**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

# 文件系统

## 详述Linux 的节点 inode

### 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

### 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。

### 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。

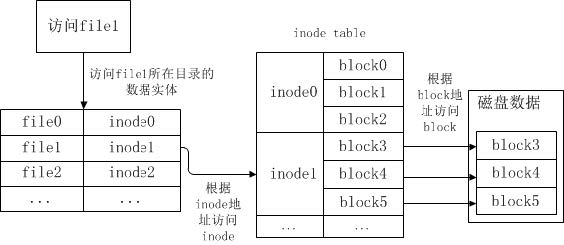
事实上，Linux中的FCB（文件控制块）只包含文件名和inode编号，二者称作基本目录项。用当户通过文件名打开文件时，Linux系统会进行3个步骤：

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

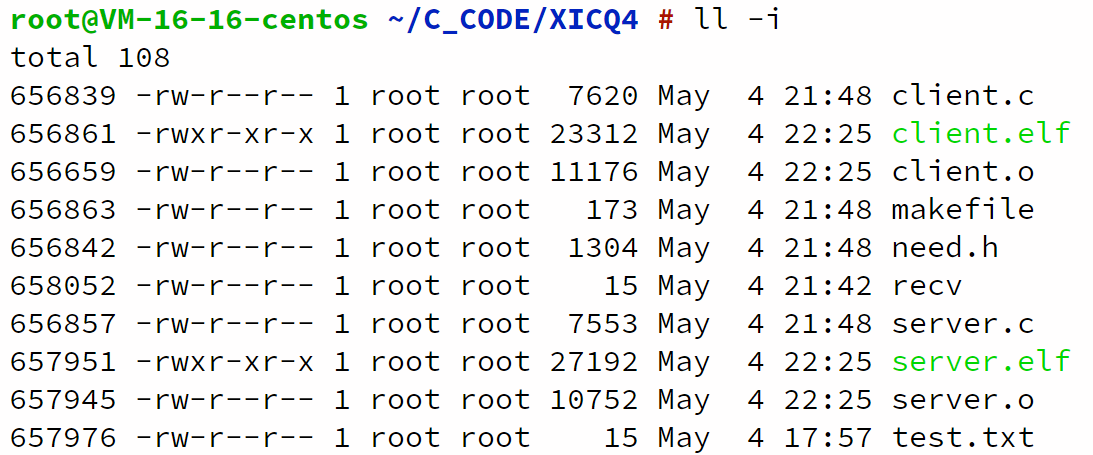
3. 根据inode信息，找到文件数据所在的block，读出数据。

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

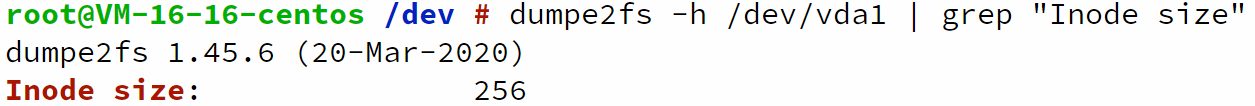


### inode实例的大小

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"

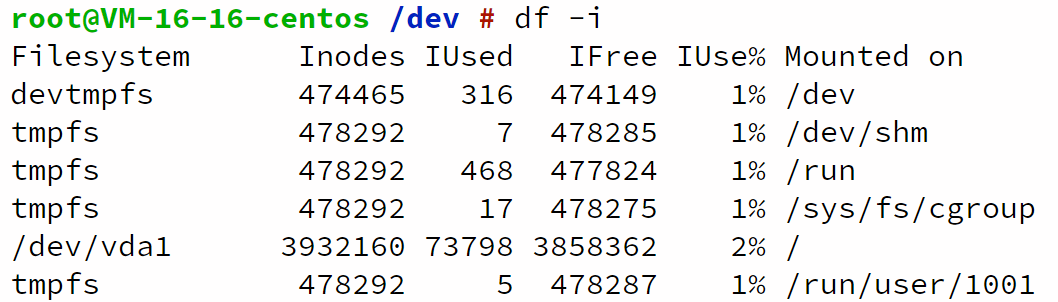
命令来查看每个inode结点的大小。



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



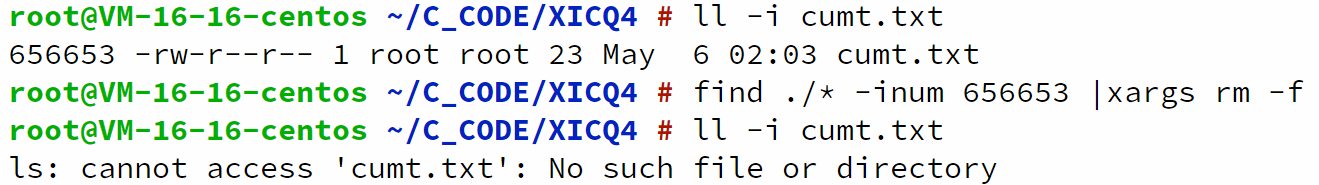
### inode机制的特有现象

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

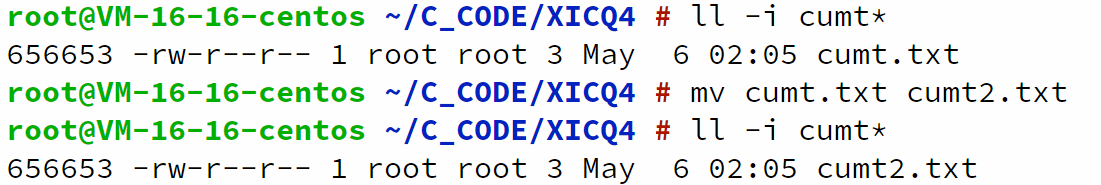
find ./\* -inum 656653 -delete 或

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



2. 移动文件或重命名文件不影响inode号码。

mv cumt.txt 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。

## 详述硬链接与软链接

### 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）。

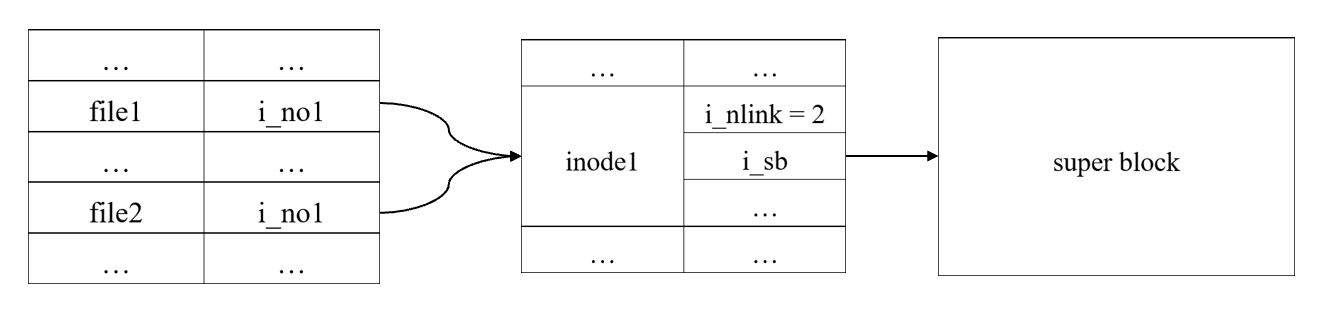
创建链接文件的命令如下：

硬链接：ln file1 file2

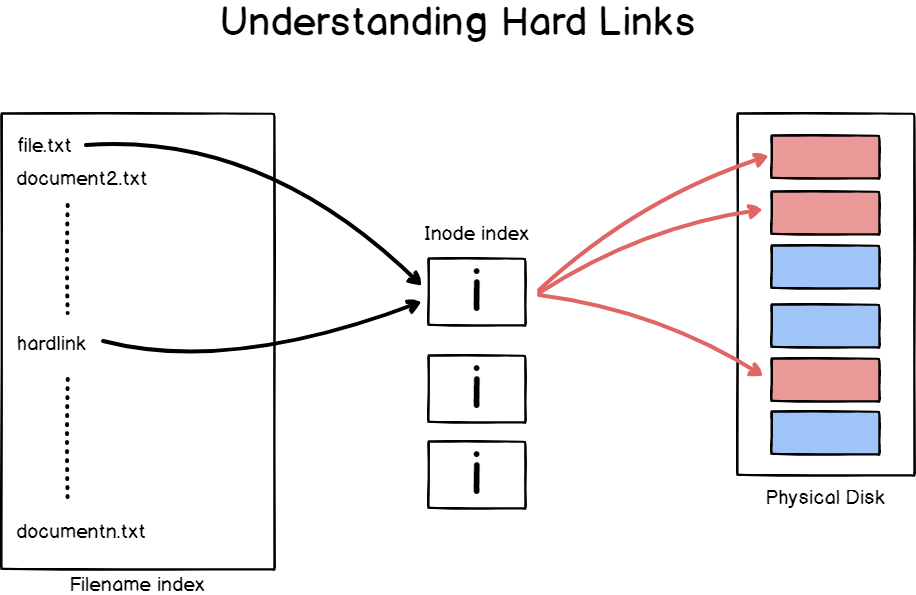
软链接：ln -s file1 file2

### 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。



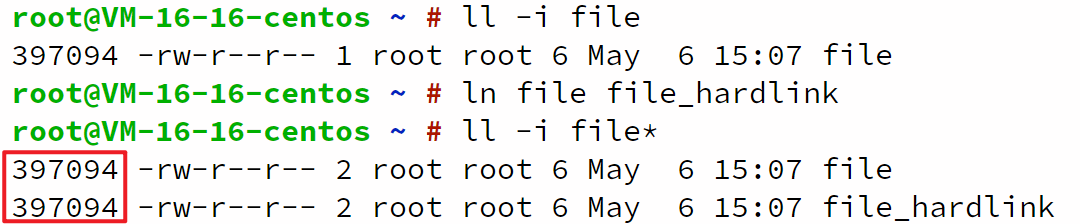
更简洁的表述如下：



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

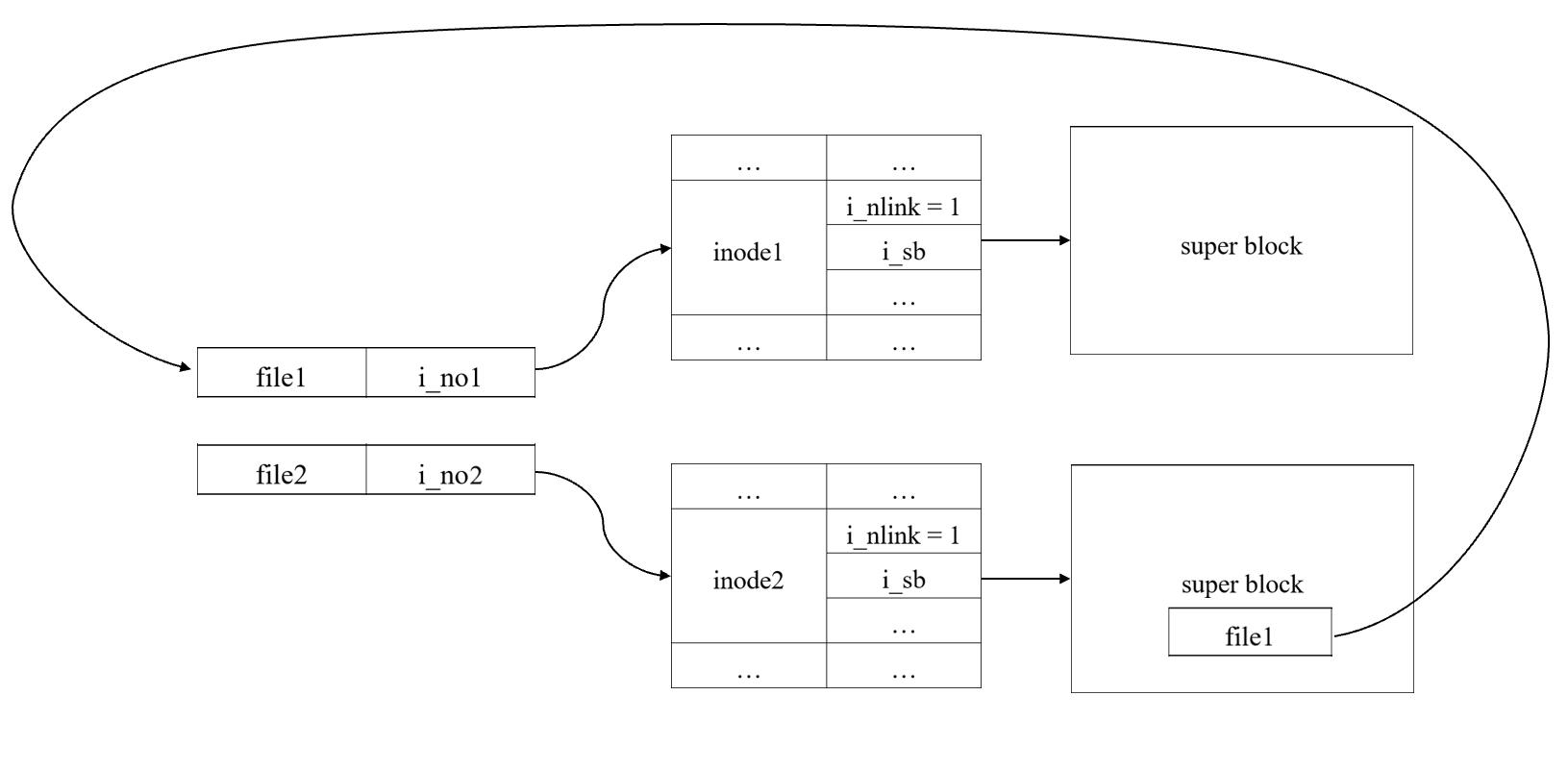
ln file file\_hardlink



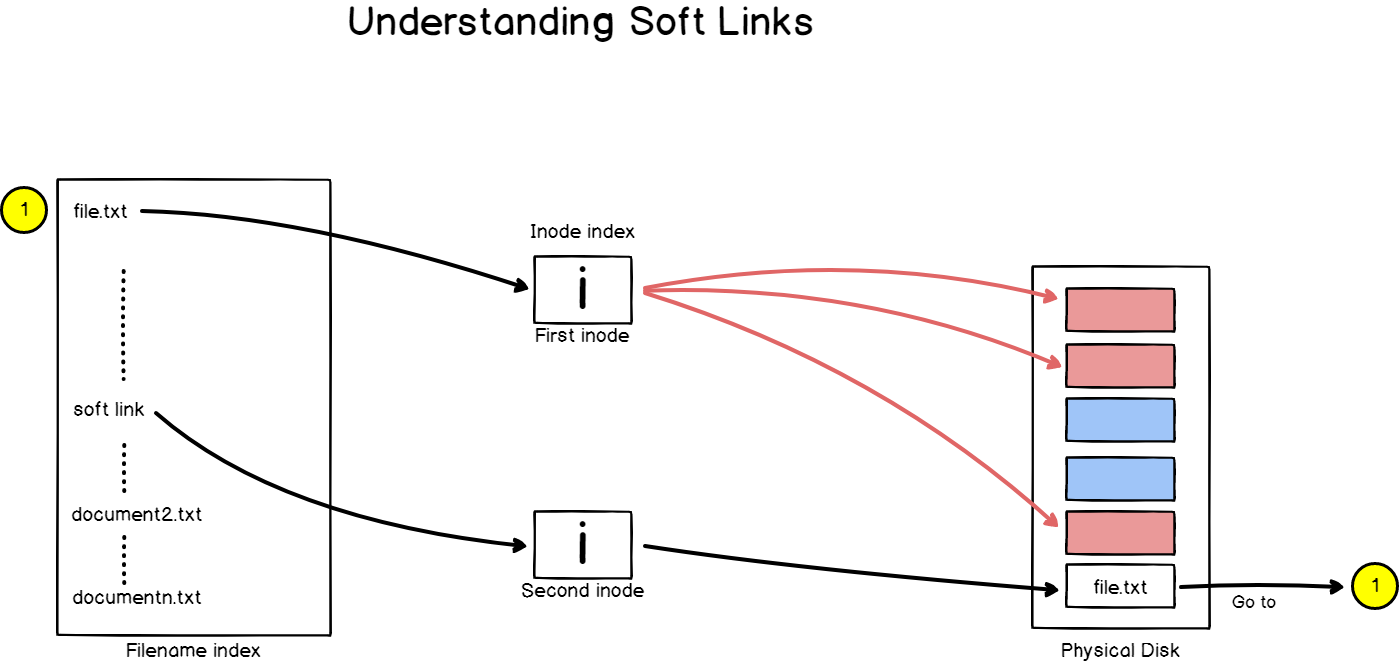
file和file\_hardlink的i\_no确实相同。

### 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。

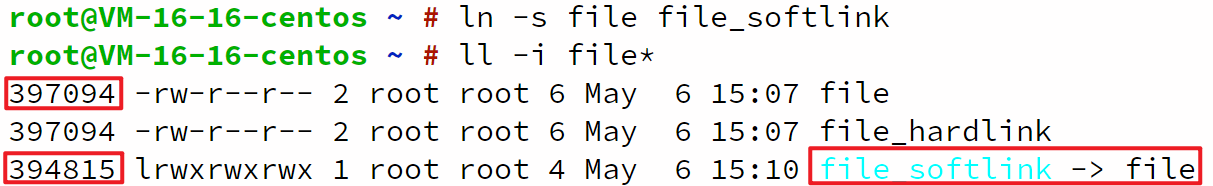


更简洁的表述如下：



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。

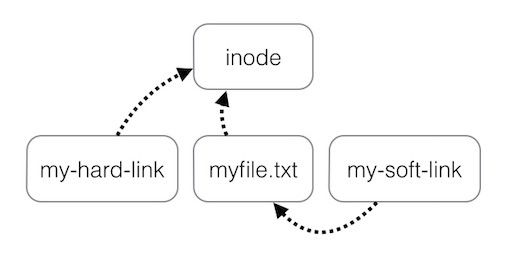
实际测试一下



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。

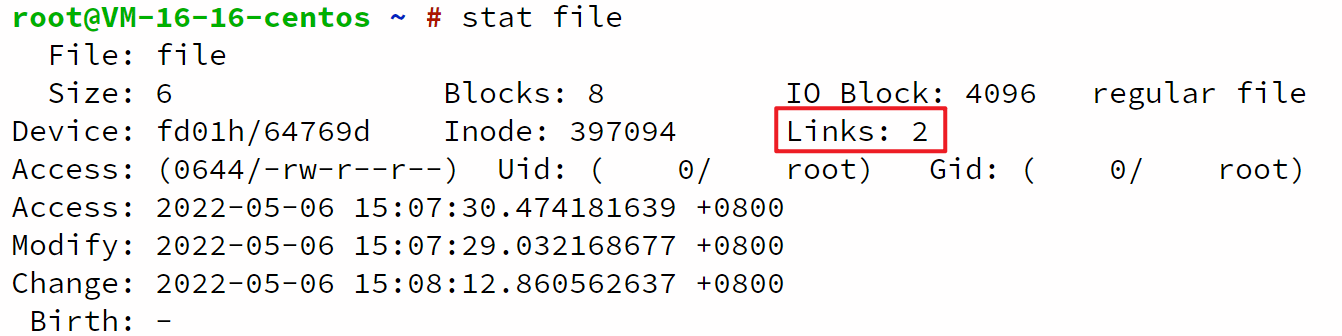
### 联系与区别

大体上，他们的关系如下：

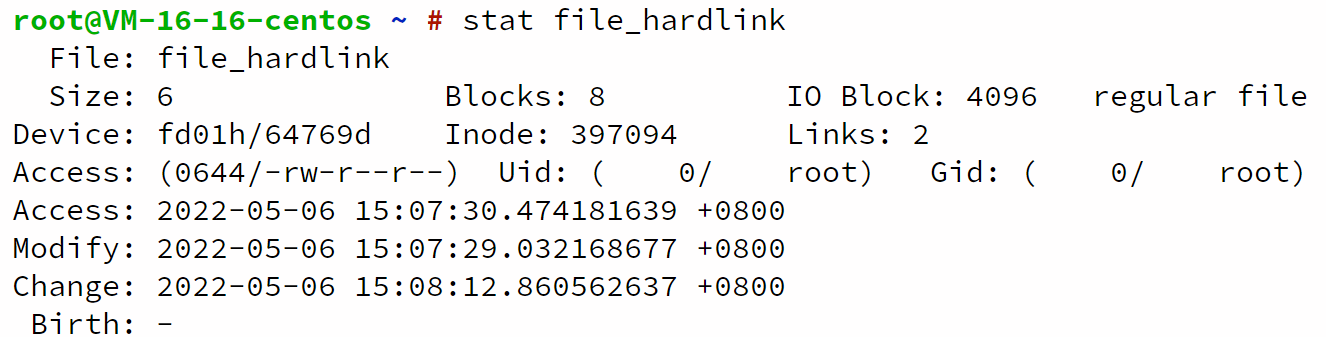


1. soft link不增加i\_nlink（硬链接计数）

对file分别建立硬链接和软链接后，使用stat命令，发现Links为2。



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。



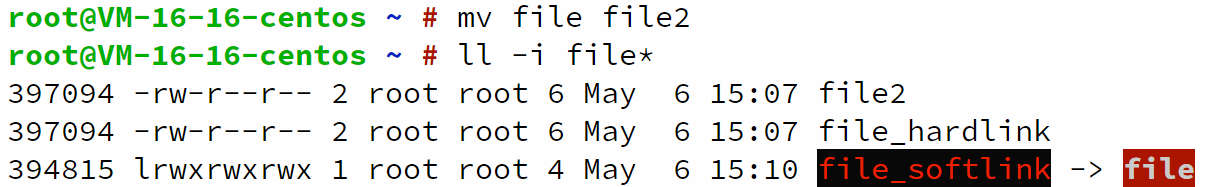
2. 硬链接不能跨越分区，但软链接可以。这个原因很简单，因为i\_no在不同分区是重新计数的，同一i\_no不指向同一inode；

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



可以看到ls用醒目的红色提醒我们这个软链接失效了。

# 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。

UNIX 系统上有各种通信和同步工具，根据功能，它们分为三类：

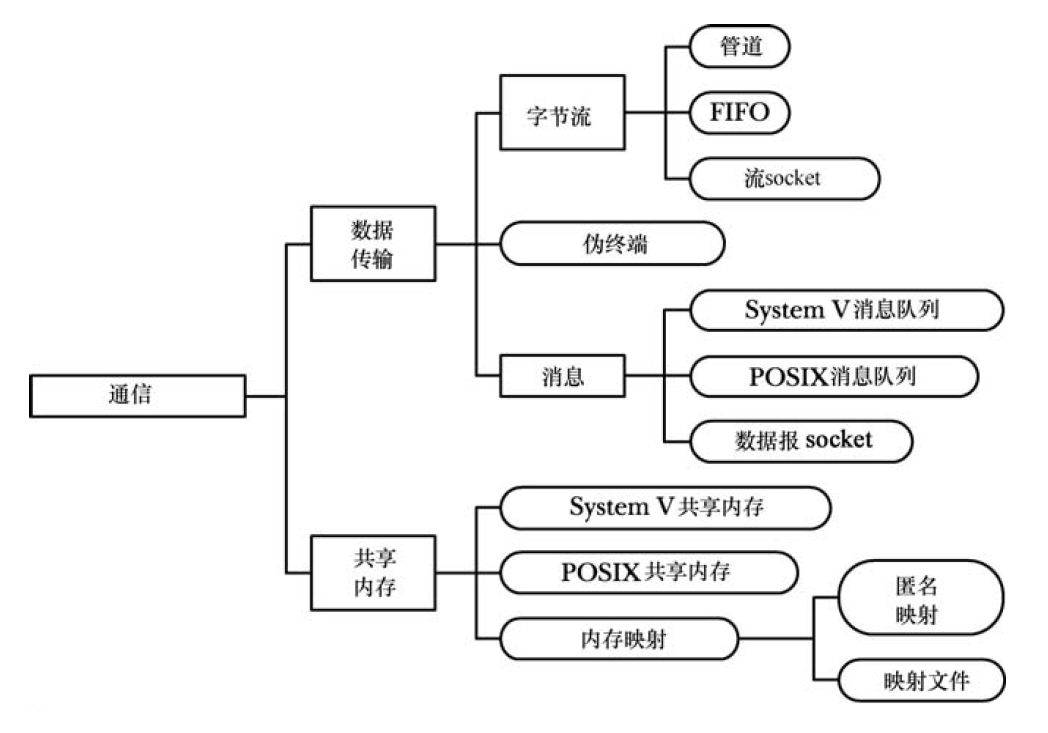
通信：这些工具关注进程之间的数据交换。

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。

### 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



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**]);**

#### **头文件**

#include <unistd.h>

#### 原型

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

#### 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。

#### 返回值

运行成功：返回0

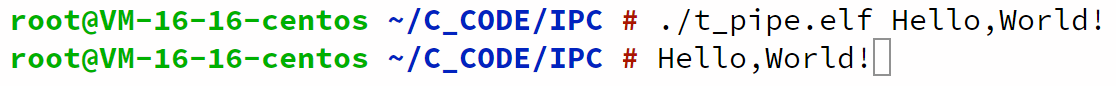
运行失败：返回-1

#### 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>  #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



“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。

### 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**);**

#### **头文件**

#include <sys/stat.h>

#### 原型

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

#### 参数

pathname：创建一个名为pathname的FIFO特殊文件

mode：指定FIFO的权限

#### 返回值

运行成功：返回0

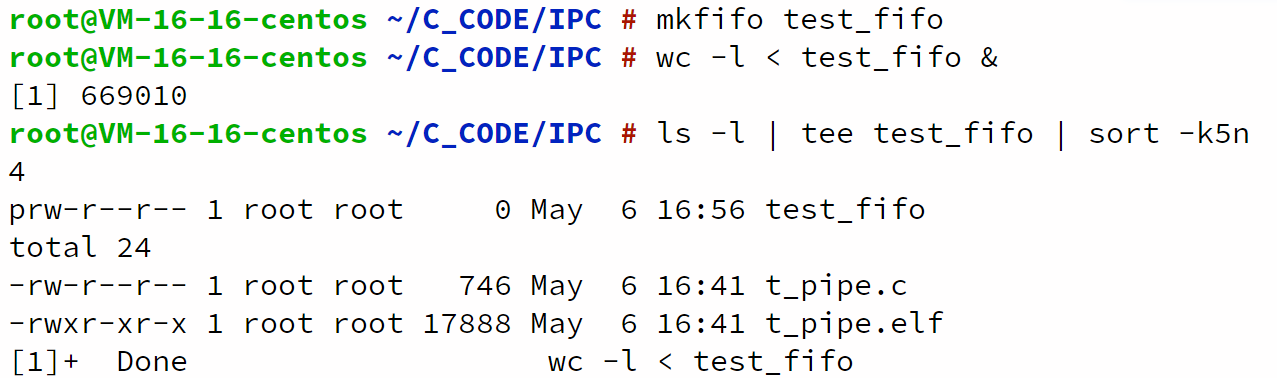
运行失败：返回-1

#### 实例

mkfifo test\_fifo

wc -l < test\_fifo &

ls -l | tee test\_fifo | sort -k5n



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。

### 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);

#### 头文件

#include <sys/types.h>

#include <sys/msg.h>

#### 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);

#### 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

#### 返回值

全部为：

运行成功：返回0

运行失败：返回-1

#### 实例

创建如下三个程序。

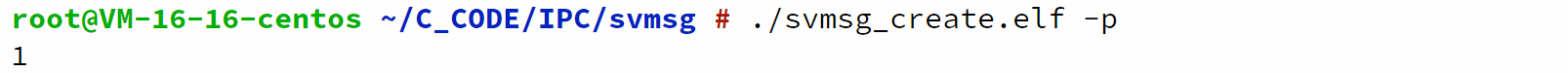
|  |  |
| --- | --- |
|  | svmsg\_create.c |
|  |
| /\* svmsg\_create.c  Experiment with the use of msgget() to create a System V message queue.  \*/  #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h>  #include <sys/stat.h>  #include "tlpi\_hdr.h"  static void /\* Print usage info, then 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。

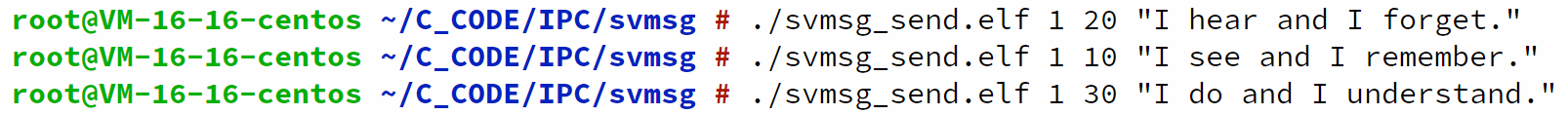
./svmsg\_create.elf -p



./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."

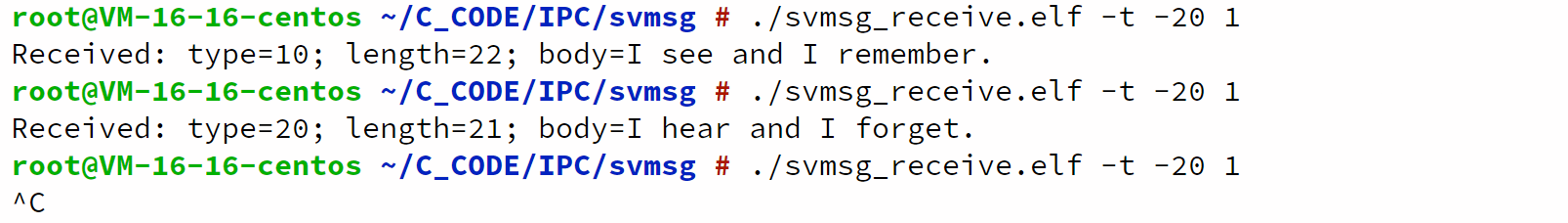


接着使用svmsg\_receive从队列中读取类型小于或等于20 的消息。

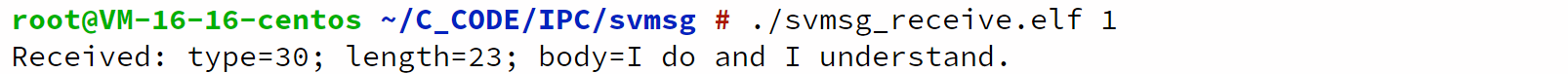
./svmsg\_receive.elf -t -20 1

./svmsg\_receive.elf -t -20 1

./svmsg\_receive.elf -t -20 1



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。



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。

### 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;

#### 头文件

#include <sys/types.h>

#include <sys/sem.h>

#### 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);

#### 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。

#### 返回值

全部为：

运行成功：返回0

运行失败：返回-1

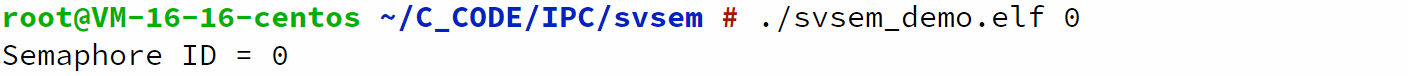
#### 实例

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



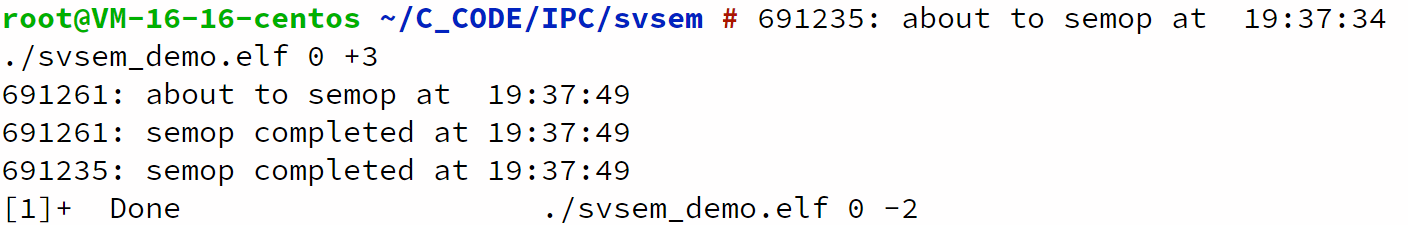
执行一个后台命令（&）将信号量值-2

./svsem\_demo.elf 0 -2 &



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



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。

### 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;

#### 头文件

#include <sys/types.h>

#include <sys/shm.h>

#### 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);

#### 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。

#### 返回值

全部为：

运行成功：返回0

运行失败：返回-1

#### 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。

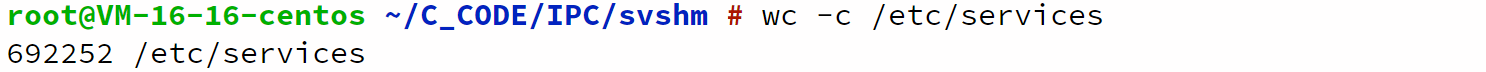
-释放（增加）写者的信号量。

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 |
|  |

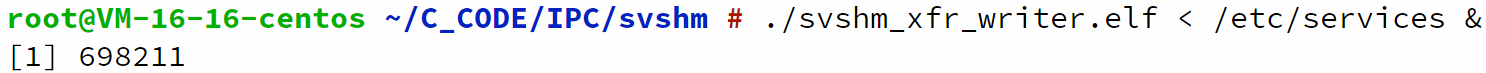
1. 打印文件的大小（单位为byte）

wc -c /etc/services



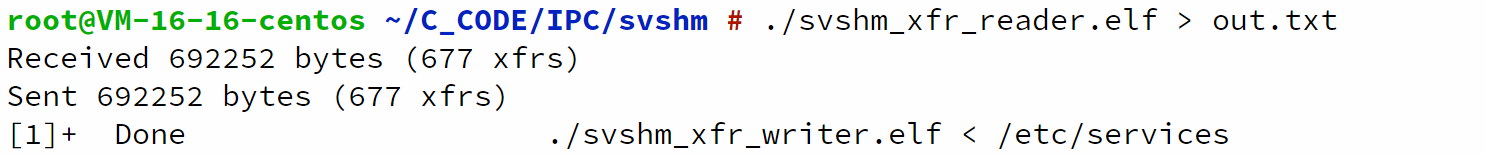
2. 调用写者，将文件/etc/services 作为输入

./svshm\_xfr\_writer.elf < /etc/services &



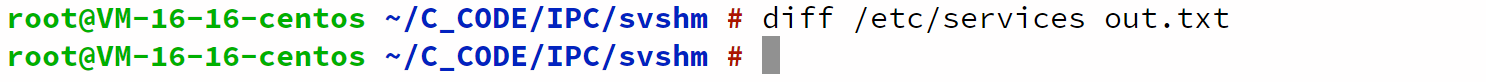
3. 调用读者，将内容输出定向到另一个文件中。

./svshm\_xfr\_reader.elf > out.txt



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



### 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

#### 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。

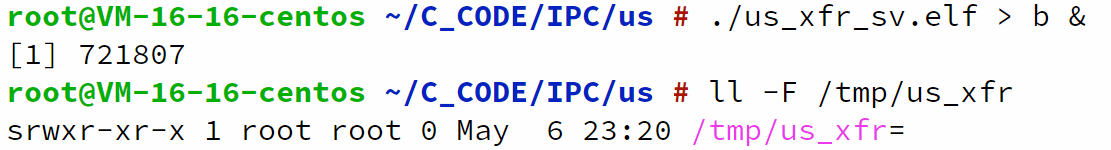
3. 执行一个循环将其标准输入复制到socket连接上。当遇到标准输入中的文件结尾时客户端就终止，其结果是客户端socket将会被关闭并且服务器在从连接的另一端的socket中读取数据时会看到文件结束。

|  |  |
| --- | --- |
|  | 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 &

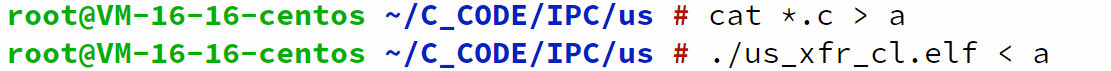
ll -F /tmp/us\_xfr



2. 创建一个客户端，将测试文件输入

cat \*.c > a

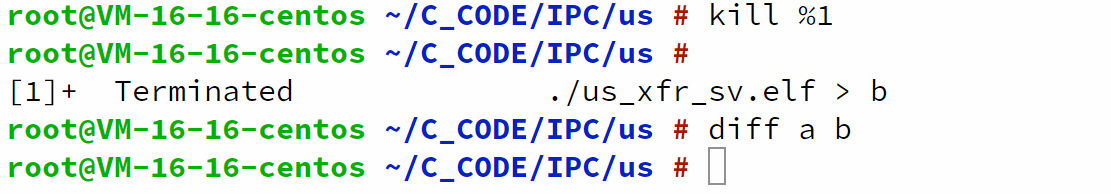
./us\_xfr\_cl.elf < a



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



diff没有输出，说明二者匹配。