4.2 -NTERPROCESS COMMUNICATION

Overview of IPC Methods

1. Pipes
2. Popen and pclose Functions
3. Coprocesses
4. FIFOs
5. System V IPC
6. Message Queues
7. Semaphores

INTRODUCTION

- ✓ IPC enables one application to control another application, and for several applications to share the same data without interfering with one another. IPC is required in all multiprocessing systems, but it is not generally supported by single-process operating systems.
- ✓ The various forms of IPC that are supported on a UNIX system are as follows :

1) Half duplex Pipes
2) FIFO's
3) Full duplex Pipes
4) Named full duplex Pipes
5) Message queues
6) Shared memory
7) Semaphores
8) Sockets
9) STREAMS

- ✓ The first seven forms of IPC are usually restricted to IPC between processes on the same host.
- ✓ The final two i.e. Sockets and STREAMS are the only two that are generally supported for IPC between processes on different hosts.

1. PIPES

- ✓ Pipes are the oldest form of UNIX System IPC. Pipes have two limitations.
- ✓ Historically, they have been half duplex (i.e., data flows in only one direction).
- ✓ Pipes can be used only between processes that have a common ancestor.
- ✓ Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parentand the child.

- ✓ A pipe is created by calling the pipe function.

```
#include <unistd.h>
int pipe(int filedes[2]);
                                Returns: 0 if OK, 1 on error.
```

- ✓ Two file descriptors are returned through the filedes argument: filedes[0] is open for reading, andfiledes[1] is open for writing.
- ✓ The output of filedes[1] is the input for filedes[0].

- ✓ Two ways to picture a half-duplex pipe are shown in Figure 1.
- ✓ The left half of the figure shows the two ends of the pipe connected in a single process. The right half ofthe figure emphasizes that the data in the pipe flows through the kernel.

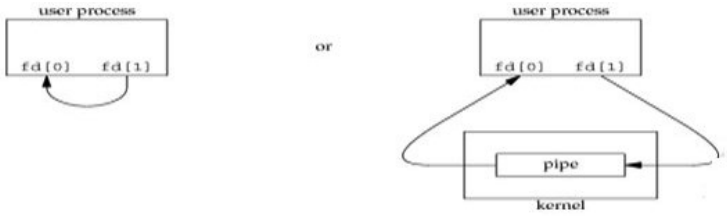


Figure 1. Two ways to view a half-duplex pipe

- ✓ A pipe in a single process is next to useless.
- ✓ Normally, the process that calls pipe then calls fork, creating an IPC channel from the parent to the child orvice versa. Figure 2 shows this scenario.

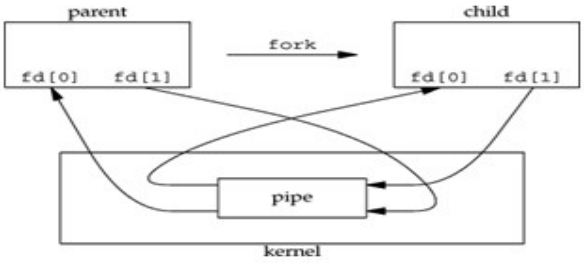


Figure 2 Half-duplex pipe after a fork

- ✓ What happens after the fork depends on which direction of data flow we want.
- ✓ For a pipe from the parent to the child, the parent closes the read end of the pipe (fd[0]), and the child closes the write end (fd[1]). Figure 3 shows the resulting arrangement of descriptors.

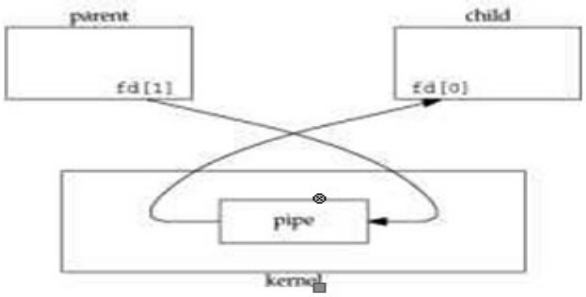


Figure 3 Pipe from parent to child

- ✓ For a pipe from the child to the parent, the parent closes fd[1], and the child closes fd[0]. When one end ofa pipe is closed, the following two rules apply. If we read from a pipe whose write end has been closed,
  - ➢ read returns 0 to indicate an end of file after all the data has been read.
  - ➢ If we write to a pipe whose read end has been closed, the signal SIGPIPE is generated. If we either ignore the signal or catch it and return from the signal handler, write returns 1 with errno set to EPIPE.

PROGRAM: shows the code to create a pipe between a parent and its child and to send data down the pipe.

```
#
i
n
c
l
u
d
e

"
a
p
u
e
.
h
"

i
n
t

m
a
i
n
(
v
o
i
d
)
{
```

```
int     n;
int     fd[2];
pid_t   pid;
char    line[MAXLINE];

i
    f

    (
    p
    i
    p
    e
    (
    f
    d
    )

    <

    0
    )

    e
    r
    r
    _
    s
    y
    s
    (
    "
    p
    i
    p
    e

    e
    r
    r
    o
    r
    "
    )
    ;
i
    f

    (
    (
    p
    i
    d

    =

    f
    o
    r
    k
    (
    )
```

```
    )

    <

    0
    )

    {

    e
    r
    r

    _
    s
    y
    s
    (
    "
    f
    o
    r
    k

    e
    r
    r
    o
    r
    "
    )
    ;
}
    e
    l
    s
    e

    i
    f

    (
    p
    i
    d

    >

    0
    )

    {
                            /*parent */ close(fd[0]);
    write(fd[1], "hello world\n", 12);
} else {                /*
    c
    h
    i
    l
    d

    *
    /
```

```
close(fd[1]);
n = read(fd[0], line, MAXLINE);
write(STDOUT_FILEN
```

```
            o
            ,

            l
            i
            n
            e
            ,

            n
            )
            ;
        }
        exit(0);
}
```

2. popen AND pcloseFUNCTIONS

- ✓ Since a common operation is to create a pipe to another process, to either read its output or send it input, the standard I/O library has historically provided the popen and pclose functions.
- ✓ These two functions handle all the dirty work that we've been doing ourselves: creating a pipe, forking a child, closing the unused ends of the pipe, executing a shell to run the command, and waiting for the command to terminate.

```
#include <stdio.h>

FILE *popen(const char *cmdstring, const char *type);

                                    Returns: file pointer if OK, NULL on error
```

```
int pclose(FILE *fp);

                          Returns: termination status of cmdstring, or 1 on error
```

- ✓ The function popen does a fork and exec to execute the cmdstring, and returns a standard I/O file pointer.

  If type is "r", the file pointer is connected to the standard output of cmdstring

Figure 4 Result of fp = popen(cmdstring, "r")

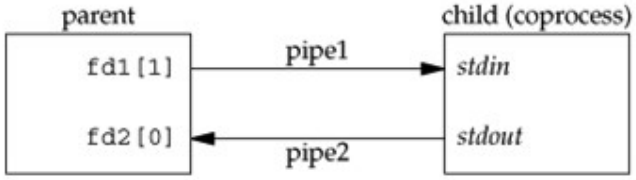
If type is "w", the file pointer is connected to the standard input of cmdstring, as shown:

Figure 5 Result of fp = popen(cmdstring, "w")

3. COPROCESSES

- ✓ A UNIX system filter is a program that reads from standard input and writes to standard output.
- ✓ Filters are normally connected linearly in shell pipelines.
- ✓ A filter becomes a coprocess when the same program generates the filter's input and reads the filter'soutput.
- ✓ A coprocess normally runs in the background from a shell, and its standard input and standard output are connected to another program using a pipe.
- ✓ The process creates two pipes: one is the standard input of the coprocess, and the other is the standard output of the coprocess. Figure 6 shows this arrangement.

Figure 6. Driving a coprocess by writing its standard input and reading its standard output



Program: Simple filter to add two numbers

```
#
i
n
c
l
u
d
e

"
a
p
u
e
.
h
"

i
n
```

```
            =
            strlen(
            line);
            if
                 (write(S
                 TDOUT_FI
                 LENO,
                 line, n)
                 != n)
                 err_sys(
                 "write
                 error");
        } else {
            if (write(STDOUT_FILENO,
                "invalid args\n", 13)
                != 13)err_sys("write
                error");
        }
    }
    exit(0);
}
```

4. FIFOs

- ✓ FIFOs are sometimes called named pipes. Pipes can be used only between related processes when a common ancestor has created the pipe.

```
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
                                                    Returns: 0 if OK, -1 on error
```

Once we have used mkfifo to create a FIFO, we open it using open. When we open a FIFO, the nonblocking flag (O_NONBLOCK) affects what happens.

In the normal case (O_NONBLOCK not specified), an open for read-only blocks until some other process opens the FIFO for writing. Similarly, an open for write-only blocks until some other process opens the FIFO for reading.

If O_NONBLOCK is specified, an open for read-only returns immediately. But an open for write-only returns 1 with errno set to ENXIOif no process has the FIFO open for reading.

There are two uses for FIFOs.

- ✓ FIFOs are used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files.
- ✓ FIFOs are used as rendezvous points in client-server applications to pass data between the clients andthe servers.

Example Using FIFOs to Duplicate Output Streams

- ✓ FIFOs can be used to duplicate an output stream in a series of shell commands.
- ✓ This prevents writing the data to an intermediate disk file. Consider a procedure that needs toprocess a filtered input stream twice. Figure shows this arrangement.

FIGURE : Procedure that processes a filtered input stream twice

- ✓ With a FIFO and the UNIX program tee(1), we can accomplish this procedure without using a temporary file. (The tee program copies its standard input to both its standard output and to the file named on its command line.)

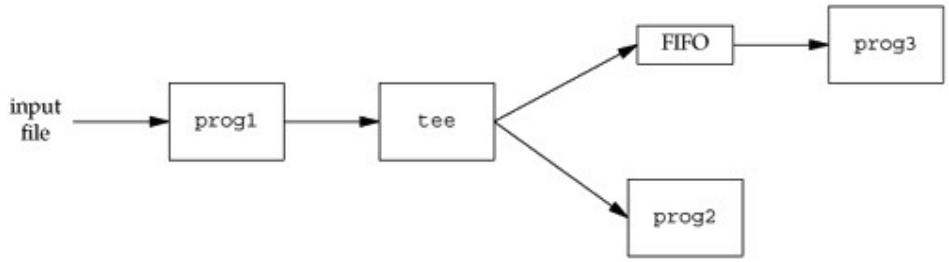
```
m
k
f
i
f
o

f
i
f
o
1

p
r
o
g
3
< fifo1 &

prog1 < infile | tee fifo1 | prog2
```

- ✓ We create the FIFO and then start prog3 in the background, reading from the FIFO. We then start prog1 and use tee to send its input to both the FIFO and prog2. Figure shows the process arrangement.



- ✓ FIGURE : Using a FIFO and tee to send a stream to two different processes
- ✓ Example Client-Server Communication Using a FIFO
- ✓ FIFO's can be used to send data between a client and a server. If we have a server that is contacted by numerous clients, each client can write its request to a well-known FIFO that the server creates. Since there are multiple writers for the FIFO, the requests sent by the clients to the server need to be less than PIPE_BUF bytes in size.
- ✓ This prevents any interleaving of the client writes. The problem in using FIFOs for this type of client server communication is how to send replies back from the server to each client.
- ✓ A single FIFO can't be used, as the clients would never know when to read their response versus responses for other clients. One solution is for each client to send its process ID with the request. The server then creates a unique FIFO for each client, using a pathname based on the client'sprocess ID.
- ✓ For example, the server can create a FIFO with the name /vtu/ ser.XXXXX, where XXXXX is replaced with the client's process ID. This arrangement works, although it is impossible for the server to tell whether a client crashes. This causes the client-specific FIFOs to be left in the file system.
- ✓ The server also must catch SIGPIPE, since it's possible for a client to send a request and terminate before reading the response, leaving the client-specific FIFO with one writer (the server) and no reader.

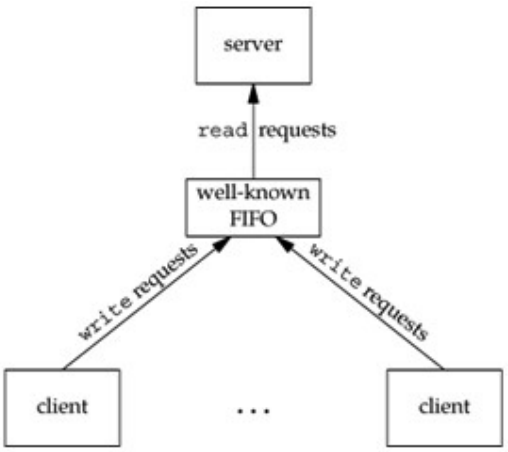


Figure : Clients sending requests to a server using a FIFO

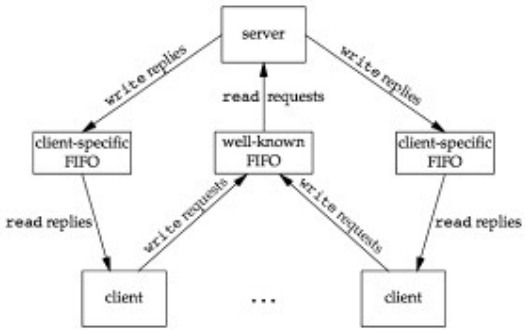


Figure: Client-server communication using FIFOs

5. System V IPC

❖ Identifiers and Keys

Each IPC structure (message queue, semaphore, or shared memory segment) in the kernel is referred to by a non- negative integer identifier. The identifier is an internal name for an IPC object. Cooperating processes need an external naming scheme to be able to rendezvous using the same IPC object. For this purpose, an IPC object is associated with a key that acts as an external name.

Whenever an IPC structure is being created, a key must be specified. The data type of this key is the primitive system data type key_t, which is often defined as a long integer in the header <sys/types.h>. This key is converted into an identifier by the kernel.

There are various ways for a client and a server to rendezvous at the same IPC structure.

- ✓ The server can create a new IPC structure by specifying a key of IPC_PRIVATE and store the returned

  identifier somewhere (such as a file) for the client to obtain. The key IPC_PRIVATE guarantees that the server creates a new IPC structure. The disadvantage to this technique is that file system operations are required for the server to write the integer identifier to a file, and then for the clients to retrieve this identifier later.

  The IPC_PRIVATE key is also used in a parent-child relationship. The parent creates a new IPC structure specifying IPC_PRIVATE, and the resulting identifier is then available to the child after the fork. The child can pass the identifier to a new program as an argument to one of the execfunctions.

- ✓ The client and the server can agree on a key by defining the key in

a common header, for example. The server then creates a new IPC structure specifying this key. The problem with this approach is that it's possible for the key to already be associated with an IPC structure, in which case the get function (msgget, semget, or shmget) returns an error. The server must handle this error, deleting the existing IPC structure, and try to create it again.

- ✓ The client and the server can agree on a pathname and project ID (the project ID is a character value between 0 and 255) and call the function ftok to convert these two values into a key. This key is then used in step 2. The only service provided by ftok is a way of generating a key from a pathname and project ID.

```
#include <sys/ipc.h>
key_t ftok(const char *path, int id);
```

Returns: key if OK, `(key_t)`-1 on error

- ✓ The path argument must refer to an existing file. Only the lower 8 bits of id are used when generating the key.
- ✓ The key created by ftok is usually formed by taking parts of the st_dev and st_ino fields in the stat structure corresponding to the given pathname and combining them with the project ID.
- ✓ If two pathnames refer to two different files, then ftok usually returns two different keys for the two pathnames. However, because both i-node numbers and keys are often stored in long integers, there can be information loss creating a key. This means that two different pathnames to different files can generate the same key if the same project ID isused.

❖ **Permission Structure**

- ✓ XSI IPC associates an ipc_perm structure with each IPC structure. This structure defines the permissions and owner and includes at least the following members:

```
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 */
      .
      .
       };
```

- ✓ All the fields are initialized when the IPC structure is created. At a

later time, we can modify the uid, gid, and mode fields by calling msgctl, semctl, or shmctl. To change these values, the calling process must be either the creator of the IPC structure or the superuser. Changing these fields is similar to calling chown or chmod for a file.

Permission	Bit
user-read	0400
user-write (alter)	0200
group-read	0040
group-write (alter)	0020
other-read	0004
other-write (alter)	0002

XSI IPC permissions

❖ **Advantages and Disadvantages**

- ✓ A fundamental problem with XSI IPC(System V IPC) is that the IPC structures are system wide and do not have a reference count. For example, if we create a message queue, place some messages on the queue, and then terminate, the message queue and its contents are not deleted. They remain in the system until specifically read or deleted by some process calling msgrcv or msgctl, by someone executing the ipcrm(1) command, or by the system being rebooted. Compare this with a pipe, which is completely removed when the last process to reference it terminates. With a FIFO, although the name stays in the file system until explicitly removed, any data left in a FIFO is removed when the last process to reference the FIFO terminates.
- ✓ Another problem with XSI IPC (System V IPC) is that these IPC structures are not known by names in the file system. We can't access them and modify their properties with the functions. Almost a dozen new system calls (msgget, semop, shmat, and so on) were added to the kernel to support these IPC objects. We can't see the IPC objects with an ls command, we can't remove them with the rm command, and we can't change their permissions with the chmod command. Instead, two new commands ipcs(1) and ipcrm(1)were added.
- ✓ Since these forms of IPC don't use file descriptors, we can't use the multiplexed I/O functions (select and poll) with them. This makes it harder to use more than one of these IPC structures at a time or to use any of these IPC structures with file or device I/O. For example, we can't have a server wait for a message to be placed on one of two message queues without some form of busy wait loop.

6. MESSAGE QUEUES

- ✓ A message queue is a linked list of messages stored within the

kernel and identified by a message queue identifier. We'll call the message queue just a queue and its identifier a queue ID.

- ✓ A new queue is created or an existing queue opened by msgget.
- ✓ New messages are added to the end of a queue by msgsnd.
- ✓ Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd when the message is added to a queue.
- ✓ Messages are fetched from a queue by msgrcv. We don't have to fetch the messages in a first-in, first- out order. Instead, we can fetch messages based on their type field.

Each queue has the following msqid_dsstructure associated with it:

```
struct msqid_ds
{
        Struct ipc_perm  msg_perm;
        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_lspid;         /*
        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 */
        .
        .
};
```

This structure defines the current status of the queue.

msgget

- ✓ The first function normally called is msgget to either open an existing queue or create a new queue.

```
#include <sys/msg.h>
int msgget(key_t key, int flag);
                                        Returns: message queue ID if OK, 1 on error
```

When a new queue is created, the following members of the msqid_dsstructure are initialized.

- ✓ The ipc_perm structure is initialized. The mode member of this structure is set to the correspondingpermission bits of flag.
- ✓ msg_qnum, msg_lspid, msg_lrpid, msg_stime, and msg_rtimeare all set to 0.
- ✓ msg_ctimeis set to the current time.
- ✓ msg_qbytesis set to the system limit.

On success, msgget returns the non-negative queue ID. This value is then used with the other threemessage queue functions.

Msgctl

- ✓ The msgctlfunction performs various operations on a queue.

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

The cmd argument specifies the command to be performed on the queue specified by msqid.

Msgsnd

- ✓ Data is placed onto a message queue by calling msgsnd.

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

int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);

                                        Returns: 0 if OK, -1 on error
```

- ✓ Each message is composed of a positive long integer type field, a non-negative length (nbytes), and the actual data bytes (corresponding to the length). Messages are always placed at the end of the queue.
- ✓ The ptr argument points to a long integer that contains the positive integer message type, and it is immediately followed by the message data. (There is no message data if nbytes is 0.) If the largest message we send is 512 bytes, we can define the following structure:

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

- ✓ The ptr argument is then a pointer to a mymesg structure. The message type can be used by the receiver tofetch messages in an order other than first in, first out.

msgrcv

- ✓ Messages are retrieved from a queue by msgrcv

```
#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
                                    Returns: size of data portion of message if OK, -1 on error.
```

The type argument lets us specify which message we want.

type == 0	The first message on the queue is returned.
type > 0	The first message on the queue whose message type equals type is returned.
type < 0	The first message on the queue whose message type is the lowest value less than or equal to the absolute value of type is returned.

7. SEMAPHORES

A semaphore is a counter used to provide access to a shared data object for multiple processes.

To obtain a shared resource, a process needs to do the following:

1. Test the semaphore that controls the resource.
2. If the value of the semaphore is positive, the process can use the resource. In this case, the process

   decrements the semaphore value by 1, indicating that it has used one unit of the resource.
3. Otherwise, if the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0. When the process wakes up, it returns to step 1.

When a process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1. If any other processes are asleep, waiting for the semaphore, they are awakened.
A common form of semaphore is called a ***binary semaphore***. It controls a single resource, and its value is initialized to 1. In general, however, a semaphore can be initialized to any positive value, with the value indicating how many units of the shared resource are available for sharing.
XSI semaphores are, unfortunately, more complicated than this. Three features contribute to this unnecessary complication.

1. A semaphore is not simply a single non-negative value. Instead, we have to define a semaphore as a set of one or more semaphore values. When we create a semaphore, we specify the number of values in the set.
2. The creation of a semaphore (semget) is independent of its initialization (semctl). This is a fatal flaw, since we cannot atomically create a new semaphore set and initialize all the values in the set.

3. Since all forms of XSI IPC remain in existence even when no process is using them, we have to worry about a program that terminates without releasing the semaphores it has been allocated. The undo feature that we describe later is supposed to handle this.

The kernel maintains a semid_dsstructure for each semaphore set:

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

Each semaphore is represented by an anonymous structure containing at least the following members:

```
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 */
.
.
.
};
```

semget

The first function to call is semget to obtain a semaphore ID.

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

When a new set is created, the following members of the semid_ds structure are initialized.

- The ipc_perm structure is initialized. The mode member of this structure is set to the correspondingpermission bits of flag.
- sem_otime is set to 0.

- sem_ctime is set to the current time.
- sem_nsems is set to nsems.
- The number of semaphores in the set is nsems. If a new set is being created (typically in the server), we must specify nsems. If we are referencing an existing set (a client), we can specify nsems as 0.

Semctl

✓ The semctl function is the catchall for various semaphore operations.

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

✓ The fourth argument is optional, depending on the command requested, and if present, is of type semun, a union of various command-specific arguments:

```
union semun
{
    intval;                          /* for SETVAL */
    struct semid_ds *buf;            /*
    for IPC_STAT and IPC_SET */unsigned
    short  *array;                   /*
    for GETALL and SETALL */
};
```

✓ The cmd argument specifies one of the above ten commands to be performed on the set specified by semid.

semop

✓ The function semop atomically performs an array of operations on a

```
#include <sys/sem.h>
int semop(int semid, struct sembuf semoparray[ ], size_t nops);
                                                    Returns: 0 if OK, -1 on error.
```

semaphore set.

✓ The semoparray argument is a pointer to an array of semaphore operations, represented by sembuf
structures:

```
struct sembuf
{
  unsigned short sem_num; /* member # in
  set (0, 1, ..., nsems-1) */short sem_op;
```

```
    /* operation (negative, 0, or positive)
    */ short    sem_flg; /* IPC_NOWAIT,
    SEM_UNDO */
};
```

- The nops argument specifies the number of operations (elements) in the array.
- The sem_op element operations are values specifying the amount by which the semaphore value isto be changed.
  - If sem_op is an integer ***greater than zero***, semop adds the value to the corresponding semaphore element value and awakens all processes that are waiting for the element to increase.
  - If sem_op is ***0*** and the semaphore element value is not 0, semop blocks the calling process (waiting for 0) and increments the count of processes waiting for a zero value of that element.
  - If sem_op is a ***negative*** number, semop adds the sem_op value to the corresponding semaphore element value provided that the result would not be negative. If the operation would make the element value negative, semop blocks the process on the event that the semaphore element value increases. If the resulting value is 0, semop wakes the processes waiting for 0.

7.16 Client-Server Properties

- Following are the some of the properties of client - server architecture :

1. Clients and servers are separate processes.
2. They may run on the same or different machines.
3. Each process can hide internal information.
4. Each process can implement its own set of business rules.
5. They communicate by peer to peer protocols.

- The simplest example of this type is client's "fork" and "exec" of server. Two half-duplex pipes can be created before the fork to allow data to be transferred in both directions. Open server is created using this type of arrangement.
- **With FIFOs,** an individual per client FIFO is also required if the server is to send data back to the client. If the client server application sends data only from the client to the server, a single well-known FIFO suffices.
- Multiple possibilities exist with **message queues.**

1. A single queue can be used between the server and all the clients, using the type field of each message to indicate the message recipient.
2. An individual message queue can be used for each client. Before sending the first request to a server, each client creates its own message queue with a key of IPC_PRIVATE. The server also has its own queue, with a key or identifier known to all clients.

- One problem with this technique is that each client-specific queue usually has only a single message on it : A request for the server or a response for a client. Either of these two techniques using message queues can be implemented using shared memory segments and a synchronization method.

7.17 Stream Pipes

VTU : July-14, 17, Jan.-15, 17, 18

- A stream pipe is a bidirectional pipe. For bidirectional data flow between a parent and child, only a single stream pipe is required.
  Fig. 7.17.1 shows stream pipe.

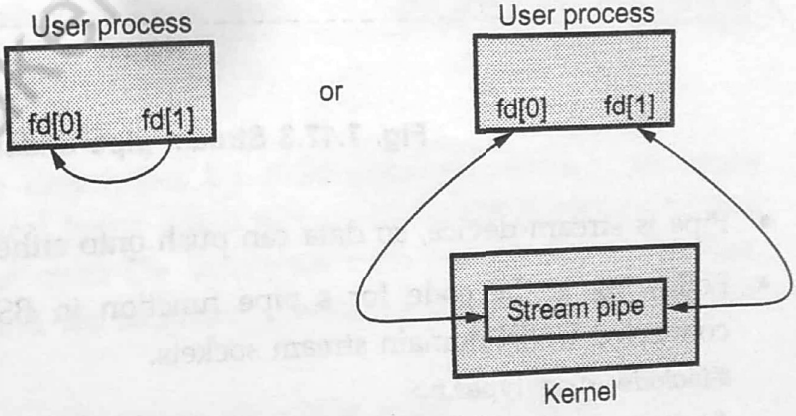


Fig. 7.17.1

- Stream pipe is full duplex. The fd[0] is used by parent process and the child uses only fd[1]. Each end of the stream pipe is full duplex, the parent reads and writes fd[0] and the child duplicates fd[1] to both standard input and standard output. Fig. 7.17.2 shows the

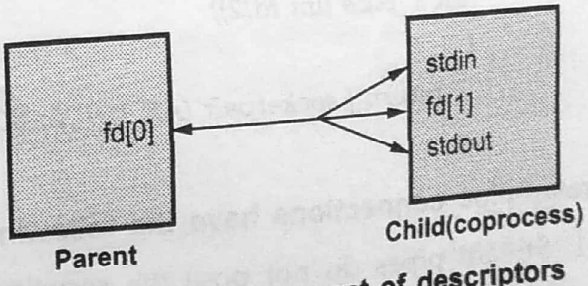


Fig. 7.17.2 Arrangement of descriptors for processes

- The function s_pipe is similar to the standard pipe function. It takes the same argument as pipe, but the returned descriptors are open for reading and writing.
- s_pipe function under SVR4 :

```
#include  "ourhdr.h"
int s_pipe (int fd[2])
{
   return ( pipe (fd) );
}
```

- Fig. 7.17.3 shows the stream pipe under SVR4.

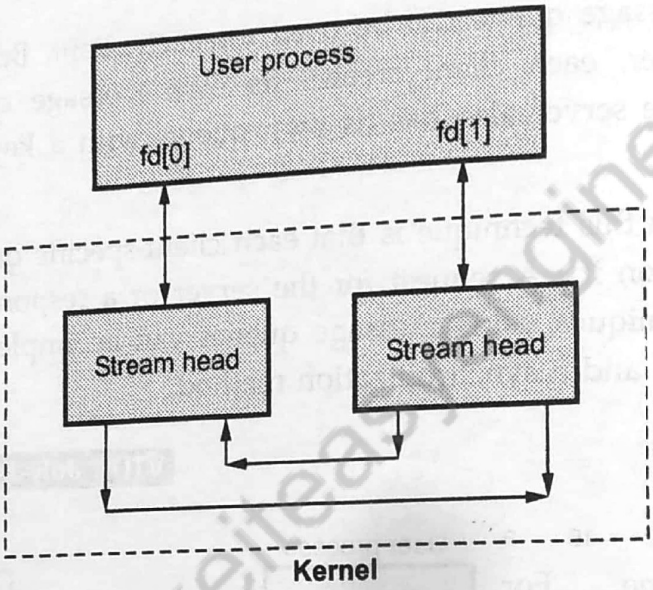


Fig. 7.17.3 Stream pipe under SVR4

- Pipe is stream device, so data can push onto either end of the pipe.
- Following is the code for s_pipe function in BSD version. It creates a pair of connected UNIX domain stream sockets.

```
#include <sys/types.h>
#include <sys/socket.h>
int s_pipe (int fd[2])
{
   return ( socketpair (AF_UNIX , SOCK_STREAM, 0, fd );
}
```

Stream-pipe connections have the following advantages :

1. Stream pipes do not pose the security risk of being overwritten or read by other programs that explicitly access the same portion of shared memory.

2. Stream-pipe connections allow distributed transactions between database servers that are on the same computer.

Stream-pipe connections have the following disadvantages :

1. Stream-pipe connections might be slower than shared-memory connections on some computers.
2. Stream pipes are not available on all platforms.
3. When you use stream pipes for client/server communications, the hostname entry is ignored.

University Questions

1. *What are pipes ? Explain the different ways to view a half duplex pipe. Write a program to create a pipe between a parent and its child and to send data down the pipe.* **VTU : July-14, Marks 10**
2. *Explain with a neat diagram, how STREAM PIPES can be used to implement client server model.* **VTU : Jan.-15, Marks 10**
3. *What are pipes ? Write a C/C++ program to send data from parent to child over a pipe.* **VTU : Jan.-17, Marks 10**
4. *Explain STREAMs - based pipes. Write a C function that is used by a server to wait for a client's connect request to arrive.* **VTU : July-17, Marks 10**
5. *Write short notes on any two of the following : Stream pipes* **VTU : Jan.-18, Marks 10**

7.18 Passing File Descriptors

- The normal inheritance of file descriptors by child processes suffices for many purposes, one of the most typical being when servers use child processes to deal with clients. When we pass an open file descriptor from one process to another, we want the passing process and the receiving process to also share the same file table entry.
- Fig. 7.18.1 shows passing open file from one process to another process.
(See Fig. 7.18.1 on next page)
- When a descriptor is passed from one process to another, sending process closes the descriptor after passing the descriptor. Closing the descriptor by the sender does not really close the file or device, since the descriptor is still considered open by the receiving process.
- In order for a file descriptor to be passed, the sender and receiver must already share a pipe or pump stream for its conveyance. Given a writable file descriptor pd1 for such an open stream and a file descriptor fd1 to be passed, the sender executes the command.

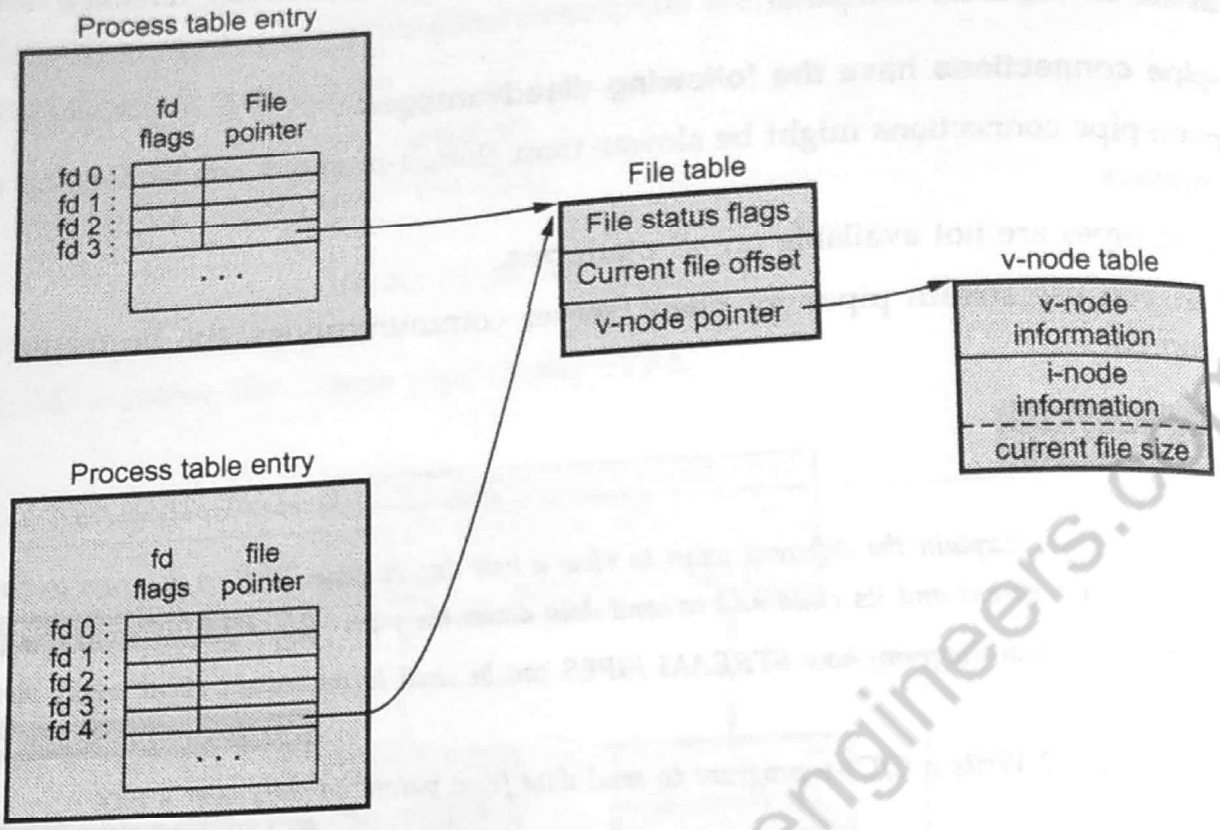


Fig. 7.18.1 Passing an open file

```
#include "ourhdr.h"
int send_fd (int spipefd,   int filedes);
int send_err (int spipefd,   int status,   const char *errmsg);
int recv_fd (int spipefd,  ssize_t (*userfun ) (int, const viod *, size_t));
```

- When a process wants to pass a descriptor to another process it calls either send_fd or send_err. The process waiting to receive the descriptor calls recv_fd.
- The function send_err calls the send_fd function after writing the error message to the s_pipe.

```
int send_err(int fd, int errcode, const char *msg)
{
   int    n;
   if ((n = strlen(msg)) > 0)
      if (writen(fd, msg, n) != n)    /* send the error message */
         return(-1);
   if (errcode >= 0)
      errcode = -1;   /* must be negative */
   if (send_fd(fd, errcode) < 0)
      return(-1);
   return(0);
}
```

7.18.1 System V Release 4

- File descriptors are exchanged using *ioctl* command *I_SENDFD* and *I_RECVED* in SRV4 on stream pipe. Third argument is used for send descriptor. Following is the program for these purposes.
- The send_fd function for STREAMS pipes :

```
#include "apue.h"
#include <stropts.h>

int  send_fd(int fd, int fd_to_send)
{
   char   buf[2];    /* send_fd()/recv_fd( ) 2-byte protocol */
     buf[0] = 0;       /* null byte flag to recv_fd( ) */
     if (fd_to_send < 0)
      {
        buf[1] = -fd_to_send;   /* nonzero status means error */
         if (buf[1] == 0)
           buf[1] = 1; /* -256, etc. would screw up protocol */
      }
       else
      {
           buf[1] = 0;    /* zero status means OK */
     }

        if (write(fd, buf, 2) != 2)
          return(-1);
        if (fd_to_send >= 0)
          if (ioctl(fd, I_SENDFD, fd_to_send) < 0)
            return(-1);
           return(0);
}
```

The strrecvfd structure :

```
struct strrecvfd
 {
    int    fd;       /* new descriptor */
    uid_t  uid;      /* effective user ID of sender */
    gid_t  gid;      /* effective group ID of sender */
    char   fill[8];
 };
```

- The *recv_fd* function reads the STREAMS pipe until the first byte of the 2-byte protocol is received. When we issue the *I_RECVFD* ioctl command, the next message on the stream head's read queue must be a descriptor from an I_SENDFD call, or we get an error.

7.19 An Open Server-Version 1

- To develop a open server program i.e. a program that is executed by a process to open one or more files. But instead of sending the contents of the file back to the calling process, the server sends back an open file descriptor.
- Following is the advantages for designing separate program on server are as follows :

1. The server can easily be contacted by any client, similar to the client calling a library function.
2. If we need to change the server, only a single program is affected.
3. The server can be a set-user-ID program, providing it with additional permissions that the client does not have.

- The client process creates an s_pipe and then calls fork and exec to invoke the server. The client sends requests across the s_pipe, and the server sends back responses across the s_pipe. Following are the application protocol between the client and the server.

1. The client sends a request of the form "open <pathname> <openmode>\0" across the s-pipe to the server.
2. The server sends back an open descriptor or an error by calling either send_fd or send_err.

The client "main" function, version 1 is as follows :

```
#include   "open.h"
#include   <fcntl.h>

#define BUFFSIZE   8192
int main(int argc, char *argv[ ])
{
   int     n, fd;
   char    buf[BUFFSIZE], line[MAXLINE];

   /* read filename to cat from stdin */
   while (fgets(line, MAXLINE, stdin) != NULL)
     {
```

```
            if (line[strlen(line) - 1] == '\n')
            line[strlen(line) - 1] = 0; /* replace newline with null */

          /* open the file */
            if ((fd = csopen(line, O_RDONLY)) < 0)
              continue;   /* csopen() prints error from server */

          /* and cat to stdout */
            while ((n = read(fd, buf, BUFFSIZE)) > 0)
             if (write(STDOUT_FILENO, buf, n) != n)
               err_sys("write error");
               if (n < 0)
               err_sys("read error");
              close(fd);
            }
          exit(0);
        }
```

The server main function, version 1

```
#include   "opend.h"

char    errmsg[MAXLINE];
int     oflag;
char    *pathname;

int main(void)
{
  int    nread;
  char   buf[MAXLINE];

  for ( ; ; )
   {  /* read arg buffer from client, process request */
      if ((nread = read(STDIN_FILENO, buf, MAXLINE)) < 0)
      err_sys("read error on stream pipe");
    else if (nread == 0)
      break;      /* client has closed the stream pipe */
    request(buf, nread, STDOUT_FILENO);
  }
  exit(0);
}
```

7.20 Client-Server Connection Functions

VTU : Jan.-16, 17

- Here we will discuss three functions that can be used by a client server to establish the functions per client connections.

```
#include "ourhdr.h"
int serv_listen (const char *name);
```

- Server listens for client connection on a well known name by calling *serv_listen*. Here name is the well known name of the server. When client wants to connect with server then client used this name. The return value is the file descriptor for the server's end of the named stream pipe.
- Once a server has called serv_listen, it calls serv_accept to wait for a client connection to arrive.

```
#include "ourhdr.h"
int serv_accept (int listenfd, uid_t  *uidptr);
```

where *listenfd* is a descriptor from *serv_listen*.

- When the client does connect to the server, a new stream pipe is automatically created, and the new descriptor is returned as the value of the function. The effective user ID of the client is stored through the pointer *uidptr*.
- A client just calls *cli_conn* to connect to a server.

```
#include "ourhdr.h"
int cli_conn (const char *name);
```

- where name specified by the client must be the same name that was advertised by the server's call to *serv_listen*.
- All above three functions are used for writing server daemons to handle multiple clients. The number of descriptor available to a single process is only limit to handle the multiple clients. Server requires one descriptor for each client connection. The client-server connections are all stream pipes, open descriptors can be passed across the connections.

7.20.1 System V Release 4

- SVR4 provides mounted streams and a streams processing module named *connld* that we can use to provide a named stream pipe with unique connections for the server. Initially server creates an unnamed stream pipe and pushes the stream processing module *connld* on one end. Fig. 7.20.1 shows SVR4 pipe after pushing "connld".
- SVR4 uses *fattach* function to attach a pathname to the end of the pipe that has the *connld* pushed onto it.

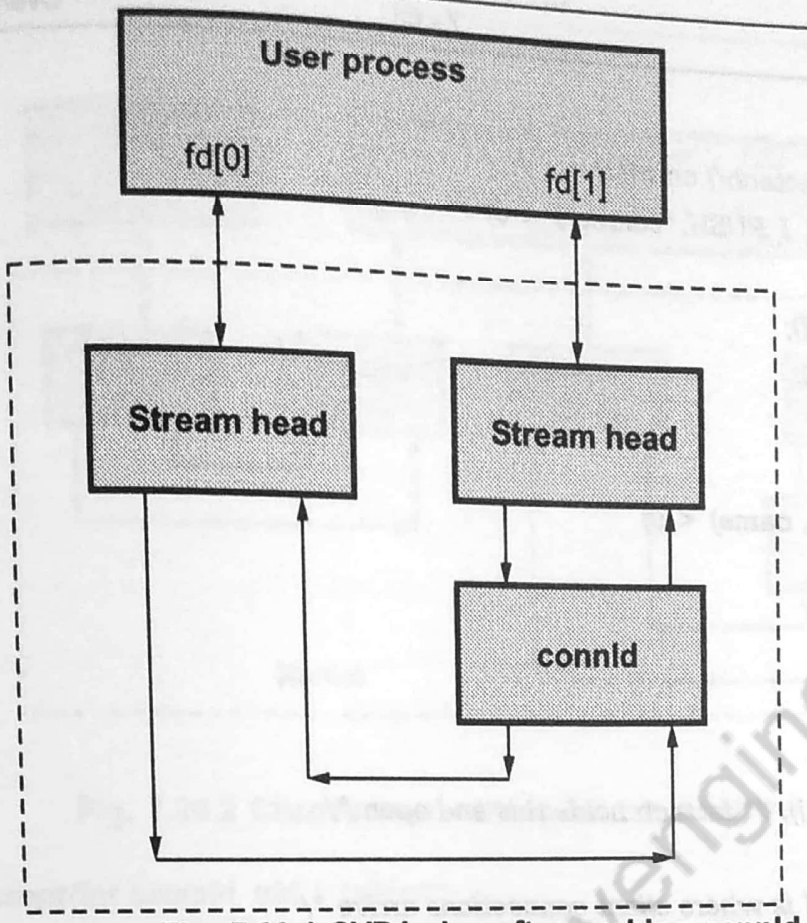


Fig. 7.20.1 SVR4 pipe after pushing connld

- The serv_listen function using STREAMS pipes is as follows :

```
#include "apue.h"
#include <fcntl.h>
#include <stropts.h>
/* pipe permissions: user rw, group rw, others rw */
#define FIFO_MODE (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)

  /* Establish an endpoint to listen for connect requests.*/
int serv_listen(const char *name)
 {
    int    tempfd;
    int    fd[2];

  /*  Create a file: mount point for fattach( ).    */
  unlink(name);
  if ((tempfd = creat(name, FIFO_MODE)) < 0)
     return(-1);
  if (close(tempfd) < 0)
     return(-2);
  if (pipe(fd) < 0)
```

```
        return(-3);
    /*  Push connld & fattach() on fd[1].    */
            if (ioctl(fd[1], I_PUSH, "connld") < 0)
             {
              close(fd[0]);
              close(fd[1]);
             return(-4);
            }
          if (fattach(fd[1], name) < 0)
           {
            close(fd[0]);
           close(fd[1]);
           return(-5);
          }
              close(fd[1]); /* fattach holds this end open */

    return(fd[0]); /* fd[0] is where client connections arrive */
}
```

- When another process call a open for the named end of the pipe, the following occurs :

1. A new pipe is created.
2. One descriptor for new pipe is passed back to the client as the return value from open.
3. The other descriptor is passed to the server on the other end of the named pipe.

- Fig. 7.20.2 shows client-server connection on a named stream pipe. The pipe between the client and server is created by the open. The file descriptor in the client (fd) is returned by the open. The new file descriptor in the server is received by the server using an *ioctl* of *I_RECVFD* on the descriptor fd[0]. Once the server has pushed *connld* onto fd[1] and attached a name to fd[1], it never specifically uses fd[1] again.

- The server waits for a client connection to arrive by calling the serv_accept function. It shown in the following program.

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <stropts.h>
#include    "ourhdr.h"

/* Wait for a client connection to arrive, and accept it.  */
```