第十五章 进程间通信

冯 晶 E-mail: gfeng@whu.edu.cn

1. UNIX系统IPC摘要

IPC 类型	SUS	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
半双工管道	•	（全）	•	•	（全）
FIFO	•	•	•	•	•
全双工管道	允许	•、UDS	UDS	UDS	•、UDS
命名全双工管道	废弃的	UDS	UDS	UDS	•、UDS
XSI 消息队列	XSI	•	•	•	•
XSI 信号量	XSI	•	•	•	•
XSI 共享存储	XSI	•	•	•	•
消息队列（实时）	MSG 选项	•	•		•
信号量	•	•	•	•	•
共享存储（实时）	SHM 选项	•	•	•	•
套接字	•	•	•	•	•
STREAMS	废弃的				•

2. 管道

- 管道是UNIX IPC的最老形式，所有的UNIX系统都支持此种通讯机制

- 管道有两种限制：

  （1）它们是半双工的，数据只能在一个方向上流动；

  （2） 它们只能在具有公共祖先的进程之间使用。通常，一个管道由一个进程创建，然后该进程调用fork()函数，此后父进程和子进程之间就可以使用该管道。

2. 管道

- 管道的创建
  - 进程使用fork()创建了子进程以后，父子进程就有各自独立的存储空间，互不影响。两个进程之间交换数据就不可能像进程内的函数调用那样，通过传递参数或者使用全局变量实现，必须通过其他的方式。

    例如：父子进程共同访问同一个磁盘文件交换数据，但是这很不方便。

  - UNIX提供了很多种进程之间通信的手段。管道是一种很简单的进程间通信方式，最早的UNIX中就提供管道机制。

2. 管道

- 管道的创建
  - 使用管道的基本方法是创建一个内核中的管道对象，进程可以得到两个文件描述符。
  - 程序像访问文件一样地访问管道，write()将数据写入管道，read()从管道中读出写入的内容。
  - 读入的顺序和写入的顺序相同，看起来管道就像是一个仅有一个字节粗细的管子一样传递数据流。
  - 在内核的活动文件目录的三级结构中，管道的两个文件描述符和普通文件一样，只是内核的inode中记文件类型为特殊文件：“管道文件”，操作系统在内核中实现管道机制。

2. 管道

- 管道的创建

```
#include <unistd.h>

int pipe(int filedes[2])        成功返回0; 出错返回-1
```

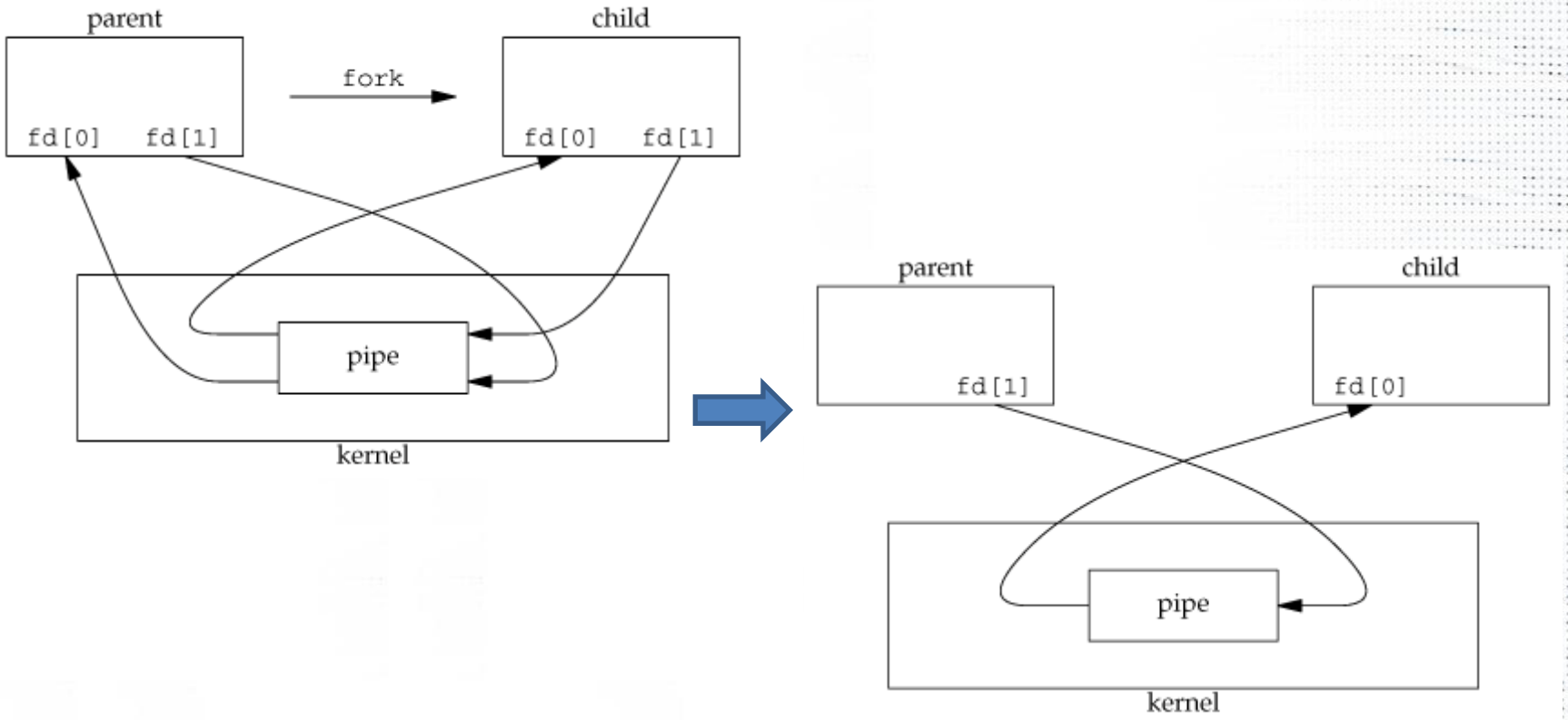
  - filedes[0]用于读管道，filedes[1]用于写管道。这样，如果进程向filedes[1]写入数据，那么就会从filedes[0]顺序读出来。
  - 如果仅有一个进程，创建一个这样的管道交换进程内的数据没什么意义。

2. 管道

- 管道的创建
  - 使用fork()创建子进程之后，文件描述符被继承。这样，父进程从filedes[1]写入的数据，子进程就可以从filedes[0]读出，从而实现父子进程之间的通信。
  - 如果父进程创建两个子进程，都继承管道的文件描述符，两个子进程间也可交换数据。
  - 一般的，有共同祖先的进程就有机会从同一个祖先继承这个祖先创建的管道的文件描述符，从而互相间通过管道通信。
  - fork以后，父子进程可关闭不再需要的文件描述符。

2. 管道

- 管道的创建


parent
child
fork
fd[0]
fd[1]
fd[0]
fd[1]
pipe
kernel
parent
child
fd[1]
fd[0]
pipe
kernel

2. 管道

- 管道的读写操作
  - 对于写操作write()来说，由于管道是内核中的一个缓冲区，缓冲区不可能无限大。若管道已满，则write()操作就会导致进程被阻塞，直到管道另一端read()将已进入管道的数据取走后，内核才把阻塞在write()的写端进程唤醒。
- 读操作的三种情况：
  - 第一种情况，管道为空，则read调用就会将进程阻塞，而不是返回0。
  - 第二种情况，管道不为空，返回读取的内容。
  - 第三种情况，管道写端已关闭，则返回0。

2. 管道

- 管道的读写操作
  - 两个独立的进程对管道的读写操作，如果未写之前，读先行一步，那么，操作系统内核在系统调用read()中让读端进程睡眠，等待写端送来数据。
  - 同样，如果写端的数据太多或者写得太快，读端来不及读，管道满了之后操作系统内核就会在系统调用write()中让写端进程睡眠，等待读端读走数据。
  - 这种同步机制，在读写速度不匹配时不会丢失数据。

2. 管道

- 管道的关闭
  - 只有所有进程中引用管道写端的文件描述符都关闭了，读端read()调用才返回0。
  - 关闭读端，不再有任何进程读，则导致写端write()调用返回–1，终止调用write()的进程。

2. 管道

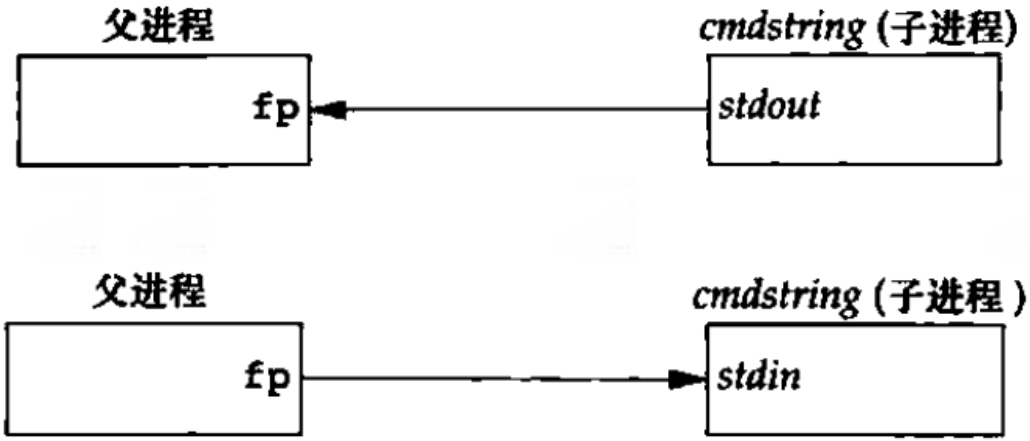
- 两个进程通过管道传送数据的例子：
  - 程序文件pwrite.c创建管道，并通过管道向程序pread.c传递数据。

2. 管道

- 应注意的问题
  - 管道传输的是一个无记录边界的字节流。写端的一次write所发送的数据，读端可能需要多次read才能读取；也有可能写端的多次write所发送的数据，读端一次全部读出积压在管道中的所有数据。
  - 父子进程需要双向通信时，应采用两个管道。
  - 父子进程使用两个管道传递数据，安排不当就有可能产生死锁。
  - 管道的缺点。管道是半双工通信通道，数据只能在一个方向上流动；管道通信只限于父子进程或者同祖先的进程间通信，而且没有保留记录边界。

3. 函数popen和pclose


```
#include <stdio.h>

FILE *popen(const char *cmdstring, const char *type);
int pclose(FILE *fp);
```



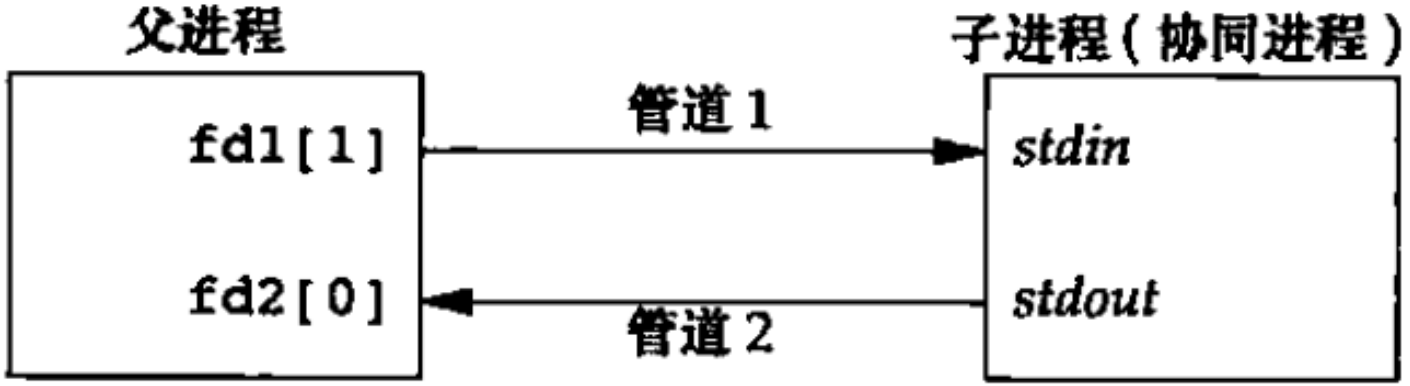
3. 函数popen和pclose


```
#include <stdio.h>

FILE  *popen (const  char *cmdstring,  const  char *type);
int  pclose(FILE  *fp);
```

```
fp  =  popen ( "ls  *.c ", "r");

fp  = popen ( "cmd  2>&1", "r");
```

4. 协同进程

- UNIX系统过滤程序从标准输入读取数据，向标准输出写数据，几个过滤程序通常在shell管道中线性连接。
- 当一个过滤程序即产生某个过滤程序的输入，又读取该过滤程序的输出时，它就变成了协同进程



5. FIFO

- 命名管道最早出现在UNIX System III，允许没有共同祖先的不相干进程访问一个FIFO管道。

- 命名管道的创建

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

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

成功返回0; 出错返回-1

5. FIFO

- 命名管道的使用

  一旦用mkfifo()创建了一个FIFO，就可以像一般的文件一样对其进行I/O操作

  - 发送者调用：

    ```
    fd = open("pipename", O_WRONLY);
    write(fd, buf, len);
    ```

  - 接收者调用：

    ```
    fd = open("pipename", O_RDONLY);
    len = read(fd, buf, sizeof buf);
    ```

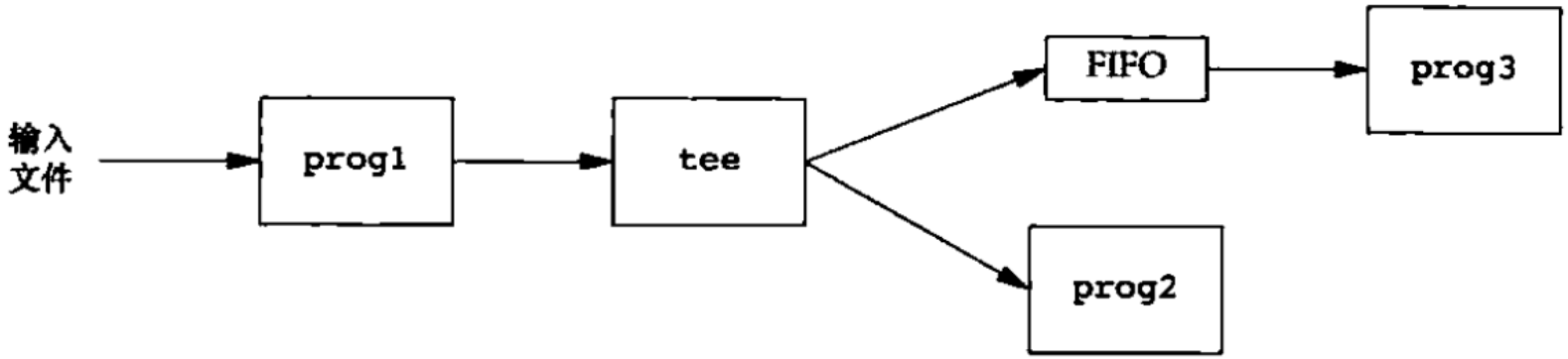
5. FIFO

- 命名管道的用途：
  - FIFO由shell命令使用以便将数据从一条管道线传送到另一条，为此无需创建中间临时文件；
  - FIFO用于客户机-服务器应用程序中，以便在客户机和服务器之间传输数据。

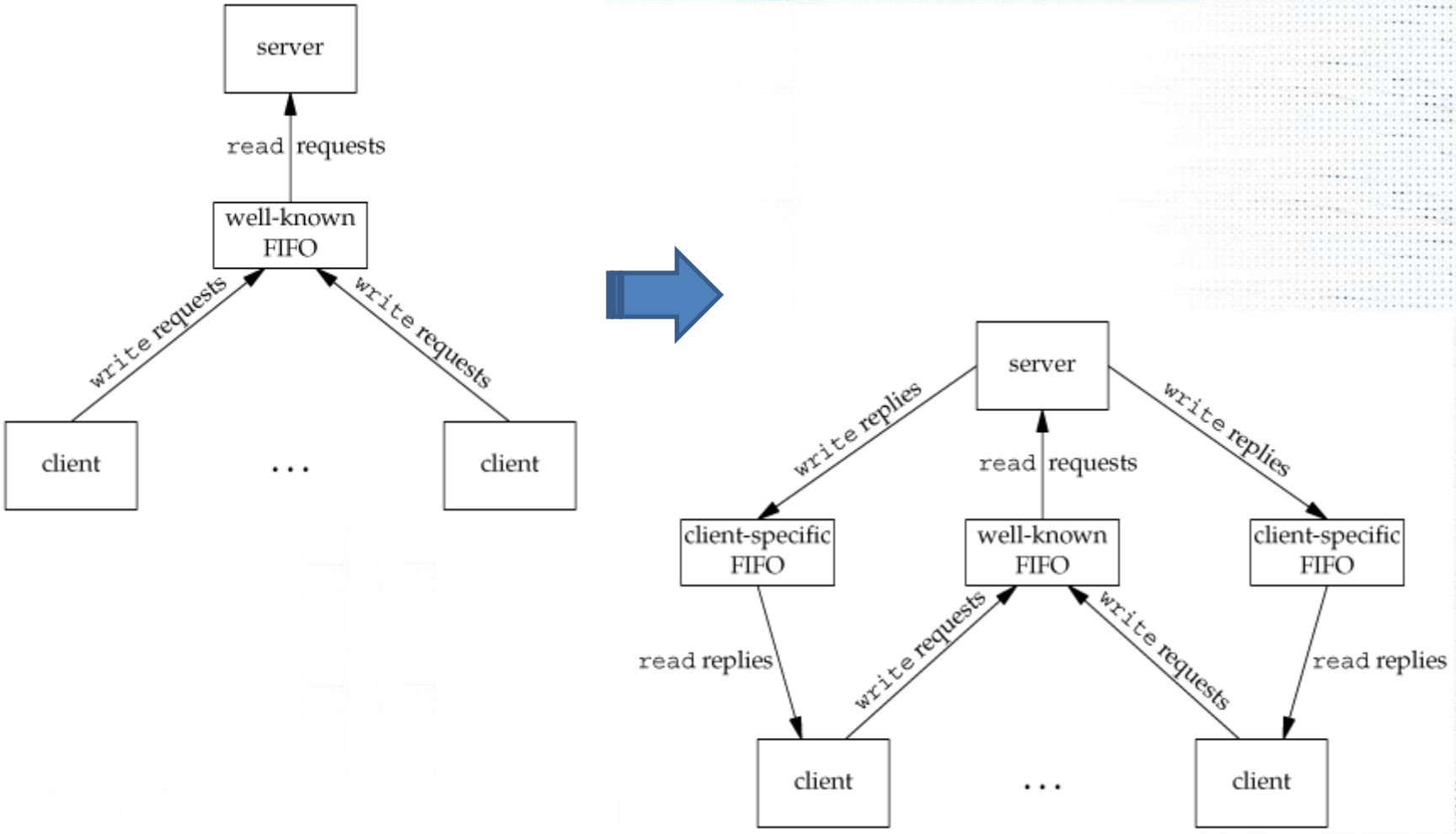
5. FIFO

- 用FIFO复制输出流

```
mkfifo fifo1
prog3 < fifo1 &
prog1 < infile | tee fifo1 | prog2
```



5. FIFO


server
read requests
well-known FIFO
write requests
write requests
client
. . .
client
server
write replies
read requests
write replies
client-specific FIFO
well-known FIFO
client-specific FIFO
read replies
write requests
write requests
read replies
client
. . .
client

6. XSI IPC

- 消息队列
- 信号量
- 共享存储器

6. XSI IPC

- 标识符和键
  - 标识符：非负整数，IPC对象的内部名
  - 键：与每个IPC对象关联，IPC对象的外部名
- 客户进程与服务器进程的通信
  - 服务器进程指定IPC_PRIVATE创建一个新的IPC结构，将返回标识存放在某处以便客户进程取用
  - 在公用头文件中定义一个客户进程和服务器进程都认可的键
  - 客户进程和服务器进程认同一个路径名和项目ID，调用fotk将两个值变换为一个键

6. XSI IPC


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

key_t  ftok (const  char *path,  int  id);
```

6. XSI IPC

- 权限结构

```
struct ipc_perm {
  uid_t  uid;  /* owner's effective user id */
  gid_t  gid;  /* owner's effective group id */
  uid_t  cuid; /* creator's effective user id */
  gid_t  cgid; /* creator's effective group id */
  mode_t mode; /* access modes */
  :
  :
};
```

6. XSI IPC

- IPC特征比较

IPC 类型	无连接?	可靠的?	流控制?	记录?	消息类型或优先级?
消息队列	否	是	是	是	是
STREAMS	否	是	是	是	是
UNIX 域流套接字	否	是	是	否	否
UNIX 域数据报套接字	是	是	否	是	否
FIFO（非 STREAMS）	否	是	是	否	否

7. 消息队列

- 消息队列是消息的链接表，存储在内核中，有消息队列标识符标识。

```
#include <sys/msg.h>
int  msgget (key_t  key,  int  flag);
```

说明	典型值			
	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
可发送的最长消息的字节数	16 384	8 192	不支持	2 048
一个特定队列的最大字节数（亦即队列中所有消息长度之和）	2 048	16 384	不支持	4 096
系统中最大消息队列数	40	16	不支持	50
系统中最大消息数	40	导出的	不支持	40

7. 消息队列

- 消息队列是消息的链接表，存储在内核中，有消息队列标识符标识。

```
struct msqid_ds {
  struct ipc_perm      msg_perm;      /* see Section 15.6.2 */
  msgqnum_t            msg_qnum;      /* # of messages on queue */
  msglen_t             msg_qbytes;    /* max # of bytes on queue */
  pid_t                msg_lspid;     /* pid of last msgsnd() */
  pid_t                msg_lrpid;     /* pid of last msgrcv() */
  time_t               msg_stime;     /* last-msgsnd() time */
  time_t               msg_rtime;     /* last-msgrcv() time */
  time_t               msg_ctime;     /* last-change time */
   :
};
```

7. 消息队列

- 消息队列是消息的链接表，存储在内核中，有消息队列标识符标识。

```
#include <sys/msg.h>
int msgctl (int msqid, int cmd, struct msqid_ds *buf);
```

IPC_STAT
IPC_SET
PIC_RMID

7. 消息队列

- 消息队列是消息的链接表，存储在内核中，有消息队列标识符标识。

```
#include <sys/msg.h>
int msgsnd(int msqid, const void *ptr, size_t nbytes,
           int flag);
```

IPC_NOWAIT

```
struct mymesg {
  long mtype;       /* positive message type */
  char mtext[512];  /* message data, of length nbytes */
};
```

7. 消息队列

- 消息队列是消息的链接表，存储在内核中，有消息队列标识符标识。

```
#include <sys/msg.h>
int msgrcv (int msqid, void *ptr, size_t nbytes,
            long type, int flag);
```

type==0：返回队列中的第一个消息
type>0：返回队列中的消息类型为type的第一个消息
type<0：返回队列中消息类型值小于等于type绝对值的消息

8. 信号量

- 信号量是一个计数器，用于为多个进程提供对共享数据对象的访问。
- 内核为每个信号集合维护的semid_ds结构

```
struct semid_ds {
  struct ipc_perm  sem_perm;    /* see Section 15.6.2 */
  unsigned short   sem_nsems;   /* # of semaphores in set */
  time_t           sem_otime;   /* last-semop() time */
  time_t           sem_ctime;   /* last-change time */
   :
};
```

8. 信号量

- 信号量是一个计数器，用于为多个进程提供对共享数据对象的访问。
- 每个信号量的结构

```
struct {
  unsigned short  semval;      /* semaphore value, always >= 0 */
  pid_t           sempid;      /* pid for last operation */
  unsigned short  semncnt;     /* # processes awaiting semval>curval */
  unsigned short  semzcnt;     /* # processes awaiting semval==0 */
   :
};
```

8. 信号量

- 信号量是一个计数器，用于为多个进程提供对共享数据对象的访问。

说明	典型值			
	FreeBSD 8.0	Linux 3. 2. 0	Mac OS X 10. 6.8	Solaris 10
任一信号量的最大值	32 767	32 767	32 767	65 535
任一信号量的最大退出时的调整值	16 384	32 767	16 384	32 767
系统中信号量集的最大数量	10	128	87 381	128
系统中信号量的最大数量	60	32 000	87 381	导出的
每个信号量集中的信号量的最大数量	60	250	87 381	512
系统中 undo 结构的最大数量	30	32 000	87 381	导出的
每个 undo 结构中 undo 项的最大数量	10	无限制	10	导出的
每个 semop 调用中操作的最大数量	100	32	5	512

8. 信号量

- 信号量是一个计数器，用于为多个进程提供对共享数据对象的访问。

```
#include <sys/sem.h>
int semget (key_t key, int nsems, int flag);
```

nsems==0：引用现有信号量集合
nsems==n：创建新集合

8. 信号量

- 信号量是一个计数器，用于为多个进程提供对共享数据对象的访问。

```
#include <sys/sem.h>
int semctl (int semid, int semnum, int cmd, ...);
```

0 – (semnum-1)

```
union semun {
        int             val;
        struct semid_ds     *buf;
        unsigned short      *array;
}
```

8. 信号量

- 信号量是一个计数器，用于为多个进程提供对共享数据对象的访问。

```
#include <sys/sem.h>
int semop (int semid, struct sembuf semoparray[],
          size_t nops);
```

```
struct sembuf {
        unsigned  short sem_num;    // 0 - (nsems-1)
        short     sem_op;           // negative, 0, pasitive
        short     sem_flg;          // IPC_NOWAIT, SEM_UNDO
}
```

8. 信号量

- 信号量是一个计数器，用于为多个进程提供对共享数据对象的访问。

操作	用户	系统	时钟
带 undo 的信号量	0.50	6.08	7.55
建议性记录锁	0.51	9.06	4.38
共享存储中的互斥量	0.21	0.40	0.25

9. 共享存储

- 共享存储允许两个或多个进程共享一个给定的存储区，因为不需要再不同进程间复制数据，所以是最快的一种IPC。

```
struct shmid_ds {
  struct ipc_perm shm_perm;    /* see Section 15.6.2 */
  size_t          shm_segsz;   /* size of segment in bytes */
  pid_t           shm_lpid;    /* pid of last shmop() */
  pid_t           shm_cpid;    /* pid of creator */
  shmatt_t        shm_nattch;  /* number of current attaches */
  time_t          shm_atime;   /* last-attach time */
  time_t          shm_dtime;   /* last-detach time */
  time_t          shm_ctime;   /* last-change time */
   :
};
```

9. 共享存储

- 共享存储允许两个或多个进程共享一个给定的存储区，因为不需要再不同进程间复制数据，所以是最快的一种IPC。

```
#include <sys/shm.h>
int shmget(key_t key, size_t size, int flag);
```

9. 共享存储

- 对共享存储段执行多种操作。

```
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

9. 共享存储

- 连接共享存储到进程自己的地址空间。

```
#include <sys/shm.h>
int shmat(int shmid, const void *addr, int flag);
```

9. 共享存储

- 连接共享存储到进程自己的地址空间。

```
#include <sys/shm.h>
int shmdt(const void *addr);
```

10. POSIX信号量

- POSIX信号量接口解决了XSI信号量接口的几个缺陷
  - POSIX信号量接口有更高性能的实现
  - POSIX信号量接口更简单：没有信号量集，类似于文件系统的操作
  - POSIX信号量在删除时表现更完美，信号量被删除时，不会立即导致信号量使用者报错，能够继续正常工作直到该信号量的最后一次引用被释放
- POSIX信号量形式
  - 未命名信号量：存在于内存中，只能在同一进程的线程使用或不同进程中已映射相同共享存储的线程使用

10. POSIX信号量

- 创建一个新的命名信号量或者使用一个现有信号量。

```
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag,
                /* mode_t mode, */
                /* unsigned int value */);
```

10. POSIX信号量

- 释放任何信号量相关的资源
- 释放命名信号量

```
#include <semaphore.h>

int  *sem_close (sem_t  *sem);

int  sem_unlink (const  char  *name);
```

10. POSIX信号量

- 信号量减1操作

```
#include <semaphore.h>

int sem_trywait (sem_t *sem);

int sem_wait (sem_t *sem);

int sem_timedwait (sem_t *restrict sem,
        const struct timespec *resctrict tsptr);
```

10. POSIX信号量

- 信号量加1操作

```
#include <semaphore.h>

int  sem_post (sem_t  *sem);
```

10. POSIX信号量

- 创建POSIX未命名信号量

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

10. POSIX信号量

- 丢弃POSIX未命名信号量

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

10. POSIX信号量

- 检索信号量的值

```
#include <semaphore.h>

int sem_getvalue(sem_t restrict *sem, int *restrict valp);
```