UNIT – V INTER-PROCESS COMMUNICATION AND SOCKETS

Part- 1 INTER-PROCESS COMMUNICATION

Inter Process Communication (IPC)

The processes can exchange information by passing open files across a fork or an exec, or through the file system. But there are different types of interprocess communication (IPC), such as:

1. Pipes (half duplex)
2. FIFOs (named pipes)
3. Message queues
4. Semaphores
5. Shared memory
6. Sockets
7. Streams

The first five forms of IPC shown above are usually restricted to IPC between processes on the same host. The final two rows, sockets and stream for IPC between processes on different hosts.

Inter process communication is the generic term describing how two processes may exchange information with each other. In general, the two processes may be running on the same machine or on different machines, although same IPC mechanisms may support only local usage (ex: signals and pipes). IPC may be an exchange of data when in two or more processes are cooperatively processing the data or other synchronization information to help two independent bits related, processes schedule work so that they don't destructively overlap.

5.1 Pipes Definition

Pipes are the oldest form of Unix IPC and are provided by all UNIX systems. They have two limitations:

1. They are half-duplex data flows only in one direction.
2. They 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 child. Despite these limitations, half-duplex pipes are still the most commonly used form of IPC.

A pipe is created by calling the 'pipe' function.

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

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

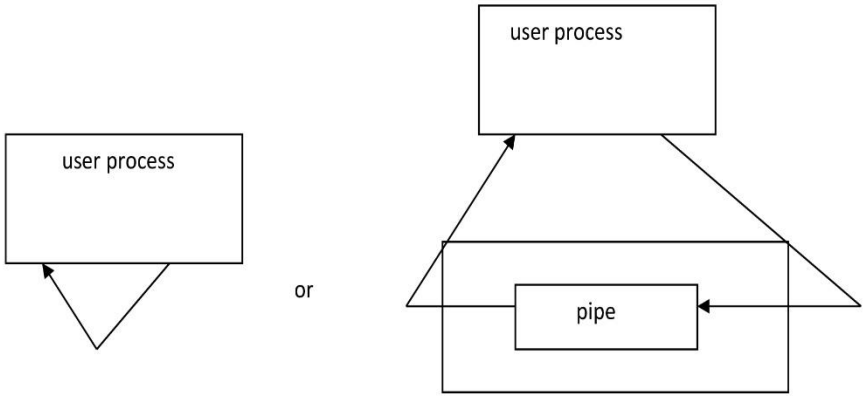
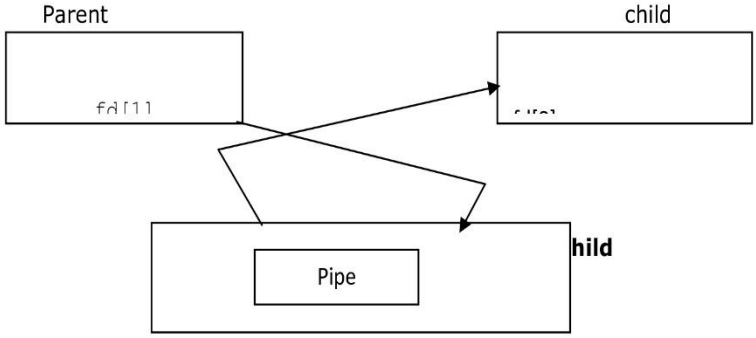


Fig 5.1: Two ways to view a UNIX pipe.

There are two ways to picture a pipe. The left half of the fig5.1 shows the two ends of the pipe connected in a single process. The right half of the figure reiterates the fact that the data in the pipe flows through the kernel.
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.



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 rules apply:

1. If we read from a pipe whose write end has been closed, after all the data has been read, read returns 0 to indicate an end of file.
2. If we write to a pipe whose read end has been closed, the signal SIGPIPE is generated. If we either ignore the signal or catch it and return from the signal or catch it and return from the signal handler, write returns an error with errno set to EPIPE.

When we are writing to a pipe (or FIFO) the constant PIPE-BUF specifies the kernel's pipe buffer size. A write of PIPE-BUF bytes or less will not be interleaved with writer from other processes to the same pipe. But if multiple processes are writing to a pipe and we write more than PIPE-BUF bytes, the data might be interleaved with the data from the other writes.
Pipes are an inter process communication mechanism that allow two or more processes to send information to each other. They are commonly used from within shells to connect the

standard output of one utility to the standard input of another. For example, to know the number of uses logged on a system:

$ who| wc _l.

The 'who' utility generates one line of output per user. This output is then 'piped' into the 'wc' utility, which, when invoked with the '- l' option, outputs the total number of lines in its input.

Bytes from 'who' flow through the pipe to 'wc'.

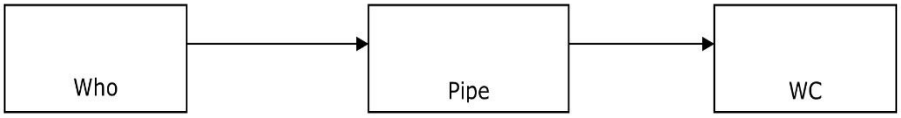


Figure 5.3: A simple pipe.

Pipe automatically buffers the output of the writer and suspends the writer if the pipe gets too full. Similarly, if a pipe empties, the reader is suspended until some more output becomes available.

All versions of UNIX support 'unnamed' pipes, which are the kind of pipes that shells use. The typical sequence of events is as follows:

1. The parent process creates an unnamed pipe, using pipe().
2. The parent process forks.
3. The writer closes its read end of the pipe and the designated reader closes its write end of the pipe.
4. The process communicates by using write() and read() calls.
5. Each process closes its active pipe descriptor when it's finished with it.

Bidirectional communication is possible only by using two pipes. Here is a small program that uses a pipe to allow the parent to read a message from its child.

Ex 1:

```
$cat pipe1.c
#include <stdio.h>
#define READ 0 /* the index of the read end of the pipe */
#define WRITE 1 /* the index of write end of the pipe */
Char *phrase = "stuff this message ";
main()
{
        int fd[2],bytesRead;
        char message[20]; /* parent process message buffer */
        pipe (fd); /* create an unnamed pipe */
        if (fork()==0) /* child, write */
        {
                close (fd[READ]); /* close unused end */
                write (fd[WRITE], phrase, strlen(phrase)+1);
/* include NULL */
                close (fd[WRITE]); /* close used end */
        }
else /* parent, node */
{
        close (fd[WRITE]); /* close unused end */
```

```
        bytesRead= read (fd[READ], message,100);
        printf("Read %d bytes: %s \n", bytesRead, message); /* send*/
        close (fd[READ]); /* close useend */
}}
```

Output:

```
        Read 18 bytes: stuff this message
```

5.2 Process Pipes

Perhaps the simplest way of passing data between two programs is with the popen and pclose functions.

These have the following prototypes:

```
#include <stdio.h>
FILE *popen(const char *command, const char *open_mode);
int pclose(FILE *stream_to_close);
```

popen

- The popen function allows a program to invoke another program as a new process and either pass data to it or receive data from it. The command string is the name of the program to run, together with any parameters. open_mode must be either "r" or "w".
- If the open_mode is "r", output from the invoked program is made available to the invoking program and can be read from the file stream FILE * returned by popen, using the usual stdio library functions for reading (for example, fread). However, if open_mode is "w", the program can send data to the invoked command with calls to fwrite. The invoked program can then read the data on its standard input. Normally, the program being invoked won't be aware that it's reading data from another process; it simply reads its standard input stream and acts on it.
- A call to popen must specify either "r" or "w"; no other option is supported in a standard implementation of popen. This means that you can't invoke another program and both read from and write to it. On failure, popen returns a null pointer. If you want bidirectional communication using pipes, the normal solution is to use two pipes, one for data flow in each direction.

Pclose

When the process started with popen has finished, you can close the file stream associated with it usingnpclose. The pclose call will return only when the process started with popen finishes. If it's still running when pclose is called, the pclose call will wait for the process to finish.

- The pclose call normally returns the exit code of the process whose file stream it is closing. If the invoking process has already executed a wait statement before calling pclose, the exit status will be lost because the invoked process has finished and pclose will return −1, with errno set to ECHILD.

5.3 Sending Output to popen

Here's a program, popen2.c, that pipes data to another. Here, you'll use od (octal dump). It is very similar to the preceding example, except you are writing down a pipe instead of reading from it. This is popen2.c.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{
FILE *write_fp;
char buffer[BUFSIZ + 1];
sprintf(buffer, "Once upon a time, there was...\n");
write_fp = popen("od -c", "w");
if (write_fp != NULL) {
fwrite(buffer, sizeof(char), strlen(buffer), write_fp);
pclose(write_fp);
exit(EXIT_SUCCESS);
}
exit(EXIT_FAILURE);
}
```

When you run this program, you should get the following output:

```
$ ./popen2
0000000 O n c e u p o n a t i m e
0000020 , t h e r e w a s . . . \n
0000037
```

5.3.1 Passing More Data

The mechanism that you've used so far simply sends or receives all the data in a single fread or fwrite. Sometimes you may want to send the data in smaller pieces, or perhaps you may not know the size of the output. To avoid having to declare a very large buffer, you can just use multiple fread or fwrite calls and process the data in parts.

Here's a program, popen3.c, that reads all of the data from a pipe

In this program, you read data from an invoked ps ax process. There's no way to know in advance how much output there will be, so you must allow for multiple reads of the pipe.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
```

```
{
FILE *read_fp;
char buffer[BUFSIZ + 1];
int chars_read;
memset(buffer, '\0', sizeof(buffer));
read_fp = popen("ps ax", "r");
if (read_fp != NULL) {
chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
while (chars_read > 0) {
buffer[chars_read - 1] = '\0';
printf("Reading %d:-\n %s\n", BUFSIZ, buffer);
chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
}
pclose(read_fp);
exit(EXIT_SUCCESS);
}
exit(EXIT_FAILURE);
}
```

The output, edited for brevity, is similar to this:

```
$ ./popen3
Reading 1024:-
PID TTY STAT TIME COMMAND
1 ? Ss 0:03 init [5]
2 ? SW 0:00 [kflushd]
3 ? SW 0:00 [kpiod]
4 ? SW 0:00 [kswapd]
5 ? SW< 0:00 [mdrecoveryd]
...
240 tty2 S 0:02 emacs draft1.txt
Reading 1024:-
368 tty1 S 0:00 ./popen3
369 tty1 R 0:00 ps –ax
```

5.3.2 popen Is Implemented

The popen call runs the program you requested by first invoking the shell, sh, passing it the command string as an argument. This has two effects, one good and the other not so good.

In Linux (as in all UNIX-like systems), all parameter expansion is done by the shell, so invoking the shell to parse the command string before the program is invoked allows any shell expansion, such as determining what files *.c actually refers to, to be done before the program starts. This is often quite useful, and it allows complex shell commands to be started with popen. Other process creation functions, such as execl, can be much more complex to invoke, because the calling process has to perform its own shell expansion.

The unfortunate effect of using the shell is that for every call to popen, a shell is invoked along with the requested program. Each call to popen then results in two extra processes being started, which makes the popen function a little expensive in terms of system resources and invocation of the target command is slower than it might otherwise have been.

Here's a program, popen4.c, that you can use to demonstrate the behavior of popen. You can count the lines in all the popen example source files by cating the files and then piping the output to wc -l,which counts the number of lines. On the command line, the equivalent command is

$ cat popen.c | wc –l*

This program uses exactly the preceding command, but through popen so that it can read the result:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{
FILE *read_fp;
char buffer[BUFSIZ + 1];
int chars_read;
memset(buffer, '\0', sizeof(buffer));
read_fp = popen("cat popen*.c | wc -l", "r");
if (read_fp != NULL) {
chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
while (chars_read > 0) {
buffer[chars_read – 1] = '\0';
printf("Reading:-\n %s\n", buffer);
chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
}
pclose(read_fp);
exit(EXIT_SUCCESS);
}
exit(EXIT_FAILURE);
}
```

When you run this program, the output is

```
$ ./popen4
Reading:-
94
```

The program shows that the shell is being invoked to expand popen*.c to the list of all files starting popen and ending in .c and also to process the pipe (|) symbol and feed the output from cat into wc. You invoke the shell, the cat program, and wc and cause an output redirection, all in a single popen call. The program that invokes the command sees only the final output.

5.3.2 The Pipe call

This function provides a means of passing data between two programs, without the overhead of invoking a shell to interpret the requested command. It also gives you more control over the reading and writing of data.

The pipe function has the following prototype:

```
#include <unistd.h>
int pipe(int file_descriptor[2]);
```

Pipe is passed (a pointer to) an array of two integer file descriptors. It fills the array with two new file descriptors and returns a zero. On failure, it returns -1 and sets errno to indicate the reason for failure.
Errors defined in the Linux manual page for pipe are

- EMFILE: Too many file descriptors are in use by the process.
- ENFILE: The system file table is full.
- EFAULT: The file descriptor is not valid.

The two file descriptors returned are connected in a special way. Any data written to file_descriptor[1] can be read back from file_descriptor[0]. The data is processed in a first in, first out basis, usually abbreviated to FIFO. This means that if you write the bytes 1, 2, 3 to file_descriptor[1], reading from file_descriptor[0] will produce 1, 2, 3. This is different from a stack, which operates on a last in, first out basis, usually abbreviated to LIFO.
Here's a program, pipe1.c, that uses pipe to create a pipe

The following example is pipe1.c. Note the file_pipes array, the address of which is passed to the pipe function as a parameter.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{
int data_processed;
int file_pipes[2];
const char some_data[] = "123";
char buffer[BUFSIZ + 1];
memset(buffer, '\0', sizeof(buffer));
if (pipe(file_pipes) == 0) {
data_processed = write(file_pipes[1], some_data, strlen(some_data));
printf("Wrote %d bytes\n", data_processed);
data_processed = read(file_pipes[0], buffer, BUFSIZ);
printf("Read %d bytes: %s\n", data_processed, buffer);
        exit(EXIT_SUCCESS);
        }
        exit(EXIT_FAILURE);
        }
```

When you run this program, the output is

```
$ ./pipe1
Wrote 3 bytes
Read 3 bytes: 123
```

The program creates a pipe using the two file descriptors in the array file_pipes[]. It then writes data into the pipe using the file descriptor file_pipes[1] and reads it back from file_pipes[0]. Notice that the pipe has some internal buffering that stores the data in between the calls to write and read.

5.4 Parent and child processes

The next logical step in our investigation of the pipe call is to allow the child process to be a different program from its parent, rather than just a different process running the same program. You do this using the exec call. One difficulty is that the new execed process needs to know which file descriptor to access. In the previous example, this wasn't a problem because the child had access to its copy of the file_pipes data. After an exec call, this will no longer be the case, because the old process has been replaced by the new child process. You can get around this by passing the file descriptor (which is, after all, just a number) as a parameter to the newly execed program.

To show how this works, you need two programs. The first is the data producer. It creates the pipe and then invokes the child, the data consumer

1. For the first program, you adapt pipe2.c to pipe3.c. The changed lines are shown shaded:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{
int data_processed;
int file_pipes[2];
const char some_data[] = "123";
char buffer[BUFSIZ + 1];
pid_t fork_result;
memset(buffer, '\0', sizeof(buffer));
if (pipe(file_pipes) == 0) {
fork_result = fork();
if (fork_result == (pid_t)-1) {
fprintf(stderr, "Fork failure");
exit(EXIT_FAILURE);
}
if (fork_result == 0) {
sprintf(buffer, "%d", file_pipes[0]);
(void)execl("pipe4", "pipe4", buffer, (char *)0);
exit(EXIT_FAILURE);
}
else {
data_processed = write(file_pipes[1], some_data,
strlen(some_data));
printf("%d - wrote %d bytes\n", getpid(), data_processed);
}
}
exit(EXIT_SUCCESS);
}
```

2. The consumer program, pipe4.c, which reads the data, is much simpler:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

```
int main(int argc, char *argv[])
{
int data_processed;
char buffer[BUFSIZ + 1];
int file_descriptor;
memset(buffer, '\0', sizeof(buffer));
sscanf(argv[1], "%d", &file_descriptor);
data_processed = read(file_descriptor, buffer, BUFSIZ);
printf("%d - read %d bytes: %s\n", getpid(), data_processed, buffer);
exit(EXIT_SUCCESS);
}
```

Remembering that pipe3 invokes the pipe4 program, you get something similar to the following output when you run pipe3:

```
$ ./pipe3
22460 - wrote 3 bytes
22461 - read 3 bytes: 12
```

The pipe3 program starts like the previous example, using the pipe call to create a pipe and then using the fork call to create a new process. It then uses sprintf to store the "read" file descriptor number of the pipe in a buffer that will form an argument of pipe4. A call to execl is used to invoke the pipe4 program. The arguments to execl are

- The program to invoke
- argv[0], which takes the program name
- argv[1], which contains the file descriptor number you want the program to read from
- (char *)0, which terminates the parameters

The pipe4 program extracts the file descriptor number from the argument string and then reads from that file descriptor to obtain the data

5.4.1 Reading closed pipes

- Most programs that read data from the standard input do so differently than the examples you've seen so far. They don't usually know how much data they have to read, so they will normally loop — reading data, processing it, and then reading more data until there's no more data to read.
- A read call will normally block; that is, it will cause the process to wait until data becomes available. If the other end of the pipe has been closed, then no process has the pipe open for writing, and the read blocks. Because this isn't very helpful, a read on a pipe that isn't open for writing returns zero rather than blocking. This allows the reading process to detect the pipe equivalent of end of file and act appropriately. Notice that this isn't the same as reading an invalid file descriptor, which read considers an error and indicates by returning–1.
- If you use a pipe across a fork call, there are two different file descriptors that you can use to write to the pipe: one in the parent and one in the child. You must close the write file descriptors of the pipe in both parent and child processes before the pipe is considered closed and a read call on the pipe will fail.

5.4.2 Pipes Used as Standard Input and Output

The one big advantage is that you can invoke standard programs, ones that don't expect a file descriptoras a parameter. In order to do this, you need to use the dup function, There are two closely related versions of dup that have the following prototypes:

#include <unistd.h>
int dup(int file_descriptor);
int dup2(int file_descriptor_one, int file_descriptor_two);

The purpose of the dup call is to open a new file descriptor, a little like the open call. The difference is that the new file descriptor created by dup refers to the same file (or pipe) as an existing file descriptor.

In the case of dup, the new file descriptor is always the lowest number available, and in the case of dup2 it's the same as, or the first available descriptor greater than, the parameter file_descriptor_two.

File Descriptor Manipulation by close and dup

The easiest way to understand what happens when you close file descriptor 0, and then call dup, is to look at how the state of the first four file descriptors changes during the sequence.

File Descriptor Number	Initially	After close of File Descriptor 0	After dup
0	Standard input	(closed)	Pipe file descriptor
1	Standard output	Standard output	Standard output
2	Standard error	Standard error	Standard error
3	Pipe file descriptor	Pipe file descriptor	Pipe file descriptor

Fig 5.4 File descriptor and call dup

The child can then use exec to invoke any program that reads standard input. In this case, you use the od command. The od command will wait for data to be available to it as if it were waiting for input from a user terminal. In fact, without some special code to explicitly detect the difference, it won't know that the input is from a pipe rather than a terminal.
The parent starts by closing the read end of the pipe file_pipes[0], because it will never read the pipe. It then writes data to the pipe. When all the data has been written, the parent closes the write end of the pipe and exits. Because there are now no file descriptors open that could write to the pipe, the od program will be able to read the three bytes written to the pipe, but subsequent reads will then return 0 bytes, indicating an end of file. When the read returns 0, the od program exits. This is analogous to running the od command on a terminal, then pressing Ctrl+D to send end of file to the od command.

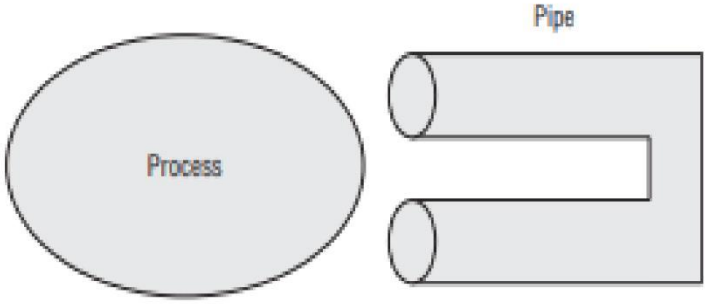


Fig 5.5 Sequence after the call to the pipe

5.5 Named Pipes: FIFOs

FIFOs are sometimes called named pipes. Pipes can be used only between related processes when a common ancestor has created the pipe. With FIFOs, however, unrelated processes can exchange data.

A FIFO is a type of file. Creating a FIFO is similar to creating a file. Indeed, the pathname for a FIFO exists in the file system.

```
#include <sys/types.h>
    #include <sys/stat.h>
        int  mkfifo (const  char *pathname, mode_t mode);
```

Returns: 0 if OK, -1 on error

The specification of mode argument for the mkfifo function is the same as for the open function. Once a FIFO is created using mkfifo then it can be opened using open. Indeed, the normal file I/O functions (read, close, write, unlink, etc.) all work with FIFOs.

When we open a FIFO, the non-blocking flag (O_NONBLOCK) affects what happens.

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

Like a pipe, if we write to a FIFO that has opened for reading, the signal SIGPIPE is generated.

There are two uses for FIFOs:

1. FIFOs are used by shell commands to pass data from on shell to pipeline to another, without creating intermediate temporary files.
2. FIFOs are used in a client-server application to pass data between the clients and server.

5.5.1 Accessing a FIFO

One very useful feature of named pipes is that, because they appear in the file system, you can use them in commands where you would normally use a filename. Before you do more programming using the FIFO file you created, let's investigate the behavior of the FIFO file using normal file commands.

Accessing a FIFO File

1. First, try reading the (empty) FIFO:

```
$ cat < /tmp/my_fifo
```

2.Now try writing to the FIFO. You will have to use a different terminal because the first command will now be hanging, waiting for some data to appear in the FIFO.

```
$ echo "Hello World" > /tmp/my_fifo
```

You will see the output appear from the cat command. If you don't send any data down the FIFO, the cat command will hang until you interrupt it, conventionally with Ctrl+C.

3. You can do both at once by putting the first command in the background:

```
$ cat < /tmp/my_fifo &
[1] 1316
$ echo "Hello World" > /tmp/my_fifo
Hello World
[1]+ Done cat </tmp/my_fifo
$
```

Unlike a pipe created with the pipe call, a FIFO exists as a named file, not as an open file descriptor, and it must be opened before it can be read from or written to. You open and close a FIFO using the same open and close functions that you saw used earlier for files,

with some additional functionality. The open call is passed the path name of the FIFO, rather than that of a regular file.

5.6 Client/Server Using FIFOs

- Let's consider how you might build a very simple client/server application using named pipes. You want to have a single-server process that accepts requests, processes them, and returns the resulting data to the requesting party: the client.
- You want to allow multiple client processes to send data to the server. In the interests of simplicity, we'll assume that the data to be processed can be broken into blocks, each smaller than PIPE_BUF bytes. Of course, you could implement this system in many ways, but we'll consider only one method as an illustration of how named pipes can be used.
- Because the server will process only one block of information at a time, it seems logical to have a single FIFO that is read by the server and written to by each of the clients. By opening the FIFO in blocking mode, the server and the clients will be automatically blocked as required.
- Returning the processed data to the clients is slightly more difficult. You need to arrange a second pipe, one per client, for the returned data. By passing the process identifier (PID) of the client in the original data sent to the server, both parties can use this to generate the unique name for the return pipe

An Example Client/Server Application

First, you need a header file, client.h, that defines the data common to both client and server programs. It also includes the required system headers, for convenience.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#define SERVER_FIFO_NAME "/tmp/serv_fifo"
#define CLIENT_FIFO_NAME "/tmp/cli_%d_fifo"
#define BUFFER_SIZE 20
struct data_to_pass_st {
pid_t client_pid;
char some_data[BUFFER_SIZE - 1];
};
```

The server creates its FIFO in read-only mode and blocks. It does this until the first client connects by opening the same FIFO for writing. At that point, the server process is unblocked and the sleep is executed, so the writes from the clients queue up. (In a real application, the sleep would be removed; we're only using it to demonstrate the correct operation of the program with multiple simultaneous clients.)

In the meantime, after the client has opened the server FIFO, it creates its own uniquely named FIFO for reading data back from the server. Only then does the client write data to the server (blocking if the pipe is full or the server's still sleeping) and then blocks on a read of its own FIFO, waiting for the reply.

On receiving the data from the client, the server processes it, opens the client pipe for writing, and writes the data back, which unblocks the client. When the client is unblocked, it can read from its pipe the data written to it by the server.

The whole process repeats until the last client closes the server pipe, causing the server's read to fail (returning 0) because no process has the server pipe open for writing. If this were a real server process that needed to wait for further clients, you would need to modify it to either

- Open a file descriptor to its own server pipe, so read always blocks rather than returning 0.
- Close and reopen the server pipe when read returns 0 bytes, so the server process blocks in the open waiting for a client, just as it did when it first started.

Both of these techniques are illustrated in the rewrite of the CD database application to use named pipes.

PART – 2 Socket Connections

Socket

A socket is a communication mechanism that allows client/server systems to be developed either locally, on a single machine, or across networks. Linux functions such as printing, connecting to databases, and serving web pages as well as network utilities such as rlogin for remote login and ftp for file transfer usually use sockets to communicate.

Sockets are created and used differently from pipes because they make a clear distinction between client and server. The socket mechanism can implement multiple clients attached to a single server.

5.1 Socket Attributes

To fully understand the system calls used in this example, you need to learn a little about UNIX networking.

Sockets are characterized by three attributes: domain, type, and protocol. They also have an address used as their name. The formats of the addresses vary depending on the domain, also known as the protocol family. Each protocol family can use one or more address families to define the address format.

Socket Domains

Domains specify the network medium that the socket communication will use. The most common socket domain is AF_INET, which refers to Internet networking that's used on many Linux local area networks and, of course, the Internet itself. The underlying protocol, Internet Protocol (IP), which only has one address family, imposes a particular way of specifying computers on a network. This is called the IP address.

Although names almost always refer to networked machines on the Internet, these are translated into lower-level IP addresses. An example IP address is 192.168.1.99. All IP addresses are represented by four numbers, each less than 256, a so-called dotted quad. When a client connects across a network via sockets, it needs the IP address of the server computer.

There may be several services available at the server computer. A client can address a particular service on a networked machine by using an IP port. A port is identified within the system by assigning a unique 16-bit integer and externally by the combination of IP

address and port number. The sockets are communication end points that must be bound to ports before communication is possible.

Servers wait for connections on particular ports. Well-known services have allocated port numbers that are used by all Linux and UNIX machines. These are usually, but not always, numbers less than 1024. Examples are the printer spooler (515), rlogin (513), ftp (21), and httpd (80). The last of these is the standard port for web servers. Usually, port numbers less than 1024 are reserved for system services and may only be served by processes with superuser privileges. X/Open defines a constant in netdb.h, IPPORT_RESERVED, to stand for the highest reserved port number.

Because there is a standard set of port numbers for standard services, computers can easily connect to each other without having to figure out the correct port. Local services may use nonstandard port addresses.

The domain in the first example is the UNIX file system domain, AF_UNIX, which can be used by sockets based on a single computer that perhaps isn't networked. When this is so, the underlying protocol is file input/output and the addresses are filenames. The address that you used for the server socket was server socket, which you saw appear in the current directory when you ran the server application.
Other domains that might be used include AF_ISO for networks based.

Socket Types
A socket domain may have a number of different ways of communicating, each of which might have different characteristics. This isn't an issue with AF_UNIX domain sockets, which provide a reliable twoway communication path. In networked domains, however, you need to be aware of the characteristics of the underlying network and how different communication mechanisms are affected by them.

Internet protocols provide two communication mechanisms with distinct levels of service: streams and datagrams.

Stream Sockets

Stream sockets (in some ways similar to standard input/output streams) provide a connection that is a sequenced and reliable two-way byte stream. Thus, data sent is guaranteed not to be lost, duplicated, or reordered without an indication that an error has occurred. Large messages are fragmented, transmitted, and reassembled. This is similar to a file stream, which also accepts large amounts of data and splits it up for writing to the low-level disk in smaller blocks. Stream sockets have predictable behavior.

Stream sockets, specified by the type SOCK_STREAM, are implemented in the AF_INET domain by TCP/IP connections. They are also the usual type in the AF_UNIX domain. We concentrate on SOCK_STREAM sockets in this chapter because they are more commonly used in programming network applications.

Datagram Sockets

In contrast, a datagram socket, specified by the type SOCK_DGRAM, doesn't establish and maintain a connection. There is also a limit on the size of a datagram that can be sent. It's transmitted as a single network message that may get lost, duplicated, or arrive out of sequence — ahead of datagrams sent after it.

Datagram sockets are implemented in the AF_INET domain by UDP/IP connections and provide an unsequenced, unreliable service. (UDP stands for User Datagram Protocol.) However, they are relatively inexpensive in terms of resources, because network connections need not be maintained. They're fast because there is no associated connection setup time.

Datagrams are useful for "single-shot" inquiries to information services, for providing regular status information, or for performing low-priority logging. They have the advantage that the death of a server doesn't unduly inconvenience a client and would not require a client restart. Because datagram-based servers usually retain no connection information, they can be stopped and restarted without disturbing their clients.

Socket Protocols

Socket protocols provide the network transportation of application data from one machine to another (or from one process to another within the same machine).

The application specifies the transport provider on the protocol parameter of the socket() API.

For the AF_INET address family, more than one transport provider is allowed. The protocols of Systems Network Architecture (SNA) and TCP/IP can be active on the same listening socket at the same time. The ALWANYNET (Allow ANYNET support) network attribute allows a customer to select whether a transport other than TCP/IP can be used for AF_INET socket applications. This network attribute can be either *YES or *NO. The default value is *NO.

For example, if the current status (the default status) is *NO, the use of AF_INET over an SNA transport is not active. If AF_INET sockets are to be used over a TCP/IP transport only, the ALWANYNET status should be set to *NO to improve CPU utilization..

The AF_INET and AF_INET6 sockets over TCP/IP can also specify a SOCK_RAW type, which means that the socket communicates directly with the network layer known as Internet Protocol (IP). The TCP or UDP transport providers normally communicate with this layer. When you use SOCK_RAW sockets, the application program specifies any protocol between 0 and 255 (except the TCP and UDP protocols). This protocol number

then flows in the IP headers when machines are communicating on the network. In fact, the application program is the transport provider, because it must provide for all the transport services that UDP or TCP transports normally provide.

For the AF_UNIX and AF_UNIX_CCSID address families, a protocol specification is not really meaningful because there are no protocol standards involved. The communications mechanism between two processes on the same machine is specific to the machine.

5.2 Creating a Socket

The socket system call creates a socket and returns a descriptor that can be used for accessing the socket.

#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);

The socket created is one end point of a communication channel. The domain parameter specifies the address family, the type parameter specifies the type of communication to be used with this socket, and protocol specifies the protocol to be employed.

The most common socket domains are AF_UNIX, which is used for local sockets implemented via the UNIX and Linux file systems, and AF_INET, which is used for UNIX network sockets. The AF_INET sockets may be used by programs communicating across a TCP/IP network including the Internet. The Windows Winsock interface also provides access to this socket domain.

The socket parameter type specifies the communication characteristics to be used for the new socket. Possible values include SOCK_STREAM and SOCK_DGRAM. SOCK_STREAM is a sequenced, reliable, connection-based two-way byte stream.

For an AF_INET domain socket, this is provided by default by a TCP connection that is established between the two end points of the stream socket when it's connected. Data may be passed in both directions along the socket connection. The TCP protocols include facilities to fragment and reassemble long messages and to retransmit any parts that may be lost in the network.

SOCK_DGRAM is a datagram service. You can use this socket to send messages of a fixed (usually small) maximum size, but there's no guarantee that the message will be delivered or that messages won't be reordered in the network. For AF_INET sockets, this type of communication is provided by UDP datagrams.

The protocol used for communication is usually determined by the socket type and domain. There is normally no choice. The protocol parameter is used where there is a choice. 0 selects the default protocol, which is used in all the examples in this chapter. The socket system call returns a descriptor that is in many ways similar to a low-level file descriptor. When the socket has been connected to another end-point socket, you can use the read and write system calls with the descriptor to send and receive data on the socket. The close system call is used to end a socket connection

5.3 Socket Addresses

Each socket domain requires its own address format. For an AF_UNIX socket, the address is described by a structure, sockaddr_un, defined in the sys/un.h include file.

struct sockaddr_un {
sa_family_t sun_family; / AF_UNIX */*
char sun_path[]; / pathname */*
};
So that addresses of different types may be passed to the socket-handling system calls, each address format is described by a similar structure that begins with a field (in this case, sun_family) that specifies the address type (the socket domain). In the AF_UNIX domain, the address is specified by a filename in the sun_path field of the structure.

On current Linux systems, the type sa_family_t, defined by X/Open as being declared in sys/un.h,is taken to be a short. Also, the pathname specified in sun_path is limited in size (Linux specifies 108 characters; others may use a manifest constant such as UNIX_MAX_PATH). Because address structures may vary in size, many socket calls require or provide as an output a length to be used for copying the particular address structure.

In the AF_INET domain, the address is specified using a structure called sockaddr_in, defined in netinet/in.h, which contains at least these members:
struct sockaddr_in {
short int sin_family; / AF_INET */*
unsigned short int sin_port; / Port number */*
struct in_addr sin_addr; / Internet address */*
};
The IP address structure, in_addr, is defined as follows:
struct in_addr {
unsigned long int s_addr;
};
The four bytes of an IP address constitute a single 32-bit value. An AF_INET socket is fully described by its domain, IP address, and port number. From an application's point of view, all sockets act like file descriptors and are addressed by a unique integer value.

5.4 Naming a socket
To make a socket (as created by a call to socket) available for use by other processes, a server program needs to give the socket a name. Thus, AF_UNIX sockets are associated with a file system pathname, as you saw in the server1 example. AF_INET sockets are associated with an IP port number.
#include <sys/socket.h>
int bind(int socket, const struct sockaddr *address, size_t address_len);
The bind system call assigns the address specified in the parameter, address, to the unnamed socket associated with the file descriptor socket. The length of the address structure is passed as address_len.

The length and format of the address depend on the address family. A particular address structure pointer will need to be cast to the generic address type (struct sockaddr *) in the call to bind.

5.5 Creating a socket queue
To accept incoming connections on a socket, a server program must create a queue to store pending requests. It does this using the listen system call.
#include <sys/socket.h>
int listen(int socket, int backlog);

A Linux system may limit the maximum number of pending connections that may be held in a queue. Subject to this maximum, listen sets the queue length to backlog. Incoming connections up to this queue length are held pending on the socket; further connections will be refused and the client's connection will fail.

This mechanism is provided by listen to allow incoming connections to be held pending while a server program is busy dealing with a previous client. A value of 5 for backlog is very common.
The listen function will return 0 on success or -1 on error. Errors include EBADF, EINVAL, and ENOTSOCK, as for the bind system call.

5.6 Accepting Connections
Once a server program has created and named a socket, it can wait for connections to be made to the socket by using the accept system call.
#include <sys/socket.h>
int accept(int socket, struct sockaddr *address, size_t *address_len);

The accept system call returns when a client program attempts to connect to the socket specified by the parameter socket. The client is the first pending connection from that socket's queue. The accept function creates a new socket to communicate with the client and returns its descriptor. The new socket will have the same type as the server listen socket.

The socket must have previously been named by a call to bind and had a connection queue allocated by listen. The address of the calling client will be placed in the sockaddr structure pointed to by address.

A null pointer may be used here if the client address isn't of interest. The address_len parameter specifies the length of the client structure. If the client address is longer than this value, it will be truncated. Before calling accept, address_len must be set to the expected address length. On return, address_len will be set to the actual length of the calling client's address structure.
If there are no connections pending on the socket's queue, accept will block (so that the program won't continue) until a client does make connection. You can change this behavior by using the O_NONBLOCK flag on the socket file descriptor, using the fcntl function in your code like this:
int flags = fcntl(socket, F_GETFL, 0);
fcntl(socket, F_SETFL, O_NONBLOCK|flags);

The accept function returns a new socket file descriptor when there is a client connection pending or -1 on error. Possible errors are similar to those for bind and listen, with the addition of EWOULDBLOCK, where O_NONBLOCK has been specified and there are no pending connections. The error EINTR will occur if the process is interrupted while blocked in accept.

5.7 Requesting Connections
Client programs connect to servers by establishing a connection between an unnamed socket and the server listen socket. They do this by calling connect.
#include <sys/socket.h>

*int connect(int socket, const struct sockaddr *address, size_t address_len);*

The socket specified by the parameter socket is connected to the server socket specified by the parameter address, which is of length address_len. The socket must be a valid file descriptor obtained by a call to socket.
If it succeeds, connect returns 0, and -1 is returned on error. Possible errors this time include the following:

Errno Value	**Description**
EBADF	An invalid file descriptor was passed in socket.
EALREADY	A connection is already in progress for this socket.
ETIMEDOUT	A connection timeout has occurred.
ECONNREFUSED	The requested connection was refused by the server

If the connection can't be set up immediately, connect will block for an unspecified timeout period. Once this timeout has expired, the connection will be aborted and connect will fail. However, if the call to connect is interrupted by a signal that is handled, the connect call will fail (with errno set to EINTR), but the connection attempt won't be aborted — it will be set up asynchronously, and the program will have to check later to see if the connection was successful.

As with accept, the blocking nature of connect can be altered by setting the O_NONBLOCK flag on thefile descriptor. In this case, if the connection can't be made immediately, connect will fail with errno set to EINPROGRESS and the connection will be made asynchronously.

Though asynchronous connections can be tricky to handle, you can use a call to select on the socket file descriptor to check that the socket is ready for writing.

5.8 Closing a socket

You can terminate a socket connection at the server and client by calling close, just as you would for lowlevel file descriptors. You should always close the socket at both ends. For the server, you should do this when read returns zero. Note that the close call may block if the socket has untransmitted data, is of a connection-oriented type, and has the SOCK_LINGER option set.

5.9 Socket Communications

Now that we have covered the basic system calls associated with sockets, let's take a closer look at the example programs. You'll try to convert them to use a network socket rather than a file system socket. The file system socket has the disadvantage that, unless the author uses an absolute pathname, it's created in the server program's current directory. To make it more generally useful, you need to create it in a globally accessible directory (such as /tmp) that is agreed between the server and its clients. For network sockets, you need only choose an unused port number.

For the example, select port number 9734. This is an arbitrary choice that avoids the standard services (you can't use port numbers below 1024 because they are reserved for system use). Other port numbers are often listed, with the services provided on them, in

[3] 23770
server waiting
server waiting
char from server = B
$ netstat –A inet
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address Foreign Address (State) User
tcp 1 0 localhost:1574 localhost:1174 TIME_WAIT root

You can see the port numbers that have been assigned to the connection between the server and the client. The local address shows the server, and the foreign address is the remote client. (Even though it's on the same machine, it's still connected over a network.) To ensure that all sockets are distinct, these client ports are typically different from the server listen socket and unique to the computer.

To enable computers of different types to agree on values for multibyte integers transmitted over a network, you need to define a network ordering. Client and server programs must convert their internal integer representation to the network ordering before transmission. They do this by using functions defined in netinet/in.h. These are

#include <netinet/in.h>
unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
These functions convert 16-bit and 32-bit integers between native host format and the standard network ordering. Their names are abbreviations for conversions — for example, "host to network, long" (htonl) and "host to network, short" (htons). For computers where the native ordering is the same as network ordering, these represent null operations.

To ensure correct byte ordering of the 16-bit port number, your server and client need to apply these
functions to the port address. The change to server3.c is
server_address.sin_addr.s_addr = htonl(INADDR_ANY);
server_address.sin_port = htons(9734);
You don't need to convert the function call, inet_addr("127.0.0.1"), because inet_addr is defined
to produce a result in network order. The change to client3.c is

address.sin_port = htons(9734);

The server has also been changed to allow connections from any IP address by using INADDR_ANY.
Now, when you run server3 and client3, you see the correct port being used for the local connection.

$ netstat
Active Internet connections
Proto Recv-Q Send-Q Local Address Foreign Address (State) User