Module-4-SYLLABUS

Changing User IDs and Group IDs, Interpreter Files, system Function, Process Accounting, User Identification, Process Times, I/O Redirection.

Overview of IPC Methods,
Overview of IPC Methods, Pipes, popen, pclose Functions, Coprocesses, FIFOs, System V IPC, Message Queues, Semaphores.

SHARED MEMORY: Shared Memory, Client-Server Properties, Stream Pipes, Passing File Descriptors**, An Open Server-Version 1, Client-Server Connection Functions.**

Module-4 :NOTES:

➢ Changing user IDs and Group IDs

When our programs need additional privileges or need to gain access to resources that they currently aren't allowed to access, they need to change their user or group ID to an ID that has the appropriate privilege or access.

Similarly, when our programs need to lower their privileges or prevent access to certain resources, they do so by changing either their user ID or group ID to an ID without the privilege or ability access to the resource.

Function prototype

```
#include <unistd.h>
int setuid(uid_t uid);
int setgid(gid_t gid);
```

Both return: 0 if OK, 1 on error

There are rules for who can change the IDs.
(Everything describe for the user ID also applies to the group ID.)

If the process has superuser privileges, the setuid function sets the real user ID, effective user ID, and saved set-user-ID to uid.

If the process does not have superuser privileges, but uid equals either the real user ID or the saved set-user-ID, setuid sets only the effective user ID to uid. The real user ID and the saved set-user-ID are not changed.

If neither of these two conditions is true, errno is set to EPERM, and an error is returned

Few statements about the three user IDs that the kernel maintains.

Only a superuser process can change the real user ID. Normally, the real user ID is set by the login program when we log in and never changes. Because login is a superuser process, it sets all three user IDs when it calls setuid.

The effective user ID is set by the exec functions only if the set-user-ID bit is set for the program file. If the set-user-ID bit is not set, the exec functions leave the effective user ID as its current value.

We can call setuid at any time to set the effective user ID to either the real user ID or the saved set-user-ID. Naturally, we can't set the effective user ID to any random value.

The saved set-user-ID is copied from the effective user ID by exec. If the file's set-user-ID bit is set, this copy is saved after exec stores the effective user ID from the file's user ID.

➢ setreuid and setregid Functions

Swapping of the real user ID and the effective user ID with the setreuid function.

Function prototype

```
#include <unistd.h>
int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);
```

Both return: 0 if OK, -1 on error

We can supply a value of 1 for any of the arguments to indicate that the corresponding ID should remain unchanged.

The rule is simple: an unprivileged user can always swap between the real user ID and the effective user ID. This allows a set-user-ID program to swap to the user's normal

permissions and swap back again later for set-user- ID operations.

➢ seteuid and setegid functions :

POSIX.1 includes the two functions seteuid and setegid. These functions are similar to setuid and setgid, but only the effective user ID or effective group ID is changed.

Function prototype

```
#include <unistd.h>
int seteuid(uid_t uid);
int setegid(gid_t gid);
```

Both return : 0 if OK, 1 on error

An unprivileged user can set its effective user ID to either its real user ID or its saved set-user-ID.
For a privileged user, only the effective user ID is set to uid. (This differs from the setuid function, which changes all three user IDs.)

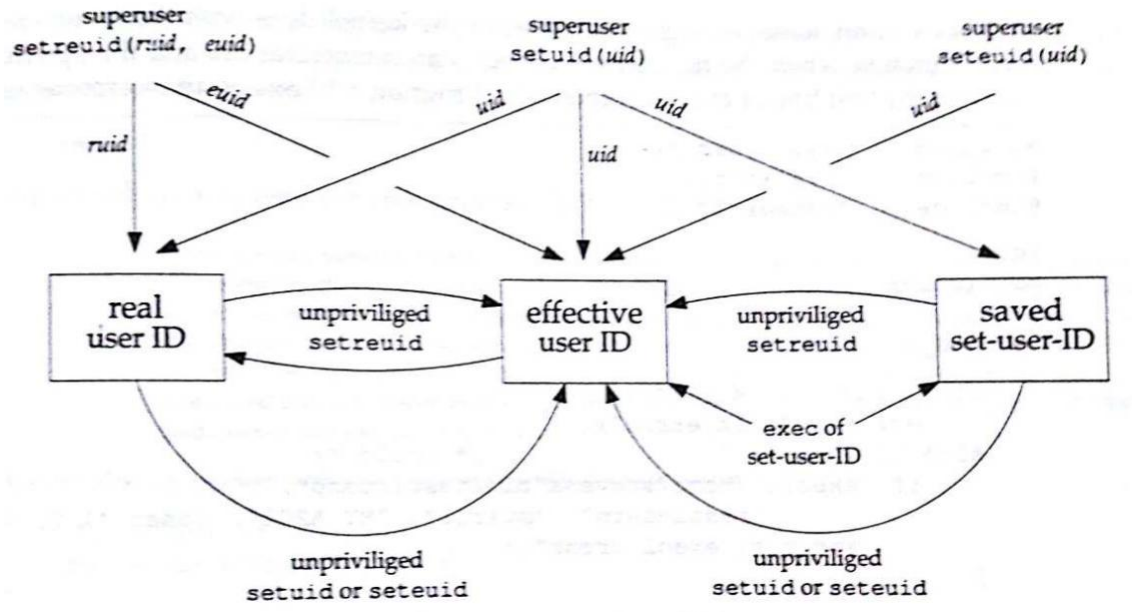


Figure: Summary of all the functions that set the various user IDs

➢ Interpreter files

These files are text files that begin with a line of the form

#! pathname [optional-argument]

The space between the exclamation point and the pathname is optional. The most common of these interpreter files begin with the line

#!/bin/sh

The pathname is normally an absolute pathname, since no special operations are performed on it (i.e., PATH is not used).

The recognition of these files is done within the kernel as part of processing the exec system call.

The actual file that gets executed by the kernel is not the interpreter file, but the file specified by the pathname on the first line of the interpreter file. Be sure to differentiate between the interpreter file a text file that begins with #! and the interpreter, which is specified by the pathname on the first line of the interpreter file.

Be aware that systems place a size limit on the first line of an interpreter file. This limit includes the #!, the pathname, the optional argument, the terminating newline, and any spaces.

A program that execs an interpreter file

```
#include
"apue.h"
#include
<sys/wait.h>
Int main(void)
{
    pid_t   pid;
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0)  {              /* child */
        if (execl("/home/sar/bin/testinterp",
                    "testinterp", "myarg1", "MY ARG2",
            (char *)0) < 0) err_sys("execl error");
    }

    if (waitpid(pid, NULL, 0) < 0) /*
        parent */ err_sys("waitpid
        error");
    exit(0);
```

```
}
```

Output:

```
$ cat
/home/sar/bin/testinterp
#!/home/sar/bin/echoarg
foo

$ ./a.out
argv[0]:
/home/sar/bin/echoarg
argv[1]: foo
argv[2]:
/home/sar/bin/testinterp
argv[3]: myarg1
argv[4]: MY ARG2
```

➢ System FUNCTION

#include <stdlib.h>

int system(const char *cmdstring);

If cmdstring is a null pointer, system returns nonzero only if a command processor is available. This feature determines whether the system function is supported on a given operating system. Under the UNIX System, system is always available.

Because system is implemented by calling fork, exec, and waitpid, there are three types of return values.

If either the fork fails or waitpid returns an error other than EINTR, system returns 1 with errno set to indicate the error.

If the exec fails, implying that the shell can't be executed, the return value is as if the shell had executed exit(127).

Otherwise, all three functions fork, exec, and waitpid succeed, and the return value from system is the termination status of the shell, in the format specified for waitpid.

➢ Process accounting

Most UNIX systems provide an option to do process accounting. When enabled, the kernel writes an accounting record each time a process terminates.

These accounting records are typically 32 bytes of binary data with the name of the command, the amount of CPU time used, the user ID and group ID, the starting time, and so on.

A function **acct** enables and disables process accounting.

A superuser executes accton with a pathname argument to enable accounting.

The accounting records are written to the specified file, which is usually /var/adm/pacct.

Accounting is turned off by executing accton without any arguments.

Each accounting record is written when the process terminates.

This means that the order of the records in the accounting file corresponds to the termination order of the processes, not the order in which they were started.

The accounting records correspond to processes, not programs.

A new record is initialized by the kernel for the child after a fork, not when a new program is executed.

The structure of the accounting records is defined in the header <sys/acct.h> and looks something like

Typedef u_short comp_t

```
struct     acct
{
  Char      ac_flag;        /* flag */
  Char      ac_stat;        /* termination status (signal & core flag only) */
  uid_t     ac_uid;         /*  real user ID */
  gid_t     ac_gid;         /*  real group ID */
  dev_t     ac_tty;         /*  controlling terminal */
```

```
  time_t ac_btime;        /*  starting calendar time */
  comp_t ac_utime;        /*  user CPU time (clock ticks) */
  comp_t ac_stime;        /*  system CPU time (clock ticks) */
  comp_t ac_etime;        /*  elapsed time (clock ticks) */
  comp_t ac_mem;          /*  average memory usage */
  comp_t ac_io;           /*  bytes transferred (by read and write) */
  comp_t ac_rw;           /*  blocks read or written */
  Char ac_comm[8];        /*  command name */
};
```

Values for ac_flag from accounting record

ac_flag	Description
AFORK	process is the result of fork, but never called exec
ASU	process used super user privileges
ACOMPAT	process used compatibility mode
ACORE	process dumped core
AXSIG	process was killed by a signal
AEXPND	expanded accounting entry

Program to generate accounting data

```
#include "apue.h"
Int main(void)
{
    pid_t   pid;
    if ((pid = fork()) < 0)
    err_sys("fork error");

    else if (pid != 0) {          /* parent */
        sleep(2);
        exit(2);                  /* terminate with exit status 2 */
    }
                                  /* first child */
    if ((pid = fork()) < 0)
    err_sys("fork error");
    else if (pid != 0) {
        sleep(4);
        abort();                  /* terminate with core dump */
    }
```

```
                                /* second child */
    if ((pid = fork()) < 0)
    err_sys(“fork error”);
    else if (pid != 0) {
        execl("/bin/dd", "dd", "if=/etc/termcap", "of=/dev/null", NULL);
        exit(7);                /* shouldn't get here */

    }

                                /* third child */

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        sleep(8);
        exit(0);                /* normal exit */
    }

                                /* fourth child */
    sleep(6);
    kill(getpid(), SIGKILL); /* terminate w/signal, no core dump */
    exit(6);                    /* shouldn't get here */
}
```

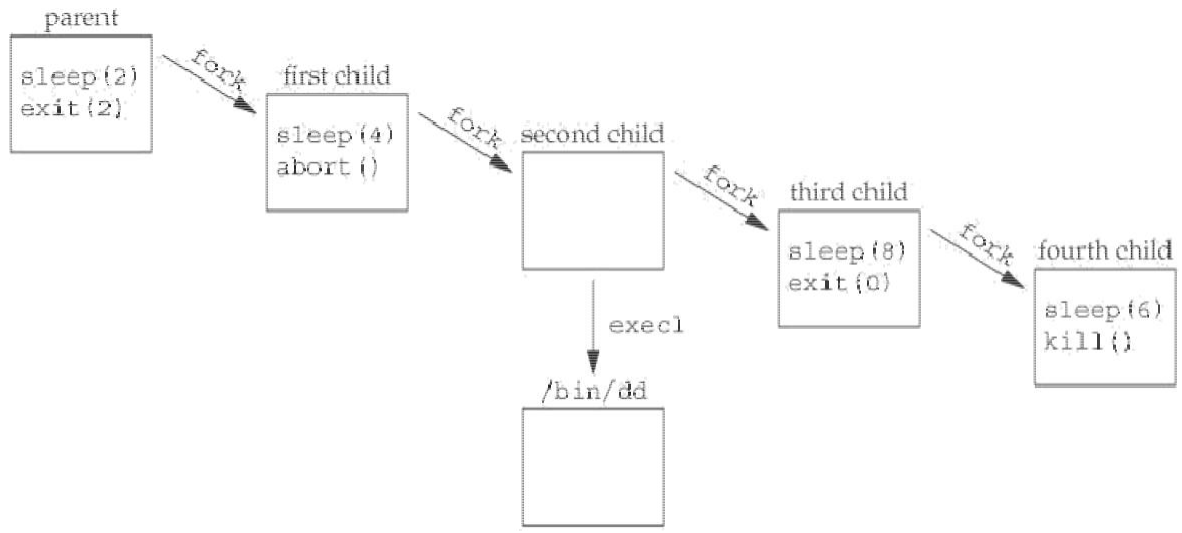


Figure. Process structure of accounting example.

➢ User identification

Any process can find out its real and effective user ID and group ID. Sometimes, however, we want to find out the login name of the user who's running the program. We could call getpwuid(getuid()), but what if a single user has multiple login names, each with the same user ID? (A person might have multiple entries in the password file with the same user ID to have a different login shell for each entry.) The system normally keeps track of the name we log in and the getlogin function provides a way to fetch that login name.

#include <unistd.h>
char *getlogin(void);

Returns : pointer to string giving login name if OK, NULL on error
This function can fail if the process is not attached to a terminal that a user logged in to.

➢ Process times

We describe three times that we can measure:

i. Wall clock time
ii. User CPU time, and
iii. System CPU time.

Any process can call the times function to obtain these values for itself and any terminated children.

Function prototype
#include <sys/times.h>
clock_t times(struct tms *buf);

Returns: elapsed wall clock time in clock ticks if OK, 1 on error

This function fills in the tms structure pointed to by buf:

struct tms
{
clock_t tms_utime; /* user CPU time*/
clock_t tms_stime; /* system CPU time */
clock_t tms_cutime; /* user CPU time, terminated children *
clock_t tms_cstime; /* system CPU time, terminated
children };

Note that the structure does not contain any measurement for the wall clock time. Instead,

the function returns the wall clock time as the value of the function, each time when times function is called.

This value is measured from some arbitrary point in the past, so we can't use its absolute value; instead, we use its relative value.

For example – we call times and save the return value. After some time if we call times again, it's subtract the earlier return value from the new return value. The difference is the clock time.

I/O REDIRECTION:

In this chapter, we will discuss in detail about the Shell input/output redirections. Most Unix system commands take input from your terminal and send the resulting output back to your terminal. A command normally reads its input from the standard input, which happens to be your terminal by default. Similarly, a command normally writes its output to standard output, which is again your terminal by default.

Output Redirection:

The output from a command normally intended for standard output can be easily diverted to a file instead. This capability is known as output redirection.

If the notation > file is appended to any command that normally writes its output to standard output, the output of that command will be written to file instead of your terminal.

Check the following who command which redirects the complete output of the command in the users file.

$ who > users

Notice that no output appears at the terminal. This is because the output has been redirected from the default standard output device (the terminal) into the specified file. You can check the users file for the complete content −

```
$ cat users
oko         tty01   Sep 12 07:30
ai          tty15   Sep 12 13:32
ruth        tty21   Sep 12 10:10
pat         tty24   Sep 12 13:07
steve       tty25   Sep 12 13:03
$
```

If a command has its output redirected to a file and the file already contains some data, that data will be lost. Consider the following example −

```
$ echo line 1 > users
$ cat users
line 1
$
```

You can use >> operator to append the output in an existing file as follows −

```
$ echo line 2 >> users
$ cat users
line 1
line 2
$
```

Input Redirection:

Just as the output of a command can be redirected to a file, so can the input of a command be redirected from a file. As the greater-than character > is used for output redirection, the less-than character < is used to redirect the input of a command.

The commands that normally take their input from the standard input can have their input redirected from a file in this manner. For example, to count the number of lines in the file users generated above, you can execute the command as follows −

```
$ wc -l users
2 users
$
```

Upon execution, you will receive the following output. You can count the number of lines in the file by redirecting the standard input of the wc command from the file users −

```
$ wc -l < users
2
$
```

Note that there is a difference in the output produced by the two forms of the wc command. In the first case, the name of the file users is listed with the line count; in the second case, it is not.

In the first case, wc knows that it is reading its input from the file users. In the second case, it only knows that it is reading its input from standard input so it does not display file name.

Interprocess communication (IPC) is a mechanism where two or more process can communicate with each other to perform the tasks.

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 processes may interact in a client server manner (that is one or more "client" processes send data to a central server process, the server process responds to each client) or in a peer-to-peer fashion (that is, any process may exchange data with others).

- Interprocess communication is supported by all UNIX systems. However, different UNIX system implement different methods for IPC.

- BSD UNIX provides sockets for processes running on different machines to communicate.

- UNIX System V.3 and V.4 supports pipes, messages, semaphores and shared memory for processes running on the same machine to communicate. And they provide Transport Level Interface (TLI) for intermachine communication.

- both BSD and UNIX System V support memory map as an intermachine communication method

➢ Pipes

Pipes are the oldest form of UNIX System IPC. They have two limitations.

1. They have been half duplex (i.e., data flows in only one direction).
2. 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 parent and the child.

The prototype of pipe is

#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, and filedes[1] is open for writing. The output of filedes[1] is the input for filedes[0].

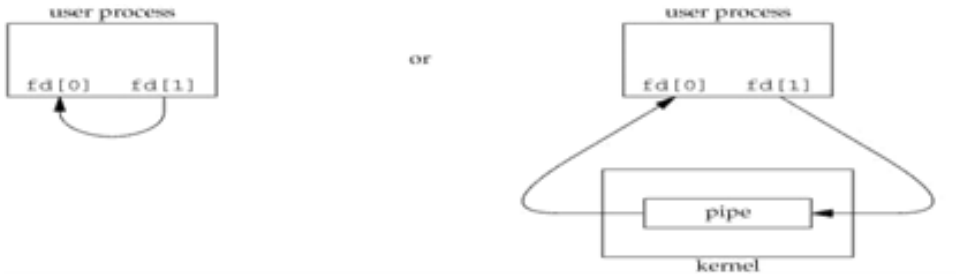


Figure: Two ways to view a half-duplex pipe

- The left half of the figure shows the two ends of the pipe connected in a single process.
- The right half of the figure shows that the data in the pipe flows through the kernel.

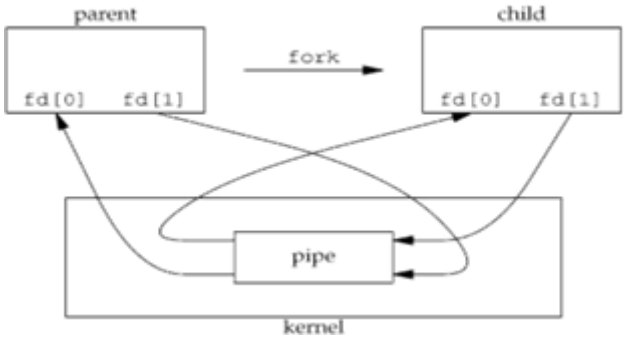
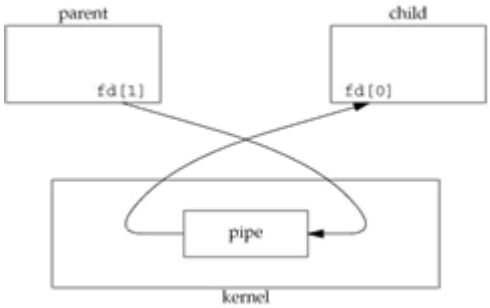


Figure: Half-duplex pipe after a `fork`

- The process that calls pipe then calls fork, creating an IPC channel from the parent to the child or vice versa.

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]).

As shown in the below figure.



For a pipe from the child to the parent, the parent closes fd[1], and the child closes fd[0].

When one end of a pipe is closed, the following two rules apply.

1. 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.

2. If we write to a pipe whose read end has been closed, the signal SIGPIPE is generated. Write returns 1 with errno set to EPIPE. When pipe is full.

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

```
#include "apue.h"
int main(void)
{
int n;
int fd[2];
pid_t pid;
char line[MAXLINE];
        if (pipe(fd) <0);
                err_sys("pipe error");
        if ((pid = fork()) < 0)
        {
        err_sys("fork error");
        }
        else if (pid > 0)     /* parent */
        {
        Close(fd[0]);
        write(fd[1], "hello world\n",12);
        }
        else {        /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line,n);
        }
exit(0);
}
```

➢ popen and pclose functions

A common operation is to create a pipe to another process, either to read its output or send it input, the standard I/O library provides the popen and pclose functions.

These two functions handles the following tasks

- 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.

The prototype is

#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 : Result of fp = popen(cmdstring, "r")

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

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

➢ Coprocesses

A UNIX system provides a filter.
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's output.
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. As shown in the below figure.

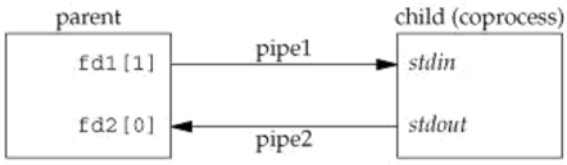


Figure : Driving a coprocess by writing its standard input and reading its standard output

Program: Simple coprocess that reads two numbers from its standard input, computes their sum, and writes the sum to its standard output.

```
#include "apue.h"
int main(void)
{
int n, int1, int2;
char line[MAXLINE];
while ((n =    read(STDIN_FILENO, line, MAXLINE)) > 0) {
line[n]        = 0;                    /* null terminate */
if (sscanf(line, "%d%d", &int1, &int2) == 2) {
sprintf(line, "%d\n", int1 + int2);
n = strlen(line);
if (write(STDOUT_FILENO, line, n)!=n)
err_sys("write error");}
else {
if (write(STDOUT_FILENO, "invalid args\n", 13) !=13)
err_sys("write error");
            }}
      exit(0);
}
```

➢ FIFOs

FIFOs are also called named pipes. Pipes can be used only between related processes when a common ancestor has created the pipe. Whereas with FIFOs unrelated processes can exchange data.

Creating a FIFO is similar to creating a file.

The prototype of FIFO is

#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.

- 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 an error with error errno.

There are two uses of FIFOs.

1. FIFOs are used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files.

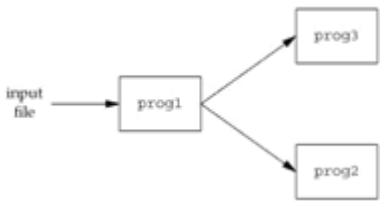
2. FIFOs are used in client-server applications to pass data between the clients and the servers.

Let us discuss each of the above uses of FIFO.

Example – using FIFOs to duplicate output stream.

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 to process a filtered input stream twice.



With a FIFO and the UNIX program tee, 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.)

mkfifo fifo1

prog3 < 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. As shown in the below figure.

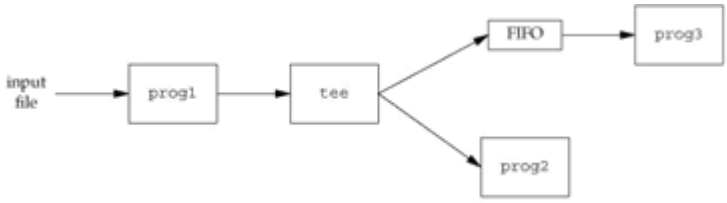


FIGURE : Using a FIFO and tee to send a stream to two different processes

Example Client-Server Communication Using a FIFO

FIFOs can be used to send data between a client and a server. If we have a server that is contacted by various 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.

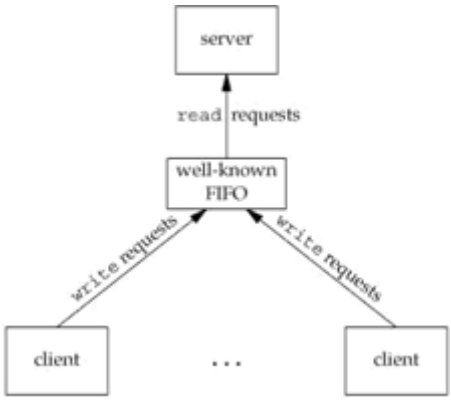


Figure : Clients sending requests to a server using a FIFO

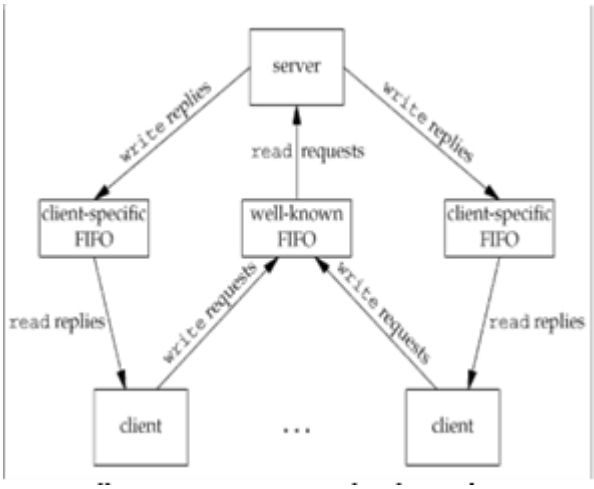


Figure : Client-server communication using FIFOs

➢ System V IPC

System V supports 3 types of IPC

1. Message queue
2. Semaphores
3. Shared memory

There are many similarities (similar features) between the above 3 types of IPC's.

1. **Identifiers and keys** - Each IPC structure (message queue, semaphore, or shared memory segment) in the kernel is referred by a non-negative integer identifier.
   To send or fetch a message to or from a message queue, we need to know the identifier of a queue.
   When an IPC structure is created and then removed, the identifier associated with that structure continuously increases until it reaches a maximum positive value for an identifier and wraps around to zero. (This is called slot usage sequence number).

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

2. **Permission Structure -** System V IPC associates an ipc_perm structure with each IPC structure. This structure defines the permissions and ownership.

```
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 */
        ulong seq;          /* slot usage sequence number */
        key_t key;          /* key */
};
```

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.

3. **Configuration Limits -** All three forms of System V IPC have built-in limits. Most of these limits can be changed by reconfiguring the kernel.

➢ Advantages and dis-advantages

- A fundamental problem with system v IPC is that the IPC structures are systemwide 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 or by rebooting the system. Compare this with a pipe, which is completely removed when the last process to reference it terminates.

- Another problem with 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.
  Many 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). 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.

Advantages

- Reliable
- Flow controlled
- Record oriented
- Con be processed in other then first-in first-out order.

➢ Message queue

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).
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_ds structure associated with it, this structure defines the current status of the queue.

```
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 */
struct msg *msg_first;       /* prt to first message on the queue*/
struct msg *msg_last;        /* prt to last message on the queue*/
time_tmsg_stime;             /* last-msgsnd() time */
time_tmsg_rtime;             /* last-msgrcv() time */
time_tmsg_ctime;             /* last-change time */
.
.
}
```

- **msgget**, is used to 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_ds structure are initialized.

- The ipc_perm structure is initialized.
- msg_qnum, msg_lspid, msg_lrpid, msg_stime, and msg_rtime are all set to 0.
- msg_ctime is set to the current time.
- msg_qbytes is set to the system limit.

On success, msgget returns the non-negative queue ID.

The **msgctl** function performs various operations on a queue.

#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
Returns: 0 if OK, 1 on error

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

Table 9.7.2 POSIX:XSI values for the cmd parameter of msgctl.

cmd	description
IPC_RMID	remove the message queue msqid and destroy the corresponding msqid_ds
IPC_SET	set members of the msqid_ds data structure from buf
IPC_STAT	copy members of the msqid_ds data structure into buf

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 */
};
```

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.

➢ 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.

System V semaphores are more complicated, Three features contribute to this unnecessary complication are.

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 System V 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_ds** structure 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** is used to obtain a semaphore ID

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

Returns: semaphore ID if OK, 1 on error

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 corresponding permission bits of flag.
- sem_otime is set to 0.
- sem_ctime is set to the current time
- sem_nsems is set to nsems.

nsems is the number of semaphores in the set. 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.

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
{
  int    val;                    /* for SETVAL */
  struct semid_ds *buf;          /* for IPC_STAT and IPC_SET */
  unsigned short   *array; /* for GETALL and SETALL */
};
```

Table 9.8.1 POSIX:XSI values for the cmd parameter of semctl.

cmd	description
GETALL	return values of the semaphore set in arg.array
GETVAL	return value of a specific semaphore element
GETPID	return process ID of last process to manipulate element
GETNCNT	return number of processes waiting for element to increment
GETZCNT	return number of processes waiting for element to become 0
IPC_RMID	remove semaphore set identified by semid
IPC_SET	set permissions of the semaphore set from arg.buf
IPC_STAT	copy members of semid_ds of semaphore set semid into arg.buf
SETALL	set values of semaphore set from arg.array
SETVAL	set value of a specific semaphore element to arg.val

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

The function **semop** atomically performs an array of operations on a semaphore set.

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

Returns: 0 if OK, 1 on error.

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 is to 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.

➢ Shared memory

Shared memory allows two or more processes to share a given region of memory. This is the fastest form of IPC because the data does not need to be copied between the client and the server.

The only trick in using shared memory is synchronizing access to a given region of shared memory among multiple processes.

If the server is placing data into a shared memory region the client shouldn't try to access the data until server is done. Often semaphores are used to synchronize shared memory access.

The kernel maintains the following structure for each shared memory segment

```
struct shmid_ds
{
struct ipc_perm shm_perm;
struct anon_map  *shm_amp;          /*pointer in kernel */
int shm_segsz;                      /* siza of segment in bytes */
ushort shm_lkcnt;                   /* number of times segment is being locked */
pid_t shm_lpid:                     /* pid of last shmop() */
p1d_t shm_cpid;                     /* pid of creator */
ulong shm_nattch:                   /* number of current attaches */
ulong shm_cnattch;                  /* used only for shminfo */
time_t shm_atime;                   /* last-attach time */
time_t shm_dtime;                   /* last-detach time */
time_t shm_ctime;                   /* last-change time */
};
```

System limits that affect shared memory

- **SHMMAX** - The maximum size in bytes of a shared memory segment. [131,072]
- **SHMMIN** - The minimum size in bytes of a shared memory segment. [1]
- **SHMMNI** - The maximum number of shared memory segments, system wide. [100]
- **SHMSEG** - The maximum number of shared memory segments, per process. [6]

shmget is used to obtain a shared memory identifier.

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

int shmget (key_t key, int size, int flag);

Returns: shared memory ID if OK, —1 on error;

When a new segment is created the following members of the shmid_ds structure are initialized.

- The ipc_perm structure is initialized.
- shm_lpid, shm_nattach, shm_atime, and shm_dtime are all set to 0.
- shm_ctime is set to the current time.

Size is the minimum size of the shared memory segment. If a new segment is being created (typically in the server) we must specify its size. If we are referencing an existing segment (a client) we can specify size as 0.

The **shmctl** function is the catchall for various shared memory operations

#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);

Return 0 if ok, -1 on error

- The **cmd argument** specifies one of the following five commands to be performed, on the segment specified by shmid.

1. **IPC_STAT** - Fetch the shmid_ds structure for this segment, storing it in the structure pointed to by buf.
2. **IPC_SET** - Set the following three fields from the structure pointed to by buf. shm_perm.uid, Shm_perm.gid, and shm perm.mode.
3. **IPC_RMID** - Remove the shared memory segment set from the system.
4. **SHM LOCK** - Lock the shared memory segment in memory. This command can be executed only by the superuser.
5. **SHM UNLOCK** - Unlock the Shared memory segment. This command can be executed only by the superuser.

- Once a shared memory Segment has been created, a process attaches it to its address space by calling **shmat.**

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

Returns: pointer to shared memory segment if OK, —1 on error

- The address in the calling process at which the segment is attached depends on the addr argument and whether the SHM_RND bit is specified in flag.

1. 1f addr is 0, the segment is attached at the first available address selected by the kernel.
2. If addr is nonzero and SHM_RND is not specified, the segment is attached at the address given by addr.
3. If addr is nonzero and SHM_RND is specified, the segment is attached at the address given by (addr — (addr modulus SHMLBA)). SHMLBA stands for ”low boundary address multiple.

If the SHM__RDONLY bit is specified in flag, the segment is attached read-only. Otherwise the segment is attached read-write.

- When we’re done with a shared memory segment we call **shmdt** to detach it.

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

Returns: 0 if OK, —1 on error

Note that this does not remove the identifier and its associated data structure from the system. The identifier remains in existence until some process (often a server) specifically removes it by calling shmctl with a command of IPC_RMID.

client server properties and client server connection functions:

Most of the Net Applications use the Client-Server architecture, which refers to two processes or two applications that communicate with each other to exchange some information. One of the two processes acts as a client process, and another process acts as a server.

Client Process

This is the process, which typically makes a request for information. After getting the response, this process may terminate or may do some other processing.

Example, Internet Browser works as a client application, which sends a request to the Web Server to get one HTML webpage.

Server Process

This is the process which takes a request from the clients. After getting a request from the client, this process will perform the required processing, gather the requested information, and send it to the requestor client. Once done, it becomes ready to serve another client. Server processes are always alert and ready to serve incoming requests.

Example − Web Server keeps waiting for requests from Internet Browsers and as soon as it gets any request from a browser, it picks up a requested HTML page and sends it back to that Browser.

Note that the client needs to know the address of the server, but the server does not need to know the address or even the existence of the client prior to the connection being established. Once a connection is established, both sides can send and receive information.

2-tier and 3-tier architectures

There are two types of client-server architectures −

- **2-tier architecture** − In this architecture, the client directly interacts with the server. This type of architecture may have some security holes and performance problems. Internet Explorer and Web Server work on two-tier architecture. Here security problems are resolved using Secure Socket Layer (SSL).
- **3-tier architectures** − In this architecture, one more software sits in between the client and the server. This middle software is called ‘middleware’. Middleware are used to perform all the security checks and load balancing in case of heavy load. A middleware takes all requests from the client and after performing the required authentication, it passes that request to the server. Then the

server does the required processing and sends the response back to the middleware and finally the middleware passes this response back to the client. If you want to implement a 3-tier architecture, then you can keep any middleware like Web Logic or WebSphere software in between your Web Server and Web Browser.

Types of Server

There are two types of servers you can have −

- **Iterative Server** − This is the simplest form of server where a server process serves one client and after completing the first request, it takes request from another client. Meanwhile, another client keeps waiting.
- **Concurrent Servers** − This type of server runs multiple concurrent processes to serve many requests at a time because one process may take longer and another client cannot wait for so long. The simplest way to write a concurrent server under Unix is to *fork* a child process to handle each client separately.

How to Make Client

The system calls for establishing a connection are somewhat different for the client and the server, but both involve the basic construct of a socket. Both the processes establish their own sockets.

The steps involved in establishing a socket on the client side are as follows −

- Create a socket with the **socket()** system call.
- Connect the socket to the address of the server using the **connect()** system call.
- Send and receive data. There are a number of ways to do this, but the simplest way is to use the **read()** and **write()** system calls.

How to make a Server

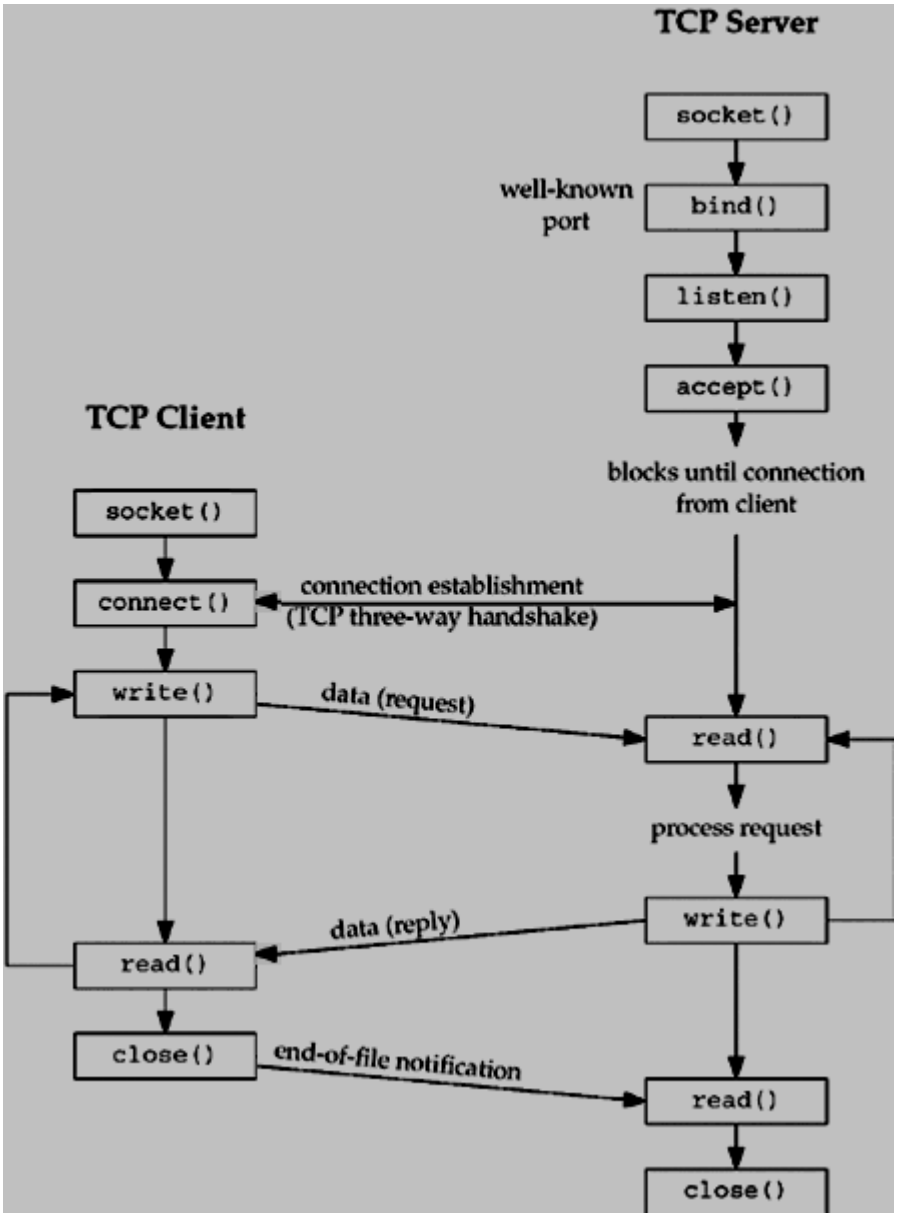
The steps involved in establishing a socket on the server side are as follows −

- Create a socket with the **socket()** system call.
- Bind the socket to an address using the **bind()** system call. For a server socket on the Internet, an address consists of a port number on the host machine.
- Listen for connections with the **listen()** system call.

- Accept a connection with the **accept()** system call. This call typically blocks the connection until a client connects with the server.
- Send and receive data using the **read()** and **write()** system calls.

Client and Server Interaction

Following is the diagram showing the complete Client and Server interaction −



The socket Function

To perform network I/O, the first thing a process must do is, call the socket function, specifying the type of communication protocol desired and protocol family, etc.

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

int socket (int family, int type, int protocol);
```

This call returns a socket descriptor that you can use in later system calls or -1 on error.

Parameters

family − It specifies the protocol family and is one of the constants shown below −

Family	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols
AF_ROUTE	Routing Sockets
AF_KEY	Ket socket

This chapter does not cover other protocols except IPv4.

type − It specifies the kind of socket you want. It can take one of the following values −

Type	Description
SOCK_STREAM	Stream socket
SOCK_DGRAM	Datagram socket
SOCK_SEQPACKET	Sequenced packet socket
SOCK_RAW	Raw socket

protocol − The argument should be set to the specific protocol type given below, or 0 to select the system's default for the given combination of family and type −

Protocol	Description
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol

The *connect* Function

The *connect* function is used by a TCP client to establish a connection with a TCP server.

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

int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

This call returns 0 if it successfully connects to the server, otherwise it returns -1 on error.

Parameters

- **sockfd** − It is a socket descriptor returned by the socket function.
- **serv_addr** − It is a pointer to struct sockaddr that contains destination IP address and port.
- **addrlen** − Set it to sizeof(struct sockaddr).

The *bind* Function

The *bind* function assigns a local protocol address to a socket. With the Internet protocols, the protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number. This function is called by TCP server only.

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

int bind(int sockfd, struct sockaddr *my_addr,int addrlen);
```

This call returns 0 if it successfully binds to the address, otherwise it returns -1 on error.

Parameters

- **sockfd** − It is a socket descriptor returned by the socket function.
- **my_addr** − It is a pointer to struct sockaddr that contains the local IP address and port.
- **addrlen** − Set it to sizeof(struct sockaddr).

You can put your IP address and your port automatically

A 0 value for port number means that the system will choose a random port, and *INADDR_ANY* value for IP address means the server's IP address will be assigned automatically.

```
server.sin_port = 0;
server.sin_addr.s_addr = INADDR_ANY;
```

NOTE − All ports below 1024 are reserved. You can set a port above 1024 and below 65535 unless they are the ones being used by other programs.

The *listen* Function

The *listen* function is called only by a TCP server and it performs two actions −

- The listen function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket.
- The second argument to this function specifies the maximum number of connections the kernel should queue for this socket.

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

int listen(int sockfd,int backlog);
```

This call returns 0 on success, otherwise it returns -1 on error.

Parameters

- **sockfd** − It is a socket descriptor returned by the socket function.
- **backlog** − It is the number of allowed connections.

The *accept* Function

The *accept* function is called by a TCP server to return the next completed connection from the front of the completed connection queue. The signature of the call is as follows −

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

int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

This call returns a non-negative descriptor on success, otherwise it returns -1 on error. The returned descriptor is assumed to be a client socket descriptor and all read-write operations will be done on this descriptor to communicate with the client.

Parameters

- **sockfd** − It is a socket descriptor returned by the socket function.
- **cliaddr** − It is a pointer to struct sockaddr that contains client IP address and port.
- **addrlen** − Set it to sizeof(struct sockaddr).

The *send* Function

The *send* function is used to send data over stream sockets or CONNECTED datagram sockets. If you want to send data over UNCONNECTED datagram sockets, you must use sendto() function.

You can use *write()* system call to send data. Its signature is as follows −

```
int send(int sockfd, const void *msg, int len, int flags);
```

This call returns the number of bytes sent out, otherwise it will return -1 on error.

Parameters

- **sockfd** − It is a socket descriptor returned by the socket function.
- **msg** − It is a pointer to the data you want to send.
- **len** − It is the length of the data you want to send (in bytes).
- **flags** − It is set to 0.

The *recv* Function

The *recv* function is used to receive data over stream sockets or CONNECTED datagram sockets. If you want to receive data over UNCONNECTED datagram sockets you must use recvfrom().

You can use *read()* system call to read the data. This call is explained in helper functions chapter.

int recv(int sockfd, void *buf, int len, unsigned int flags);

This call returns the number of bytes read into the buffer, otherwise it will return -1 on error.

Parameters

- **sockfd** − It is a socket descriptor returned by the socket function.
- **buf** − It is the buffer to read the information into.
- **len** − It is the maximum length of the buffer.
- **flags** − It is set to 0.

The *sendto* Function

The *sendto* function is used to send data over UNCONNECTED datagram sockets. Its signature is as follows −

int sendto(int sockfd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen);

This call returns the number of bytes sent, otherwise it returns -1 on error.

Parameters

- **sockfd** − It is a socket descriptor returned by the socket function.
- **msg** − It is a pointer to the data you want to send.
- **len** − It is the length of the data you want to send (in bytes).
- **flags** − It is set to 0.
- **to** − It is a pointer to struct sockaddr for the host where data has to be sent.
- **tolen** − It is set it to sizeof(struct sockaddr).

The *recvfrom* Function

The *recvfrom* function is used to receive data from UNCONNECTED datagram sockets.

int recvfrom(int sockfd, void *buf, int len, unsigned int flags struct sockaddr *from, int *fromlen);

This call returns the number of bytes read into the buffer, otherwise it returns -1 on error.

Parameters

- **sockfd** − It is a socket descriptor returned by the socket function.
- **buf** − It is the buffer to read the information into.
- **len** − It is the maximum length of the buffer.
- **flags** − It is set to 0.
- **from** − It is a pointer to struct sockaddr for the host where data has to be read.
- **fromlen** − It is set it to sizeof(struct sockaddr).

The *close* Function

The *close* function is used to close the communication between the client and the server. Its syntax is as follows −

int close(int sockfd);

This call returns 0 on success, otherwise it returns -1 on error.

Parameters

- **sockfd** − It is a socket descriptor returned by the socket function.

The *shutdown* Function

The *shutdown* function is used to gracefully close the communication between the client and the server. This function gives more control in comparison to the *close* function. Given below is the syntax of *shutdown* −

int shutdown(int sockfd, int how);

This call returns 0 on success, otherwise it returns -1 on error.

Parameters

- **sockfd** − It is a socket descriptor returned by the socket function.
- **how** − Put one of the numbers −
  - **0** − indicates that receiving is not allowed,
  - **1** − indicates that sending is not allowed, and
  - **2** − indicates that both sending and receiving are not allowed. When *how* is set to 2, it's the same thing as close().

The *select* Function

The *select* function indicates which of the specified file descriptors is ready for reading, ready for writing, or has an error condition pending.

When an application calls *recv or recvfrom*, it is blocked until data arrives for that socket. An application could be doing other useful processing while the incoming data stream is empty. Another situation is when an application receives data from multiple sockets.

Calling *recv or recvfrom* on a socket that has no data in its input queue prevents immediate reception of data from other sockets. The select function call solves this problem by allowing the program to poll all the socket handles to see if they are available for non-blocking reading and writing operations.

Given below is the syntax of *select* −

```
int select(int  nfds, fd_set  *readfds, fd_set  *writefds, fd_set *errorfds, struct timeval *timeout);
```

This call returns 0 on success, otherwise it returns -1 on error.

Parameters

- **nfds** − It specifies the range of file descriptors to be tested. The select() function tests file descriptors in the range of 0 to nfds-1
- **readfds** − It points to an object of type *fd_set* that on input, specifies the file descriptors to be checked for being ready to read, and on output, indicates which file descriptors are ready to read. It can be NULL to indicate an empty set.

- **writefds** − It points to an object of type *fd_set* that on input, specifies the file descriptors to be checked for being ready to write, and on output, indicates which file descriptors are ready to write. It can be NULL to indicate an empty set.
- **exceptfds** − It points to an object of type *fd_set* that on input, specifies the file descriptors to be checked for error conditions pending, and on output indicates, which file descriptors have error conditions pending. It can be NULL to indicate an empty set.
- **timeout** − It points to a timeval struct that specifies how long the select call should poll the descriptors for an available I/O operation. If the timeout value is 0, then select will return immediately. If the timeout argument is NULL, then select will block until at least one file/socket handle is ready for an available I/O operation. Otherwise *select* will return after the amount of time in the timeout has elapsed OR when at least one file/socket descriptor is ready for an I/O operation.

The return value from select is the number of handles specified in the file descriptor sets that are ready for I/O. If the time limit specified by the timeout field is reached, select return 0. The following macros exist for manipulating a file descriptor set −

- **FD_CLR(fd, &fdset)** − Clears the bit for the file descriptor fd in the file descriptor set *fdset.*
- **FD_ISSET(fd, &fdset)** − Returns a non-zero value if the bit for the file descriptor *fd* is set in the file descriptor set pointed to by *fdset*, and 0 otherwise.
- **FD_SET(fd, &fdset)** − Sets the bit for the file descriptor fd in the file descriptor set fdset.
- **FD_ZERO(&fdset)** − Initializes the file descriptor set fdset to have zero bits for all file descriptors.

The behavior of these macros is undefined if the fd argument is less than 0 or greater than or equal to FD_SETSIZE.

Example

```
fd_set fds;

struct timeval tv;

/* do socket initialization etc.
tv.tv_sec = 1;
tv.tv_usec = 500000;

/* tv now represents 1.5 seconds */
```

```
FD_ZERO(&fds);

/* adds sock to the file descriptor set */
FD_SET(sock, &fds);

/* wait 1.5 seconds for any data to be read from any single socket */
select(sock+1, &fds, NULL, NULL, &tv);

if (FD_ISSET(sock, &fds)) {
  recvfrom(s, buffer, buffer_len, 0, &sa, &sa_len);
  /* do something */
}
else {
  /* do something else */
}
```

PASSING FILE DESCRIPTORS

FILE DESCRIPTOR:

In Unix and operating systems, a **file descriptor** (**FD**, less frequently **fields**) is an abstract handler used to access a file or other i/o resource, such as a pipe or network socket. File descriptors form part of the POSIX application programming interface(API). A file descriptor is a non-negative integer, generally represented in the C programming language as the type int (negative values being reserved to indicate "no value" or an error condition).

Each Unix process (except a daemon) should expect to have three standard POSIX file descriptors, corresponding to the three standard streams:

Integer value	Name	<unistd.h> symbolic constant [1]	<stdio.h> file stream stream [2]
0	Standard input	STDIN_FILENO	stdin

1	Standard output	STDOUT_FILENO	stdout
2	Standard error	STDERR_FILENO	stderr

Passing File Descriptors

The ability to pass an open file descriptor between processes is powerful. It can lead to different ways of designing client-server applications. It allows one process (a server) to do everything that is required to open a file (like translating a network name to a network address) and simply pass back to the calling process descriptor that can be used with all the I/O functions. All the details involved in opening the file or device are hidden from the client.

when a descriptor is passed from one process to another, the sending process, after passing the descriptor, closes the descriptor. Closing the descriptor by the sender doesn't close the file or device, since the descriptor is still considered open by the receiving process (even if the receiver hasn't received the descriptor yet).

We define the following three functions that we use to send and receive file descriptors.

int send_fd(int *fd*, int *fd_to_send*);

int send_err(int *fd*, int *status*, const char **errmsg*);

Both return: 0 if OK, 1 on error

int recv_fd(int *fd*, ssize_t (**userfunc*)(int, const void *, size_t));

Returns: file descriptor if OK, else negative value on error

A process (normally a server) that wants to pass a descriptor to another process calls either send_fd or send_err. The process waiting to receive the descriptor (the client) calls recv_fd.

The send_fd function sends the descriptor *fd_to_send* across using the STREAMS pipe or UNIX domain socket represented by *fd*.

The send_err function sends the *errmsg* using *fd*, followed by the *status* byte. The value of *status* must be in the range 1 through 255.

Clients call recv_fd to receive a descriptor. If all is OK (the sender called send_fd), the non-negative

descriptor is returned as the value of the function. Otherwise, the value returned is the *status* that was sent by send_err (a negative value in the range 1 through -255).

The first argument to *userfunc* is the constant STDERR_FILENO, followed by a pointer to the error message and its length.

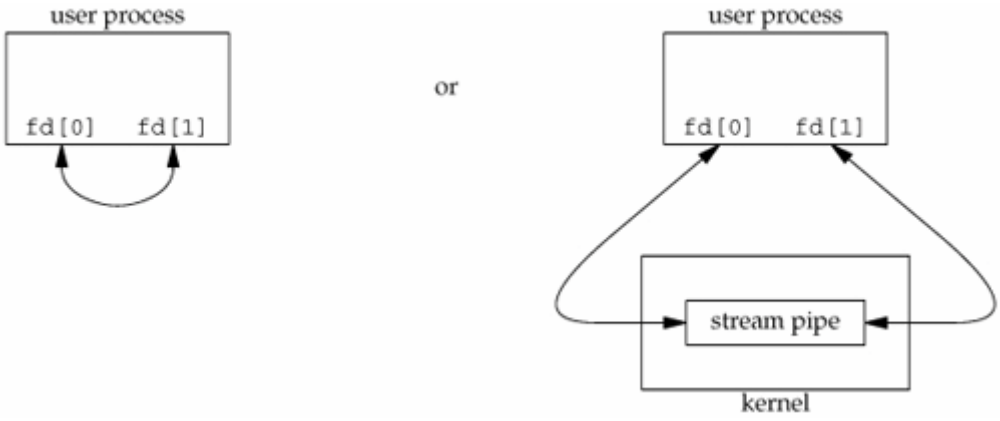
STREAMS-Based Pipes

A STREAMS-based pipe ("STREAMS pipe," for short) is a bidirectional (full-duplex) pipe. To obtain bidirectional data flow between a parent and a child, only a single STREAMS pipe is required.

STREAMS pipes are supported by Solaris and are available in an optional add-on package with Linux.

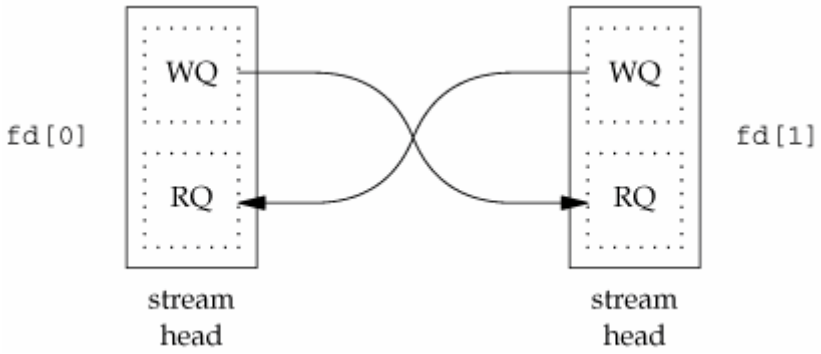
Figure shows the two ways to view a STREAMS pipe. The only difference between this picture and Figure is that the arrows have heads on both ends; since the STREAMS pipe is full duplex, data can flow in both directions.

Figure . Two ways to view a STREAMS pipe



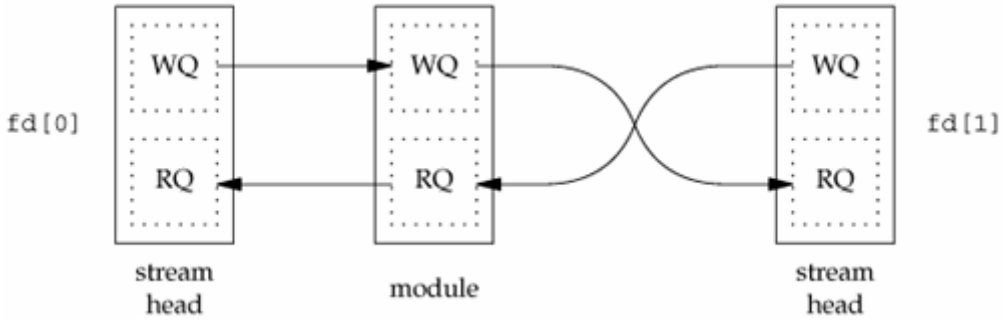
If we look inside a STREAMS pipe, we see that it is simply two stream heads, with each write queue (WQ) pointing at the other's read queue (RQ). Data written to one end of the pipe is placed in messages on the other's read queue.

Figure . Inside a STREAMS pipe



Since a STREAMS pipe is a stream, we can push a STREAMS module onto either end of the pipe to process data written to the pipe. But if we push a module on one end, we can't pop it off the other end. If we want to remove it, we need to remove it from the same end on which it was pushed.

Figure . Inside a STREAMS pipe with a module



Assuming that we don't do anything fancy, such as pushing modules, a STREAMS pipe behaves just like a non-STREAMS pipe, except that it supports most of the STREAMS ioctl commands described in streamio(7). In , we'll see an example of pushing a module on a STREAMS pipe to provide unique connections when we give the pipe a name in the file system.

Example

Let's redo the coprocess example, , with a single STREAMS pipe. Figure shows the new main function. The add2 coprocess is the same . We call a new function, s_pipe, to create a single STREAMS pipe. (We show versions of this function for both STREAMS pipes and UNIX domain sockets shortly.)

The parent uses only fd[0], and the child uses only fd[1]. Since each end of the STREAMS pipe is full duplex, the parent reads and writes fd[0], and the child duplicates fd[1] to both standard input and standard output. shows the resulting descriptors. Note that this example also works with full-duplex pipes that are not based on STREAMS, because it doesn't make use of any STREAMS features other than the full-duplex nature of STREAMS-based pipes.

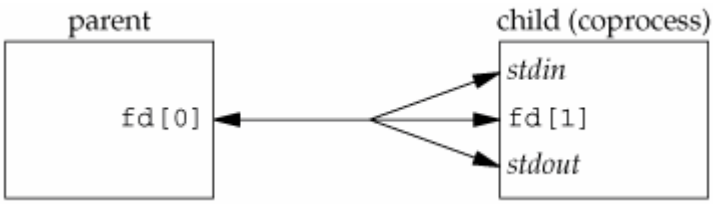
Rago [1993] covers STREAMS-based pipes in more detail. Recall from that FreeBSD supports full-duplex pipes, but these pipes are not based on the STREAMS mechanism.

Figure . Program to drive the add2 filter, using a STREAMS pipe

```
#include "apue.h"

static void sig_pipe(int);      /* our signal handler */

int
main(void)
{
    int     n;
    int     fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (signal(SIGPIPE, sig_pipe) == SIG_ERR)
        err_sys("signal error");

    if (s_pipe(fd) < 0)         /* need only a single stream pipe */
        err_sys("pipe error");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {                       /* parent */
        close(fd[1]);
        while (fgets(line, MAXLINE, stdin) != NULL) {
            n = strlen(line);
            if (write(fd[0], line, n) != n)
                err_sys("write error to pipe");
            if ((n = read(fd[0], line, MAXLINE)) < 0)
                err_sys("read error from pipe");
            if (n == 0) {
                err_msg("child closed pipe");
                break;
            }
            line[n] = 0; /* null terminate */
            if (fputs(line, stdout) == EOF)
                err_sys("fputs error");
        }
        if (ferror(stdin))
            err_sys("fgets error on stdin");
        exit(0);
    } else {                                    /* child */
        close(fd[0]);
        if (fd[1] != STDIN_FILENO &&
          dup2(fd[1], STDIN_FILENO) != STDIN_FILENO)
            err_sys("dup2 error to stdin");
        if (fd[1] != STDOUT_FILENO &&
          dup2(fd[1], STDOUT_FILENO) != STDOUT_FILENO)
              err_sys("dup2 error to stdout");
        if (execl("./add2", "add2", (char *)0) < 0)
```

```
        err_sys("execl error");
    }
    exit(0);
}
static void
sig_pipe(int signo)
{
    printf("SIGPIPE caught\n");
    exit(1);
}
```

Figure . Arrangement of descriptors for coprocess



We define the function s_pipe to be similar to the standard pipe function. Both functions take the same argument, but the descriptors returned by s_pipe are open for reading and writing.

ExampleSTREAMS-Based s_pipe Function

Figure shows the STREAMS-based version of the s_pipe function. This version simply calls the standard pipe function, which creates a full-duplex pipe.

Figure 17.6. STREAMS version of the s_pipe function

```
#include "apue.h"
/*
 * Returns a STREAMS-based pipe, with the two file descriptors
 * returned in fd[0] and fd[1].
 */
int
s_pipe(int fd[2])
{
    return(pipe(fd));
}
```

Naming STREAMS Pipes

Normally, pipes can be used only between related processes: child processes inheriting pipes from their parent processes., here we saw that unrelated processes can communicate using FIFOs, but this provides

only a one-way communication path. The STREAMS mechanism provides a way for processes to give a pipe a name in the file system. This bypasses the problem of dealing with unidirectional FIFOs.

We can use the fattach function to give a STREAMS pipe a name in the file system.

```
#include <stropts.h>

int fattach(int filedes, const char *path);
```

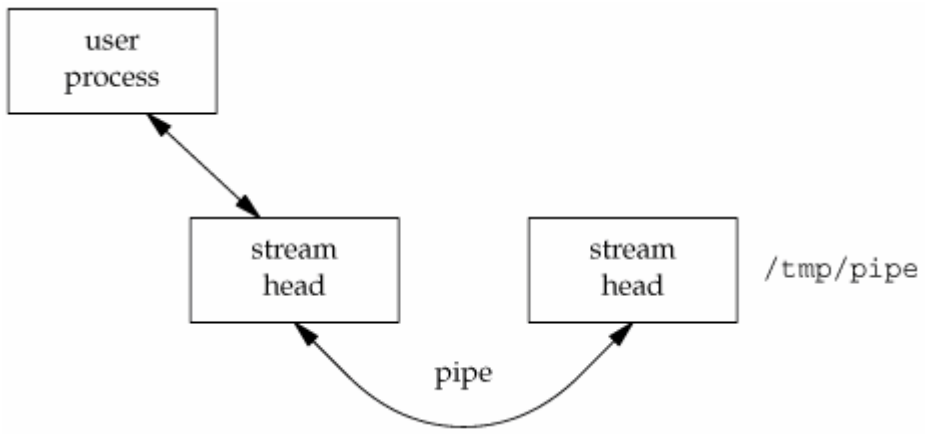
Returns: 0 if OK, 1 on error

The *path* argument must refer to an existing file, and the calling process must either own the file and have write permissions to it or be running with superuser privileges.

Once a STREAMS pipe is attached to the file system namespace, the underlying file is inaccessible. Any process that opens the name will gain access to the pipe, not the underlying file. Any processes that had the underlying file open before fattach was called, however, can continue to access the underlying file. Indeed, these processes generally will be unaware that the name now refers to a different file.

Figure shows a pipe attached to the pathname /tmp/pipe. Only one end of the pipe is attached to a name in the file system. The other end is used to communicate with processes that open the attached filename. Even though it can attach any kind of STREAMS file descriptor to a name in the file system, the fattach function is most commonly used to give a name to a STREAMS pipe.

Figure . A pipe mounted on a name in the file system



A process can call fdetach to undo the association between a STREAMS file and the name in the file system.

```
#include <stropts.h>

int fdetach(const char *path);
```

Returns: 0 if OK, 1 on error

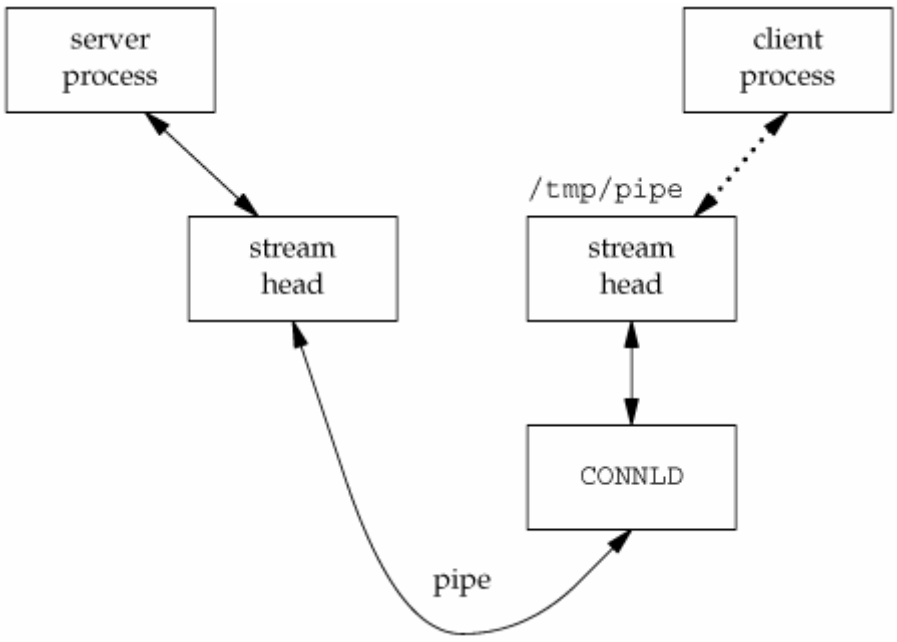
After fdetach is called, any processes that had accessed the STREAMS pipe by opening the *path* will still continue to access the stream, but subsequent opens of the *path* will access the original file residing in the file system.

Unique Connections

Although we can attach one end of a STREAMS pipe to the file system namespace, we still have problems if multiple processes want to communicate with a server using the named STREAMS pipe. Data from one client will be interleaved with data from the other clients writing to the pipe. Even if we guarantee that the clients write less than PIPE_BUF bytes so that the writes are atomic, we have no way to write back to an individual client and guarantee that the intended client will read the message. With multiple clients reading from the same pipe, we cannot control which one will be scheduled and actually read what we send.

The connld STREAMS module solves this problem. Before attaching a STREAMS pipe to a name in the file system, a server process can push the connld module on the end of the pipe that is to be attached. This results in the configuration shown in Figure .

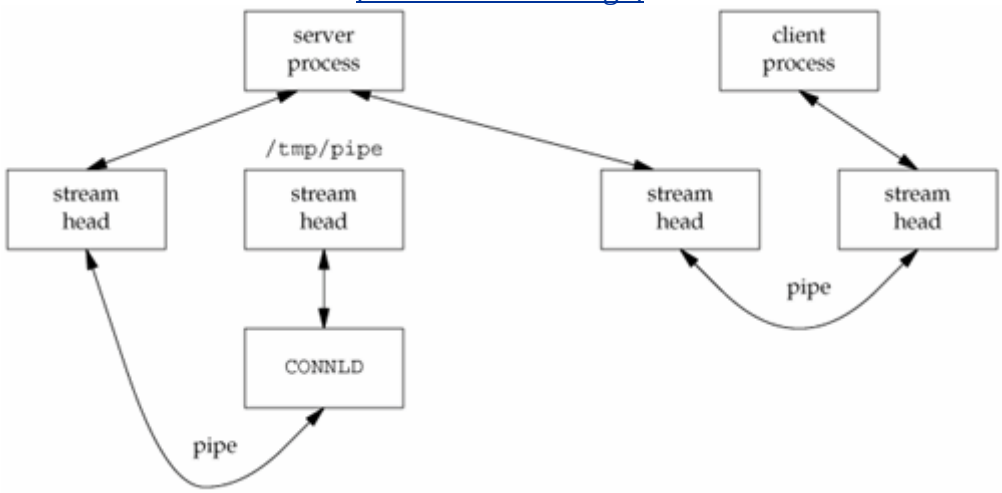
Figure . Setting up connld for unique connections



In Figure , the server process has attached one end of its pipe to the path /tmp/pipe. We show a dotted line to indicate a client process in the middle of opening the attached STREAMS pipe. Once the open completes, we have the configuration shown in Figure .

Figure .Using connld to make unique connections

[View full size image]



The client process never receives an open file descriptor for the end of the pipe that it opened. Instead, the operating system creates a new pipe and returns one end to the client process as the result of opening /tmp/pipe. The system sends the other end of the new pipe to the server process by passing its file descriptor over the existing (attached) pipe, resulting in a unique connection between the client process and the server process. We'll see the mechanics of passing file descriptors using STREAMS pipes .
The fattach function is built on top of the mount system call. This facility is known as *mounted streams*. Mounted streams and the connld module were developed by Presotto and Ritchie [1990] for the Research UNIX system. These mechanisms were then picked up by SVR4.

We will now develop three functions that can be used to create unique connections between unrelated processes. These functions mimic the connection-oriented socket functions discussed in . We use STREAMS pipes for the underlying communication mechanism here, but we'll see alternate implementations of these functions that use UNIX domain sockets.

```
#include "apue.h"

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

Returns: file descriptor to listen on if OK, negative value on error

```
int serv_accept(int listenfd, uid_t *uidptr);
```

Returns: new file descriptor if OK, negative value on error

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

Returns: file descriptor if OK, negative value on error

The serv_listen function can be used by a server to announce its willingness to listen for client connect requests on a well-known name (some pathname in the file system). Clients will use this name when they want to connect to the server. The return value is the server's end of the STREAMS pipe.

Figure 17.10. The serv_listen function using STREAMS pipes

```
#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.
 * Returns fd if all OK, <0 on error
 */
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 */
```

```
}
```

The serv_accept function is used by a server to wait for a client's connect request to arrive. When one arrives, the system automatically creates a new STREAMS pipe, and the function returns one end to the server. Additionally, the effective user ID of the client is stored in the memory to which *uidptr* points.

Figure. The serv_accept function using STREAMS pipes

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

/*
 * Wait for a client connection to arrive, and accept it.
 * We also obtain the client's user ID.
 * Returns new fd if all OK, <0 on error.
 */
int
serv_accept(int listenfd, uid_t *uidptr)
{
    struct strrecvfd    recvfd;
    if (ioctl(listenfd, I_RECVFD, &recvfd) < 0)
        return(-1);     /* could be EINTR if signal caught */
    if (uidptr != NULL)
        *uidptr = recvfd.uid;   /* effective uid of caller */
    return(recvfd.fd);  /* return the new descriptor */
}
```

A client calls cli_conn to connect to a server. The *name* argument specified by the client must be the same name that was advertised by the server's call to serv_listen. On return, the client gets a file descriptor connected to the server.

Figure. The cli_conn function using STREAMS pipes

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

/*
 * Create a client endpoint and connect to a server.
 * Returns fd if all OK, <0 on error.
 */
int
cli_conn(const char *name)
{
    int     fd;

    /* open the mounted stream */
    if ((fd = open(name, O_RDWR)) < 0)
```

```
        return(-1);
    if (isastream(fd) == 0) {
        close(fd);
        return(-2);
    }
    return(fd);
}
```

We double-check that the returned descriptor refers to a STREAMS device, in case the server has not been started but the pathname still exists in the file system., we'll see how these three functions are used.