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 19-01 班 学生姓名 许万鹏 学 号 05191643

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 5 月 6 日

1 文件系统

1.1 详述 Linux 的节点 inode

1.1.1 inode 简介

inode，中文译作“索引节点”，用于存放文件和目录除文件名外的其他管理信息，是 UNIX 操作系统中的一种数据结构，本质是结构体。

在文件系统中，每个文件对应一个 inode 实例，且有唯一的编号 i_no（下述）。在 UNIX 中创建文件系统时会创建大量 inode 实例，它们被集中存放于磁盘上的 inode 区（inode table），但文件系统磁盘中只有大约 1%的空间会分配给 inode 表，可见其高效性。

当内核访问存储设备上的一个文件时，会在内存中创建 inode 的一个副本——活动 inode。

inode 存在于两个双向链表中:

1. inode 所在文件系统的 super_block 的 s_inodes 链表；
2. 根据 inode 的使用状态存在于以下三个循环双向链表中的某个链表：

   未用的：inode_unused 链表

   正在使用的：inode_in_use 链表

   脏的：super block 中的 s_dirty 链表

1.1.2 inode 结构体

在最新的 Linux 内核 linux-5.17.3 中，其定义如下（已省略非活动预处理器块）：

include/linux/fs.h

```
/*
 * Keep mostly read-only and often accessed (especially for
 * the RCU path lookup and 'stat' data) fields at the beginning
 * of the 'struct inode'
 */
struct inode {
    umode_t             i_mode;
    unsigned short      i_opflags;
    kuid_t              i_uid;
    kgid_t              i_gid;
    unsigned int        i_flags;
    ...
    const struct inode_operations   *i_op;
    struct super_block  *i_sb;
    struct address_space    *i_mapping;
    ...
```

```
/* Stat data, not accessed from path walking */
unsigned long       i_ino;
/*
 * Filesystems may only read i_nlink directly.  They shall use the
 * following functions for modification:
 *
 *    (set|clear|inc|drop)_nlink
 *    inode_(inc|dec)_link_count
 */
union {
    const unsigned int i_nlink;
    unsigned int    __i_nlink;
};
dev_t               i_rdev;
loff_t              i_size;
struct timespec64   i_atime;
struct timespec64   i_mtime;
struct timespec64   i_ctime;
spinlock_t          i_lock; /* i_blocks, i_bytes, maybe i_size */
unsigned short      i_bytes;
u8                  i_blkbits;
u8                  i_write_hint;
blkcnt_t            i_blocks;
...
/* Misc */
unsigned long       i_state;
struct rw_semaphore i_rwsem;

unsigned long       dirtied_when;   /* jiffies of first dirtying */
unsigned long       dirtied_time_when;

struct hlist_node   i_hash;
struct list_head    i_io_list;  /* backing dev IO list */
...
struct list_head    i_lru;      /* inode LRU list */
struct list_head    i_sb_list;
struct list_head    i_wb_list;  /* backing dev writeback list */
union {
    struct hlist_head   i_dentry;
    struct rcu_head     i_rcu;
};
atomic64_t          i_version;
atomic64_t          i_sequence; /* see futex */
atomic_t            i_count;
```

```
    atomic_t            i_dio_count;
    atomic_t            i_writecount;
    ...
    union {
        const struct file_operations    *i_fop; /* former ->i_op->default_file_ops */
        void (*free_inode)(struct inode *);
    };
    struct file_lock_context    *i_flctx;
    struct address_space    i_data;
    struct list_head    i_devices;
    union {
        struct pipe_inode_info  *i_pipe;
        struct cdev     *i_cdev;
        char            *i_link;
        unsigned        i_dir_seq;
    };

    __u32               i_generation;
    ...
    void                *i_private; /* fs or device private pointer */
} __randomize_layout;
```

include/linux/fs.h

结构体 inode 包含以下四类内容：

1. 描述文件信息的域：包含文件类型和访问权限、文件主 id 和其所属组的 id、文件大小和文件占用块数、块大小、inode 使用的块数和可用块的剩余数、文件最后访问时间、文件最后修改时间、块设备链表指针和字符设备链表指针、物理设备号、与该 inode 建立的连接数等；

2. 用于 inode 管理的域：包含哈希链表、inode 所在文件系统的超级块链表头节点、分类索引链表、指向该 inode 的目录项链表头；

3. 用于 inode 操作的域：包含 inode 标识符、指向 inode_operations 结构的指针、指向 file_operations 结构的指针、指向具体文件系统的超级块的指针、使用该 inode 的进程数等；

4. union 联合体：存放多种具体文件系统的 inode 信息。因此根据具体文件系统不同，可将 union 解释为不同的数据结构。

描述文件信息的域中的变量：

变量	意义
i_mode	文件类型和访问权限
i_uid	创建文件的用户的标识符
i_gid	创建文件的用户所属的组标识符
i_ino	索引节点的编号

i_size	文件长度
i_blocks	文件的块数，即文件长度除以块长度的商
i_bytes	文件长度除以块长度的余数
i_blkbits	是块长度以 2 为底的对数（块长度是 2 的 i_bllcbits 次幂）
i_atime (access time)	上一次访问文件的时间
i_mtime (modified time)	上一次修改文件数据的时间
i_ctime (change time)	上一次修改文件索引节点的时间
i_sb	指向文件所属的文件系统的超级块
i_mapping	指向文件的地址空间
i_count	索引节点的引用计数
i_nlink	硬链接计数
i_rdev	设备号
i_bdev（这里没有设置）	指向块设备
i_cdev	指向字符设备
i_pipe	用于实现管道的 inode 的相关信息

其他域的成员多数指向复合数据结构，例如：

i_op 指向索引节点操作集合 inode_operations，他们用于操作目录和文件属性。

inode_operations 存放与 inode 关联的方法，其定义如下：

include/linux/fs.h

```
struct inode_operations {
    struct dentry * (*lookup) (struct inode *,struct dentry *, unsigned int);
    const char * (*get_link) (struct dentry *, struct inode *, struct delayed_call *);
    int (*permission) (struct user_namespace *, struct inode *, int);
    struct posix_acl * (*get_acl)(struct inode *, int, bool);

    int (*readlink) (struct dentry *, char __user *,int);

    int (*create) (struct user_namespace *, struct inode *,struct dentry *,
            umode_t, bool);
    int (*link) (struct dentry *,struct inode *,struct dentry *);
    int (*unlink) (struct inode *,struct dentry *);
    int (*symlink) (struct user_namespace *, struct inode *,struct dentry *,
        const char *);
    int (*mkdir) (struct user_namespace *, struct inode *,struct dentry *, umode_t);
    int (*rmdir) (struct inode *,struct dentry *);
    int (*mknod) (struct user_namespace *, struct inode *,struct dentry *,
            umode_t,dev_t);
    int (*rename) (struct user_namespace *, struct inode *, struct dentry *,
```

```
            struct inode *, struct dentry *, unsigned int);
    int (*setattr) (struct user_namespace *, struct dentry *, struct iattr *);
    int (*getattr) (struct user_namespace *, const struct path *,
            struct kstat *, u32, unsigned int);
    ssize_t (*listxattr) (struct dentry *, char *, size_t);
    int (*fiemap)(struct inode *, struct fiemap_extent_info *, u64 start, u64 len);
    int (*update_time)(struct inode *, struct timespec64 *, int);
    int (*atomic_open)(struct inode *, struct dentry *, struct file *, unsigned open_flag,
                umode_t create_mode);
    int (*tmpfile) (struct user_namespace *, struct inode *, struct dentry *, umode_t);
    int (*set_acl)(struct user_namespace *, struct inode *, struct posix_acl *, int);
    int (*fileattr_set)(struct user_namespace *mnt_userns,
                struct dentry *dentry, struct fileattr *fa);
    int (*fileattr_get)(struct dentry *dentry, struct fileattr *fa);
} ____cacheline_aligned;
```

include/linux/fs.h

这些方法被许多耳熟能详的系统调用所使用，例如：

lookup 方法用来在一个目录下查找文件。

系统调用 open 和 creat 调用 create 方法来创建普通文件，

系统调用 link 调用 link 方法来创建硬链接，

系统调用 symljnk 调用 symlink 方法来创建符号链接，

系统调用 mkdir 调用 mkdir 方法来创建目录，

系统调用 mknod 调用 mknod 方法来创建字符设备文件、块设备文件、命名管道和套接字。

系统调用 unlink 调用 unlink 方法来删除硬链接，

系统调用 rmdir 调用 rmdir 方法来删除目录。

系统调用 rename 调用 rename 方法来给重命名文件。

系统调用 chmod 调用 setattr 方法来设置文件的属性，

系统调用 stat 调用 getattr 方法来读取文件的属性。

系统调用 listxattr 调用 listxattr 方法来列出文件的所有扩展属性。

1.1.3 inode 编号（i_no）

在简介中我们提到过每个文件拥有唯一的 inode 编号——i_no，每个 i_no 指向一个 inode 区的一个 inode 实例。

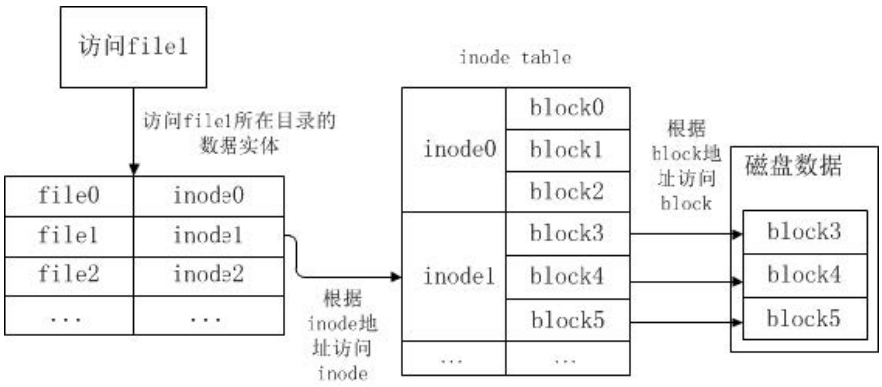
事实上，Linux 中的 FCB（文件控制块）只包含文件名和 inode 编号，二者称作基本目录项。用当户通过文件名打开文件时，Linux 系统会进行 3 个步骤：

1. 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。

可用下图描述这个过程。



我们可以使用 ls 命令的-i 选项查看 inode 编号。

ll -i 或 ls -li

```
root@VM-16-16-centos ~/C_CODE/XICQ4 # ll -i
total 108
656839 -rw-r--r-- 1 root root  7620 May  4 21:48 client.c
656861 -rwxr-xr-x 1 root root 23312 May  4 22:25 client.elf
656659 -rw-r--r-- 1 root root 11176 May  4 22:25 client.o
656863 -rw-r--r-- 1 root root   173 May  4 21:48 makefile
656842 -rw-r--r-- 1 root root  1304 May  4 21:48 need.h
658052 -rw-r--r-- 1 root root    15 May  4 21:42 recv
656857 -rw-r--r-- 1 root root  7553 May  4 21:48 server.c
657951 -rwxr-xr-x 1 root root 27192 May  4 22:25 server.elf
657945 -rw-r--r-- 1 root root 10752 May  4 22:25 server.o
657976 -rw-r--r-- 1 root root    15 May  4 17:57 test.txt
```

1.1.4 inode 实例的大小

inode 实例的大小是固定的，具体大小视具体系统而定，我们可以使用

dumpe2fs -h /dev/vda1 | grep "Inode size"

命令来查看每个 inode 结点的大小。

```
root@VM-16-16-centos /dev # dumpe2fs -h /dev/vda1 | grep "Inode size"
dumpe2fs 1.45.6 (20-Mar-2020)
Inode size:               256
```

在我的 CentOS 8 上，inode 节点大小为 256 bytes

我们也可以查看每个硬盘分区的 inode 总数和已经使用的数量，使用命令

df -i

```
root@VM-16-16-centos /dev # df -i
Filesystem      Inodes IUsed   IFree IUse% Mounted on
devtmpfs        474465   316  474149    1% /dev
tmpfs           478292     7  478285    1% /dev/shm
tmpfs           478292   468  477824    1% /run
tmpfs           478292    17  478275    1% /sys/fs/cgroup
/dev/vda1      3932160 73798 3858362    2% /
tmpfs           478292     5  478287    1% /run/user/1001
```

1.1.5 inode 机制的特有现象

1. 直接删除 inode 节点可以起到删除文件的作用。

使用 i_no 删除文件的两种方式

find ./* -inum 656653 -delete 或

find ./* -inum 656653 | xargs rm -f

```
root@VM-16-16-centos ~/C_CODE/XICQ4 # ll -i cumt.txt
656653 -rw-r--r-- 1 root root 23 May  6 02:03 cumt.txt
root@VM-16-16-centos ~/C_CODE/XICQ4 # find ./* -inum 656653 |xargs rm -f
root@VM-16-16-centos ~/C_CODE/XICQ4 # ll -i cumt.txt
ls: cannot access 'cumt.txt': No such file or directory
```

2. 移动文件或重命名文件不影响 inode 号码。

mv cumt.txt cumt2.txt

```
root@VM-16-16-centos ~/C_CODE/XICQ4 # ll -i cumt*
656653 -rw-r--r-- 1 root root 3 May  6 02:05 cumt.txt
root@VM-16-16-centos ~/C_CODE/XICQ4 # mv cumt.txt cumt2.txt
root@VM-16-16-centos ~/C_CODE/XICQ4 # ll -i cumt*
656653 -rw-r--r-- 1 root root 3 May  6 02:05 cumt2.txt
```

3. Linux 可以在不关闭软件的情况下进行更新，不需要重启。这是因为打开一个文件以后，系统就以 inode 号码来识别这个文件，不再考虑文件名。

1.2 详述硬链接与软链接

1.2.1 简介

链接（link）是 Linux 中用于建立文件系统对象之间联系的一种方法。链接有两种类型——硬链接（hard link，简称 link）、符号链接/软链接（symbolic link/soft link）。

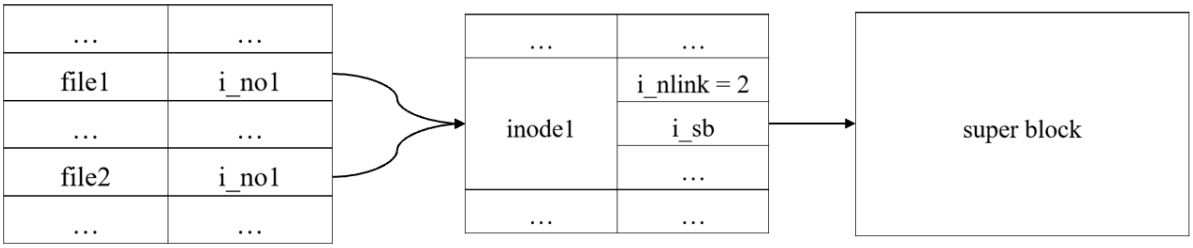
创建链接文件的命令如下：

硬链接：ln file1 file2

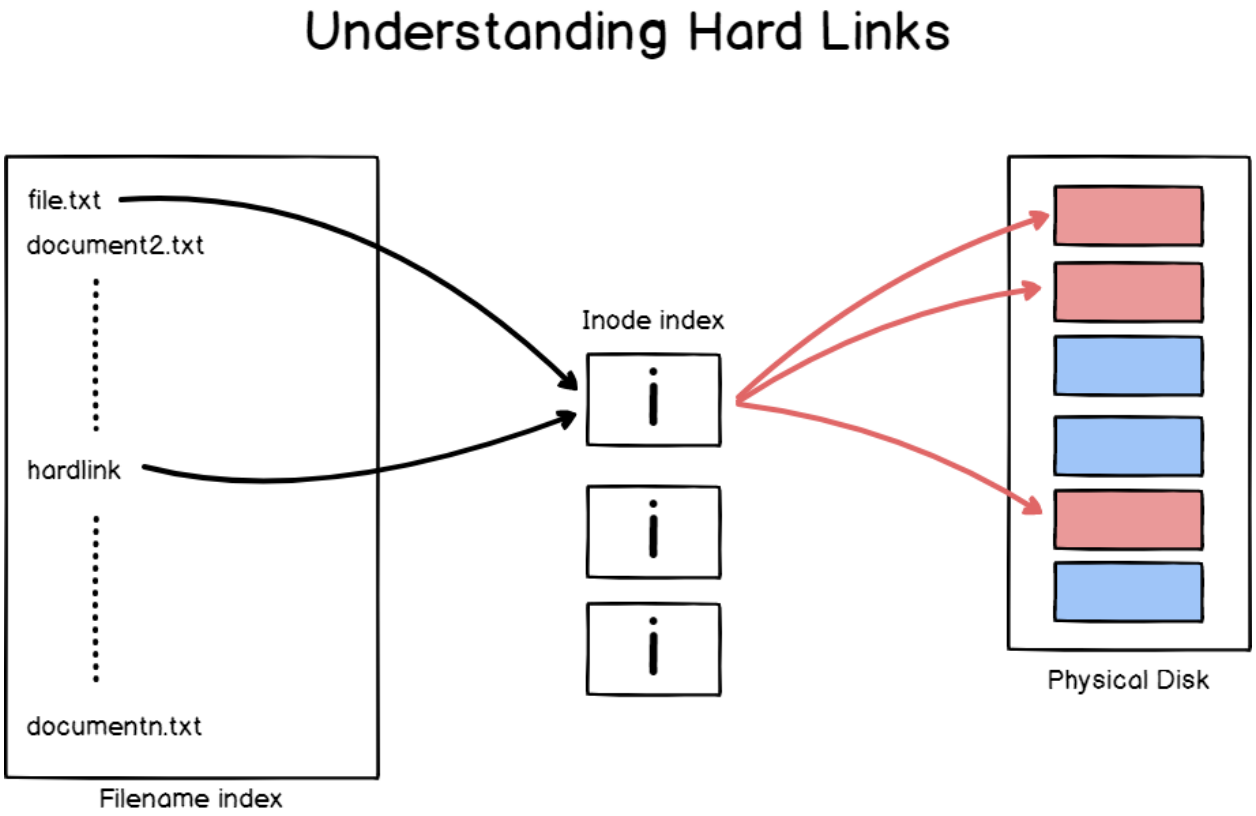
软链接：ln -s file1 file2

1.2.2 硬链接

事实上，硬链接原理十分简单，就是新建一个 inode 号相同的文件，通过相同的 inode **直接地**访问原文件的数据。



更简洁的表述如下：



因为 file 和 file_hardlink 的 i_no 相同，所以他们指向同样的 inode，最后找到同样的 blocks。

我们来实际测试一下

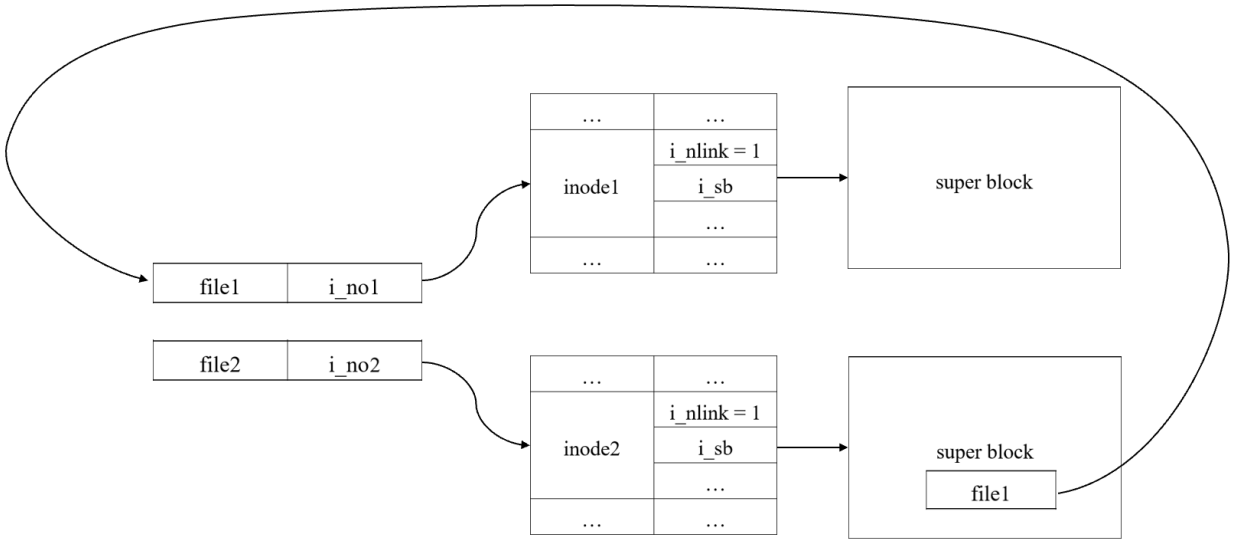
ln file file_hardlink

```
root@VM-16-16-centos ~ # ll -i file
397094 -rw-r--r-- 1 root root 6 May  6 15:07 file
root@VM-16-16-centos ~ # ln file file_hardlink
root@VM-16-16-centos ~ # ll -i file*
397094 -rw-r--r-- 2 root root 6 May  6 15:07 file
397094 -rw-r--r-- 2 root root 6 May  6 15:07 file_hardlink
```

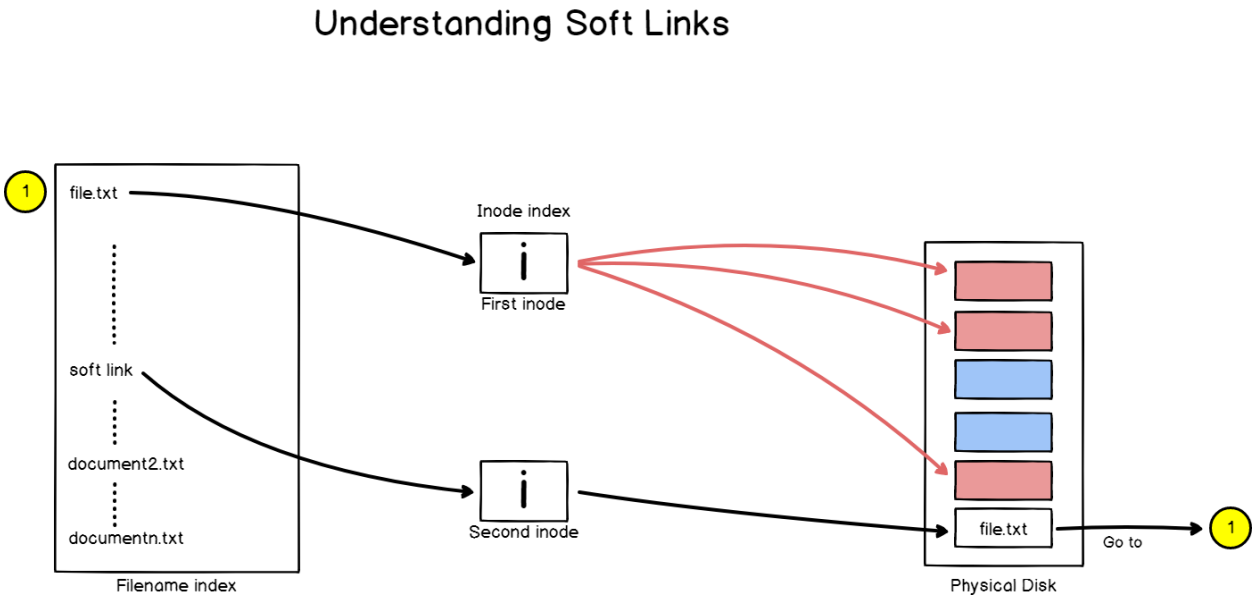
file 和 file_hardlink 的 i_no 确实相同。

1.2.3 软链接

软链接的原理要稍微复杂一些，它通过存储原文件的路径，**间接地**访问原文件的数据。



更简洁的表述如下：



file_softlink 虽然不与 file 指向同一 inode，但 file_softlink 指向的 inode 指向的数据是

file 的路径，找到 file 后访问 file 指向的 inode，从而访问 file 的数据。

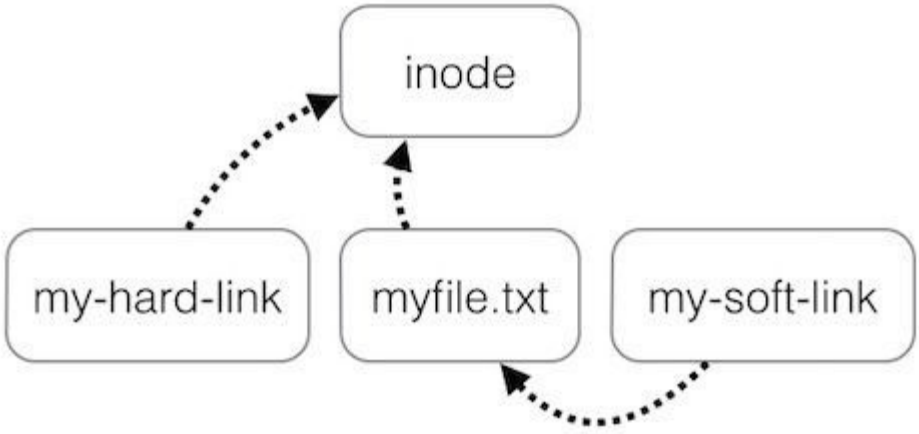
实际测试一下

```
root@VM-16-16-centos ~ # ln -s file file_softlink
root@VM-16-16-centos ~ # ll -i file*
397094 -rw-r--r-- 2 root root 6 May  6 15:07 file
397094 -rw-r--r-- 2 root root 6 May  6 15:07 file_hardlink
394815 lrwxrwxrwx 1 root root 4 May  6 15:10 file_softlink -> file
```

可以发现 file 和 file_softlink 的 i_no 是不同的，ls 命令还为我们指出了 file_softlink 所指向的文件。

1.2.4 联系与区别

大体上，他们的关系如下：



1. soft link 不增加 i_nlink（硬链接计数）

对 file 分别建立硬链接和软链接后，使用 stat 命令，发现 Links 为 2。

```
root@VM-16-16-centos ~ # stat file
  File: file
  Size: 6               Blocks: 8          IO Block: 4096   regular file
Device: fd01h/64769d    Inode: 397094      Links: 2
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2022-05-06 15:07:30.474181639 +0800
Modify: 2022-05-06 15:07:29.032168677 +0800
Change: 2022-05-06 15:08:12.860562637 +0800
 Birth: -
```

此外，因为 i_nlink 存储于 inode 中，file 和 file_hardlink 指向同一 inode，所以他们的 statistics 相同。

```
root@VM-16-16-centos ~ # stat file_hardlink
  File: file_hardlink
  Size: 6               Blocks: 8          IO Block: 4096   regular file
Device: fd01h/64769d    Inode: 397094      Links: 2
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2022-05-06 15:07:30.474181639 +0800
Modify: 2022-05-06 15:07:29.032168677 +0800
Change: 2022-05-06 15:08:12.860562637 +0800
 Birth: -
```

2. 硬链接不能跨越分区，但软链接可以。这个原因很简单，因为 i_no 在不同分区是重新计数的，同一 i_no 不指向同一 inode；

3. 硬链接直接访问文件数据，软链接间接访问文件数据；

4. 只有当 i_no 相同的所有文件和硬链接文件均不存在时才会删除 inode；

5. 当软链接指向的文件路径不存在（重命名或删除）时，软链接失效。

mv file file2

```
root@VM-16-16-centos ~ # mv file file2
root@VM-16-16-centos ~ # ll -i file*
397094 -rw-r--r-- 2 root root 6 May  6 15:07 file2
397094 -rw-r--r-- 2 root root 6 May  6 15:07 file_hardlink
394815 lrwxrwxrwx 1 root root 4 May  6 15:10 file_softlink -> file
```

可以看到 ls 用醒目的红色提醒我们这个软链接失效了。

2 进程通信

2.1 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。

UNIX 系统上有各种通信和同步工具，根据功能，它们分为三类：

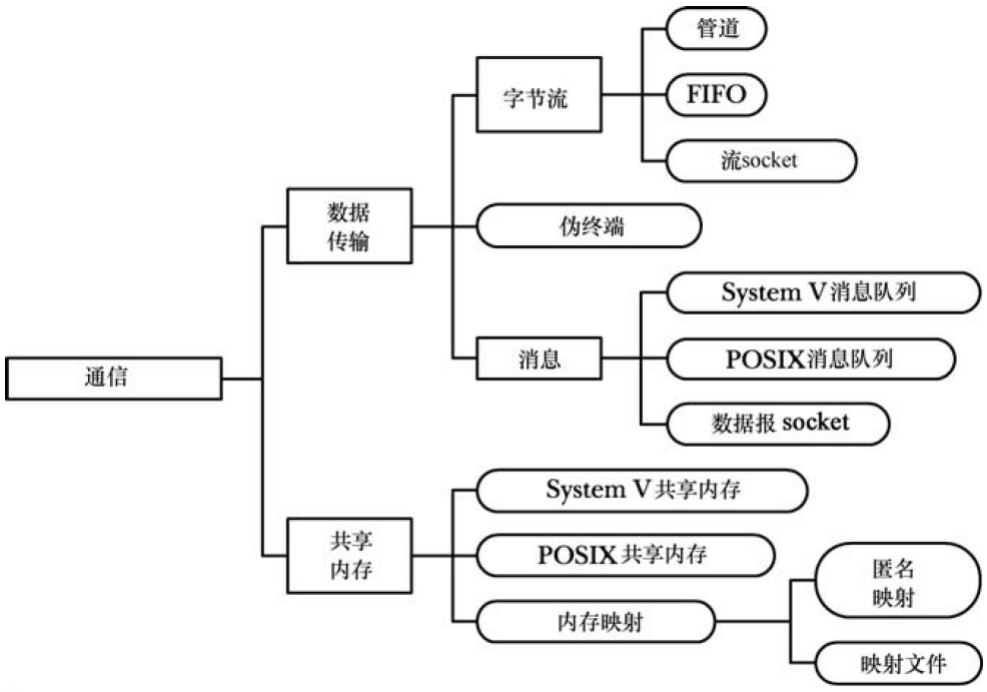
通信：这些工具关注进程之间的数据交换。

同步：这些进程关注进程和线程操作之间的同步。

信号：尽管信号的主要作用并不在此，但在特定场景下仍然可以将它作为一种同步技术。更罕见的是信号还可以作为一种通信技术：信号编号本身是一种形式的信息，并且可以在实时信号上绑定数据（一个整数或指针）。

尽管其中一些工具关注的是同步，但通用术语**进程间通信**（IPC）通常指代所有这些工具。

在这一部分我们主要关注“通信”，可以总结如下图：



接下来我们逐个介绍它们。

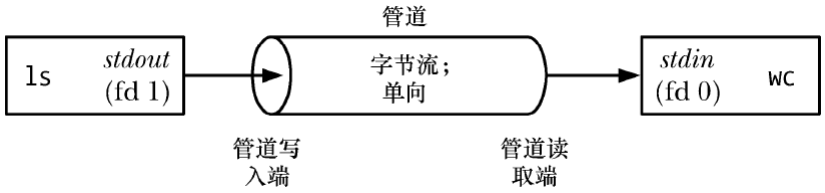
2.1.1 管道（Pipe）

管道是 UNIX 系统上最古老的 IPC 方法，它可以用来在**相关**进程之间传递数据，最常见的应用是：给定两个运行不同程序（命令）的进程，在 shell 中让一个进程的输出作为另一个进程的输入。

例如这个两个程序

```
root@VM-16-16-centos ~ # ls | wc -l
19
```

用管道描述即为：



管道的重要特征：

1. 一个管道是一个字节流
2. 从管道中读取数据

   试图从一个当前为空的管道中读取数据将会被阻塞直到至少有一个字节被写入到管道中为止。如果管道的写入端被关闭了，那么从管道中读取数据的进程在读完管道中剩余的所有数据之后将会看到文件结束（即 read()返回 0）。
3. 管道是单向的
4. 可以确保写入不超过 PIPE_BUF 字节的操作是原子的
5. 管道的容量是有限的

unistd.h 中的 pipe

```
int pipe(int pipefd[2]);
```

(1) 头文件

```
#include <unistd.h>
```

(2) 原型

```
int pipe(int pipefd[2]);
```

(3) 参数

pipefd[2]：数组 pipefd 用于返回两个引用管道末端的文件描述符。

pipefd[0]指的是管道的读取端。

pipefd[1]指的是管道的写入端。

(4) 返回值

运行成功：返回 0

运行失败：返回-1

(5) 实例

我们编写一个简单的程序，父进程通过管道向子进程写一个字符串，子进程将其打印。

t_pipe.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```c
#include <string.h>

int main(int argc, char *argv[])
{
    int fd[2];                          /* 两个文件描述符 */
    pid_t pid;                          /* 进程 ID */
    char buffer[20];                    /* 数据缓冲区 */
    int return_code;                    /* 返回值 */

    return_code = pipe(fd);
    if(return_code == -1)               /* 创建管道 */
        perror("pipe");

    pid = fork();
    switch(pid) {
        case -1:
            perror("fork");
            exit(EXIT_FAILURE);
        case 0:                         /* 子进程 */
            close(fd[1]);               /* 关闭写端 */
            read(fd[0], buffer, 20);
            printf("%s", buffer);
        default:                        /* 父进程 */
            close(fd[0]);               /* 关闭读端 */
            write(fd[1], argv[1], strlen(argv[1]));
    }
    exit(EXIT_SUCCESS);
}
```

t_pipe.c

./t_pipe.elf

```
root@VM-16-16-centos ~/C_CODE/IPC # ./t_pipe.elf Hello,World!
root@VM-16-16-centos ~/C_CODE/IPC # Hello,World!
```

“相关”的概念就在于此：通信的子进程之间需要通过一个共同的祖先进程创建管道。

2.1.2 有名管道（FIFO）

FIFO 是管道概念的一个变体，它们之间的一个重要差别在于 FIFO 可以用于**任意**进程间的通信。

一旦打开了 FIFO，就能在它上面使用与操作管道和其他文件的系统调用一样的 I/O 系统调用了（如 read()、write()和 close()）。与管道一样，FIFO 也有一个写入端和读取端，并且从管道中读取数据的顺序与写入的顺序是一样的。FIFO 的名称也由此而来：先入先出。

FIFO 有时候也被称为命名管道。与管道一样，当所有引用 FIFO 的描述符都被关闭之后，所有未被读取的数据会被丢弃。

使用 mkfifo 命令可以在 shell 中创建一个 FIFO。

mkfifo [-m mode] pathname

pathname 是创建的 FIFO 的名称，-m 选项用来指定权限 mode，其工作方式与 chmod 命令一样。

sys/stat.h 中的 mkfifo

```
int mkfifo(const char *pathname, mode_t mode);
```

(1) 头文件

```
#include <sys/stat.h>
```

(2) 原型

```
int mkfifo(const char *pathname, mode_t mode);
```

(3) 参数

pathname：创建一个名为 pathname 的 FIFO 特殊文件

mode：指定 FIFO 的权限

(4) 返回值

运行成功：返回 0

运行失败：返回-1

(5) 实例

mkfifo test_fifo

wc -l < test_fifo &

ls -l | tee test_fifo | sort -k5n

```
root@VM-16-16-centos ~/C_CODE/IPC # mkfifo test_fifo
root@VM-16-16-centos ~/C_CODE/IPC # wc -l < test_fifo &
[1] 669010
root@VM-16-16-centos ~/C_CODE/IPC # ls -l | tee test_fifo | sort -k5n
4
prw-r--r-- 1 root root     0 May  6 16:56 test_fifo
total 24
-rw-r--r-- 1 root root   746 May  6 16:41 t_pipe.c
-rwxr-xr-x 1 root root 17888 May  6 16:41 t_pipe.elf
[1]+  Done                    wc -l < test_fifo
```

这个程序创建了一个名为 test_fifo 的 FIFO，然后在后台启动一个 wc 命令，该命令会打开 FIFO 以读取数据（这个操作会阻塞直到有进程打开 FIFO 写入数据为止），接着执行一条管道线将 ls 的输出发送给 tee，tee 会将输出传递给管道线中的下一个命令 sort，同时还会将输出发送给名为 test_fifo 的 FIFO。（sort 的-k5n 选项会导致 ls 的输出按照第五个以

空格分隔的字段的数值升序排序。）

下面的消息队列、信号量、共享内存，事实上都是 System V 版，另外还有 POSIX 版，就不在这里讲述了。

2.1.3 消息队列（Message queue）

消息队列，是消息的链接表，存放在内核中。一个消息队列由一个标识符（即队列 ID）来标识。

关于 System V 消息队列的系统调用位于 sys/msg.h

```
/* Message queue control operation.  */
extern int msgctl (int __msqid, int __cmd, struct msqid_ds *__buf) __THROW;

/* Get messages queue.  */
extern int msgget (key_t __key, int __msgflg) __THROW;

/* Receive message from message queue.

   This function is a cancellation point and therefore not marked with
   __THROW.  */
extern ssize_t msgrcv (int __msqid, void *__msgp, size_t __msgsz,
                       long int __msgtyp, int __msgflg);

/* Send message to message queue.

   This function is a cancellation point and therefore not marked with
   __THROW.  */
extern int msgsnd (int __msqid, const void *__msgp, size_t __msgsz,
                   int __msgflg);
```

(1) 头文件

```
#include <sys/types.h>
#include <sys/msg.h>
```

(2) 原型

```
int msgctl (int msqid, int cmd, struct msqid_ds *buf);
int msgget (key_t key, int msgflg);
ssize_t msgrcv (int msqid, void *msgp, size_t msgsz, long int msgtyp, int msgflg);
int msgsnd (int msqid, const void *msgp, size_t msgsz, int msgflg);
```

(3) 参数

msqid：消息队列的标识符。

key：一个键（即通常是值 IPC_PRIVATE 或 ftok()返回的一个键）。

msgflg：一个指定施加于新消息队列之上的权限或检查一个既有队列的权限的位掩码。此外，在 msgflg 参数中还可以将下列标记中的零个或多个标记取 OR（|）以控制 msgget()的操作。

msgsz：指定 mtext 字段中包含的字节数。

msgtyp：根据 mtype 字段的值来选择消息。

maxmsgsz：指定 msgp 缓冲区中 mtext 字段的最大可用空间。

cmd：指定在队列上执行的操作

(4) 返回值

全部为：

运行成功：返回 0

运行失败：返回-1

(5) 实例

创建如下三个程序。

svmsg_create.c

```
/* svmsg_create.c

   Experiment with the use of msgget() to create a System V message queue.
*/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/stat.h>
#include "tlpi_hdr.h"

static void             /* Print usage info, then exit */
usageError(const char *progName, const char *msg)
{
    if (msg != NULL)
        fprintf(stderr, "%s", msg);
    fprintf(stderr, "Usage: %s [-cx] {-f pathname | -k key | -p} "
                    "[octal-perms]\n", progName);
    fprintf(stderr, "    -c           Use IPC_CREAT flag\n");
    fprintf(stderr, "    -x           Use IPC_EXCL flag\n");
    fprintf(stderr, "    -f pathname  Generate key using ftok()\n");
    fprintf(stderr, "    -k key       Use 'key' as key\n");
    fprintf(stderr, "    -p           Use IPC_PRIVATE key\n");
    exit(EXIT_FAILURE);
}

int
```

```
main(int argc, char *argv[])
{
    int numKeyFlags;            /* Counts -f, -k, and -p options */
    int flags, msqid, opt;
    unsigned int perms;
    long lkey;
    key_t key;

    /* Parse command-line options and arguments */

    numKeyFlags = 0;
    flags = 0;

    while ((opt = getopt(argc, argv, "cf:k:px")) != -1) {
        switch (opt) {
        case 'c':
            flags |= IPC_CREAT;
            break;

        case 'f':               /* -f pathname */
            key = ftok(optarg, 1);
            if (key == -1)
                errExit("ftok");
            numKeyFlags++;
            break;

        case 'k':               /* -k key (octal, decimal or hexadecimal) */
            if (sscanf(optarg, "%li", &lkey) != 1)
                cmdLineErr("-k option requires a numeric argument\n");
            key = lkey;
            numKeyFlags++;
            break;

        case 'p':
            key = IPC_PRIVATE;
            numKeyFlags++;
            break;

        case 'x':
            flags |= IPC_EXCL;
            break;

        default:
            usageError(argv[0], "Bad option\n");
```

```
        }
    }

    if (numKeyFlags != 1)
        usageError(argv[0], "Exactly one of the options -f, -k, "
                            "or -p must be supplied\n");

    perms = (optind == argc) ? (S_IRUSR | S_IWUSR) :
                getInt(argv[optind], GN_BASE_8, "octal-perms");

    msqid = msgget(key, flags | perms);
    if (msqid == -1)
        errExit("msgget");

    printf("%d\n", msqid);
    exit(EXIT_SUCCESS);
}
```

svmsg_create.c

svmsg_send.c

```
/* svmsg_send.c

   Usage: svmsg_send [-n] msqid msg-type [msg-text]

   Experiment with the msgsnd() system call to send messages to a
   System V message queue.

   See also svmsg_receive.c.
*/
#include <sys/types.h>
#include <sys/msg.h>
#include "tlpi_hdr.h"

#define MAX_MTEXT 1024

struct mbuf {
    long mtype;                         /* Message type */
    char mtext[MAX_MTEXT];              /* Message body */
};

static void             /* Print (optional) message, then usage description */
usageError(const char *progName, const char *msg)
{
```

```
    if (msg != NULL)
        fprintf(stderr, "%s", msg);
    fprintf(stderr, "Usage: %s [-n] msqid msg-type [msg-text]\n", progName);
    fprintf(stderr, "    -n       Use IPC_NOWAIT flag\n");
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    int msqid, flags, msgLen;
    struct mbuf msg;                    /* Message buffer for msgsnd() */
    int opt;                            /* Option character from getopt() */

    /* Parse command-line options and arguments */

    flags = 0;
    while ((opt = getopt(argc, argv, "n")) != -1) {
        if (opt == 'n')
            flags |= IPC_NOWAIT;
        else
            usageError(argv[0], NULL);
    }

    if (argc < optind + 2 || argc > optind + 3)
        usageError(argv[0], "Wrong number of arguments\n");

    msqid = getInt(argv[optind], 0, "msqid");
    msg.mtype = getInt(argv[optind + 1], 0, "msg-type");

    if (argc > optind + 2) {            /* 'msg-text' was supplied */
        msgLen = strlen(argv[optind + 2]) + 1;
        if (msgLen > MAX_MTEXT)
            cmdLineErr("msg-text too long (max: %d characters)\n", MAX_MTEXT);

        memcpy(msg.mtext, argv[optind + 2], msgLen);

    } else {                            /* No 'msg-text' ==> zero-length msg */
        msgLen = 0;
    }

    /* Send message */

    if (msgsnd(msqid, &msg, msgLen, flags) == -1)
```

```
        errExit("msgsnd");

    exit(EXIT_SUCCESS);
}
```

svmsg_send.c

svmsg_receive.c

```
/* svmsg_receive.c

   Usage: svmsg_receive [-nex] [-t msg-type] msqid [max-bytes]

   Experiment with the msgrcv() system call to receive messages from a
   System V message queue.

   See also svmsg_send.c.
*/
#define _GNU_SOURCE             /* Get definition of MSG_EXCEPT */
#include <sys/types.h>
#include <sys/msg.h>
#include "tlpi_hdr.h"

#define MAX_MTEXT 1024

struct mbuf {
    long mtype;                 /* Message type */
    char mtext[MAX_MTEXT];      /* Message body */
};

static void
usageError(const char *progName, const char *msg)
{
    if (msg != NULL)
        fprintf(stderr, "%s", msg);
    fprintf(stderr, "Usage: %s [options] msqid [max-bytes]\n", progName);
    fprintf(stderr, "Permitted options are:\n");
    fprintf(stderr, "    -e       Use MSG_NOERROR flag\n");
    fprintf(stderr, "    -t type  Select message of given type\n");
    fprintf(stderr, "    -n       Use IPC_NOWAIT flag\n");
#ifdef MSG_EXCEPT
    fprintf(stderr, "    -x       Use MSG_EXCEPT flag\n");
#endif
    exit(EXIT_FAILURE);
}
```

```
int
main(int argc, char *argv[])
{
    int msqid, flags, type;
    ssize_t msgLen;
    size_t maxBytes;
    struct mbuf msg;            /* Message buffer for msgrcv() */
    int opt;                    /* Option character from getopt() */

    /* Parse command-line options and arguments */

    flags = 0;
    type = 0;
    while ((opt = getopt(argc, argv, "ent:x")) != -1) {
        switch (opt) {
        case 'e':       flags |= MSG_NOERROR;   break;
        case 'n':       flags |= IPC_NOWAIT;    break;
        case 't':       type = atoi(optarg);    break;
#ifdef MSG_EXCEPT
        case 'x':       flags |= MSG_EXCEPT;    break;
#endif
        default:        usageError(argv[0], NULL);
        }
    }

    if (argc < optind + 1 || argc > optind + 2)
        usageError(argv[0], "Wrong number of arguments\n");

    msqid = getInt(argv[optind], 0, "msqid");
    maxBytes = (argc > optind + 1) ?
                getInt(argv[optind + 1], 0, "max-bytes") : MAX_MTEXT;

    /* Get message and display on stdout */

    msgLen = msgrcv(msqid, &msg, maxBytes, type, flags);
    if (msgLen == -1)
        errExit("msgrcv");

    printf("Received: type=%ld; length=%ld", msg.mtype, (long) msgLen);
    if (msgLen > 0)
        printf("; body=%s", msg.mtext);
    printf("\n");
```

```
    exit(EXIT_SUCCESS);
}
```

svmsg_receive.c

首先使用 IPC_PRIVATE 键创建了一个消息队列，然后向队列中写入了三条不同类型的消息。

./svmsg_create.elf -p

```
root@VM-16-16-centos ~/C_CODE/IPC/svmsg # ./svmsg_create.elf -p
1
```

./svmsg_send.elf 1 20 "I hear and I forget."

./svmsg_send.elf 1 10 "I see and I remember."

./svmsg_send.elf 1 30 "I do and I understand."

```
root@VM-16-16-centos ~/C_CODE/IPC/svmsg # ./svmsg_send.elf 1 20 "I hear and I forget."
root@VM-16-16-centos ~/C_CODE/IPC/svmsg # ./svmsg_send.elf 1 10 "I see and I remember."
root@VM-16-16-centos ~/C_CODE/IPC/svmsg # ./svmsg_send.elf 1 30 "I do and I understand."
```

接着使用 svmsg_receive 从队列中读取类型小于或等于 20 的消息。

./svmsg_receive.elf -t -20 1

./svmsg_receive.elf -t -20 1

./svmsg_receive.elf -t -20 1

```
root@VM-16-16-centos ~/C_CODE/IPC/svmsg # ./svmsg_receive.elf -t -20 1
Received: type=10; length=22; body=I see and I remember.
root@VM-16-16-centos ~/C_CODE/IPC/svmsg # ./svmsg_receive.elf -t -20 1
Received: type=20; length=21; body=I hear and I forget.
root@VM-16-16-centos ~/C_CODE/IPC/svmsg # ./svmsg_receive.elf -t -20 1
^C
```

上面最后一条命令会阻塞，因为队列中已经没有类型小于或等于 20 的消息了。因此需要输入 Control-C 来终止这个命令，然后执行一个从队列中读取任意类型的消息的命令。

```
root@VM-16-16-centos ~/C_CODE/IPC/svmsg # ./svmsg_receive.elf 1
Received: type=30; length=23; body=I do and I understand.
```

注：这三句话译自“不闻不若闻之，闻之不若见之，见之不若知之，知之不若行之”。我个人感觉只翻译了后三句，而且破坏了递进逻辑，虽然很好听就是了。

2.1.4 信号量（Signal）

System V 信号量虽然可以用来通信，但它不是用来在进程间传输数据的。相反，它们用来同步进程的动作。信号量的一个常见用途是同步对一块共享内存的访问以防止出现一个进程在访问共享内存的同时另一个进程更新这块内存的情况。

一个信号量是一个由内核维护的整数，其值被限制为大于或等于 0。在一个信号量上可以执行各种操作（即系统调用），包括：

1. 将信号量设置成一个绝对值；

2. 在信号量当前值的基础上加上一个数量；

3. 在信号量当前值的基础上减去一个数量；

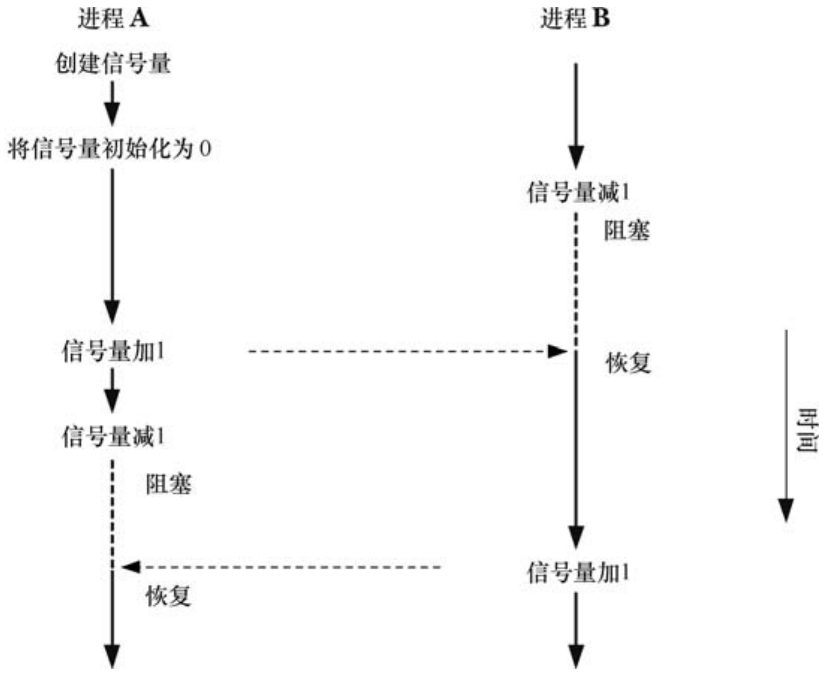
4. 等待信号量的值等于 0。

和《操作系统》课程中学的一样，上面操作中的后两个可能会导致调用进程阻塞。

当减小一个信号量的值时，内核会将所有试图将信号量值降低到 0 之下的操作阻塞，如果信号量的当前值不为 0，那么等待信号量的值等于 0 的调用进程将会发生阻塞。

不管是何种情况，调用进程会一直保持阻塞直到其他一些进程将信号量的值修改为一个允许这些操作继续向前的值，在那个时刻内核会唤醒被阻塞的进程。

下图是使用一个信号量来同步两个交替将信号量的值在 0 和 1 之间切换的进程的动作。



信号量操作位于/usr/include/sys/sem.h

```
/* Semaphore control operation.  */
extern int semctl (int __semid, int __semnum, int __cmd, ...) __THROW;

/* Get semaphore.  */
extern int semget (key_t __key, int __nsems, int __semflg) __THROW;

/* Operate on semaphore.  */
extern int semop (int __semid, struct sembuf *__sops, size_t __nsops) __THROW;
```

(1) 头文件

```
#include <sys/types.h>
#include <sys/sem.h>
```

(2) 原型

```
int semctl (int semid, int semnum, int cmd, ...);
int semget (key_t key, int nsems, int semflg);
int semop (int semid, struct sembuf *sops, size_t nsops);
```

(3) 参数

key：一个键（通常使用值 IPC_PRIVATE 或由 ftok()返回的键）。

nsems：如果使用 semget()创建一个新信号量集，那么 nsems 会指定集合中信号量的数量，并且其值必须大于 0。如果使用 semget()来获取一个既有集的标识符，那么 nsems 必须要小于或等于集合的大小（否则会发生 EINVAL 错误）。无法修改一个既有集中的信号量数量。

semflg ：一个位掩码，它指定了施加于新信号量集之上的权限或需检查的一个既有集合的权限。此外，在 semflg 中可以通过对标记中的零个或多个取 OR 来控制 semget()的操作。

(4) 返回值

全部为：

运行成功：返回 0

运行失败：返回-1

(5) 实例

我们可以用如下程序体会信号量操作。

svsem_demo.c

```
/* svsem_demo.c

   A simple demonstration of System V semaphores.
*/
#include <sys/types.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include "curr_time.h"                  /* Declaration of currTime() */
#include "semun.h"                      /* Definition of semun union */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int semid;

    if (argc < 2 || argc > 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s init-value\n"
```

```
                "    or: %s semid operation\n", argv[0], argv[0]);

    if (argc == 2) {                    /* Create and initialize semaphore */
        union semun arg;

        semid = semget(IPC_PRIVATE, 1, S_IRUSR | S_IWUSR);
        if (semid == -1)
            errExit("semid");

        arg.val = getInt(argv[1], 0, "init-value");
        if (semctl(semid, /* semnum= */ 0, SETVAL, arg) == -1)
            errExit("semctl");

        printf("Semaphore ID = %d\n", semid);

    } else {                            /* Perform an operation on first semaphore */

        struct sembuf sop;              /* Structure defining operation */

        semid = getInt(argv[1], 0, "semid");

        sop.sem_num = 0;                /* Specifies first semaphore in set */
        sop.sem_op = getInt(argv[2], 0, "operation");
                                        /* Add, subtract, or wait for 0 */
        sop.sem_flg = 0;                /* No special options for operation */

        printf("%ld: about to semop at  %s\n", (long) getpid(), currTime("%T"));
        if (semop(semid, &sop, 1) == -1)
            errExit("semop");

        printf("%ld: semop completed at %s\n", (long) getpid(), currTime("%T"));
    }

    exit(EXIT_SUCCESS);
}
```

svsem_demo.c

创建一个信号量并将其初始化为 0

./svsem_demo.elf 0

```
root@VM-16-16-centos ~/C_CODE/IPC/svsem # ./svsem_demo.elf 0
Semaphore ID = 0
```

执行一个后台命令（&）将信号量值-2

./svsem_demo.elf 0 -2 &

```
root@VM-16-16-centos ~/C_CODE/IPC/svsem # ./svsem_demo.elf 0 -2 &
[1] 691235
```

因为信号量成了负数，所以这个命令阻塞了，现在执行一个命令对其+3

./svsem_demo.elf 0 +3

```
root@VM-16-16-centos ~/C_CODE/IPC/svsem # 691235: about to semop at  19:37:34
./svsem_demo.elf 0 +3
691261: about to semop at  19:37:49
691261: semop completed at 19:37:49
691235: semop completed at 19:37:49
[1]+  Done                    ./svsem_demo.elf 0 -2
```

这个信号量增加操作会立即成功，并且会导致后台命令中的信号量缩减操作能够向前执行，因为在执行该操作之后不会导致信号量值小于 0。

2.1.5 共享内存（Shared memory）

共享内存允许两个或多个进程共享物理内存的同一块区域（通常被称为段）。由于一个共享内存段会成为一个进程用户空间内存的一部分，因此这种 IPC 机制无需内核介入。所有需要做的就是让一个进程将数据复制进共享内存中，并且这部分数据会对其他所有共享同一个段的进程可用。

共享内存的系统调用位于/usr/include/sys/shm.h

```
/* The following System V style IPC functions implement a shared memory
   facility.  The definition is found in XPG4.2.  */

/* Shared memory control operation.  */
extern int shmctl (int __shmid, int __cmd, struct shmid_ds *__buf) __THROW;

/* Get shared memory segment.  */
extern int shmget (key_t __key, size_t __size, int __shmflg) __THROW;

/* Attach shared memory segment.  */
extern void *shmat (int __shmid, const void *__shmaddr, int __shmflg)
     __THROW;

/* Detach shared memory segment.  */
extern int shmdt (const void *__shmaddr) __THROW;
```

(1) 头文件

```
#include <sys/types.h>
#include <sys/shm.h>
```

(2) 原型

```
int shmctl (int __shmid, int __cmd, struct shmid_ds *__buf);
int shmget (key_t __key, size_t __size, int __shmflg);
```

```
void *shmat (int __shmid, const void *__shmaddr, int __shmflg);
int shmdt (const void *__shmaddr);
```

(3) 参数

key：通常是 IPC_PRIVATE 值或由 ftok()返回的键当使用

size：一个正整数，表示需分配的段的字节数。内核是以系统分页大小的整数倍来分配共享内存的，因此实际上 size 会被提升到最近的系统分页大小的整数倍。如果使用 shmget()来获取一个既有段的标识符，那么 size 对段不会产生任何效果，但它必须要小于或等于段的大小。

shmflg：指定施加于新共享内存段上的权限或需检查的既有内存段的权限。此外，在 shmflg 中还可以对标记中的零个或多个取 OR 来控制 shmget()的操作。

shmaddr：配合 shmflg 位掩码参数中 SHM_RND 位的设置控制段是如何被附加上去的。

(4) 返回值

全部为：

运行成功：返回 0

运行失败：返回-1

(5) 实例

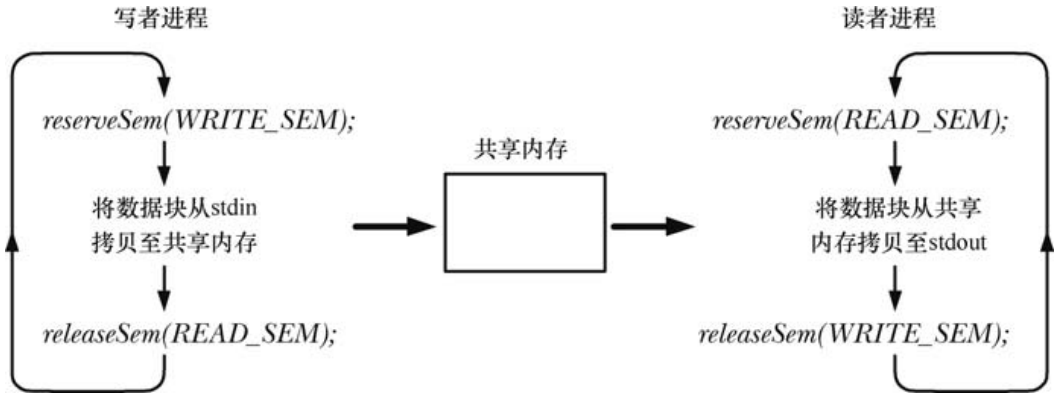
这个应用程序由两个程序构成：写者和读者。写者从标准输入中读取数据块并将数据复制（“写”）到一个共享内存段中。读者将共享内存段中的数据块复制（“读”）到标准输出中。实际上，程序在某种程度上将共享内存当成了管道来处理。

两个程序使用了二元信号量协议（initSemAvailable()、initSemInUse()、reserveSem()以及 releaseSem()函数）中的一对 System V 信号量来确保：

1. 一次只有一个进程访问共享内存段；

2. 进程交替地访问段（即写者写入一些数据，然后读者读取这些数据，然后写者再次写入数据，以此类推）。

下图概述了这两个信号量的使用。注意写者对两个信号量进行了初始化，这样它就成为两个程序中第一个能够访问共享内存段的程序了，即写者的信号量初始时是可用的，而读者的信号量初始时是正在被使用中的。



这个应用程序的源代码由三个文件构成。第一个文件是由读者程序和写者程序共享的头文件。这个头文件定义了 shmseg 结构，程序使用了这个结构来声明指向共享内存段的指针，这样就能给共享内存段中的字节规定一种结构。

svshm_xfr.h

```
/*  svshm_xfr.h

   Header file used by the svshm_xfr_reader.c and svshm_xfr_writer.c programs.
*/
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include "binary_sems.h"        /* Declares our binary semaphore functions */
#include "tlpi_hdr.h"

/* Hard-coded keys for IPC objects */

#define SHM_KEY 0x1234          /* Key for shared memory segment */
#define SEM_KEY 0x5678          /* Key for semaphore set */

#define OBJ_PERMS (S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP)
                                /* Permissions for our IPC objects */

/* Two semaphores are used to ensure exclusive, alternating access
   to the shared memory segment */

#define WRITE_SEM 0             /* Writer has access to shared memory */
#define READ_SEM 1              /* Reader has access to shared memory */

#ifndef BUF_SIZE                /* Allow "cc -D" to override definition */
#define BUF_SIZE 1024           /* Size of transfer buffer */
#endif
```

```
struct shmseg {                   /* Defines structure of shared memory segment */
    int cnt;                      /* Number of bytes used in 'buf' */
    char buf[BUF_SIZE];           /* Data being transferred */
};
```

svshm_xfr.h

svshm_xfr_writer.c 是写者程序。这个程序按序完成下列任务。

1. 创建一个包含两个信号量的集合，写者和读者程序会使用这两个信号量来确保它们交替地访问共享内存段。信号量被初始化为使写者首先访问共享内存段。由于是由写者来创建信号量集的，因此必须在启动读者之前启动写者。

2. 创建共享内存段并将其附加到写者的虚拟地址空间中系统所选择的一个地址处。

3. 进入一个循环将数据从标准输入传输到共享内存段。每个循环迭代需要按序完成下面的任务：

-预留（减小）写者的信号量。

-从标准输入中读取数据并将数据复制到共享内存段。

-释放（增加）读者的信号量。

4. 当标准输入中没有可用的数据时循环终止。在最后一次循环中，写者通过传递一个长度为 0 的数据块（shmp->cnt 为 0）来通知读者没有更多的数据了。

5. 在退出循环时，写者再次预留其信号量，这样它就能知道读者已经完成了对共享内存的最后一次访问了。写者随后删除了共享内存段和信号量集。

svshm_xfr_writer.c

```
/*  svshm_xfr_writer.c

    Read buffers of data data from standard input into a System V shared memory
    segment from which it is copied by svshm_xfr_reader.c

    We use a pair of binary semaphores to ensure that the writer and reader have
    exclusive, alternating access to the shared memory. (I.e., the writer writes
    a block of text, then the reader reads, then the writer writes etc). This
    ensures that each block of data is processed in turn by the writer and
    reader.

    This program needs to be started before the reader process as it creates the
    shared memory and semaphores used by both processes.

    Together, these two programs can be used to transfer a stream of data through
    shared memory as follows:

        $ svshm_xfr_writer < infile &
        $ svshm_xfr_reader > out_file
```

```
*/
#include "semun.h"              /* Definition of semun union */
#include "svshm_xfr.h"

int
main(int argc, char *argv[])
{
    int semid, shmid, bytes, xfrs;
    struct shmseg *shmp;
    union semun dummy;

    /* Create set containing two semaphores; initialize so that
       writer has first access to shared memory. */

    semid = semget(SEM_KEY, 2, IPC_CREAT | OBJ_PERMS);
    if (semid == -1)
        errExit("semget");

    if (initSemAvailable(semid, WRITE_SEM) == -1)
        errExit("initSemAvailable");
    if (initSemInUse(semid, READ_SEM) == -1)
        errExit("initSemInUse");

    /* Create shared memory; attach at address chosen by system */

    shmid = shmget(SHM_KEY, sizeof(struct shmseg), IPC_CREAT | OBJ_PERMS);
    if (shmid == -1)
        errExit("shmget");

    shmp = shmat(shmid, NULL, 0);
    if (shmp == (void *) -1)
        errExit("shmat");

    /* Transfer blocks of data from stdin to shared memory */

    for (xfrs = 0, bytes = 0; ; xfrs++, bytes += shmp->cnt) {
        if (reserveSem(semid, WRITE_SEM) == -1)         /* Wait for our turn */
            errExit("reserveSem");

        shmp->cnt = read(STDIN_FILENO, shmp->buf, BUF_SIZE);
        if (shmp->cnt == -1)
            errExit("read");

        if (releaseSem(semid, READ_SEM) == -1)          /* Give reader a turn */
```

```
            errExit("releaseSem");

        /* Have we reached EOF? We test this after giving the reader
           a turn so that it can see the 0 value in shmp->cnt. */

        if (shmp->cnt == 0)
            break;
    }

    /* Wait until reader has let us have one more turn. We then know
       reader has finished, and so we can delete the IPC objects. */

    if (reserveSem(semid, WRITE_SEM) == -1)
        errExit("reserveSem");

    if (semctl(semid, 0, IPC_RMID, dummy) == -1)
        errExit("semctl");
    if (shmdt(shmp) == -1)
        errExit("shmdt");
    if (shmctl(shmid, IPC_RMID, 0) == -1)
        errExit("shmctl");

    fprintf(stderr, "Sent %d bytes (%d xfrs)\n", bytes, xfrs);
    exit(EXIT_SUCCESS);
}
```

svshm_xfr_writer.c

读者程序。它将共享内存段中的数据块传输到标准输出中。读者按序完成了下面的任务。

1. 获取写者程序创建的信号量集合共享内存段的 ID。

2. 附加共享内存段供只读访问。

3. 进入一个循环从共享内存段中传输数据。在每个循环迭代中需要按序完成下面的任务。

-预留（减小）读者的信号量。

-检查 shmp->cnt 是否为 0，如果为 0 就退出循环。

-将共享内存段中的数据块写入标准输出中。

-释放（增加）写者的信号量。

4. 在退出循环之后分离共享内存段并释放写者的信号量，这样写者程序就能够删除 IPC 对象了。

svshm_xfr_reader.c

```
/* svshm_xfr_reader.c

   Read data from a System V shared memory using a binary semaphore lock-step
   protocol; see svshm_xfr_writer.c
*/
#include "svshm_xfr.h"

int
main(int argc, char *argv[])
{
    int semid, shmid, xfrs, bytes;
    struct shmseg *shmp;

    /* Get IDs for semaphore set and shared memory created by writer */

    semid = semget(SEM_KEY, 0, 0);
    if (semid == -1)
        errExit("semget");

    shmid  = shmget(SHM_KEY, 0, 0);
    if (shmid == -1)
        errExit("shmget");

    /* Attach shared memory read-only, as we will only read */

    shmp = shmat(shmid, NULL, SHM_RDONLY);
    if (shmp == (void *) -1)
        errExit("shmat");

    /* Transfer blocks of data from shared memory to stdout */

    for (xfrs = 0, bytes = 0; ; xfrs++) {
        if (reserveSem(semid, READ_SEM) == -1)          /* Wait for our turn */
            errExit("reserveSem");

        if (shmp->cnt == 0)                     /* Writer encountered EOF */
            break;
        bytes += shmp->cnt;

        if (write(STDOUT_FILENO, shmp->buf, shmp->cnt) != shmp->cnt)
            fatal("partial/failed write");

        if (releaseSem(semid, WRITE_SEM) == -1)         /* Give writer a turn */
            errExit("releaseSem");
```

```
    }

    if (shmdt(shmp) == -1)
        errExit("shmdt");

    /* Give writer one more turn, so it can clean up */

    if (releaseSem(semid, WRITE_SEM) == -1)
        errExit("releaseSem");

    fprintf(stderr, "Received %d bytes (%d xfrs)\n", bytes, xfrs);
    exit(EXIT_SUCCESS);
}
```

svshm_xfr_reader.c

1. 打印文件的大小（单位为 byte）

wc -c /etc/services

```
root@VM-16-16-centos ~/C_CODE/IPC/svshm # wc -c /etc/services
692252 /etc/services
```

2. 调用写者，将文件/etc/services 作为输入

./svshm_xfr_writer.elf < /etc/services &

```
root@VM-16-16-centos ~/C_CODE/IPC/svshm # ./svshm_xfr_writer.elf < /etc/services &
[1] 698211
```

3. 调用读者，将内容输出定向到另一个文件中。

./svshm_xfr_reader.elf > out.txt

```
root@VM-16-16-centos ~/C_CODE/IPC/svshm # ./svshm_xfr_reader.elf > out.txt
Received 692252 bytes (677 xfrs)
Sent 692252 bytes (677 xfrs)
[1]+  Done                    ./svshm_xfr_writer.elf < /etc/services
```

4. 调用 diff 命令不产生任何输出，这说明读者产生的输出文件中的内容与写者使用的输入文件中的内容是一样的。

diff /etc/services out.txt

```
root@VM-16-16-centos ~/C_CODE/IPC/svshm # diff /etc/services out.txt
root@VM-16-16-centos ~/C_CODE/IPC/svshm #
```

2.1.1 套接字（Socket）

Socket 是一种 IPC 方法，它允许位于同一主机（计算机）或使用网络连接起来的不同主机上的应用程序之间交换数据。

关于套接字的 socket、bind、listen、accept、connect、read、write、recv、recvfrom、send、sendto、close 等系统调用请查看本课程配套报告中的 4.2 节，这里不再赘述，只给出实例

(1) 实例

这是一个简单地使用了 UNIX domain 中的流 socket 的客户端-服务器应用程序。

us_xfr.h

```
/* us_xfr.h

   Header file for us_xfr_sv.c and us_xfr_cl.c.

   These programs employ a socket in /tmp. This makes it easy to compile
   and run the programs. However, for a security reasons, a real-world
   application should never create sensitive files in /tmp. (As a simple of
   example of the kind of security problems that can result, a malicious
   user could create a file using the name defined in SV_SOCK_PATH, and
   thereby cause a denial of service attack against this application.)
*/
#include <sys/un.h>
#include <sys/socket.h>
#include "tlpi_hdr.h"

#define SV_SOCK_PATH "/tmp/us_xfr"

#define BUF_SIZE 100
```

us_xfr.h

us_xfr_sv.c 执行以下任务：

1. 创建一个 socket。
2. 删除所有与路径名一致的既有文件，这样就能将 socket 绑定到这个路径名上。
3. 为服务器 socket 构建一个地址结构，将 socket 绑定到该地址上，将这个 socket 标记为监听 socket。
4. 执行一个无限循环来处理进入的客户端请求。每次循环迭代执行下列任务。
   - 接受一个连接，为该连接获取一个新 socket cfd。
   - 从已连接的 socket 中读取所有数据并将这些数据写入到标准输出中。
   - 关闭已连接的 socket cfd。
5. 服务器必须要手工终止（如向其发送一个信号）。

us_xfr_sv.c

```
/* us_xfr_sv.c

   An example UNIX stream socket server. Accepts incoming connections
   and copies data sent from clients to stdout.

   See also us_xfr_cl.c.
*/
```

```
#include "us_xfr.h"
#define BACKLOG 5

int
main(int argc, char *argv[])
{
    struct sockaddr_un addr;
    int sfd, cfd;
    ssize_t numRead;
    char buf[BUF_SIZE];
    int return_code;

    sfd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sfd == -1)
        errExit("socket");

    /* Construct server socket address, bind socket to it,
       and make this a listening socket */

    if (strlen(SV_SOCK_PATH) > sizeof(addr.sun_path) - 1)
        fatal("Server socket path too long: %s", SV_SOCK_PATH);

    if (remove(SV_SOCK_PATH) == -1 && errno != ENOENT)
        errExit("remove-%s", SV_SOCK_PATH);

    memset(&addr, 0, sizeof(struct sockaddr_un));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, SV_SOCK_PATH, sizeof(addr.sun_path) - 1);

    return_code = bind(sfd, (struct sockaddr*)&addr, sizeof(struct sockaddr_un));
    if (return_code == -1)
        errExit("bind");

    return_code = listen(sfd, BACKLOG);
    if (return_code == -1)
        errExit("listen");

    while(1) {          /* Handle client connections iteratively */

        /* Accept a connection. The connection is returned on a new
           socket, 'cfd'; the listening socket ('sfd') remains open
           and can be used to accept further connections. */

        cfd = accept(sfd, NULL, NULL);
```

```
        if (cfd == -1)
            errExit("accept");

        /* Transfer data from connected socket to stdout until EOF */

        while ((numRead = read(cfd, buf, BUF_SIZE)) > 0)
            if (write(STDOUT_FILENO, buf, numRead) != numRead)
                fatal("partial/failed write");

        if (numRead == -1)
            errExit("read");

        if (close(cfd) == -1)
            errMsg("close");
    }
}
```

us_xfr_sv.c

us_xfr_sv.c 执行下列任务。

1. 创建一个 socket。

2. 为服务器 socket 构建一个地址结构并连接到位于该地址处的 socket。

3. 执行一个循环将其标准输入复制到 socket 连接上。当遇到标准输入中的文件结尾时客户端就终止，其结果是客户端 socket 将会被关闭并且服务器在从连接的另一端的 socket 中读取数据时会看到文件结束。

us_xfr_sv.c

```
/* us_xfr_cl.c

   An example UNIX domain stream socket client. This client transmits contents
   of stdin to a server socket.

   See also us_xfr_sv.c.
*/
#include "us_xfr.h"

int
main(int argc, char *argv[])
{
    struct sockaddr_un addr;
    int sfd;
    ssize_t numRead;
    char buf[BUF_SIZE];

    sfd = socket(AF_UNIX, SOCK_STREAM, 0);      /* Create client socket */
```

```
    if (sfd == -1)
        errExit("socket");

    /* Construct server address, and make the connection */

    memset(&addr, 0, sizeof(struct sockaddr_un));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, SV_SOCK_PATH, sizeof(addr.sun_path) - 1);

    if (connect(sfd, (struct sockaddr *) &addr, sizeof(struct sockaddr_un)) == -1)
        errExit("connect");

    /* Copy stdin to socket */

    while ((numRead = read(STDIN_FILENO, buf, BUF_SIZE)) > 0)
        if (write(sfd, buf, numRead) != numRead)
            fatal("partial/failed write");

    if (numRead == -1)
        errExit("read");

    exit(EXIT_SUCCESS);                         /* Closes our socket; server sees EOF
*/
}
```

us_xfr_sv.c

1. 在后台运行服务器，查看生成的临时文件

./us_xfr_sv.elf > b &

ll -F /tmp/us_xfr

```
root@VM-16-16-centos ~/C_CODE/IPC/us # ./us_xfr_sv.elf > b &
[1] 721807
root@VM-16-16-centos ~/C_CODE/IPC/us # ll -F /tmp/us_xfr
srwxr-xr-x 1 root root 0 May  6 23:20 /tmp/us_xfr=
```

2. 创建一个客户端，将测试文件输入

cat *.c > a

./us_xfr_cl.elf < a

```
root@VM-16-16-centos ~/C_CODE/IPC/us # cat *.c > a
root@VM-16-16-centos ~/C_CODE/IPC/us # ./us_xfr_cl.elf < a
```

3. 终止服务器并检查服务器的输出是否与客户端的输入匹配。

kill %1

diff a b

```
root@VM-16-16-centos ~/C_CODE/IPC/us # kill %1
root@VM-16-16-centos ~/C_CODE/IPC/us #
[1]+  Terminated              ./us_xfr_sv.elf > b
root@VM-16-16-centos ~/C_CODE/IPC/us # diff a b
root@VM-16-16-centos ~/C_CODE/IPC/us #
```

diff 没有输出，说明二者匹配。