UNIT-V

Inter Process Communication :Pipes

What Is a Pipe?

- We use the term pipe to mean connecting a data flow from one process to another. Generally you attach, or pipe, the output of one process to the input of another.
- For shell commands, this is done using the pipe character to join the commands, such as
- cmd1 | cmd2
- The shell arranges the standard input and output of the two commands, so that
- ❑ The standard input to cmd1 comes from the terminal keyboard.
- ❑ The standard output from cmd1 is fed to cmd2 as its standard input.
- ❑ The standard output from cmd2 is connected to the terminal screen.

Process Pipes

- 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.
- 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 using pclose. 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.

Reading Output from an External Program

- The uname -a command prints system information, including the machine type, the OS name, version and release, and the machine's network name.
- Having initialized the program, you open the pipe to uname, making it readable and setting read_fp to point to the output. At the end, the pipe pointed to by read_fp is closed.

Reading Output from an External Program

- #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("uname -a", "r");
- if (read_fp != NULL) {
  chars_read = fread(buffer,
  sizeof(char), BUFSIZ,
  read_fp); if (chars_read >
  0) { printf("Output was:-
  \n%s\n", buffer); }
  pclose(read_fp);
  exit(EXIT_SUCCESS); }
  exit(EXIT_FAILURE);
- }

Sending Output to popen

- you can see that it is very similar to the preceding example, except you are writing down a pipe instead of reading from it.
- #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);
- }
- The program uses popen with the parameter “w” to start the od -c command, so that it can send data to that command. It then sends a string that the od -c command receives and processes; the od -c command then prints the result of the processing on its standard output.

- od (**Unix**) od is a program for displaying ("**dumping**") data in various human-readable output formats. The name is an acronym for "**octal dump**" since it defaults to printing in the **octal** data format. It can also display output in a variety of other formats, including hexadecimal, decimal, and ASCII.

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.

Passing More Data

```
lude <unistd.h>
lude <stdlib.h>
lude <stdio.h>
lude <string.h>

ain()

ILE *read_fp;
har buffer[BUFSIZ + 1];
nt chars_read;

emset(buffer, '\0', sizeof(buffer));
ead_fp = popen("ps ax", "r");
f (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);

xit(EXIT_FAILURE);
```

Passing More Data

- The program uses popen with an “r” parameter in a similar fashion to popen1.c. This time, it continues reading from the file stream until there is no more data available.
- Notice that, although the ps command takes some time to execute, Linux arranges the process scheduling so that both programs run when they can. If the reader process, popen3, has no input data, it’s suspended until some becomes available.
- If the writer process, ps, produces more output than can be buffered, it’s suspended until the reader has consumed some of the data.

How 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.

How popen Is Implemented

- 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

Program for popen Starts a Shell

- #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);
- }

How above code Works

- $ ./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.

The Pipe Call

- You've seen the high-level popen function, but now let's move on to look at the lower-level pipe function. 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 manual 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, 3to 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

- It’s important to realize that these are file descriptors, not file streams, so you must use the lower level read and write system calls to access the data, rather than the stream library functions fread and fwrite

Program with The pipe Function

- **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**

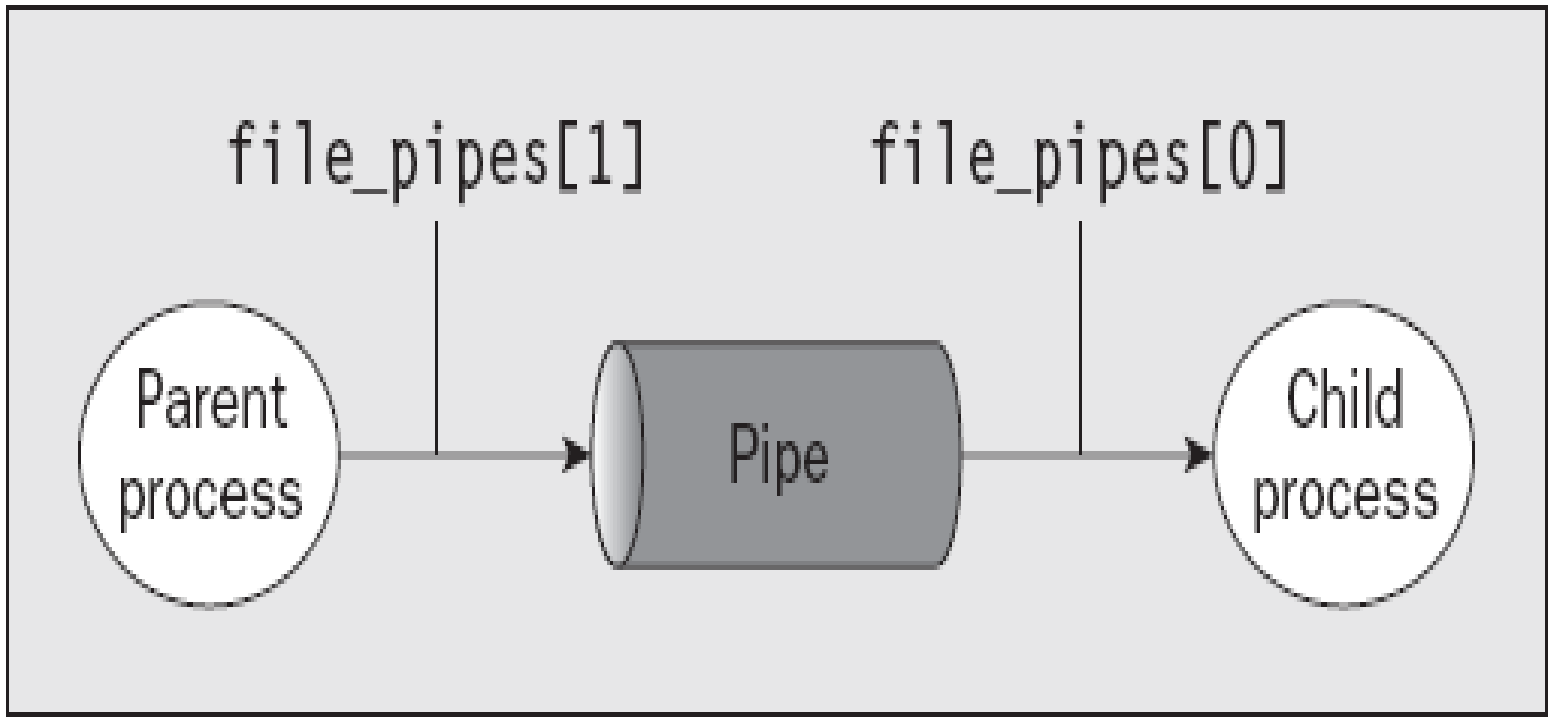
How above code Works

- 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.
- You should be aware that the effect of trying to write using file_descriptor[0], or read using file_descriptor[1], is undefined, so the behavior could be very strange and may change without warning. On the authors' systems, such calls fail with a –1 return value, which at least ensures that it's easy to catch this mistake.

- At first glance, this example of a pipe doesn’t seem to offer us anything that we couldn’t have done with a simple file. The real advantage of pipes comes when you want to pass data between two processes.
- when a program creates a new process using the fork call, file descriptors that were previously open remain open. By creating a pipe in the original process and then forking to create a new process, you can pass data from one process to the other down the pipe.

Pipes across a fork

- 1. This is pipe2.c. It starts rather like the first example, up until you make the call to fork.
- #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 == -1) {
- fprintf(stderr, “Fork failure”); exit(EXIT_FAILURE);
- } 2. You’ve made sure the fork worked, so if fork_result equals zero, you’re in the child process:
- if (fork_result == 0) { data_processed = read(file_pipes[0], buffer, BUFSIZ); printf(“Read %d bytes: %s\n”, data_processed, buffer); exit(EXIT_SUCCESS); }
- 3. Otherwise, you must be in the parent process:
- else { data_processed = write(file_pipes[1], some_data, strlen(some_data)); printf(“Wrote %d bytes\n”, data_processed); }
- } exit(EXIT_SUCCESS);
- }


file_pipes[1]
file_pipes[0]
Parent process
Pipe
Child process

How above code Works

- First, the program creates a pipe with the pipe call. It then uses the fork call to create a new process. If the fork was successful, the parent writes data into the pipe, while the child reads data from the pipe. Both parent and child exit after a single write and read. If the parent exits before the child, you might see the shell prompt between the two outputs.
- Although the program is superficially very similar to the first pipe example, we've taken a big step forward by being able to use separate processes for the reading and writing.

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.

Producer 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.

```
#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);
}
```

Consumer Program

```
#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);
}
```

How above code Works

- 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.

Reading Closed Pipes

- we need to look a little more carefully at the file descriptors that are open. Up to this point you have allowed the reading process simply to read some data and then exit, assuming that Linux will clean up the files as part of the process termination.
- 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. You’ll see an example of this later when we return to this subject in more detail to look at the O_NONBLOCK flag and FIFOs.

Pipes Used as Standard Input and Output

- You arrange for one of the pipe file descriptors to have a known value, usually the standard input, 0, or the standard output, 1. This is slightly more complex to set up in the parent, but it allows the child program to be much simpler.
- The one big advantage is that you can invoke standard programs, ones that don't expect a file descriptor as 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);

Pipes Used as Standard Input and Output

- 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.
- You can get the same effect as dup and dup2 by using the more general fcntl call, with a command F_DUPFD. Having said that, the dup call is easier to use because it's tailored specifically to the needs of creating duplicate file descriptors. It's also very commonly used, so you'll find it more frequently in existing programs than fcntl and F_DUPFD.

Pipes Used as Standard Input and Output

- **So how does dup help in passing data between processes?** The trick is knowing that the standard input file descriptor is always 0 and that dup always returns a new file descriptor using the lowest available number. By first closing file descriptor 0 and then calling dup, the new file descriptor will have the number 0.
- Because the new descriptor is a duplicate of an existing one, standard input will have been changed to access the file or pipe whose file descriptor you passed to dup. You will have created two file descriptors that refer to the same file or pipe, and one of them will be the standard input.

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. This is shown in the following table.

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

Pipes and dup

- the child program to have its stdin file descriptor replaced with the read end of the pipe you create. You'll also do some tidying up of file descriptors so the child program can correctly detect the end of the data in the pipe.
- #include <unistd.h>
- #include <stdlib.h>
- #include <stdio.h>
- #include <string.h>
- int main()
- {

Pipes and dup

```
int data_processed;
int file_pipes[2];
const char some_data[] = “123”;
pid_t fork_result;
if (pipe(file_pipes) == 0) {
fork_result = fork();
if (fork_result == (pid_t)-1) {
fprintf(stderr, “Fork failure”);
exit(EXIT_FAILURE);
    } if (fork_result == (pid_t)0) {
close(0);
dup(file_pipes[0]);
close(file_pipes[0]);
close(file_pipes[1]);
execlp(“od”, “od”, “-c”, (char *)0);
exit(EXIT_FAILURE);
}
else {
close(file_pipes[0]);
data_processed =
write(file_pipes[1], some_data,
strlen(some_data));
close(file_pipes[1]);
printf(“%d - wrote %d bytes\n”,
(int)getpid(), data_processed);
}
}
exit(EXIT_SUCCESS);
}
```

How above code Works

- the program creates a pipe and then forks, creating a child process. At this point, both the parent and child have file descriptors that access the pipe, one each for reading and writing, so there are four open file descriptors in total.
- The child closes its standard input with close(0) and then calls dup(file_pipes[0]). This duplicates the file descriptor associated with the read end of the pipe as file descriptor 0, the standard input. The child then closes the original file descriptor for reading from the pipe, file_pipes[0]. Because the child will never write to the pipe, it also closes the write file descriptor associated with the pipe, file_pipes[1]. It now has a single file descriptor associated with the pipe: file descriptor 0, its standard input.

continues

- 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

Named Pipes: FIFOs

- So far, you have only been able to pass data between related programs, that is, programs that have been started from a common ancestor process. Often this isn't very convenient, because you would like unrelated processes to be able to exchange data.
- You do this with *FIFOs*, often referred to as *named pipes*. A named pipe is a special type of file (remember that everything in Linux is a file!) that exists as a name in the file system but behaves like the unnamed pipes that you've met already.
- You can create named pipes from the command line and from within a program. Historically, the command-line program for creating them was mknod:
- $ **mknod *filename* p**

Named Pipes: FIFOs

- However, the mknod command is not in the X/Open command list, so it may not be available on all UNIX-like systems. The preferred command-line method is to use
- $ **mkfifo *filename***
- From inside a program, you can use two different calls:
- **#include <sys/types.h>**
- **#include <sys/stat.h>**
- **int mkfifo(const char *filename, mode_t mode);**
- **int mknod(const char *filename, mode_t mode | S_IFIFO, (dev_t) 0);**

Creating a Named Pipe

- #include <unistd.h>
- #include <stdlib.h>
- #include <stdio.h>
- #include <sys/types.h>
- #include <sys/stat.h>
- int main()
- {
- int res = mkfifo("/tmp/my_fifo", 0777);
- if (res == 0) printf("FIFO created\n");
- exit(EXIT_SUCCESS);
- }

- You can create and look for the pipe with
- $ **./fifo1**
- FIFO created
- $ **ls -lF /tmp/my_fifo**
- prwxr-xr-x 1 rick users 0 2007-06-16 17:18 /tmp/my_fifo|
- Notice that the first character of output is a p, indicating a pipe. The | symbol at the end is added by the
- ls command's -F option and also indicates a pipe.

How above code Works

- The program uses the mkfifo function to create a special file. Although you ask for a mode of 0777, this is altered by the user mask (umask) setting (in this case 022), just as in normal file creation, so the resulting file has mode 755. If your umask is set differently, for example to 0002, you will see different permissions on the created file.
- You can remove the FIFO just like a conventional file by using the rm command, or from within a program by using the unlink system call.

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
- 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.

Accessing a FIFO

- 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
- $

How It Works

- Because there was no data in the FIFO, the cat and echo programs both block, waiting for some data to arrive and some other process to read the data, respectively.
- Looking at the third stage, the cat process is initially blocked in the background. When echo makes some data available, the cat command reads the data and prints it to the standard output. Notice that the cat program then exits without waiting for more data. It doesn’t block because the pipe will have been closed when the second command putting data in the FIFO completed, so calls to read in the cat program will return 0 bytes, indicating the end of file.