MODULE-4

Chapter 8 Process Control

Topics

Process Control Continuation:
Changing User IDs and Group IDs, Interpreter files, System function, Process Accounting, User Identification, Process Times, I/O Redirection.
IPC Methods:
Pipes, popen, pclose functions, Coprocesses, FIFOs, System V IPC, Message Queues, Semaphores.
Shared Memory:
Client-Server Properties, Stream Pipes, Passing File Descriptors, An Open Server-Version 1, Client-Server Functions.

8.11 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.

```
#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. Let's consider only the user ID for now. (Everything we 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 1 is returned.

We can make a 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(1) 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 execstores the effective user ID from the file's user ID.

ID	exec		setuid(uid)	
	set-user-ID bit off	set-user-ID bit on	superuser	Unprivileged user
real user ID	unchanged	unchanged	set to uid	unchanged
effective user ID	unchanged	set from user ID of program file	set to uid	set to uid
saved set-user ID	copied from effective user ID	copied from effective user ID	set to uid	unchanged

The above table summarizes the various ways these three user IDs can be changed.

setreuid and setregid Functions

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

```
#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.

```
#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.)

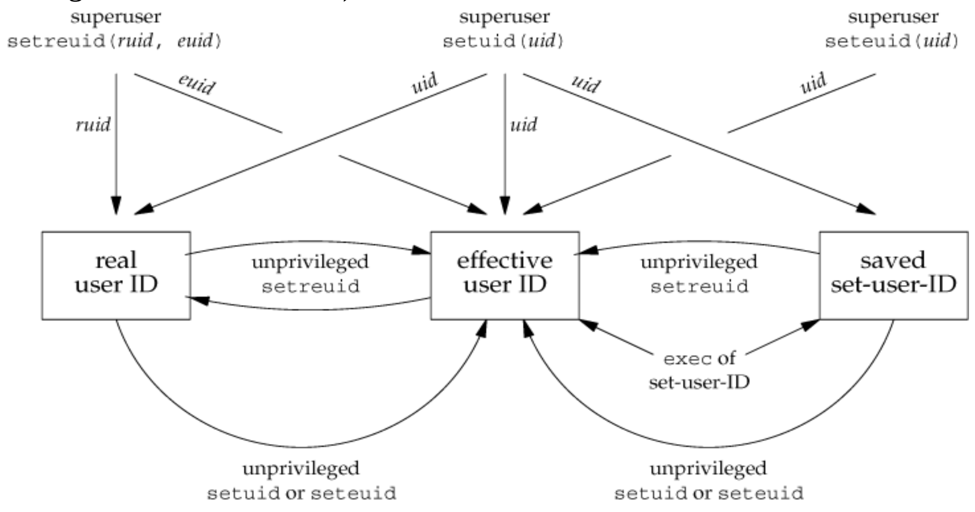


Figure: Summary of all the functions that set the various user Ids

8.12 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);
}
```

8.13 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.

Program: The system function, without signal handling

```
#include<sys/wait.h>
#include<errno.h>
#include<unistd.h>
int system(const char *cmdstring)       /* version without signal handling */
{
       pid_t   pid;
       int     status;
       if (cmdstring == NULL)
       return(1);    /* always a command processor with UNIX */

       if ((pid = fork()) < 0) {
       status = -1; /* probably out of processes */
       } else if (pid == 0) {     /* child */
       execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
       exit(127);    /* execl error */
       } else {/* parent */
       while (waitpid(pid, &status, 0) < 0) {
               if (errno != EINTR) {
                    status = -1; /* error other than EINTR from waitpid() */
                    break;
                    }
               }
       }

       return(status);
}
```

Program: Calling the system function

```
#include "apue.h"
#include <sys/wait.h>
int main(void)
{
       int status;

       if ((status = system("date")) < 0) err_sys("system() error");
       pr_exit(status);

       if ((status = system("nosuchcommand")) < 0) err_sys("system() error");
       pr_exit(status);

       if ((status = system("who; exit 44")) < 0) err_sys("system() error");
       pr_exit(status);

       exit(0);
}
```

Program: Execute the command-line argument using system

```
#include "apue.h"
int main(int argc, char *argv[])
{
       int     status;
       if (argc < 2)
       err_quit("command-line argument required");
       if ((status = system(argv[1])) < 0) err_sys("system() error");
       pr_exit(status);
       exit(0);
}
```

Program: Print real and effective user IDs

```
#include "apue.h"
int main(void)
```

```
{
        printf("real uid = %d, effective uid = %d\n", getuid(), geteuid());
        exit(0);
}
```

8.14 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 a small amount 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 superuser executes accton with a pathname argument to enable accounting.
- The accounting records are written to the specified file, which is usually /var/account/acct. Accounting is turned off by executing accton without any arguments.
- The data required for the accounting record, such as CPU times and number of characters transferred, is kept by the kernel in the process table and initialized whenever a new process is created, as in the child after a fork.
- 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;   /* 3-bit base 8 exponent; 13-bit fraction */
struct  acct
{
  char   ac_flag;     /* flag  */
  char   ac_stat;     /* termination status (signal & core flag only) */
                      /*(Solaris 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) */
                      /* "blocks" on BSD systems */
  comp_t ac_rw;       /* blocks read or written */
                      /* (not present on BSD systems) */
  char   ac_comm[8];  /* command name: [8] for Solaris, */
                      /* [10] for Mac OS X, [16] for FreeBSD, and */
                      /* [17] for Linux */
};
```

Values for `ac_flag` from accounting record

ac_flag	Description
AFORK	process is the result of `fork`, but never called `exec`

ASU	process used superuser 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

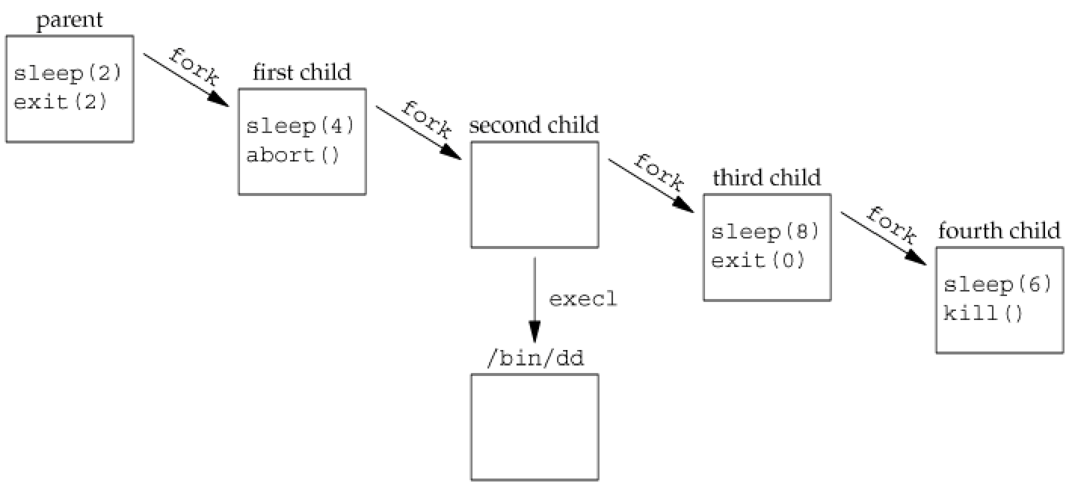
```
#include "apue.h"
int main(void)
{
    pid_t  pid;
    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {        /* parent */
        sleep(2);
        exit(2);                /* terminate with exit status 2 */
    }

                                /* 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 */
}
```



Process structure for accounting example

8.15 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 getloginfunction 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.

- *getlogin* can fail if the process is not attached to a terminal that a user logged in to.
- These types of processes are called *daemons*.
- Given login name, we can use the information to look up the user in the password file using *getpwnam*.
- Example – to determine the login shell.
- LOGNAME environment variable is initialized with user's login name by *login* and inherited by the login shell.
- But it can be modified by user, so *getlogin* should be used instead.

8.16 PROCESS TIMES

We describe three times that we can measure: wall clock time, user CPU time, and system CPU time. Any process can call the timesfunction to obtain these values for itself and any terminated children.

```
#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 tmsstructure pointed to by buf:

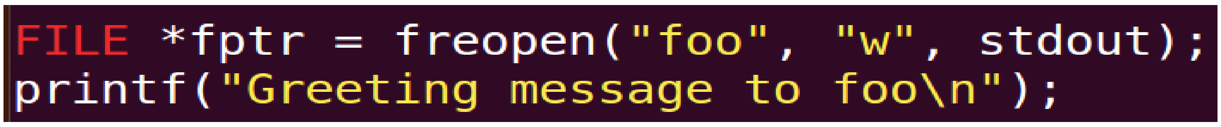
```
struct tms {
clock_t tms_utime;    /* user CPU time */
clock_t tms_stime;    /* system CPU time */
clock_t tms_cutime; /* user CPU time, terminated children */
clock_t tms_cstime; /* system CPU time, terminated children */
};
```

- 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 it's 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. *Example*: call *times* and save the return value. At a later time, call *times* again and subtract the earlier return value from the new value. Difference between the two times = wall clock time. Two structure fields for child processes (*tms_cutime*, *tms_cstime*) contain values only for child processes that were waited for using *wait*, *waitid*, or *waitpid*. All *clock_t* values returned by *times* function are converted to seconds using the number of clock ticks per second – the _SC_CLK_TCK value returned by *sysconf*.

8.17 I/O REDIRECTION

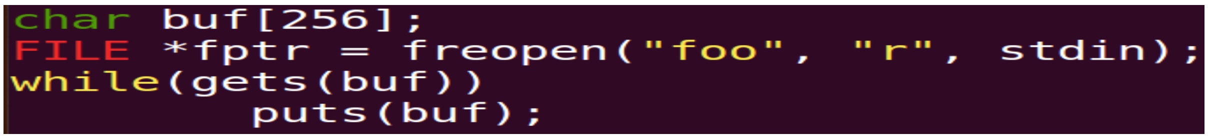
Process can use the C library function *reopen* to change its standard input and standard output ports to refer to text files instead of the console.

- Example to change process standard output to file *foo*:



```
FILE *fptr = freopen("foo", "w", stdout);
printf("Greeting message to foo\n");
```

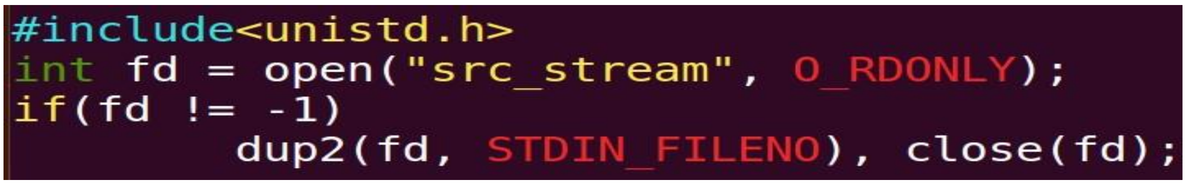
- Example to change process standard input to file *foo*:



```
char buf[256];
FILE *fptr = freopen("foo", "r", stdin);
while(gets(buf))
          puts(buf);
```

- *freopen* function relies on *open* and *dup2* system calls to do redirection of standard input or standard output.

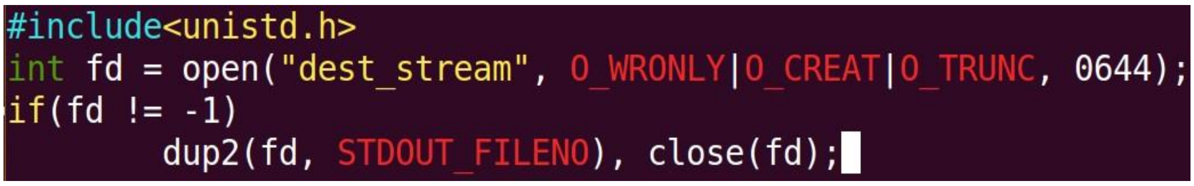
- To redirect standard input of a process from file *src_stream:*



```
#include<unistd.h>
int fd = open("src_stream", O_RDONLY);
if(fd != -1)
        dup2(fd, STDIN_FILENO), close(fd);
```

- *src_stream* file is now referenced by the STDIN_FILENO descriptor of the process.

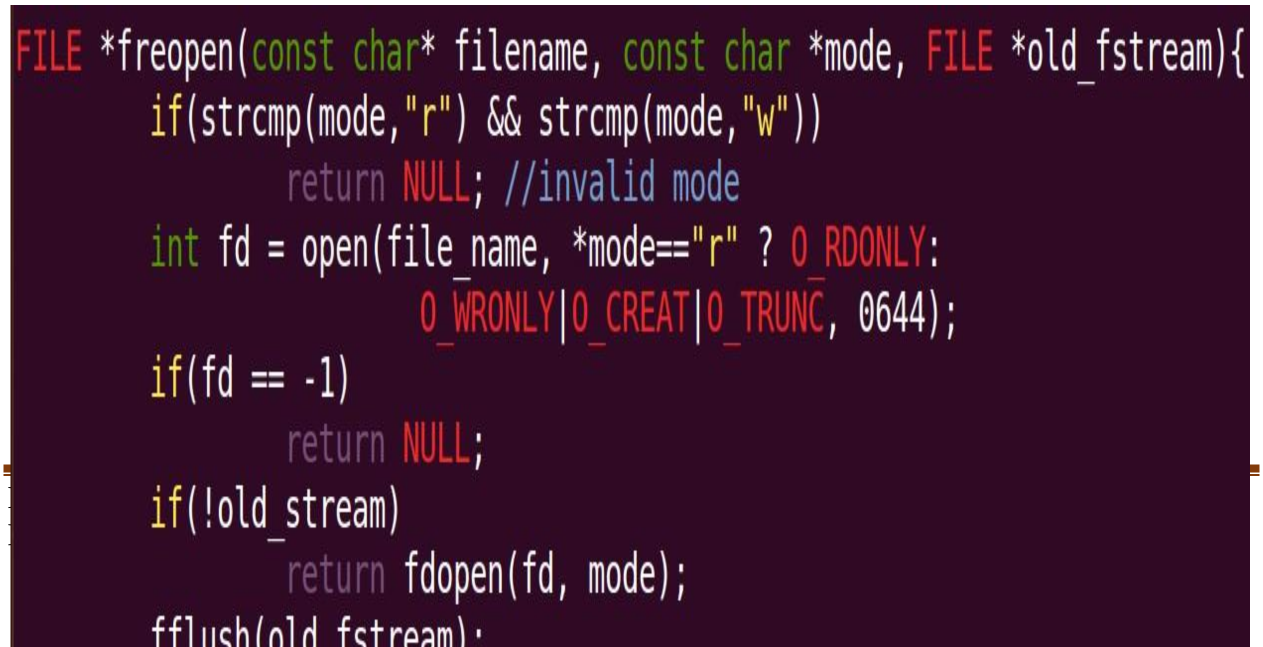
- To redirect standard output of a process to file *dest_stream:*



```
#include<unistd.h>
int fd = open("dest_stream", O_WRONLY|O_CREAT|O_TRUNC, 0644);
if(fd != -1)
        dup2(fd, STDOUT_FILENO), close(fd);
```

- *dest_stream* file is now referenced by the STDOUT_FILENO descriptor of the process.

Implementation of *freopen* function



```
FILE *freopen(const char* filename, const char *mode, FILE *old_fstream){
        if(strcmp(mode,"r") && strcmp(mode,"w"))
                return NULL; //invalid mode
        int fd = open(file_name, *mode=="r" ? O_RDONLY:
                        O_WRONLY|O_CREAT|O_TRUNC, 0644);
        if(fd == -1)
                return NULL;
        if(!old_stream)
                return fdopen(fd, mode);
        fflush(old_fstream);
```

Chapter 9 Overview of IPC Methods

9.1 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.

PIPES

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

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.

A pipe is created by calling the pipefunction.

```
#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].

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

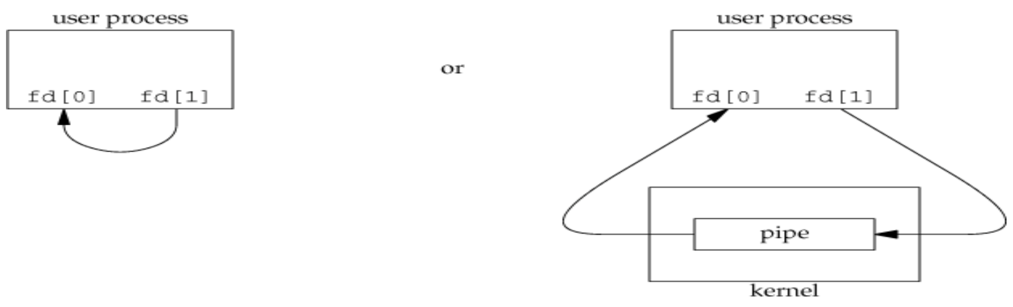


Figure: Two ways to view a half-duplex pipe

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 or vice versa. Below Figure shows this scenario.

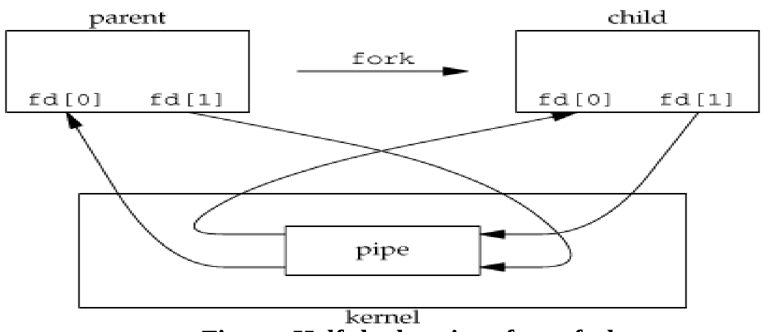


Figure: Half-duplex pipe after a fork

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 shows the resulting arrangement of descriptors.

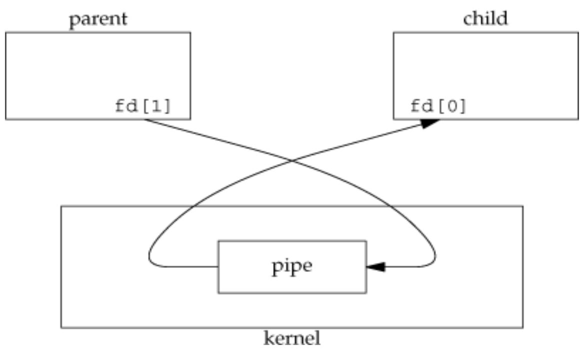


Figure: Pipe from parent to child

For a pipe from the child to the parent, the parent closes fd[1], and the child closes fd[0]. When one end of a 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 SIGPIPEis generated. If we either ignore the signal or catch it and return from the signal handler, writereturns 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.

```
#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);
}
```

9.2 popen AND pclose FUNCTIONS

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, NULLon error

```
int pclose(FILE *fp);
```

Returns: termination status of cmdstring, or 1 on error

The function popendoes a forkand execto 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.

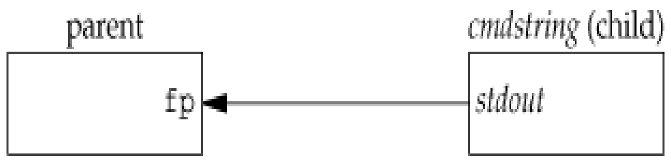


Figure: Result of fp = popen(cmdstring, "r")

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

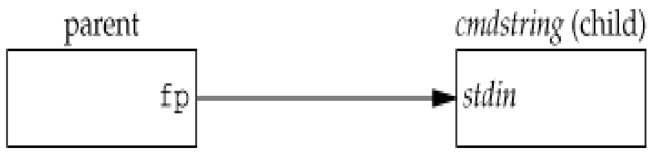


Figure: Result of fp = popen(cmdstring, "w")

9.3 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'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. Figure shows this arrangement.

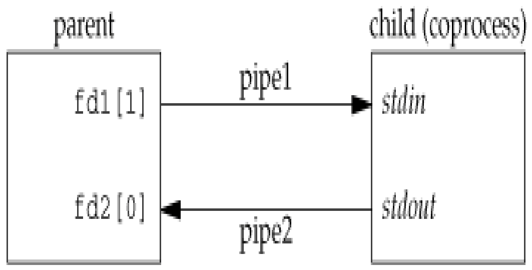


Figure: Driving a coprocess by writing its standard input and reading its standard output

Program: Simple filter to add two numbers

```
#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_FILEN
                O, line, n) != n)
                err_sys("write
                error");
        } else {
            if (write(STDOUT_FILENO, "invalid
                args\n", 13) != 13)
                err_sys("write error");
        }
    }
    exit(0);
}
```

9.4 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_NONBLOCKis specified, an openfor read-only returns immediately. But an openfor write-only returns 1 with errnoset 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 and the 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 to process a filtered input stream twice. Figure shows this arrangement.

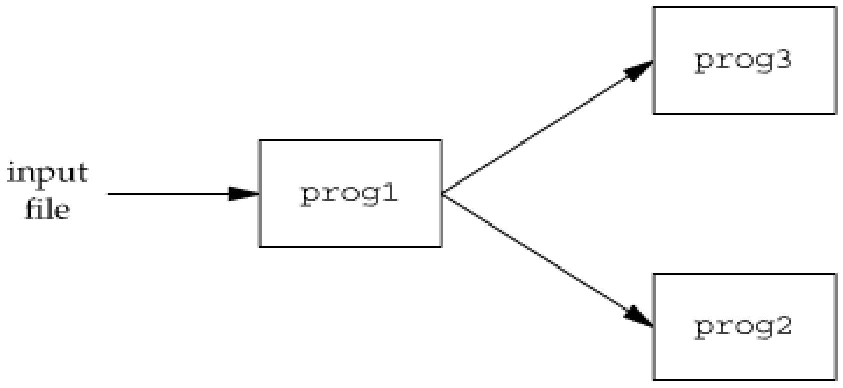


FIGURE: Procedure that processes a filtered input stream twice

With a FIFO and the UNIX program tee(1), we can accomplish this procedure without using a temporary file. (The teeprogram 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 teeto send its input to both the FIFO and prog2. Figure shows the process arrangement.

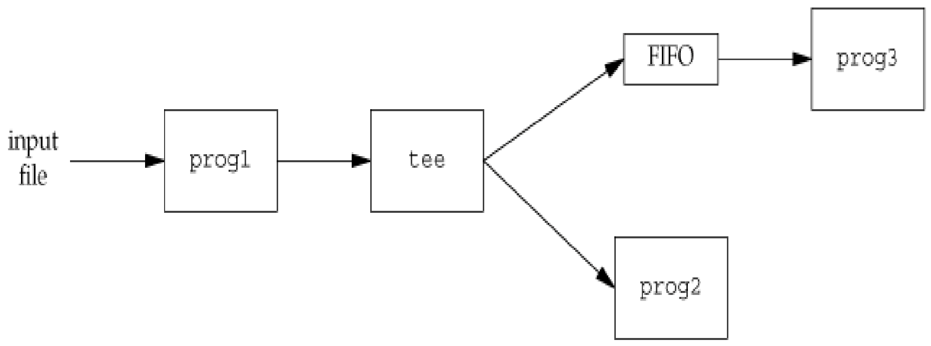


FIGURE: Using a FIFO and teeto send a stream to two different processes

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's process 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 crash. 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.

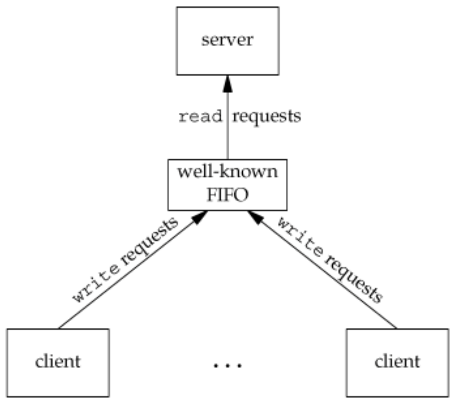


Figure: Clients sending requests to a server using a FIFO

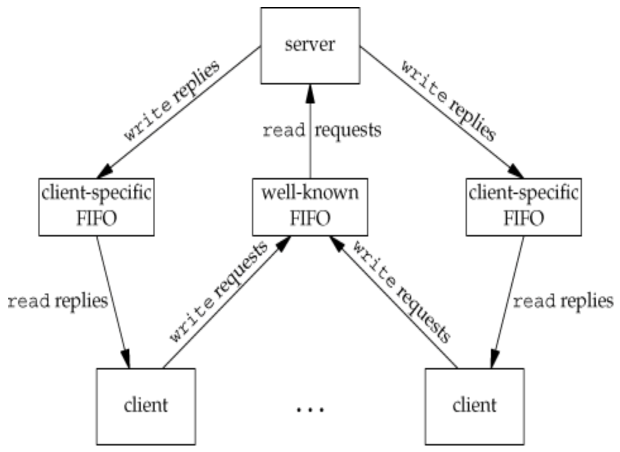


Figure: Client-server communication using FIFOs

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

Figure: XSI IPC permissions

Configuration Limits

All three forms of XSI IPC have built-in limits that we may encounter. Most of these limits can be changed by reconfiguring the kernel. We describe the limits when we describe each of the three forms of IPC.

Advantages and Disadvantages

- ➢ A fundamental problem with XSI 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, 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 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 busywait loop.

9.5 SYSTEM V IPC

Linux supports three types of interprocess communication mechanisms which first appeared in Unix System V (1983). These are message queues, semaphores and shared memory. These System V IPC mechanisms all share common authentication methods. Processes may access these resources only by passing a unique reference identifier to the kernel via system calls. Access to these System V IPC objects is checked using access permissions, much like accesses to files are checked. The access rights to the System V IPC object is set by the creator of the object via system calls. The object's reference identifier is used by each mechanism as an index into a table of resources. It is not a straight forward index but requires some manipulation to generate the index. All Linux data structures representing System V IPC objects in the system include an ipc_perm structure which contains the owner and creator processes user and group identifiers. the access mode for this object (owner, group and other) and the IPC object's key. The key is used as a way of locating the System V IPC object's reference identifier. Two sets of keys are supported: public and private. If the key is public then any process in the system, subject to rights checking, can find the reference identifier for the System V IPC object. System V IPC objects can never be referenced with a key, only by their reference identifier.

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 msgsndwhen 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_ds structure associated with it:

```
struct msqid_ds
{
    struct ipc perm  msg perm;     /* see Section 15.6.2 */
    msgqnum t        msg qnum;     /* # of messages on queue */
    msglen t         msg qbytes;   /* max # of bytes on queue */
    pid t            msg lspid;    /* pid of last msgsnd() */
    pid_t            msg_lrpid;    /* pid of last msgrcv() */
    time_t           msg_stime;    /* last-msgsnd() time */
    time_t           msg_rtime;    /* last-msgrcv() time */
    time_t           msg_ctime;    /* last-change time */
 .
 .
 .
};
```

This structure defines the current status of the queue.

The first function normally called is msggetto 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 corresponding permission 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 three message queue functions.

The msgctlfunction 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 */
 };
```

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

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.
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;  /* see
     Section 15.6.2 */ unsigned short
                      sem_nsems; /* # of
     semaphores in set */ time_t sem_otime;
     /* last-semop() time */
     time_t           sem_ctime; /* last-change time */
     .
     .
     .
   };
```

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 */
      .
      .
      .
   };
```

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);
```
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.

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.

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 */
    };
```

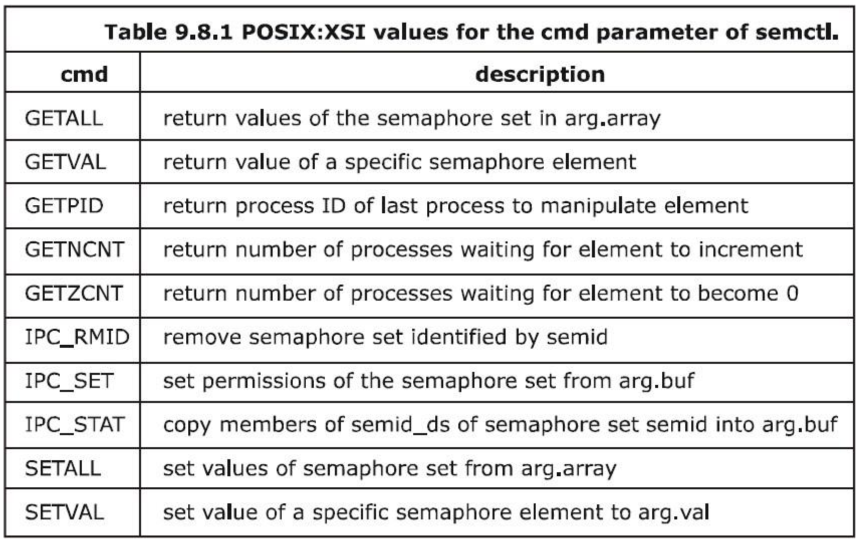


Table 9.8.1 POSIX:XSI values for the cmd parameter of semctl.

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 semopatomically 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 semop array 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.

Chapter 10 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 among multiple processes. If the server is placing data into a shared memory region, the client shouldn't try to access the data until the server is done. Often, semaphores are used to synchronize shared memory access.

The kernel maintains a structure with at least the following members for each shared memory segment:

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

The type shmatt_t is defined to be an unsigned integer at least as large as an unsigned short.

The first function called is usually shmget, to obtain a shared memory identifier.

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

Returns: shared memory ID if OK, -1 on error

we described the rules for converting the *key* into an identifier and whether a new segment is created or an existing segment is referenced. When a new segment is created, the following members of the shmid_ds structure are initialized.

- The ipc_perm structure is initialized and described previously. The mode member of this structure is set to the corresponding permission bits of *flag*.
  - shm_lpid, shm_nattch, shm_atime, and shm_dtime are all set to 0.
  - shm_ctimeis set to the current time.
  - shm_segszis set to the *size* requested.

The *size* parameter is the size of the shared memory segment in bytes. Implementations will usually round up this size to a multiple of the system's page size, but if an application specifies *size* as a value other than an integral multiple of the system's page size, the remainder of the last page will be unavailable for use. If a new segment is being created (typically by the server), we must specify its *size*.

If we are referencing an existing segment (a client), we can specify *size* as 0. When a new segment is created, the contents of the segment are initialized with zeros. 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 )
```

Returns: 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*.

IPC_STAT	Fetch the shmid_ds structure for this segment, storing it in the structure pointed to by *buf*.
IPC_SET	Set the following three fields from the structure pointed to by *buf* in the shmid_ds structure associated with this shared memory segment: shm_perm.uid, shm_perm.gid, and shm_perm.mode. This command can be executed only by a process whose effective user ID equals shm_perm.cuid or shm_perm.uid or by a process with superuser privileges.
IPC_RMID	Remove the shared memory segment set from the system. Since an attachment count is maintained for shared memory segments (the shm_nattch field in the shmid_ds structure), the segment is not removed until the last process using the segment terminates or detaches it.Regardless of whether the segment is still in use, the segment's identifier is immediately removed so that shmat can no longer attach the segment. This command can be executed only by a process whose effective user ID equals shm_perm.cuid or shm_perm.uid or by a process with superuser privileges.

Two additional commands are provided by Linux and Solaris, but are not part of the Single UNIX Specification.

- ➢ SHM_LOCK Lock the shared memory segment in memory. This command can be executed only by the superuser.
- ➢ 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, const 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*.

➢ If *addr* is 0, the segment is attached at the first available address selected by the kernel. This is the recommended technique.

➢ If *addr* is nonzero and SHM_RND is not specified, the segment is attached at the address given by *addr*.

➢ If *addr* is nonzero and SHM_RND is specified, the segment is attached at the address given by (*addr* -(*addr* modulus SHMLBA)). The SHM_RND command stands for ‘‘round.’’ SHMLBA stands for ‘‘low boundary address multiple’’ and is always a power of 2. What the arithmetic does is round the address down to the next multiple of SHMLBA.

- If the SHM_RDONLY bit is specified in *flag*, the segment is attached as read-only. Otherwise, the segment is attached as read–write.
- The value returned by shmat is the address at which the segment is attached, or -1 if an error occurred.

- If shmat succeeds, the kernel will increment the shm_nattch counter in the shmid_ds structure associated with the shared memory segment.

When we're done with a shared memory segment, we call shmdt to detach it. 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.

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

Returns: 0 if OK, -1 on error

The *addr* argument is the value that was returned by a previous call to shmat. If successful, shmdt will decrement the shm_nattch counter in the associated shmid_ds structure.

10.1 CLIENT SERVER PROPERTIES

Some of the properties of clients and servers that are affected by the various types of IPC used between them.

The simplest type of relationship is to have the client fork and exec the desired server. Two half-duplex pipes can be created before the fork to allow data to be transferred in both directions. The server that is executed can be a set-user-ID program, giving it special privileges. Also, the server can determine the real identity of the client by looking at its real user ID.

With this arrangement, we can build an ***open server***. It opens files for the client instead of the client calling the open function. This way, additional permission checking can be added, above and beyond the normal UNIX system user/group/other permissions. We assume that the server is a set-user-ID program, giving it additional permissions (root permission, perhaps). The server uses the real user ID of the client to determine whether to give it access to the requested file. This way, we can build a server that allows certain users permissions that they don't normally have.

In this example, since the server is a child of the parent, all the server can do is pass back the contents of the file to the parent. Although this works fine for regular files, it can't be used for special device files, for example. We would like to be able to have the server open the requested file and pass back the file descriptor. Whereas a parent can pass a child an open descriptor, a child cannot pass a descriptor back to the parent.

The server is a daemon process that is contacted using some form of IPC by all clients. We can't use pipes for this type of clientserver. A form of named IPC is required, such as FIFOs or message queues. With FIFOs, we saw that an individual per client FIFO is also required if the server is to send data back to the client. If the clientserver 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. For example, the clients can send their requests with a type field of 1. Included in the request must be the client's process ID. The server then sends the response with the type field set to the client's process ID. The server receives only the messages with a type field of 1 (the fourth argument for msgrcv), and the clients receive only the messages with a type field equal to their process IDs.
2. Alternatively, 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. The client sends its first request to the server's well-known queue, and this request must contain the message queue ID of the client's queue. The server sends its first response to the client's queue, and all future requests and responses are exchanged on this queue.

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. This seems wasteful of a limited system wide resource (a message queue), and a FIFO can be used instead. Another problem is that the server has to read messages from multiple queues. Neither select nor poll works with message queues.

Either of these two techniques using message queues can be implemented using shared memory segments and a synchronization method.

10.2 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.

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

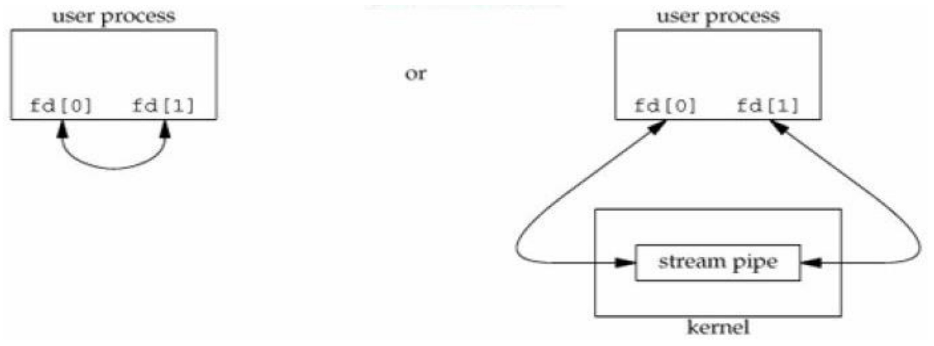


Figure: Two ways to view the stream pipes

If we look inside a STREAMS pipe (Below Figure), 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.

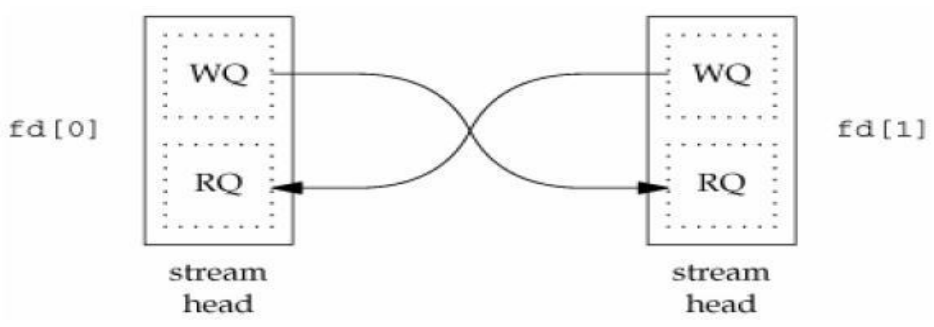


Figure: Inside a STREAMS pipe

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 (Below Figure). 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.

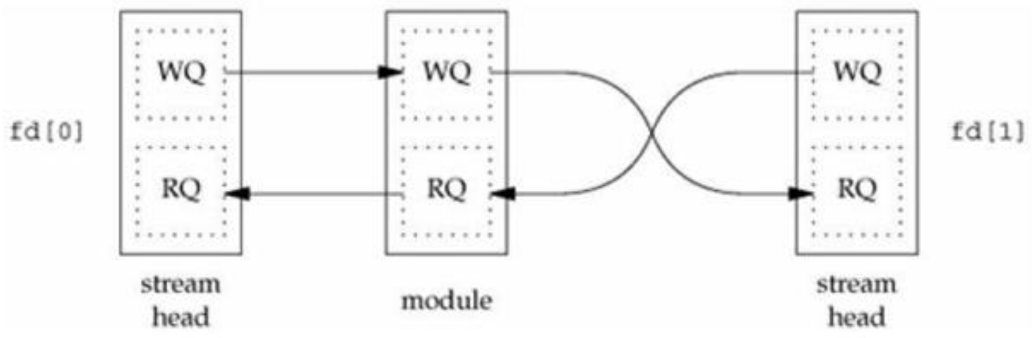


Figure: Inside a STREAMS pipe with a module

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.

Naming Streams Pipes

Normally, pipes can be used only between related processes: child processes inheriting pipes from their parent processes. In Section 15.5, 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 bypass 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.

Below 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.

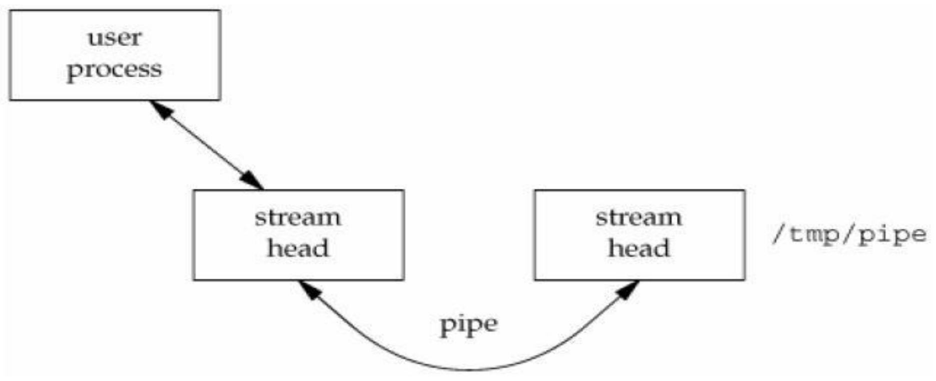


Figure: A pipe mounted on a name in the file system

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 below figure.

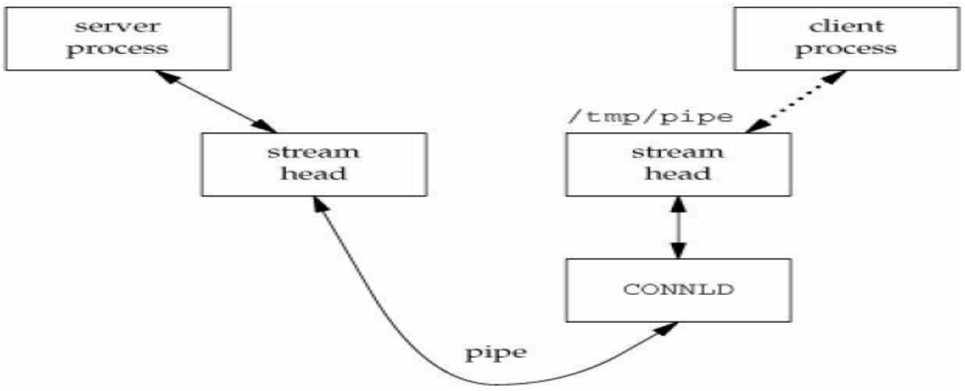


Figure: Setting up connld for unique connections

In above 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 below Figure.

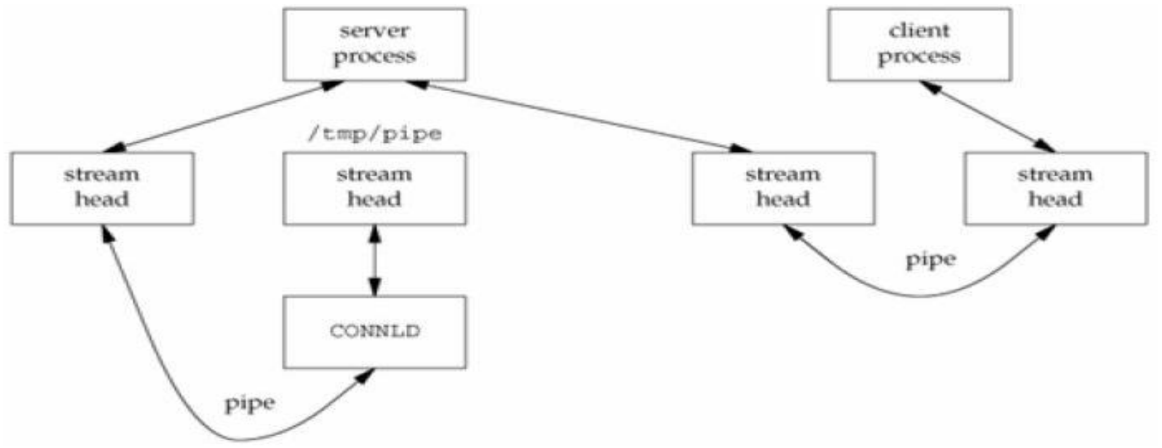


Figure: Using connld to make unique connections

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 descriptor 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.

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 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.

10.3 PASSING FILE DESCRIPTORS

The ability to pass an open file descriptor between processes is powerful. It can lead to different ways of designing clientserver applications. It allows one process (typically a server) to do everything that is required to open a file (involving such details as translating a network name to a network address, dialing a modem, negotiating locks for the file, etc.) and simply pass back to the calling process a 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.

We must be more specific about what we mean by "passing an open file descriptor" from one process to another. Although they share the same v-node, each process has its own file table entry. When we pass an open file descriptor from one process to another, we want the passing process and the receiving process to share the same file table entry. Below figure shows the desired arrangement.

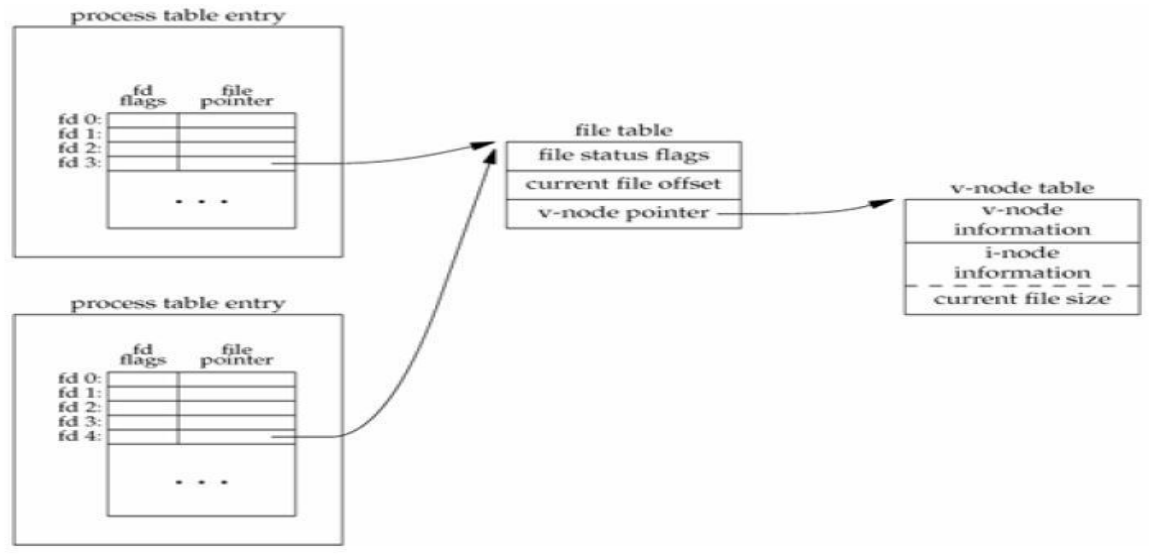


Figure: Passing an open file from the top process to the bottom process

Technically, we are passing a pointer to an open file table entry from one process to another. This pointer is assigned the first available descriptor in the receiving process. (Saying that we are passing an open descriptor mistakenly gives the impression that the descriptor number in the receiving process is the same as in the sending process, which usually isn't true.) Having two processes share an open file table is exactly what happens after a fork.

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

We define the following three functions that we use in this chapter to send and receive file descriptors. Later in this section, we'll show the code for these three functions for both STREAMS and sockets.

```
#include "apue.h"
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, 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*.
- ➢ We'll use the term *s-pipe* to refer to a bidirectional communication channel that could beimplemented as either a STREAMS pipe or a UNIX domain stream socket.
- ➢ 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). Additionally, if an error message was sent by the server, the client's *userfunc* is called to process the message. The first argument to *userfunc* is the constant STDERR_FILENO, followed by a pointer to the error message and its length. The return value from *userfunc* is the number of bytes written or a negative number on error. Often, the client specifies the normal write function as the *userfunc*.
- ➢ We implement our own protocol that is used by these three functions. To send a descriptor, send_fd sends two bytes of 0, followed by the actual descriptor. To send an error, send_err sends the *errmsg*, followed by a byte of 0, followed by the absolute value of the *status* byte (1 through 255).
- ➢ The recv_fd function reads everything on the s-pipe until it encounters a null byte. Any characters readup to this point are passed to the caller's *userfunc*. The next byte read by recv_fd is the status byte.
- ➢ If the status byte is 0, a descriptor was passed; otherwise, there is no descriptor to receive.
- ➢ The function send_err calls the send_fd function after writing the error message to the s-pipe.

10.4 AN OPEN SERVER, VERSION 1

Using file descriptor passing, we now develop an open server: 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. This lets the server work with any type of file (such as a device or a socket) and not simply regular files. It also means that a minimum of information is exchanged using IPC: the filename and open mode from the client to the server, and the returned descriptor from the server to the client. The contents of the file are not exchanged using IPC.

There are several advantages in designing the server to be a separate executable program
The server can easily be contacted by any client, similar to the client calling a library function.

- We are not hard coding a particular service into the application, but designing a general facility that others can reuse.

- If we need to change the server, only a single program is affected. Conversely, updating a library function can require that all programs that call the function be updated (i.e., relinked with the link editor). Shared libraries can simplify this updating.
- The server can be a set-user-ID program, providing it with additional permissions that the client does not have. Note that a library function (or shared library function) can't provide this capability.

The client process creates an s-pipe (either a STREAMS-based pipe or a UNIX domain socket pair) 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.

We define the following 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. The <*openmode*>is the numeric value, in ASCII decimal, of the second argument to the open function. This request string is terminated by a null byte.

The server sends back an open descriptor or an error by calling either send_fd or send_err.

2. This is an example of a process sending an open descriptor to its parent. We'll modify this example to use a single daemon server, where the server sends a descriptor to a completely unrelated process.

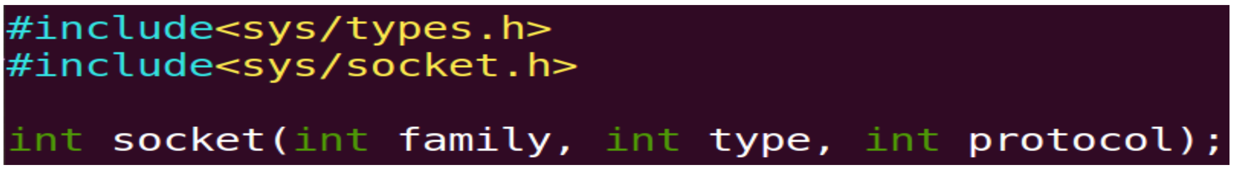
We first have the header, open.h, which includes the standard headers and defines the function prototypes

10.5 CLIENT-SERVER CONNECTION FUNCTIONS

Functions used in client-server communication:

- socket()
- connect()
- bind()
- listen()
- accept()
- send() / write()
- recv() / read()
- close()

- socket() - necessary to perform network communication. Mainly specifies protocol type and family used for communication.



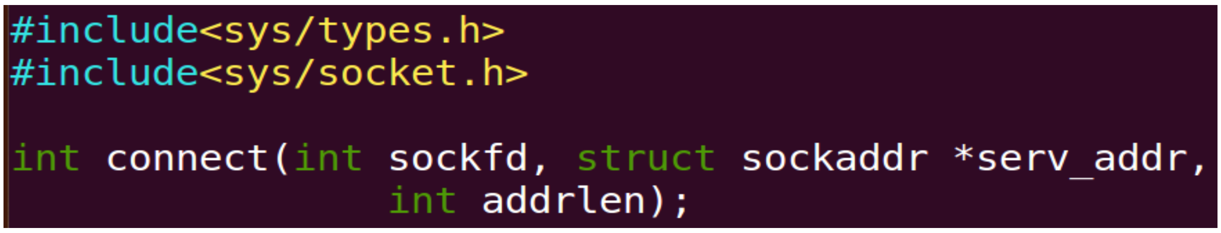
```
#include<sys/types.h>
#include<sys/socket.h>

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

Returns: socket descriptor if successful, -1 on error.

Parameters: family (protocol family), type (kind of socket), protocol (specific protocol type or 0 for system default)

- connect() - used by TCP client to establish connection with a TCP server.

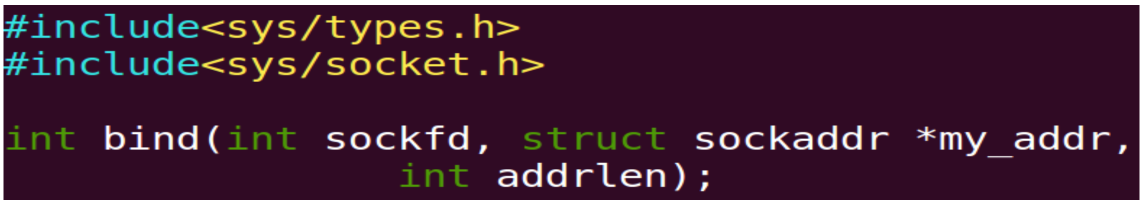


```
#include<sys/types.h>
#include<sys/socket.h>

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

Returns: 0 if successful, -1 on error.
Parameters: sockfd (socket descriptor), serv_addr (pointer to socket structure that contains destination IP and port), addrlen (size of socket structure)

- bind() - assigns local protocol address to a socket

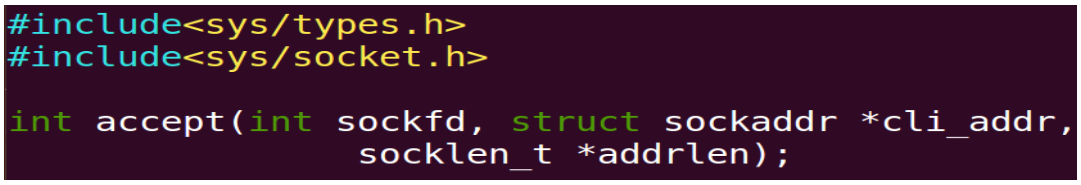


```
#include<sys/types.h>
#include<sys/socket.h>

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

Returns: 0 if successful, -1 on error.
Parameters: sockfd (socket descriptor), my_addr (pointer to socket structure that contains local IP and port), addrlen (size of socket structure)

- accept() - called by a TCP server to return the next completed connection from the front of the completed connection queue.

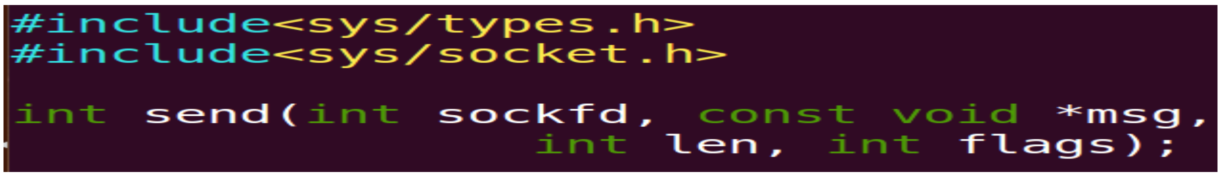


```
#include<sys/types.h>
#include<sys/socket.h>

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

Returns: non-negative if successful, -1 on error.
Parameters: sockfd (socket descriptor), cli_addr (pointer to socket structure that contains client IP and port), addrlen (size of socket structure)

- send() / write() - used to send data over stream sockets or connected datagram sockets.

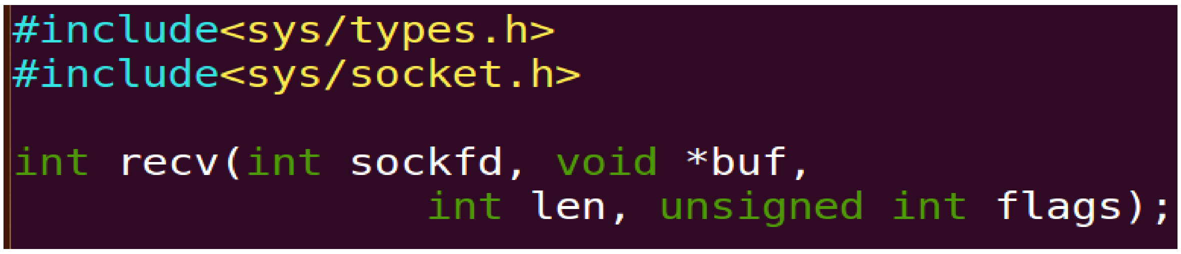


```
#include<sys/types.h>
#include<sys/socket.h>

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

Returns: number of bytes sent if successful, -1 on error.
Parameters: sockfd (socket descriptor), msg (pointer to data that must be sent), len (length of data to be sent), flags (set to 0)

- recv() / read() - used to receive data over stream sockets or connected datagram sockets.

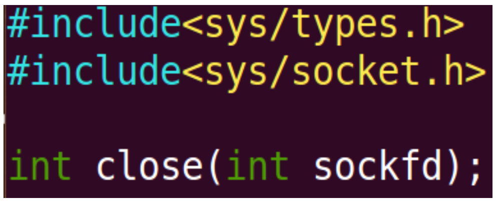


```c
#include<sys/types.h>
#include<sys/socket.h>

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

Returns: number of bytes read into the buffer if successful, -1 on error.
Parameters: sockfd (socket descriptor), buf (pointer to buffer that reads incoming information), len (maximum length of buffer), flags (set to 0)

- close() - used to close the communication between the client and the server



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

int close(int sockfd);
```

Returns: 0 if successful, -1 on error.
Parameters: sockfd (socket descriptor