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 2019-1 班 学生姓名 李春阳 学 号 10193657

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

2022 年 3 月 20 日

目录

1 文件系统

1.1. 详述 Linux 的节点 inode

1.1.1 inode 简介

inode 是理解 Unix/Linux 文件系统和硬盘储存的基础。理解 inode，要从文件储存说起。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”。

1.1.2 inode 的内容

node 包含文件的元信息，具体来说有以下内容：

- 文件的字节数
- 文件的拥有者 uid
- 文件的所属组 gid
- 文件的 r、w、x 权限
- 文件的时间戳
  - ctime：文件的 inode 上一次变动的时间
  - mtime：文件内容上一次变动的时间
  - atime：文件上一次打开的时间
- 硬链接数
- 文件数据 block 的位置

总之，除了文件名以外的所有文件信息，都存在 inode 之中。

1.1.3 inode 的大小

inode 也会消耗硬盘空间，所以硬盘格式化的时候，操作系统自动将硬盘分成两个区域。一个是数据区，存放文件数据；另一个是 inode 区（inode table），存放 inode 所包含的信息。

每个 inode 节点的大小，一般是 128 字节或 256 字节。inode 节点的总数，在格式化的时候就给定，一般是每 1kb 或每 2kb 就设置一个 inode。假定在一块 1GB 的硬盘上，每个 inode 节点的大小为 128 字节，每 1kb 就设置一个 inode，那么 inode table 的大小就会达到 128MB，占整块硬盘的 12.8%。

查看每个硬盘分区的 inode 总数和已经使用的数量，可以使用命令：

df -i

```
[lcy@VM-16-10-centos /]$ df -i
Filesystem      Inodes  IUsed   IFree IUse% Mounted on
devtmpfs        482291    322  481969    1% /dev
tmpfs           485021      9  485012    1% /dev/shm
tmpfs           485021    505  484516    1% /run
tmpfs           485021     16  485005    1% /sys/fs/cgroup
/dev/vda1      3932160 167175 3764985    5% /
tmpfs           485021      1  485020    1% /run/user/0
tmpfs           485021      1  485020    1% /run/user/1003
[lcy@VM-16-10-centos /]$
```

查看每个 inode 节点的大小，可以使用命令：

sudo dumpe2fs -h /dev/hda | grep "Inode size"

```
[lcy@VM-16-10-centos /]$ dumpe2fs -h /dev/hda | grep "Inode size"
dumpe2fs 1.42.9 (28-Dec-2013)
dumpe2fs: No such file or directory while trying to open /dev/hda
[lcy@VM-16-10-centos /]$
```

由于每个文件都必须有一个 inode，因此有可能发生 inode 已经用光，但是硬盘还未存满的情况。这时，就无法在硬盘上创建新文件。

1.1.4 inode 号码

每个 inode 都有一个号码，操作系统用 inode 号码来识别文件。对于系统来说，文件名只是 inode 号码便于识别的别称或绰号。

表面上，用户通过文件名，打开文件。实际上，系统内部这个过程分为三步：

1. 系统找到这个文件名对应的 inode 号码
2. 通过 inode 号码，获取 inode 信息
3. 根据 inode 信息，找到文件数据所在的 block，读出数据

查看文件所对应的 inode 号码：

ls -i example

```
[lcy@VM-16-10-centos /]$ cd sts-2.1.2/
[lcy@VM-16-10-centos sts-2.1.2]$ ls
finalAnalysisReport.txt  sts-2.1.2  test
[lcy@VM-16-10-centos sts-2.1.2]$ cd test/
[lcy@VM-16-10-centos test]$ ls
__init__.py  test1.txt  test4.txt  test.py  test.txt
[lcy@VM-16-10-centos test]$ ls -i
2753067 __init__.py  2753066 test1.txt  2753070 test4.txt  2753071 test.py  2753068 test.txt
[lcy@VM-16-10-centos test]$ ls -i test.py
2753071 test.py
[lcy@VM-16-10-centos test]$
```

1.1.5 目录文件

Unix/Linux 系统中，目录（directory）也是一种文件。打开目录，实际上就是打开目录文件。

目录文件的结构非常简单，就是一系列目录项（dirent）的列表。每个目录项，由两部分组成：所包含文件的文件名，该文件名对应的 inode 号码

列出整个目录文件，即文件名和 inode 号码：

```
[lcy@VM-16-10-centos /]$ cd sts-2.1.2/
[lcy@VM-16-10-centos sts-2.1.2]$ ls
finalAnalysisReport.txt  sts-2.1.2  test
[lcy@VM-16-10-centos sts-2.1.2]$ cd test/
[lcy@VM-16-10-centos test]$ ls
__init__.py  test1.txt  test4.txt  test.py  test.txt
[lcy@VM-16-10-centos test]$ ls -i
2753067 __init__.py  2753066 test1.txt  2753070 test4.txt  2753071 test.py  2753068 test.txt
[lcy@VM-16-10-centos test]$ ls -i test.py
2753071 test.py
[lcy@VM-16-10-centos test]$
```

查看文件的详细信息，必须根据 inode 号码，访问 inode 节点，读取信息。 ls -l 命令列出文件的详细信息。

```
[lcy@VM-16-10-centos test]$ ls -l
total 69716
-rwxr-xr-x 1 root root        0 Mar 13 23:18 __init__.py
-rwxr-xr-x 1 root root 71303168 Mar 13 23:18 test1.txt
-rw-r--r-- 1 root root    68640 Mar 17 15:30 test4.txt
-rwxr-xr-x 1 root root     7980 Mar 17 15:29 test.py
-rw-r--r-- 1 root root     6336 Mar 14 10:13 test.txt
[lcy@VM-16-10-centos test]$
```

操作系统读取磁盘文件的流程是这样的：

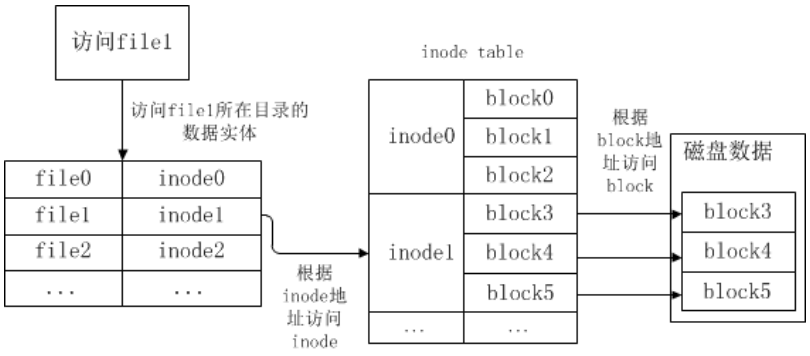
（1）根据给定的文件的所在目录，获取该目录的数据实体，根据数据实体中的数据项，找到对应文件的 inode；

（2）根据文件 inode，找到 inodeTable；

（3）根据 inodeTable 中的对应关系，找到对应的 block；

（4）读取文件。

系统读取磁盘文件流程示意图如下：



举例来说，如果想要读取/etc/passwd 文件，读取流程如下：

（1）获取根目录/的 inode。透过挂载点的信息找到根目录的 inode 号为 2；

（2）根据根目录的 inode，找到根目录的数据实体 block，可以理解为一个文件到 inode 号的映射表，找到目录 etc 的 inode 号；

（3）根据目录 etc 的 inode 号，读取目录 etc 的数据实体 block，并找到文件 passwd 的 inode 号；

（4）根据/etc/passwd 文件的 inode 号，即可获取/etc/passwd 文件的数据实体 block，完成文件的读取。

```
[lcy@VM-16-10-centos test]$ cd /
[lcy@VM-16-10-centos /]$ ll -di /
2 dr-xr-xr-x. 22 root root 4096 Mar 19 10:17 /
[lcy@VM-16-10-centos /]$ ll -di /etc
262147 drwxr-xr-x. 95 root root 12288 Mar  9 21:48 /etc
[lcy@VM-16-10-centos /]$ ll -i /etc/passwd
262850 -rw-r--r-- 1 root root 1315 Mar  9 09:07 /etc/passwd
[lcy@VM-16-10-centos /]$
```

1.1.6 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，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。

1.1.7 inode 查看命令

1.stat

功能：列出文件大小，文件所占的块数，块的大小，主设备号和次设备号，inode number，链接数，访问权限，uid,gid,atime,mtime,ctime

```
[lcy@VM-16-10-centos ~]$ ls
test
[lcy@VM-16-10-centos ~]$ stat test/
  File: 'test/'
  Size: 4096            Blocks: 8          IO Block: 4096   directory
Device: fd01h/64769d    Inode: 1445069     Links: 2
Access: (0775/drwxrwxr-x)  Uid: ( 1003/     lcy)   Gid: ( 1003/     lcy)
Access: 2022-03-09 20:54:00.930888072 +0800
Modify: 2022-03-10 14:00:43.200596954 +0800
Change: 2022-03-10 14:00:43.200596954 +0800
 Birth: -
[lcy@VM-16-10-centos ~]$
```

2.df -l

功能：查看硬盘的 i 节点总数和使用的个数.文件系统，总块数，已用块数，可用块数，已用所占比例，挂载点

```
[lcy@VM-16-10-centos ~]$ df -l
Filesystem     1K-blocks    Used Available Use% Mounted on
devtmpfs         1929164       0   1929164   0% /dev
tmpfs            1940084      32   1940052   1% /dev/shm
tmpfs            1940084     672   1939412   1% /run
tmpfs            1940084       0   1940084   0% /sys/fs/cgroup
/dev/vda1       61795096 6499212  52679260  11% /
tmpfs             388020       0    388020   0% /run/user/0
tmpfs             388020       0    388020   0% /run/user/1003
[lcy@VM-16-10-centos ~]$
```

3.ls -i

功能：查看目录下各文件的 inode number

```
[lcy@VM-16-10-centos ~]$ ls -i
1445069 test
[lcy@VM-16-10-centos ~]$ cd /
[lcy@VM-16-10-centos /]$ ls -i
   3505 bin     262147 etc          11 lost+found  2883585 patch       33 sbin          8193 tmp
 262152 boot    393721 home     393722 media             1 proc    393725 srv           14 usr
 399635 data        25 lib      393723 mnt          393219 root   2752513 sts-2.1.2  393220 var
   1026 dev         27 lib64    393724 opt            1197 run          1 sys       1048577 www
[lcy@VM-16-10-centos /]$
```

1.2. 详述硬链接与软链接

1.2.1 硬链接

一般情况下，文件名和 inode 号码是"一一对应"的关系，每个 inode 号码对应一个文件名（每个文件默认有一个硬链接）。但是，Unix/Linux 系统允许多个文件名指向同一个 inode 号码。

这意味着，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后，会影响所有文件名。但是，删除一个文件名，不影响另一个文件名的访问。这种情况就被称为"硬链接"（hard link）。

创建一个硬链接，就会为文件创建了一个新的文件名。硬链接有两个重要局限性：

硬链接不能链接不在同一系统的文件。也就是说硬链接不能链接与文件不在同一磁盘分区上的文件；

硬链接不能链接目录。

一个硬链接和文件本身没有什么区别。当你列出一个包含硬链接的文件时，不会有特殊的链接指示说明。当一个硬链接被删除时，文件本身的内容仍然存在（也就是说，它所占的磁盘空间不会被重新分配），直到所有关联这个文件的硬链接都删掉。

运行上面的命令后，多出一个 def.txt 文件，而且两个文件的 inode 号相同。inode 信息中的“链接数”会增加 1。

```
[lcy@VM-16-10-centos test]$ ls -l
total 4
-rw-rw-r-- 1 lcy lcy  0 Mar 19 10:47 abc.txt
-rw-rw-r-- 1 lcy lcy 23 Mar 10 14:00 lst.py
-rw-rw-r-- 1 lcy lcy  0 Mar  9 20:53 test1.sh
[lcy@VM-16-10-centos test]$ ls -i
1444951 abc.txt  1445116 lst.py  1445070 test1.sh
[lcy@VM-16-10-centos test]$ ls -li
total 4
1444951 -rw-rw-r-- 1 lcy lcy  0 Mar 19 10:47 abc.txt
1445116 -rw-rw-r-- 1 lcy lcy 23 Mar 10 14:00 lst.py
1445070 -rw-rw-r-- 1 lcy lcy  0 Mar  9 20:53 test1.sh
[lcy@VM-16-10-centos test]$ ln abc.txt def.txt
[lcy@VM-16-10-centos test]$ ls -li
total 4
1444951 -rw-rw-r-- 2 lcy lcy  0 Mar 19 10:47 abc.txt
1444951 -rw-rw-r-- 2 lcy lcy  0 Mar 19 10:47 def.txt
1445116 -rw-rw-r-- 1 lcy lcy 23 Mar 10 14:00 lst.py
1445070 -rw-rw-r-- 1 lcy lcy  0 Mar  9 20:53 test1.sh
[lcy@VM-16-10-centos test]$
```

反过来，删除一个文件名，就会使得 inode 节点中的"链接数"减 1。当这个值减到 0，表明没有文件名指向这个 inode，系统就会回收这个 inode 号码，以及其所对应 block 区域。

目录文件的硬链接

创建目录时，默认会生成两个目录项： . 和 .. 。 . 相当于当前目录的硬链接； .. 相当于父目录的硬链接。所以，目录的硬链接总数，等于 2 加上它的子目录总数（含隐藏目录）。其实使用 ln -d 命令也允许 root 用户尝试建立目录硬链接。

这些都说明系统限制对目录进行硬链接只是一个硬性规定，并不是逻辑上不允许或技术上不可行。为什么操作系统要进行这个限制呢？

由于 Linux 操作系统中的目录是以 / 为节点的树状结构，对目录的硬链接有可能破坏这种结构，甚至形成循环如： /usr/bin -> /usr/ ，在使用遍历目录的命令时（如： ls -R ）系统就会陷入无限循环中。软链接的 inode 号码不一样，所以不会出现这种问题。

1.2.2 软链接

除了硬链接以外，还有一种软链接，创建软链接是为了克服硬链接的局限性。

软链接是通过创建一个特殊类型的文件(指针)链接到文件或目录。就像是 Windows 的快捷方式，当然，符号链接早于 Windows 的快捷方式很多年。

文件 A 和文件 B 的 inode 号码虽然不一样，但是文件 A 的内容是文件 B 的路径。读取文件 A 时，系统会自动将访问指向文件 B。因此，无论打开哪一个文件，最终读取的都是文件 B。但是，文件 A 依赖于文件 B 而存在，如果删除了文件 B，打开文件 A 就会报错："No such file or directory"。这是软链接与硬链接最大的不同：文件 A 指向文件 B 的文件名，而不是文件 B 的 inode 号码。

软链接的应用

想象这样一个情景，一个程序需要使用 foo_1.1 文件中的共享资源，由于 foo 经常改变版本号。每次升级后都得将使用 foo_1.1 的所有程序更新到 foo_1.2 文件，那么每次更新 foo 版本后，都要重复上边的工作。

符号链接能很好的解决这个问题。比如，创建一个 foo 的软链接指向 foo_1.2。这时，当一个程序访问 foo 时，实际上是访问 foo_1.2。当升级到 foo_1.3 时，只需要更新软链接指向。这不仅解决了版本升级问题，而且还允许在系统中保存两个不同的版本，如果 foo_1.3 有错误，再更新回原来的 foo_1.2 链接就可以。

展示如下：

```
[lcy@VM-16-10-centos test1]$ ls -l
total 0
-rw-rw-r-- 1 lcy lcy 0 Mar 19 10:50 abc.txt
[lcy@VM-16-10-centos test1]$ ls -i
926025 abc.txt
[lcy@VM-16-10-centos test1]$ ls -li
total 0
926025 -rw-rw-r-- 1 lcy lcy 0 Mar 19 10:50 abc.txt
[lcy@VM-16-10-centos test1]$ ln -s abc.txt aaa
[lcy@VM-16-10-centos test1]$ ls -l
total 0
lrwxrwxrwx 1 lcy lcy 7 Mar 19 10:50 aaa -> abc.txt
-rw-rw-r-- 1 lcy lcy 0 Mar 19 10:50 abc.txt
[lcy@VM-16-10-centos test1]$ ls -li
total 0
924994 lrwxrwxrwx 1 lcy lcy 7 Mar 19 10:50 aaa -> abc.txt
926025 -rw-rw-r-- 1 lcy lcy 0 Mar 19 10:50 abc.txt
[lcy@VM-16-10-centos test1]$
```

2 进程通信

2.1 进程通讯

2.1.1 进程的概念

进程是操作系统的概念，每当我们执行一个程序时，对于操作系统来讲就创建了一个进程,在这个过程中，伴随着资源的分配和释放。可以认为进程是一个程序的一次执行过程。

2.1.2 进程通信的概念

进程用户空间是相互独立的，一般而言是不能相互访问的。但很多情况下进程间需要互相通信，来完成系统的某项功能。进程通过与内核及其它进程之间的互相通信来协调它们的行为。

2.1.3 进程通信的应用场景

- 数据传输：一个进程需要将它的数据发送给另一个进程，发送的数据量在一个字节到几兆字节之间。
- 共享数据：多个进程想要操作共享数据，一个进程对共享数据的修改，别的进程应该立刻看到。
- 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）。
- 资源共享：多个进程之间共享同样的资源。为了作到这一点，需要内核提供锁和同步机制。
- 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。

2.1.4 进程通讯的方式

1. 管道 pipe：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。

2. 命名管道 FIFO：有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。

3. 流管道 s_pipe：去除了第一种限制,可以双向传输。

4. 消息队列 MessageQueue：消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

5.共享内存 SharedMemory：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。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。

6. 信号量 Semaphore：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

7. 套接字 Socket：套解口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同及其间的进程通信。

8. 信号（sinal）： 信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。

2.2 管道 pipe

2.2.1 概叙

最基本的进程间通信方式，从本质上说，管道是一种特殊的文件，存在于内存中，但是它没有名字，只能用于有亲缘关系的进程间（fork）。

2.2.2 实现机制

管道是由内核管理的一个缓冲区，它的一端连接一个进程的输出，另一端连接一个进程的输入。管

道的缓冲区不需要很大，它被设计为环形的数据结构，当两个进程都终止后，管道的生命周期也会被结束。

2.2.3 特点

- 半双工，数据只能在一个方向上流动，若要双向通信，需要建立两个管道
- 只能在具有公共祖先的进程之间使用，通常是 fork 的父子进程之间使用管道通信
- 管道中的数据遵循先入先出的原则
- 管道所传送的数据是无格式的，这要求管道的读出方与写入方必须事先约定好数据的格式
- 管道不是普通的文件，不属于某个文件系统，其只存在于内存中，对应一块缓冲区，不同系统大小不一定相同
- 管道数据被读走后抛弃，为了释放空间以便写更多数据

2.2.4 读写规则

1) 阻塞性：
   - 默认用 read 函数读数据是阻塞（无数据时）的，write 函数也是阻塞的（当管道满了时）
   - 可通过 functl 来改变阻塞特性，如

   funtl(fd,F_SETFL,0);设为阻塞

   funtl(fd,F_SETFL,0_NOBLOCK);设为非阻塞
2) 退出：
   - 当写一个读端口已被关闭的管道时，写进程会收到 SIGPIPE 信号（默认退出，若从信号处理函数返回后则 write 函数出错返回，errno=EPIPE）
   - 当读一个写端口已被关闭的管道时，在所有数据都被读取后， read 返回 0，以指示达到了文件结束处

2.2.5 函数

int pipe(int pipefd[2]); 功能：创建无名管道 参数：pipefd 返回两个管道的文件描述符，fd[0]用于读，fd[1]用于写 返回值;成功返回 0，失败返回-1，原因保存在 errno 中

常用方式如下：

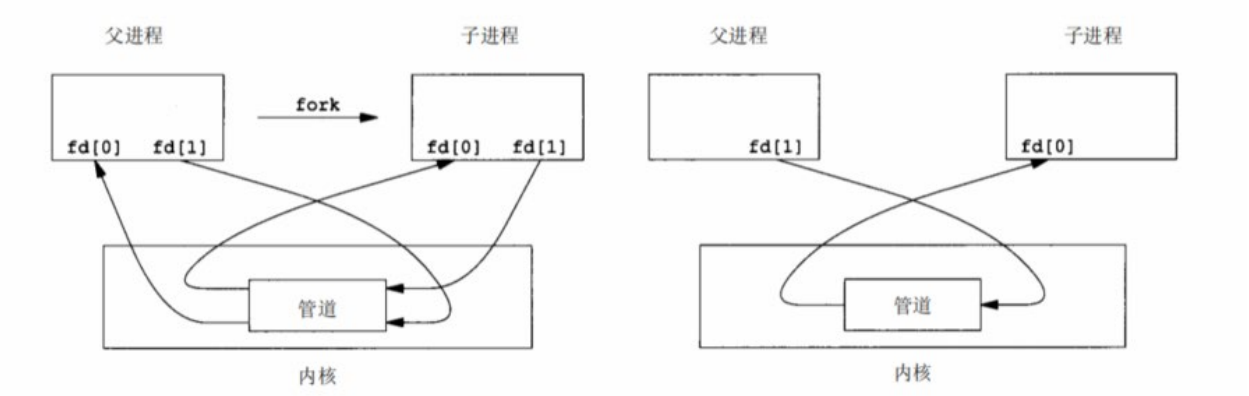


图14-2 fork之后的半双工管道　　图14-3 从父进程到子进程的管道

2.3 命名管道

2.3.1 概述

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。所以，FIFO 的两个特性：

和管道一样，FIFO 仅提供半双工的数据通信，即只支持单向的数据流；

和管道不同的是，FIFO 可以支持任意两个进程间的通信。

2.3.2 读写规则

A. 阻塞性：

一般情况中（open 时未指定 O_NOBLOCK），当你用只读/只写方式打开 FIFO 的时候会阻塞，直到其他进程用只写/只读方式打开该 FIFO

如果指定了 O_NOBLOCK，则只读打开立即返回。但是，如果没有进程已经为读而打开一个 FIFO，那么只写打开将出错返回，其 errno 是 ENXIO。

B. 退出：

类似于管道，若写一个尚无进程为读而打开的 FIFO，则产生信号 SIGPIPE。

若某个 FIFO 的最后一个写进程关闭了该 FIFO，则将为该 FIFO 的读进程产生一个文件结束标志

2.3.3 函数

int mkfifo(const char *pathname, mode_t mode); 功能：该函数创建一个命名管道的特殊文件 pathname，mode 指定了 FIFO 的读写权限（例如 0777） 返回值：成功返回 0，失败返回-1，失败原因保存在 errno

2.4 流管道

2.4.1 概述

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，其实都是把一些列合并到 popen 函数中完成。

2.4.2 函数

1. popen

FILE* popen(char* command ,char* type); 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：表示的是读写方式，只能是其中一种方式，不能读写同时进行。 ”r”：文件指针连接到 command 的标准输出 “w” :文件指针连接到 command 的标准输入

2. pclose

int pclose(FILE* stream); stream：要关闭的文件流。 popen 函数其实是对管道操作的一些包装，所完成的工作有以下几步： 创建一个管道。 fork 一个子进程。 在父子进程中关闭不需要的文件描述符。 执行 exec 函数族调用。 执行函数中所指定的命令。

2.5 消息队列

2.5.1 概叙

消息队列是消息的链接表 ,存放在内核中并由消息队列标识符标识。我们将称消息队列为“队列”,其标识符为“队列 ID”

2.5.2 特点

- 消息不一定要以先进先出的次序读取，编程时可以按消息的类型读取
- 消息被读取后删除（和管道相同）
- 消息队列有唯一标识符
- 只有内核重启或手动删除才可以删除消息队列

2.5.3 预设值

名称	说明	典型值
MSGMAX	可发送的最长消息的字节长度	2048
MSGMNB	特定队列的最大字节长度 (亦即队列中所有消息之和)	4096
MSGMNI	系统中最大消息队列数	50
MSGTOL	系统中最大消息数	50

2.5.4 函数

1. ftok

key_t ftok(const char *pathname, int proj_id);
功能：获取唯一键值
Pathname：任意路径名（必须已存在）
proj_id：1-255
key_t：成功返回消息队列的标识符，失败返回-1

2. msgget

int msgget(key_t key, int msgflg);
功能：创建一个新的或者打开一个已经存在的消息队列，只要有相同的 key 值就可以获得相同的消息队列
Key：ftok 函数返回的键值
Msgflg：IPC_CREAT 创建消息队列，且后面需要跟着权限，IPC_EXCL 检查消息队列是否存在
返回值（int）：成功返回消息队列的标识符，失败返回-1

3. msgsnd

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
功能:将新消息添加到消息队列
Msqid：消息队列的标识符
Msgp：待发送消息结构体的地址
Msgsz：消息正文的字节数
Msgflg：函数的控制属性,如下所示:
0：当消息队列满时，msgsnd 将会阻塞，直到消息能写进消息队列或者消息队列被删除，IPC_NOWAIT:当消息队列满了，msgsnd 函数将不会等待，会立即出错返回 EAGAIN
返回值（int）：成功返回 0；错误返回-1

4. msgrcv

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
功能：从标识符为 msqid 的消息队列中接收一个消息。
Msqid：消息队列的标识符
msgp：存放消息结构体的地址
msgsz：消息正文的字节数
msgflg：函数的控制属性:
0：msgrcv 调用阻塞直到接收消息成功为止。
MSG_NOERROR:若返回的消息字节数比 nbytes 字节数多,则消息就会截短到 nbytes 字节,且不通知消息发送进程，若未指定则此时消息不会从队列中移除，且函数调用返回-1。
IPC_NOWAIT:调用进程会立即返回。若没有收到消息则立即返回-1。
IPC_EXCEPT 时，与 msgtype 配合使用返回队列中第一个类型不为 msgtype 的消息
返回值（ssize_t)：成功返回消息数据部分的长度；错误返回-1

5. msgctl

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
功能:消息队列控制函数
Msqid:消息队列的标识符
Cmd:取值如下:
IPC_RMID：删除由 msqid 指示的消息队列，将它从系统中删除并破坏相关数据结构。
IPC_STAT：将 msqid 相关的数据结构中各个元素的当前值存入到由 buf 指向的结构中。
IPC_SET：将 msqid 相关的数据结构中的元素设置为由 buf 指向的结构中的对应值
Buf:msqid_ds 数据类型的地址，用来存放或更改消息队列的属性
返回值(int):成功返回 0；错误返回-1

6. shell 操作消息队列

ipcs -q 查看消息队列
ipcrm -q msgid 删除消息队列

2.6 信号

2.6.1 概述

信号(signal)是一种软中断，信号机制是进程间通信的一种方式，采用异步通信方式。Linux 系统共定义了 64 种信号，分为两大类：可靠信号与不可靠信号，前 32 种信号为不可靠信号，后 32 种为可靠信号。

- 不可靠信号：也称为非实时信号,不支持排队,信号可能会丢失，比如发送多次相同的信号，进程只能收到一次. 信号值取值区间为 1~31；
- 可靠信号： 也称为实时信号，支持排队，信号不会丢失，发多少次，就可以收到多少次. 信号值取值区间为 32~64

2.6.2 信号表

在终端，可通过 kill -l 查看所有的 signal 信号

```
C:\Program Files\cmder\cmder
# ssh lcy@124.221.87.113
lcy@124.221.87.113's password:
Last login: Sat Mar 19 10:46:32 2022 from 122.192.219.205
[lcy@VM-16-10-centos ~]$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
[lcy@VM-16-10-centos ~]$
```

取值	名称	解释	默认动作
1	SIGHUP	挂起	
2	SIGINT	中断	
3	SIGQUIT	退出	
4	SIGILL	非法指令	
5	SIGTRAP	断点或陷阱指令	
6	SIGABRT	abort 发出的信号	
7	SIGBUS	非法内存访问	
8	SIGFPE	浮点异常	
9	SIGKILL	kill 信号	不能被忽略、处理和阻塞
10	SIGUSR1	用户信号 1	
11	SIGSEGV	无效内存访问	
12	SIGUSR2	用户信号 2	
13	SIGPIPE	管道破损，没有读端的管道写数据	
14	SIGALRM	alarm 发出的信号	
15	SIGTERM	终止信号	
16	SIGSTKFLT	栈溢出	
17	SIGCHLD	子进程退出	默认忽略
18	SIGCONT	进程继续	
19	SIGSTOP	进程停止	不能被忽略、处理和阻塞
20	SIGTSTP	进程停止	
21	SIGTTIN	进程停止，后台进程从终端读数据时	
22	SIGTTOU	进程停止，后台进程想终端写数据时	
23	SIGURG	I/O 有紧急数据到达当前进程	默认忽略
24	SIGXCPU	进程的 CPU 时间片到期	
25	SIGXFSZ	文件大小的超出上限	
26	SIGVTALRM	虚拟时钟超时	
27	SIGPROF	profile 时钟超时	
28	SIGWINCH	窗口大小改变	默认忽略
29	SIGIO	I/O 相关	
30	SIGPWR	关机	默认忽略
31	SIGSYS	系统调用异常	

对于 signal 信号，绝大部分的默认处理都是终止进程或停止进程，或 dump 内核映像转储。 上述的 31 的信号为非实时信号，其他的信号 32-64 都是实时信号。

2.6.3 信号产生

1 硬件方式

用户输入：比如在终端上按下组合键 ctrl+C，产生 SIGINT 信号；

硬件异常：CPU 检测到内存非法访问等异常，通知内核生成相应信号，并发送给发生事件的进程；

2 软件方式

通过系统调用，发送 signal 信号：kill()，raise()，sigqueue()，alarm()，setitimer()，abort()

kernel,使用 kill_proc_info() 等

native,使用 kill() 或者 raise()等

java,使用 Procees.sendSignal()等

2.6.4 信号注册和注销

1 注册

在进程 task_struct 结构体中有一个未决信号的成员变量 struct sigpending pending。每个信号在进程中注册都会把信号值加入到进程的未决信号集。

非实时信号发送给进程时，如果该信息已经在进程中注册过，不会再次注册，故信号会丢失；

实时信号发送给进程时，不管该信号是否在进程中注册过，都会再次注册。故信号不会丢失；

2 注销

非实时信号：不可重复注册，最多只有一个 sigqueue 结构；当该结构被释放后，把该信号从进程未决信号集中删除，则信号注销完毕；

实时信号：可重复注册，可能存在多个 sigqueue 结构；当该信号的所有 sigqueue 处理完毕后，把该信号从进程未决信号集中删除，则信号注销完毕；

2.6.5 信号处理

内核处理进程收到的 signal 是在当前进程的上下文，故进程必须是 Running 状态。当进程唤醒或者调度后获取 CPU，则会从内核态转到用户态时检测是否有 signal 等待处理，处理完，进程会把相应的未决信号从链表中去掉。

1 处理时机

signal 信号处理时机： 内核态 -> signal 信号处理 -> 用户态：

在内核态，signal 信号不起作用；

在用户态，signal 所有未被屏蔽的信号都处理完毕；

当屏蔽信号，取消屏蔽时，会在下一次内核转用户态的过程中执行；

2 处理方式

进程对信号的处理方式：有 3 种

默认 接收到信号后按默认的行为处理该信号。 这是多数应用采取的处理方式。

自定义 用自定义的信号处理函数来执行特定的动作

忽略 接收到信号后不做任何反应。

3 信号安装

进程处理某个信号前，需要先在进程中安装此信号。安装过程主要是建立信号值和进程对相应信息值的动作。

信号安装函数

signal()：不支持信号传递信息，主要用于非实时信号安装；

sigaction():支持信号传递信息，可用于所有信号安装；

其中 sigaction 结构体

sa_handler:信号处理函数

sa_mask：指定信号处理程序执行过程中需要阻塞的信号；

sa_flags：标示位

SA_RESTART：使被信号打断的 syscall 重新发起。

SA_NOCLDSTOP：使父进程在它的子进程暂停或继续运行时不会收到 SIGCHLD 信号。

SA_NOCLDWAIT：使父进程在它的子进程退出时不会收到 SIGCHLD 信号，这时子进程如果退出也不会成为僵 尸进程。

SA_NODEFER：使对信号的屏蔽无效，即在信号处理函数执行期间仍能发出这个信号。

SA_RESETHAND：信号处理之后重新设置为默认的处理方式。

SA_SIGINFO：使用 sa_sigaction 成员而不是 sa_handler 作为信号处理函数。

函数原型：

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
signum：要操作的 signal 信号。
act：设置对 signal 信号的新处理方式。
oldact：原来对信号的处理方式。
返回值：0 表示成功，-1 表示有错误发生。

4 信号发送

kill()：用于向进程或进程组发送信号；

sigqueue()：只能向一个进程发送信号，不能像进程组发送信号；主要针对实时信号提出，与 sigaction() 组合使用，当然也支持非实时信号的发送；

alarm()：用于调用进程指定时间后发出 SIGALARM 信号；

setitimer()：设置定时器，计时达到后给进程发送 SIGALRM 信号，功能比 alarm 更强大；

abort()：向进程发送 SIGABORT 信号，默认进程会异常退出。

raise()：用于向进程自身发送信号；

5 信号相关函数

信号集操作函数

sigemptyset(sigset_t *set)：信号集全部清 0；
sigfillset(sigset_t *set)： 信号集全部置 1，则信号集包含 linux 支持的 64 种信号；
sigaddset(sigset_t *set, int signum)：向信号集中加入 signum 信号；
sigdelset(sigset_t *set, int signum)：向信号集中删除 signum 信号；
sigismember(const sigset_t *set, int signum)：判定信号 signum 是否存在信号集中。

信号阻塞函数

sigprocmask(int how, const sigset_t *set, sigset_t *oldset))； 不同 how 参数，实现不同功能
SIG_BLOCK：将 set 指向信号集中的信号，添加到进程阻塞信号集；
SIG_UNBLOCK：将 set 指向信号集中的信号，从进程阻塞信号集删除；
SIG_SETMASK：将 set 指向信号集中的信号，设置成进程阻塞信号集；
sigpending(sigset_t *set))：获取已发送到进程，却被阻塞的所有信号；
sigsuspend(const sigset_t *mask))：用 mask 代替进程的原有掩码，并暂停进程执行，直到收到信号再恢复原有掩码并继续执行进程。

2.7 共享内存

2.7.1 概叙

共享内存就是允许两个或多个进程共享一定的存储区。就如同 malloc() 函数向不同进程返回了指向同一个物理内存区域的指针。当一个进程改变了这块地址中的内容的时候，其它进程都会察觉到这个更改。因为数据不需要在客户机和服务器端之间复制，数据直接写到内存，不用若干次数据拷贝，所以这是最快的一种 IPC。(共享内存没有任何的同步与互斥机制，所以要使用信号量来实现对共享内存的

存取的同步。)

2.7.2 函数

1. shmget

int shmget(key_t key, size_t size, int shmflg);
功能：根据键值创建或者打开共享内存
Key：ftoke 返回的键值
Size：该共享存储段的长度(字节)
Shmflg：标识函数的行为及共享内存的权限类似
返回值（int)：成功返回共享内存的标识符，失败返回-1

2. shmat

void *shmat(int shmid, const void *shmaddr, int shmflg);
功能：共享内存的映射
一般用法：void addr = shmat(shmid,NULL,0);shmid 为 shmget 返回的键值，后两个参数这样写表明系统自动分配
成功返回映射到当前进程的地址，失败返回-1

3. shmdt

int shmdt(const void *shmaddr);
功能：解除映射，只是和当前进程分离，不删除共享内存
shmaddr：shmat 函数映射到当前进程的地址
成功返回 0，失败返回-1

4. shmctl

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
功能：共享内存控制函数
对共享内存进行各种控制，如修改共享内存的属性，或删除共享内存。其中参数含义和消息队列类似常用的也就是删除共享内存,如：
shmctl(shmid,IPC_RMID,NULL);
成功返回 0；错误返回-1

5. shell 中操作共享内存

查看：ipcs -m
删除：ipcrm -m shmid

2.8 信号量

2.8.1 概叙

信号量与已经介绍过的 IPC 机构（管道、FIFO 以及消息列队）不同。它是一个计数器，用于多进程对共享数据对象的存取。为了获得共享资源，进程需要执行下列操作：

1) 测试控制该资源的信号量。
2) 若此信号量的值为正，则进程可以使用该资源。进程将信号量值减 1，表示它使用了一个资源单位。
3) 若此信号量的值为 0，则进程进入睡眠状态，直至信号量值大于 0。若进程被唤醒后，它返回

至(第(1)步)。

4) 当进程不再使用由一个信息量控制的共享资源时，该信号量值增 1。如果有进程正在睡眠等待此信号量，则唤醒它们。

常用的信号量形式被称之为双态信号量(binary semaphore)。它控制单个资源，其初始值为 1。但是，一般而言，信号量的初值可以是任一正值，该值说明有多少个共享资源单位可供共享应用。 不幸的是，系统 V 的信号量与此相比要复杂得多。三种特性造成了这种并非必要的复杂性：

- 信号量并非是一个非负值，而必需将信号量定义为含有一个或多个信号量值的集合。 当创建一个信号量时，要指定该集合中的各个值
- 创建信息量（semget）与对其赋初值（semctl）分开。这是一个致命的弱点，因为不能原子地创建一个信号量集合，并且对该集合中的所有值赋初值。
- 即使没有进程正在使用各种形式的系统 V IPC，它们仍然是存在的，所以不得不为这种程序担心，它在终止时并没有释放已经分配给它的信号量。

2.8.2 预设值

名称	说明	典型值
SEMVM	任一信号量的最大值	32767
SEMAE	任一信号量的最大终止时调整值	16384
SEMMN	系统中信号量集的最大数	10
SEMMNS	系统中信号量集的最大数	60
SEMMSL	每个信号量集中的最大信号量数	25

2.8.3 函数

1.semget

int semget(key_t key, int nsems, int semflg);
功能：创建一个新的或打开一个已经存在的信号量
Key：ftoke 返回的键值无
Nsems：该集合中的信号量数。如果是创建新集合，则必须指定（一般为 1）。如果引用一个现存的集合，则将其指定为 0
Semflg：IPC_CREAT 创建消息队列，且后面需要跟着权限
IPC_EXCL 检查消息队列是否存在
返回值（int）：成功返回信号量标识，失败返回-1

2. semop

int semop(int semid, struct sembuf *sops, size_t nsops);
功能：修改信号量的值
Semid：信号量标识，由 semget 返回
Sops：信号量操作数组
Nsops：信号量操作数组的数量
返回值（int）：成功返回 0，失败返回-1

3. semctl

int semctl(int semid, int semnum, int cmd, ...)
功能：信号量控制函数
Semid：信号量标识，由 semget 返回
Semnum：一般写 0
Cmd：SETVAL:初始化信号量，IPC_RMID：删除信号量

返回值（int）：成功返回信号量标识，失败返回-1

4. shell 操作信号量

ipcs　-s 查看消息队列 ipcrm　-s　msgid　删除消息队列

2.9 套接字

2.9.1 概述

套接字是通信端点的抽象，其英文 socket，即为插座，孔的意思。如果两个机子要通信，中间要通过一条线，这条线的两端要连接通信的双方，这条线在每一台机子上的接入点则为 socket，即为插孔，所以在通信前，我们在通信的两端必须要建立好这个插孔，同时为了保证通信的正确，端和端之间的插孔必须要一一对应，这样两端便可以正确的进行通信了，而这个插孔对应到我们实际的操作系统中，就是 socket 文件，我们再创建它之后，就会得到一个操作系统返回的对于该文件的描述符，然后应用程序可以通过使用套接字描述符访问套接字，向其写入输入，读出数据。

站在更贴近系统的层级去看，两个机器间的通信方式，无非是要通过运输层的 TCP/UDP，网络层 IP，因此 socket 本质是编程接口(API)，对 TCP/UDP/IP 的封装，TCP/UDP/IP 也要提供可供程序员做网络开发所用的接口，这就是 Socket 编程接口。

2.9.2 套接字的属性

套接字的特性由 3 个属性确定，它们分别是：域、类型和协议。

1、套接字的域

它指定套接字通信中使用的网络介质，最常见的套接字域是 AF_INET，它指的是 Internet 网络。当客户使用套接字进行跨网络的连接时，它就需要用到服务器计算机的 IP 地址和端口来指定一台联网机器上的某个特定服务，所以在使用 socket 作为通信的终点，服务器应用程序必须在开始通信之前绑定一个端口，服务器在指定的端口等待客户的连接。另一个域 AF_UNIX 表示 UNIX 文件系统，它就是文件输入/输出，而它的地址就是文件名。

2、套接字类型

因特网提供了两种通信机制：流（stream）和数据报（datagram），因而套接字的类型也就分为流套接字和数据报套接字。这里主要讲流套接字。

流套接字由类型 SOCK_STREAM 指定，它们是在 AF_INET 域中通过 TCP/IP 连接实现，同时也是 AF_UNIX 中常用的套接字类型。流套接字提供的是一个有序、可靠、双向字节流的连接，因此发送的数据可以确保不会丢失、重复或乱序到达，而且它还有一定的出错后重新发送的机制。

与流套接字相对的是由类型 SOCK_DGRAM 指定的数据报套接字，它不需要建立连接和维持一个连接，它们在 AF_INET 中通常是通过 UDP/IP 协议实现的。它对可以发送的数据的长度有限制，数据报作为一个单独的网络消息被传输,它可能会丢失、复制或错乱到达，UDP 不是一个可靠的协议，但是它的速度比较高，因为它并一需要总是要建立和维持一个连接。

3、套接字协议

只要底层的传输机制允许不止一个协议来提供要求的套接字类型，我们就可以为套接字选择一个特定的协议。通常只需要使用默认值。

2.9.3 基于流套接字的客户/服务器的工作流程

1、服务器端

首先服务器应用程序用系统调用 socket 来创建一个套接安，它是系统分配给该服务器进程的类似文件描述符的资源，它不能与其他的进程共享。

接下来，服务器进程会给套接字起个名字，我们使用系统调用 bind 来给套接字命名。然后服务器进程就开始等待客户连接到这个套接字。

然后，系统调用 listen 来创建一个队列并将其用于存放来自客户的进入连接。

最后，服务器通过系统调用 accept 来接受客户的连接。它会创建一个与原有的命名套接不同的新套接字，这个套接字只用于与这个特定客户端进行通信，而命名套接字（即原先的套接字）则被保留下来继续处理来自其他客户的连接。

2、客户端

基于 socket 的客户端比服务器端简单，同样，客户应用程序首先调用 socket 来创建一个未命名的套接字，然后将服务器的命名套接字作为一个地址来调用 connect 与服务器建立连接。

一旦连接建立，我们就可以像使用底层的文件描述符那样用套接字来实现双向数据的通信。

2.9.4 流式 socket 的接口及作用

socket 的接口函数声明在头文件 sys/types.h 和 sys/socket.h 中。

1、创建套接字——socket 系统调用

该函数用来创建一个套接字，并返回一个描述符，该描述符可以用来访问该套接字，它的原型如下：

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

函数中的三个参数分别对应前面所说的三个套接字属性。protocol 参数设置为 0 表示使用默认协议。

int socket(int domain, int type, int protocol); domain：即协议域，又称为协议族（family）。常用的协议族有，AF_INET、AF_INET6、AF_LOCAL（或称 AF_UNIX，Unix 域 socket）、AF_ROUTE 等等。协议族决定了 socket 的地址类型，在通信中必须采用对应的地址，如 AF_INET 决定了要用 ipv4 地址（32 位的）与端口号（16 位的）的组合、AF_UNIX 决定了要用一个绝对路径名作为地址。 type：指定 socket 类型。常用的 socket 类型有，SOCK_STREAM、SOCK_DGRAM、SOCK_RAW、SOCK_PACKET、SOCK_SEQPACKET 等等（socket 的类型有哪些？）。 protocol：故名思意，就是指定协议。常用的协议有，IPPROTO_TCP、IPPTOTO_UDP、IPPROTO_SCTP、IPPROTO_TIPC 等，它们分别对应 TCP 传输协议、UDP 传输协议、STCP 传输协议、TIPC 传输协议（这个协议我将会单独开篇讨论！）。 注意：并不是上面的 type 和 protocol 可以随意组合的，如 SOCK_STREAM 不可以跟 IPPROTO_UDP 组合。当 protocol 为 0 时，会自动选择 type 类型对应的默认协议。

2、命名（绑定）套接字——bind 系统调用

该函数把通过 socket 调用创建的套接字命名，从而让它可以被其他进程使用。对于 AF_UNIX，调用该函数后套接字就会关联到一个文件系统路径名，对于 AF_INET，则会关联到一个 IP 端口号。函数原型如下：

int bind(int socket, const struct sockaddr *address, size_t address_len); 成功时返回 0，失败时返回-1；

3、创建套接字队列（监听）——listen 系统调用

该函数用来创建一个队列来保存未处理的请求。成功时返回 0，失败时返回-1，其原型如下：

int listen(int socket, int backlog); backlog 用于指定队列的长度，等待处理的进入连接的个数最多不能超过这个数字，否则往后的连接将被拒绝，导致客户的连接请求失败。调用后，程序一直会监听这个 IP 端口，如果有连接请求，就把它加入到这个队列中。

4、接受连接——accept 系统调用

该系统调用用来等待客户建立对该套接字的连接。accept 系统调用只有当客户程序试图连接到由 socket 参数指定的套接字上时才返回，也就是说，如果套接字队列中没有未处理的连接，accept 将阻塞直到有客户建立连接为止。accept 函数将创建一个新套接字来与该客户进行通信，并且返回新套接字的描述符，新套接字的类型和服务器监听套接字类型是一样的。它的原型如下：

```
int accept(int socket, struct sockaddr *address, size_t *address_len);
```

address 为连接客户端的地址，参数 address_len 指定客户结构的长度，如果客户地址的长度超过这个值，它将会截断。

5、请求连接——connect 系统调用

该系统调用用来让客户程序通过在一个未命名套接字和服务器监听套接字之间建立连接的方法来连接到服务器。它的原型如下：

```
int connect(int socket, const struct sockaddr *address, size_t address_len);
```

参数 socket 指定的套接字连接到参数 addres 指定的服务器套接字。成功时返回 0，失败时返回-1.

6、关闭 socket——close 系统调用

该系统调用用来终止服务器和客户上的套接字连接，我们应该总是在连接的两端（服务器和客户）关闭套接字。

```
#include <unistd.h>
int close(int fd);
```

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

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