这个字符串包含一个 shell 命令，并被送到/bin/sh 以 c 参数执行，即由 shell 来执行。
- type：
  - ❄ r：文件指针连接到 command 的标准输出，即该命令的结果产生输出。
  - ❄ w：文件指针连接到 command 的标准输入，即该命令的结果产生输出。

④ 返回值

- 成功：返回文件流指针。
- 失败：返回-1，并设置 error。

（3）pclose()函数

① 函数原型

```
int pclose(FILE *stream)
```

② 头文件

```
stdio.h
```

③ 参数

- stream：要关闭的文件流

④ 返回值

- 成功：返回由 popen()所执行的进程的退出码
- 失败：返回-1，并设置 error

2.3 信号量

2.3.1 信号量概述

（1）为了防止出现因多个程序同时访问一个共享资源而引发的一系列问题，我们需要一种方法，它可以通过生成并使用令牌来授权，使得在任一时刻只能有一个执行线程访问代码的临界区域。临界区域是指执行数据更新的代码需要独占式执行的区域。而信号量就可以提供这样的一种访问机制，让一个临界区同一时间只有一个线程在访问它，也就是说信号量是用

来调协进程对共享资源的访问的。

(2) 信号量是一个特殊的变量，程序对其的访问都是原子操作，且只允许对它进行等待（即 P(信号变量))和发送（即 V(信号变量))信息操作。最简单的信号量是只能取 0 和 1 的变量，这也是信号量最常见的一种形式，叫做二进制信号量。而可以取多个正整数的信号量被称为通用信号量。

2.3.2 信号量工作原理

(1) 信号量只能进行等待和发送信号两种操作：

① P(sv)：

sv >0 时，sv = sv - 1

sv = 0 时，挂起正在执行的进程

② V(sv)：

如果有其他进程因等待 sv 而被挂起，就让它恢复运行。

如果没有进程因等待 sv 而挂起，就给它加 1。

(2) 主要作为进程间以及同一进程不同线程之间的同步手段。

2.3.3 信号量的特点

(1) 信号量用于进程间同步，若要在进程间传递数据需要结合共享内存。

(2) 信号量基于操作系统的 PV 操作，程序对信号量的操作都是原子操作。

(3) 每次对信号量的 PV 操作不仅限于对信号量值加 1 或减 1，而且可以加减任意正整数。

(4) 支持信号量组。

2.3.4 函数介绍

(1) 创建信号量

① 函数原型

```
int semget (key_t key, int nsem, int oflag)
```

② 头文件

```
sys/sem.h
```

③ 返回值

- 成功时返回信号量集的 IPC 标识符(一个正整数)
- 失败，则返回-1，errno 被设定成以下的某个值：
  - ❄ EACCES：没有访问该信号量集的权限
  - ❄ EEXIST：信号量集已经存在，无法创建
  - ❄ EINVAL：参数 nsems 的值小于 0 或者大于该信号量集的限制；或者是该 key 关联的信号量集已存在，并且 nsems 大于该信号量集的信号量数
  - ❄ ENOENT：信号量集不存在，同时没有使用 IPC_CREAT
  - ❄ ENOMEM：没有足够的内存创建新的信号量集
  - ❄ ENOSPC：超出系统限制

④ 参数

- key: 所创建或打开信号量集的键值，需要是唯一的非零整数。
- nsem: 创建的信号量集中的信号量的个数，该参数只在创建信号量集时有效，一般为 1。若用于访问一个已存在的集合，那么就可以把该参数指定为 0。
- oflag:
  - ❄ 调用函数的操作类型，有两个值：

    IPC_CREATE：若信号量已存在，返回该信号量标识符

    IPC_EXCL：若信号量已存在，返回错误

- ❄ 也可用于设置信号量集的访问权限：SEM_R (read)和 SEM_A (alter)
- ❄ 两者通过 or 表示

（2）打开信号量

① 函数原型

```
int semop (int semid, struct sembuf * opsptr, size_t nops)
```

② 头文件

```
sys/sem.h
```

③ 参数

- semid：信号量标识符
- opsptr：指向一个信号量操作数组，信号量操作由 sembuf 结构表示：

```
struct sembuf {
short sem_num;   // 除非使用一组信号量，否则它为 0
short sem_op;    // 信号量在一次操作中需要改变的数据，通常是两个数，
                 // 一个是-1，即 P (等待)操作，一个是+1，即 V (发送信号)操作
short sem_flg;   //说明函数 semop 的行为。通常为 SEM_UNDO，使操作系统跟踪
                 // 信号，并在进程没有释放该信号量而终止时，操作系统自动释放该
                 // 进程持有的信号量
};
```

- nops：规定 opsptr 数组中元素个数
  - ❄ sem_op>0：进程释放占用的资源数，该值加到信号量的值上(V 操作)
  - ❄ sem_op<0：要获取该信号量控制的资源数，信号量值减去该值的绝对值(P 操作)
  - ❄ sem_op=0：调用进程希望等待到该信号量值变成 0

（3）操作信号量

① 函数原型

```
int semctl (int semid, int semnum, int cmd, [union semun sem_union])
```

② 头文件

```
sys/sem.h
```

③ 返回值

- 成功，则为一个正数
- 失败，则为-1，同时置 errno 为以下值之一：
  - ❄ EACCESS：权限不够
  - ❄ EFAULT：指向的地址无效
  - ❄ EIDRM：信号量集已经删除
  - ❄ EINVAL：信号量集不存在，或者 semid 无效
  - ❄ EPERM EUID：没有 cmd 的权利
  - ❄ ERANGE：信号量值超出范围

④ 参数

- semnum：指定信号集中的哪个信号(操作对象)
- cmd：指定将执行的命令：

命令	含义
IPC_STAT	读取一个信号量集的数据结构 semid_ds，并将其存储在 semun 中的 buf 参数中
IPC_SET	设置信号量集的数据结构 semid_ds 中的元素 ipc_perm，其值取自 semun 中的 buf 参数

IPC_RMID	删除不再使用的信号量
GETALL	用于读取信号量集中的所有信号量的值
GETNCNT	返回正在等待资源的进程数目
GETPID	返回最后一个执行 semop 操作的进程的 PID
GETVAL	返回信号量集中的一个单个的信号量的值
GETZCNT	返回这在等待完全空闲的资源的进程数目
SETALL	设置信号量集中的所有的信号量的值
SETVAL	设置信号量集中的一个单独的信号量的值

- 第三个参数是可选项，它取决于参数 cmd，其结构如下：

```
union semun{
  int val;    //用于 SETVAL，信号量的初始值
  struct semid_ds *buf;    //用于 IPC_STAT 和 IPC_SET 的缓冲区
  unsigned short *array;    //用于 GETALL 和 SETALL 的数组
};
```

2.4 消息队列

2.4.1 消息队列概述

（1）消息队列是消息的链接表，包括 Posix 消息队列和 system V 消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。消息队列是随内核持续的。

（2）每个消息队列都有一个队列头，用结构 struct msg_queue 来描述。队列头中包含了该消息队列的大量信息，包括消息队列键值、用户 ID、组 ID、消息队列中消息数目等，甚至记录了最近对消息队列读写进程的 ID。读者可以访问这些信息，也可以设置其中的某些信息。

2.4.2 消息队列的特点

（1）在读取消息队列中内容的时候可以选择性的读取(根据消息的类型来读取)。

（2）如果消息队列中消息的类型一样，接收的时候只能按照发送的先后顺序去接收。

（3）如果接收的消息类型没有，会导致接收阻塞。

（4） 如果消息队列不删除，接收消息的时候，那些没有接收的消息依然在队列中存放着。

2.4.3 消息队列中的数据结构

（1）msqid_ds 内核数据结构： Linux 内核维护每个消息队列的结构体，它保存着消息队列当前状态信息。

```
struct msqid_ds{
  struct ipc_perm msg_perm;    //所有者和权限
  time_t msg_stime;    //最后一次调用 msgsnd 的时间
  time_t msg_rtime;    //最后一次调用 msgrcv 的时间
  time_t msg_ctime;    //队列最后一次变动的时间
  unsigned long __msg_cbytes;    //当前队列中字节数(不标准)
  msgqnum_t msg_qnum;    //当前队列中消息数
  msglen_t msg_qbytes;    //队列中允许的最大字节数
  pid_t msg_lspid;    //最后一次调用 msgsnd 的 PID
  pid_t msg_lrpid;    //最后一次调用 msgrcv 的 PID
```

```
};
```

（2）Linux 内核在结构体 ipc_perm 中保存消息队列的一些重要的信息，如消息队列关联的键值、消息队列的用户 id、组 id 等。

```
struct ipc_perm{
  key_t key;   //消息队列键值
  uid_t uid;   //有效的拥有者 UID
  gid_t gid;   //有效的拥有者 GID
  uid_t cuid;   //有效的创建者 UID
  gid_t cgid;   //有效的创建者 GID
  unsigned short mode;   //权限
  unsigned short seq;   //队列号
};
```

2.4.4 函数介绍

（1）创建消息队列

① 函数原型

```
int msgget(key_t key,int msgflg)
```

② 头文件

```
sys/types.h
sys/ipc.h
sys/msg.h
```

③ 参数

- key：命名消息队列的键，一般用 ftok 函数获取。
- msgflg：消息队列的访问权限，可以与以下键或操作：IPC_CREAT：不存在则创建，存在则返回已有的 qid。

④ 返回值

- 成功：返回以 key 命名的消息队列的标识符(非零正整数)。
- 失败：返回-1。

（2）发送消息(将消息添加到消息队列)

① 函数原型

```
int msgsnd(int msgid,const void* msgp,size_t magsz,int msgflg)
```

② 头文件

```
sys/types.h
sys/ipc.h
sys/msg.h
```

③ 参数

- msgid：由 msgget 函数返回的消息队列标识符。
- msgp：将发往消息队列的消息结构体指针。
- msgsz：消息长度，是消息结构体中待传递数据的大小(不是整个结构体的大小)。
- msgflg：
  - IPC_NOWAIT：消息队列满时返回-1。
  - 0：消息队列满时阻塞。

④ 返回值

- 成功：消息数据的一份副本将被放到消息队列中，并返回 0。
- 失败：返回-1。

（3）接收消息(从消息队列中获取消息)

① 函数原型

```
ssize_t msgrcv(int qid,void *msgp,size_t msgsz,long msgtype,int msgflg)
```

② 头文件

```
sys/types.h
sys/ipc.h
sys/msg.h
```

③ 参数

- msgid：由 msgget 函数返回的消息队列标识符。
- msgp：将发往消息队列的消息结构体指针。
- msgsz：消息长度，是消息结构体中待传递数据的大小(不是整个结构体的大小)。
- msgflg：
  - IPC_NOWAIT：消息队列满时返回-1。
  - 0：消息队列满时阻塞。
- msgtype：可以实现一种简单的接收优先级。如果 msgtype 为 0，就获取队列中的第一个消息。如果它的值大于零，将获取具有相同消息类型的第一个信息。如果它小于零，就获取类型等于或小于 msgtype 的绝对值的第一个消息。

④ 返回值

- 成功：返回放到接收缓存区中的字节数，消息被复制到由 msgp 指向的用户分配的缓存区中，然后删除消息队列中的对应消息。
- 失败：返回-1。

（4）控制消息队列

① 函数原型

```
int msgctl(int msgid,int cmd,struct msgid_ds *buf)
```

② 头文件

```
sys/types.h
sys/ipc.h
sys/msg.h
```

③ 参数

- msgid：由 msgget 函数返回的消息队列标识符。
- cmd：将要采取的动作，它可以取 3 个值之一：
  - IPC_STAT：用来获取消息队列信息，并存储在 buf 指向的 msgid_ds 结构中。
  - IPC_SET：用来设置消息队列的属性，要设置的属性存储在 buf 指向的 msgid_ds 结构中。
  - IPC_RMID：删除 msqid 标识的消息队列。
- buf：指向 msgid_ds 权限结构，它至少包括以下成员：

```
struct msgid_ds
{
  uid_t shm_perm.uid;
  uid_t shm_perm.gid;
  mode_t shm_perm.mode;
  ……
};
```

④ 返回值

- 成功：返回 0。
- 失败：返回-1。

2.4.5 消息队列可能存在的隐患

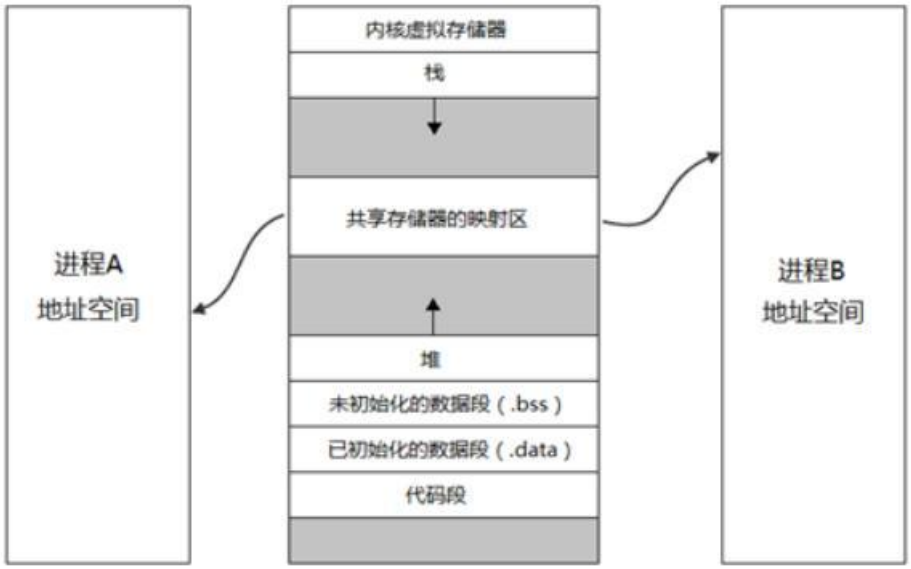
当 64 位应用向 32 位应用发送一消息时，如果它在 8 字节字段中设置的值大于 32 位应用中 4 字节类型字段可表示值，那么 32 位应用在其 mtype 字段中得到的是一个截短了的值，于是也就丢失了信息。

2.5 共享内存

2.5.1 共享内存概述

（1）共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。

（2）共享内存是最快的 IPC 方式，往往与其他通信机制，如信号量配合使用，来实现进程间的同步和通信。



2.5.2 函数介绍

（1）创建/获取共享内存：shmget 函数

① 函数原型

```
int shmget(key_t key, size_t size, int shmflg)
```

② 头文件

```
sys/ipc.h
sys/shm.h
```

③ 参数

- key：共享内存段的名字，通常用 ftok()函数获取。
  - ❄ 0(IPC_PRIVATE)：会建立新共享内存对象
  - ❄ 大于 0 的 32 位整数：视参数 shmflg 来确定操作。通常要求此值来源于 ftok 返回的 IPC 键值
- size：以字节为单位的共享内存大小。内核以页为单位分配内存，但最后一页的剩余部分内存不可用。
  - ❄ 大于 0 的整数：新建的共享内存大小，以字节为单位
  - ❄ 0：只获取共享内存时指定为 0

- shmflg：九个比特的权限标志(其作用与文件 mode 模式标志相同)，并与 IPC_CREAT 或时创建共享内存段。
  - ❄ 0：取共享内存标识符，若不存在则函数会报错
  - ❄ IPC_CREAT：当 shmflg&IPC_CREAT 为真时，如果内核中不存在键值与 key 相等的共享内存，则新建一个共享内存；如果存在这样的共享内存，返回此共享内存的标识符
  - ❄ IPC_CREAT|IPC_EXCL：如果内核中不存在键值与 key 相等的共享内存，则新建一个消息队列；如果存在这样的共享内存则报错

④ 返回值

- 成功：非负整数，即该共享内存段的标识码。
- 失败：返回-1。
  - ❄ EINVAL：参数 size 小于 SHMMIN 或大于 SHMMAX
  - ❄ EEXIST：预建立 key 所指的共享内存，但已经存在
  - ❄ EIDRM：参数 key 所指的共享内存已经删除
  - ❄ ENOSPC：超过了系统允许建立的共享内存的最大值(SHMALL)
  - ❄ ENOENT：参数 key 所指的共享内存不存在，而参数 shmflg 未设 IPC_CREAT
  - ❄ EACCES：没有权限
  - ❄ ENOMEM：核心内存不足

(2) 将共享内存段连接到进程地址空间：shmat 函数(挂接)

① 函数原型

```
void *shmat(int shmid, const void *shmaddr, int shmflg)
```

② 头文件

```
sys/ipc.h
sys/shm.h
```

③ 说明

- 共享内存段刚创建时不能被任何进程访问，必须将其连接到一个进程的地址空间才能使用。

④ 参数

- shmid：由 shmget 返回的共享内存标识
- shmaddr：指定共享内存连接到当前进程中的地址位置。
  - ❄ shmaddr = NULL：核心自动选择一个地址
  - ❄ shmaddr<>NULL 且 shmflg = SHM_RND：以 shmaddr 为连接地址
  - ❄ shmaddr<>NULL 且 shmflg = SHM_RND：连接的地址会自动向下调整为 SHMLBA 的整数倍，公式：shmaddr-(shmaddr%SHMLBA)
- shmflg：它有两个可能取值，用来控制共享内存连接的地址
  - ❄ SHM_RND：以 shmaddr 为连接地址
  - ❄ SHM_RDONLY：表示连接操作用来只读共享内存

⑤ 返回值

- 成功：指向共享内存第一个节的指针。
- 失败：返回-1。
  - ❄ EACCES：无权限以指定方式连接共享内存
  - ❄ EINVAL：无效的参数 shmid 或 shmaddr
  - ❄ ENOMEM：核心内存不足

(3) 将共享内存段与当前进程脱离：shmdt 函数(去关联)

① 函数原型

```
int shmdt(const void *shmaddr)
```

② 头文件

```
sys/ipc.h
sys/shm.h
```

③ 功能

- 将共享内存从当前进程中分离
- 共享内存分离并未删除它，只是使得该共享内存对当前进程不再可用

④ 参数

- shmaddr：shmat 所返回的指针

⑤ 返回值

- 成功：返回 0。
- 失败：返回-1。

（4）共享内存控制：shmctl 函数

① 函数原型

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

② 头文件

```
sys/ipc.h
sys/shm.h
```

③ 参数

- shmid：由 shmget 返回的共享内存标识码
- cmd：将要采取的动作，有三个可取的值
  - IPC_STAT：把 shmid_ds 结构中的数据设置为共享内存的当前关联值
  - IPC_SET：如果有足够的权限，把共享内存的当前值设置为 shmid_ds 数据结构中给出的值
  - IPC_RMID：删除共享内存段
- buf：用于保存共享内存模式状态和访问权限，至少包含以下成员

```
struct shmid_ds {
  uid_t shm_perm.uid;
  uid_t shm_perm.gid;
  mode_t shm_perm.mode; }
```

④ 返回值

- 成功：返回 0。
- 失败：返回-1。
  - EACCESS：参数 cmd 为 IPC_STAT，确无权限读取该共享内存
  - EFAULT：参数 buf 指向无效的内存地址
  - EIDRM：标识符为 msqid 的共享内存已被删除
  - EINVAL：无效的参数 cmd 或 shmid
  - EPERM：参数 cmd 为 IPC_SET 或 IPC_RMID，却无足够的权限执行

2.6 信号

2.6.1 信号概述

（1）信号是 UNIX 和 Linux 系统响应某些条件而产生的一个事件，接收到该信号的进程会相应地采取一些行动。通常信号是由一个错误产生的。但它们还可以作为进程间通信或修改行为的

一种方式，明确地由一个进程发送给另一个进程。一个信号的产生叫生成，接收到一个信号叫捕获。

(2) 在 Linux 中，每个信号都有一个名字和编号，这些名字都以 SIG 开头，例如 SIGIO、SIGCHLD 等。信号定义在 signal.h 头文件中，信号名都定义为正整数。具体的信号名称可以使用 kill-l 来查看信号的名字以及序号，信号是从 1 开始编号的，不存在 0 号信号。kill 对于信号 0 又特殊的应用。

2.6.2 信号的产生

(1) 由硬件产生，如从键盘输入 Ctrl+C 可以终止进程

(2) 由其他进程发送，如 shell 下用命令 kill -信号标号 PID 可以向指定进程发送信号

(3) 异常，进程异常时会发送信号

2.6.3 信号的处理

(1) 信号处理概述

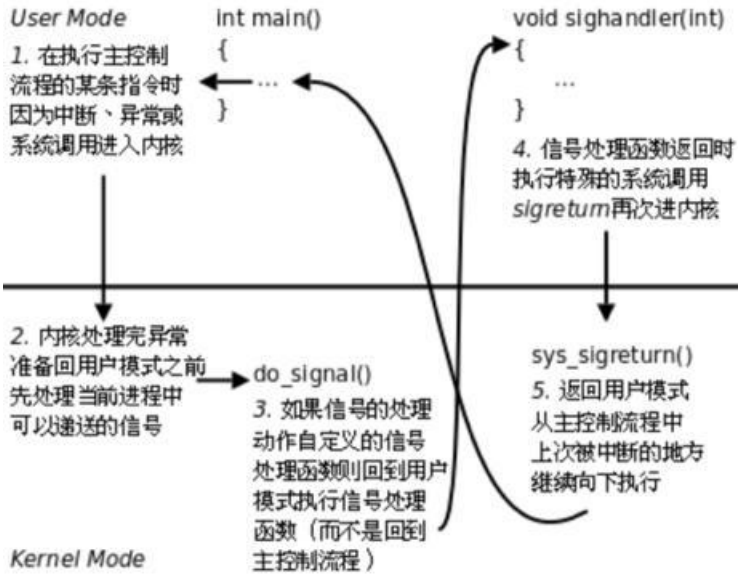
① 信号是由操作系统来处理的，说明信号的处理在内核态

② 信号不一定会被立即处理，有时会储存在信号的信号表中

③ 信号处理的三种方式：

- 忽略
- 默认处理方式：操作系统设定的默认处理方式
- 自定义信号处理方式：可自定义信号处理函数

④ 信号处理过程：



(2) 信号处理函数：signal

① 函数原型

```
void (*signal(int sig, void (*func)(int) ) ) (int)
```

② 头文件

```
signal.h
```

③ 功能

- 用于处理指定的信号，主要通过忽略和恢复其默认行为来工作

④ 参数

- sig：信号值
- func：信号处理函数指针，参数为信号值

❄ 信号处理函数的原型必须为 void func(int)，或者是下面的特殊值：

SIG_IGN：忽略信号的处理

SIG_DFL：恢复信号的默认处理

⑤ 返回值

- 成功：返回信号处理程序之前的值
- 失败：返回 SIG_ER

（2）信号处理函数：sigaction

① 函数原型

```
int sigaction(int sig,const struct sigaction *act,struct sigaction *oact)
```

② 头文件

```
signal.h
```

③ 功能

- 设置与信号 sig 关联的动作

④ 参数

- sig：信号值
- act：指定信号的动作
- oact：保存原信号的动作

⑤ 返回值

- 成功：返回 0
- 失败：-1，错误原因存于 error 中

⑥ 说明

- sigaction 函数有阻塞的功能：默认情况下，在信号处理函数未完成之前，发生的新的 SIGINT 信号将被阻塞，同时对后续来的 SIGINT 信号进行排队合并处理。

⑦ sigaction 结构体

```
stuct sigaction
{
 void (*)(int) sa_handle;   //处理函数指针，相当于 signal 函数的 func 参数
 sigset_t sa_mask;   //被屏蔽的信号，可以消除信号间的竞态
 int sa_flags;   //处理函数执行完后，信号处理方式修改。如 SA_RESETHAND，
                 //将信号处理方式重置为 SIG_DFL
}
```

2.6.4 信号的阻塞

（1）信号阻塞概述

① 阻塞是阻止进程收到该信号，此时信号处于未决状态，放入进程的未决信号表中，当解除对该信号的阻塞时，未决信号会被进程接收。

（2）信号阻塞函数：sigprocmask

① 函数原型

```
int sigprocmask(int how,const sigset_t *set,sigset_t *oset)
```

② 头文件

```
signal.h
```

③ 功能

- 可以用来检测或改变信号屏蔽字

④ 参数

- how：设置 block 阻塞表的方式
  - SIG_BLOCK：将信号集添加到 block 表中
  - SIG_UNBLOCK：将信号集从 block 表中删除
  - SIG_SETMASK：将信号集设置为 block 表
- set：为指向信号集的指针，在此专指新设的信号集，如果仅想读取现在的屏蔽值，可将其置为 NULL。
- oset：也是指向信号集的指针，在此存放原来的信号集。

⑤ 返回值

- 成功：返回 0
- 失败：-1，错误原因存于 error 中

（3）未决信号获取函数：sigpending

① 函数原型

```
int sigpending(sigset_t *set)
```

② 头文件

```
signal.h
```

③ 功能

- 获取当前进程所有未决的信号

④ 参数

- set：存储获得的当前进程的 pending 未决表中的信号集

⑤ 返回值

- 成功：返回 0
- 失败：-1，错误原因存于 error 中

2.6.5 信号的发送

（1）信号发送函数：kill

① 函数原型

```
int kill(pid_t pid, int signo)
```

② 头文件

```
signal.h
sys/types.h
```

③ 功能

- 发送信号给进程或进程组，可以是本身或其它进程

④ 参数

- pid：
  - >0：发送给进程 ID
  - 0：发送给所有和当前进程在同一个进程组的进程
  - -1：发送给所有的进程表中的进程(进程号最大的除外)
  - <-1：发送给进程组号为-PID 的每一个进程
- signo：信号值

⑤ 返回值

- 成功：返回 0
- 失败：-1，错误原因存于 error 中
  - 给定的信号无效(errno = EINVAL)
  - 发送权限不够(errno = EPERM)
  - 目标进程不存在(errno = ESRCH)

（2）信号发送函数：raise

① 函数原型

```
int raise(int sig)
```

② 头文件

```
signal.h
```

③ 功能

- 给当前进程发送指定信号

④ 参数

- sig：要发送的信号码，下面是一些重要的标准信号常量：
  - SIGABRT：程序异常终止。
  - SIGFPE：算术运算出错，如除数为 0 或溢出（不一定是浮点运算）。
  - SIGILL：非法函数映象，如非法指令，通常是由于代码中的某个变体或者尝试执行数据导致的。
  - SIGINT：中断信号，如 ctrl-C，通常由用户生成。
  - SIGSEGV：非法访问存储器，如访问不存在的内存单元。
  - SIGTERM：发送给本程序的终止请求信号。

⑤ 返回值

- 成功：返回 0
- 失败：-1，错误原因存于 error 中

（3）信号发送函数：abort

① 函数原型

```
void abort(void)
```

② 头文件

```
stdlib.h
```

③ 功能

- 发送 SIGABRT 信号，让进程异常终止，发生转储

（4）信号发送函数：pause

① 函数原型

```
int pause(void)
```

② 头文件

```
unistd.h
```

③ 功能

- pause()函数用于将调用进程挂起直至捕捉到信号为止，这个函数通常可以用于判断信号是否已到

④ 返回值

- 成功：返回 0
- 失败：-1，同时把 errno 设置为 EINTR

（5）信号发送函数：alarm

① 函数原型

```
unsigned int alarm(unsigned int seconds)
```

② 头文件

```
unistd.h
```

③ 功能

- 发送 SIGALRM 闹钟信号

④ 参数

- seconds：系统经过 seconds 秒后向进程发送 SIGALRM 信号

⑤ 返回值

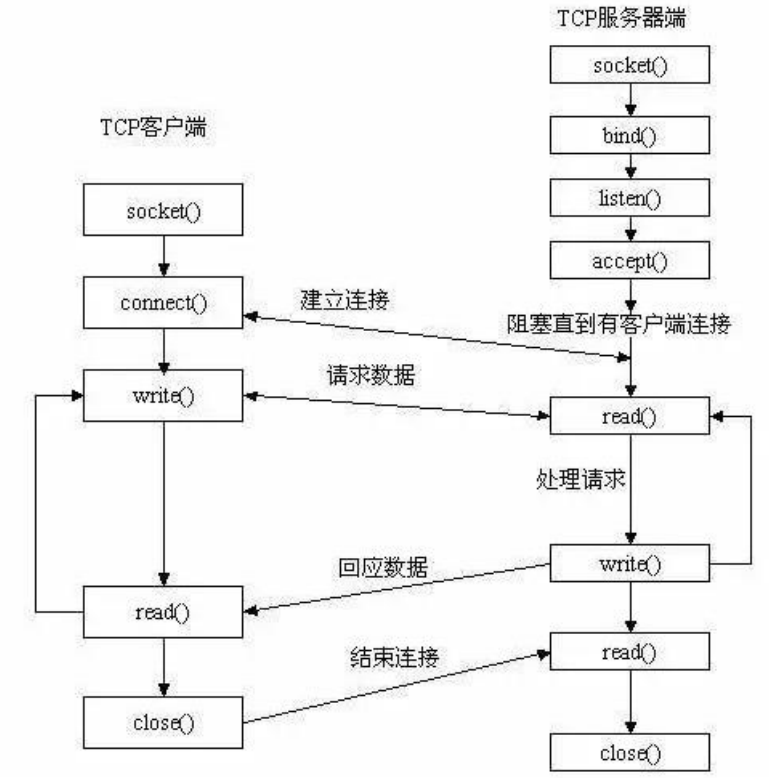
- 成功：如果调用 alarm()前，进程中已经设置了闹钟时间，则返回上一个闹钟的剩余时间，否则返回 0
- 失败：返回-1

2.7 套接字

2.7.1 套接字通信概述

（1）Socket 是在应用层和传输层之间的一个抽象层，它把 TCP/IP 层复杂的操作抽象为几个简单的接口，供应用层调用实现进程在网络中的通信。Socket 起源于 UNIX，在 Unix 一切皆文件的思想下，进程间通信就被冠名为文件描述符（file desciptor）。Socket 是一种“打开—读/写—关闭”模式的实现，服务器和客户端各自维护一个“文件”，在建立连接打开后，可以向文件写入内容供对方读取或者读取对方内容，通讯结束时关闭文件。

（2）Socket 保证了不同计算机之间的通信，也就是网络通信。对于网站，通信模型是服务器与客户端之间的通信。两端都建立了一个 Socket 对象，然后通过 Socket 对象对数据进行传输。通常服务器处于一个无限循环，等待客户端的连接，大致流程如下：



2.7.2 函数介绍

（1）socket

① 函数原型

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

② 头文件

```
sys/types.h
sys/socket.h
```

③ 功能

- socket 系统调用常用于创建一个 socket 描述符。

④ 参数

- protocol：由于指定了 type，此值一般为 0
- domain：程序采用的通讯协族
  - AF_UNIX：只用于单一的 Unix 系统进程间通信
  - AF_INET：用于 Internet 通信
- type：采用的通讯协议
  - SOCK_STREAM：使用 TCP 协议
  - SOCK_DGRAM：使用 UDP 协议

⑤ 返回值

- 成功：返回 socket 描述符
- 失败：返回-1。

（2）bind

① 函数原型

```
int bind(int sockfd，struct sockaddr* servaddr，int addrlen)
```

② 头文件

```
sys/types.h
sys/socket.h
```

③ 功能

- bind 系统调用常用于 server 程序，绑定被侦听的端口。

④ 参数

- sockfd：由 socket 调用返回的文件描述符
- servaddr：出于兼容性，一般使用 sockaddr_in 结构
- addrlen：servaddr 结构的长度

⑤ 返回值

- 成功：返回 0。
- 失败：返回-1。

（3）connect

① 函数原型

```
int connect(int sockfd，struct sockaddr* servaddr，int addrlen)
```

② 头文件

```
sys/types.h
sys/socket.h
```

③ 功能

- connect 系统调用常用于 Client 程序，连接到某个 Server。

④ 参数

- sockfd：socket 返回的文件描述符
- servaddr：被连接的服务器端地址和端口信息，出于兼容性，一般使用 sockaddr_in 结构
- addrlen：servaddr 的长度

⑤ 返回值

- 成功：返回 0。
- 失败：返回-1。

（4）listen

① 函数原型

```
int listen(int sockfd，int backlog)
```

② 头文件

```
sys/types.h
sys/socket.h
```

③ 功能

- listen 系统调用常用于 server 程序，侦听 bind 绑定的套接字。

④ 参数

- sockfd：被 bind 的文件描述符（socket()建立的）
- backlog：设置 Server 端请求队列的最大长度

⑤ 返回值

- 成功：返回 0。
- 失败：返回-1。

(5) accept

① 函数原型

```
int accept(int sockfd，struct sockaddr* addr，int *addrlen)
```

② 头文件

```
sys/types.h
sys/socket.h
```

③ 功能

- Server 用它响应连接请求，建立与 Client 连接

④ 参数

- sockfd：listen 后的文件描述符（socket()建立的）
- addr：返回 Client 的 IP、端口等信息，确切格式由套接字的地址类别（如 TCP 或 UDP）决定。若 addr 为 NULL，则 addrlen 应置为 NULL
- addrlen：返回真实的 addr 所指向结构的大小，只要传递指针就可以，但必须先初始化为 addr 所指向结构的大小

⑤ 返回值

- 成功：Server 用于与 Client 进行数据传输的文件描述符
- 失败：-1，相应地设定全局变量 errno。

(6) recv

① 函数原型

```
ssize_t recv(int sockfd，void *buf，size_t nbytes，int flags)
```

② 头文件

```
sys/types.h
sys/socket.h
```

③ 功能

- recv 系统调用常用于 TCP 协议中接收信息。

④ 参数

- sockfd：接收端套接字描述符
- buf：指向容纳接收信息的缓冲区的指针
- nbytes：buf 缓冲区的大小
- flags：接收标志，一般置为 0 或：
  - MSG_DONTWAIT：仅本操作非阻塞
  - MSG_OOB：发送或接收带外数据
  - MSG_PEEK：窥看外来消息
  - MSG_WAITALL：等待所有数据

⑤ 返回值

- 成功：返回实际接收的字节数
- 失败：返回-1，相应地设定全局变量 errno
- 为 0：表示对端已经关闭

(7) recvfrom

① 函数原型

```
ssize_t recvfrom(int sockfd，void *buf，size_t len，int flags，struct sockaddr *from，socklen_t *fromlen)
```

② 头文件

```
sys/types.h
sys/socket.h
```

③ 功能

- Recvfrom 系统调用常用于 UDP 协议中接收信息。

④ 参数

- sockfd：socket 描述符
- buf：指向容纳接收 UDP 数据报的缓冲区的指针
- len：buf 缓冲区的大小
- flags：接收标志，一般为 0
- from：指明数据的来源
- fromlen：传入函数之前初始化为 from 的大小，返回之后存放 from 实际大小

⑤ 返回值

- 成功：返回实际接收的字节数
- 失败：返回-1，相应地设定全局变量 errno
- 为 0：表示对端已经关闭

(8) send

① 函数原型

```
ssize_t send(int sockfd，const void *buf，size_t nbytes，int flags)
```

② 头文件

```
sys/types.h
sys/socket.h
```

③ 功能

- send 系统调用常用于 TCP 协议中发送信息。

④ 参数

- sockfd：指定发送端套接字描述符
- buf：存放要发送数据的缓冲区
- nbytes：实际要发送的数据的字节数
- flags：接收标志，一般置为 0 或：
  - MSG_DONTROUTE：绕过路由表查找
  - MSG_DONTWAIT：仅本操作非阻塞
  - MSG_OOB：发送或接收带外数据

⑤ 返回值

- 成功：返回已发送的字节数
- 失败：返回-1，相应地设定全局变量 errno

(9) sendto

① 函数原型

```
int sendto(int sockfd，const void *msg，int len，unsigned int flag，const struct sockaddr *to，
int tolen)
```

② 头文件

```
sys/types.h
sys/socket.h
```

③ 功能

- sendto 系统调用常用于 UDP 协议中发送信息。

④ 参数

- sock：将要从其发送数据的套接字
- buf：指向将要发送数据的缓冲区
- len：数据缓冲区的长度
- flags：一般是 0
- to：指明数据的目的地
- tolen：to 内存区的长度

⑤ 返回值

- 成功：实际传送出去的字符数
- 失败：返回-1，错误原因存于 errno 中

（10）close

① 函数原型

```
int close(int sockfd)
```

② 头文件

```
sys/types.h
sys/socket.h
```

③ 功能

- close 系统调用常用于 TCP，关闭特定的 socket 连接

④ 参数

- sockfd：要关闭的 socket 描述符

⑤ 返回值

- 成功：返回 0
- 失败：返回-1，并置错误码 errno
  - EBADF：sockfd 不是一个有效描述符
  - EINTR：close 函数被信号中断
  - EIO：IO 错误

（11）read

① 函数原型

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

② 头文件

```
unistd.h
```

③ 功能

- read 系统调用负责从 socket 描述符对应的 socket 中读取内容。

④ 参数

- fd：文件描述符
- buf：指向内存块的指针，存放从文件中读出的字节
- count：写入到 buf 中的字节数

⑤ 返回值

- 成功：返回实际所读取的字节数，如果已经读取到文件的结尾则返回 0
- 失败：返回小于 0 的数

（12）write

① 函数原型

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

② 头文件

```
unistd.h
```

③ 功能

- write 系统调用常用于向发送缓冲区写数据。

④ 参数

- fd：文件描述符
- buf：指向内存块的指针，从该内存块读出数据写入文件
- count：写入到文件中的字节数

⑤ 返回值

- 成功：返回写入的字节数
- 失败：返回-1 并设置 error