Lab-10: Inter Process Communication - Pipes

1. Objectives

   Demonstrate the ability for processes to communicate using pipe.
   Employ the dup and dup2 system calls in pipes.
   Use file descriptors for inter-process communications.
   Perform inter-process communications using named pipes.

2. Introduction

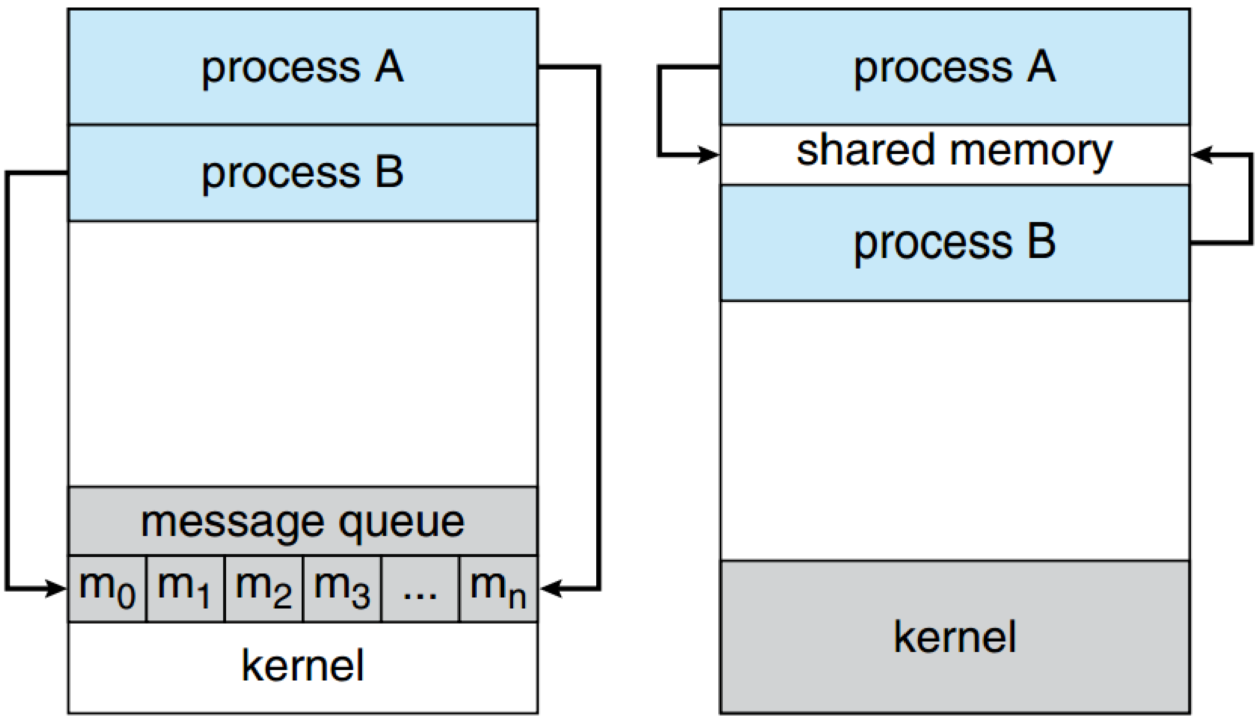
INTER PROCESS COMMUNICATION

Having multiple processes running is all good-and-well. Hey, it is one of the main reasons why the concept of an OS was introduced, remember? Right, **good job!**

It would make sense, though, if different processes were able to communicate with each other. That's what this Section is about.

There are two main techniques to facilitate communication between multiple processes. These two techniques are shown in image below.

1. Shared memory
2. Message passing



The two main techniques for inter process communication

source: SILBERSCHATZ, A., GALVIN, P.B., and GAGNE, G. *Operating System Concepts*. 9th ed. Hoboken: Wiley, 2013.

2.1. Shared memory

Shared memory is … memory that is shared. Normally multiple processes are not allowed to read/write to each other's memory space. This is enforced by the OS and we'll later discuss some details on how it does this. Errors, similar to the one in the example below, are generated by the OS if a processes try to access areas that it is not allowed to access.

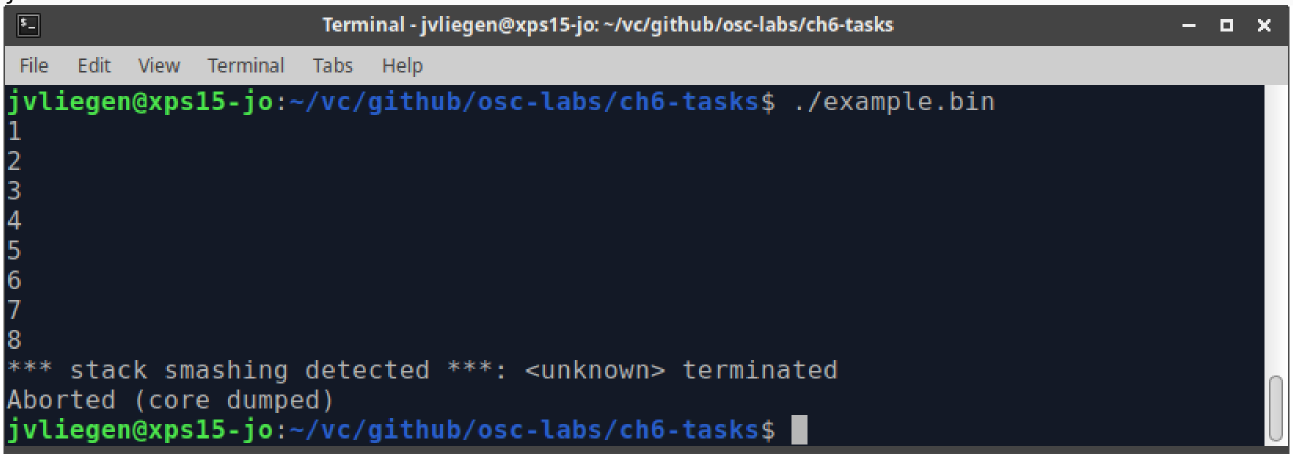
```c
#include <stdio.h>

int main(void) {
  int i, my_array[8];

  for(i=0;i<=20;i++) {
    my_array[i] = i+1;
  }

  for(i=0;i<8;i++) {
    printf("%d\n", my_array[i]);
  }

  return 0;
}
```


Terminal - jvliegen@xps15-jo: ~/vc/github/osc-labs/ch6-tasks
File Edit View Terminal Tabs Help

```
jvliegen@xps15-jo:~/vc/github/osc-labs/ch6-tasks$ ./example.bin
1
2
3
4
5
6
7
8
*** stack smashing detected ***: <unknown> terminated
Aborted (core dumped)
jvliegen@xps15-jo:~/vc/github/osc-labs/ch6-tasks$
```

An example of memory protection that given by the OS

Note that, depending on the OS, the error message might vary: on MacOS, it's simply `[1] 28507 abort ./example.bin`.

The code above exceeds the allowed stack space. Try to find out why this happens.

The technique of using **shared memory** allows other processes to gain access certain regions of the address space. Both processes have to be aware that the memory is **not** protected by the OS. A programming API for using shared memory is provided by POSIX (Portable Operating System Interface, a set of standards implemented by most UNIX OSes). The example below shows a **producer** on the left (a process which puts data inside of the shared memory) and a **consumer** on the right (which uses the data it gets from the producer).

```c
// PRODUCER
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/shm.h>
#include <sys/mman.h>

int main() {
  const int SIZE = 4096; /* buffersize (bytes) */
  const char *name = "OS"; /* shared memory object name */
  const char *data_0 = "Hello";
  const char *data_1 = "World!";
  int shm_fd; /* shared memory file descriptor */
  void *ptr;

  /* create the shared memory file descriptor */
  shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

  /* configure the size of the shared memory file */
  ftruncate(shm_fd, SIZE);

  /* memory map the shared memory file */
  ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

  /* write to the shared memory file */
  sprintf(ptr,"%s",data_0);
  ptr += strlen(data_0);
  sprintf(ptr,"%s",data_1);
  ptr += strlen(data_1);

  return 0;
}

// CONSUMER

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/shm.h>
#include <sys/mman.h>


int main() {
  const int SIZE = 4096; /* buffersize (bytes) */
  const char *name = "OS"; /* shared memory object name */


  int shm_fd; /* file descriptor */
  void *ptr;

  /* open the shared memory file */
  shm_fd = shm_open(name, O_RDONLY, 0666);


  /* memory map the shared memory file */
  ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);


  /* read from the shared memory file */
  printf("%s",(char *)ptr);

  /* remove the shared memory file */
  shm_unlink(name);

  return 0;
}
```

source: SILBERSCHATZ, A., GALVIN, P.B., and GAGNE, G. *Operating System Concepts*. 9th ed. Hoboken: Wiley, 2013.

Note: to compile these on Linux, you need to pass the `-lrt` flag to gcc ("link with library rt", see the librt interface documentation) like so: `gcc -o producer producer.c -lrt`. MacOS relies less on auxiliary external libraries; the flag can safely be omitted there.

This very simple example uses shared memory. Try to find answers to the questions below:

- Try to find out what these programs do
- What is the size of the memory that is shared?
- Can a *producer* read from the shared memory?
- Can a *consumer* write to the shared memory?
- How do both processes know which data is shared? In other words, how does the consumer decide which memory it connects to?
- Do both processes have to be active at the same time for the memory sharing to work? Why (not)?

Using shared memory is handy, but also not ideal in terms of security. For example, can you see any way in the API above to make sure only authorized programs can read/write to the shared memory? In this setup, any program which knows the name of the shared memory block and runs with sufficient permissions (remember `chmod`?) can also access the shared memory.

2.2. Message passing

The second technique for for InterProcess Communication (IPC) comes in the form of message passing. This method is a bit more restricted than using raw shared memory, but also easier to use and safer because of that. Here we touch on 2 different mechanisms for achieving this: **signals** and **pipes**.

Signals

Signals are the cheapest form of IPC. They literally allow one process to send a signal to another process, through the use of the function `kill()`. Although due to historical reasons the name might be a bit misleading (originally, the only defined signal was used to stop a process, other uses only came later), it can be used to send different signals. Signals here are very simply numbers with a predefined meaning (an "enum" if you will) that you can send to another process. Depending on the number received, a different action is expected to be taken. For example, if you use the `Ctrl+c` keyboard shortcut to stop a running program, the shell sends a `SIGINT` (signal interrupt) to the current program. Similarly, `Ctrl+z` will send a `SIGTSTP` signal.

The snippet below shows the different types of signals that can be sent. Many of them have to do with stopping other processes, but with subtly different effects (for example, stop a process—immediately—versus allowing it to safely exit itself).

```
jvliegen@localhost:~/$ kill -L
 1) SIGHUP   2) SIGINT   3) SIGQUIT  4) SIGILL   5) SIGTRAP
 6) SIGABRT  7) SIGBUS   8) SIGFPE   9) SIGKILL 10) SIGUSR1
11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN 22) SIGTTOU 23) SIGURG  24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH  29) SIGIO 30) SIGPWR
31) SIGSYS  34) SIGRTMIN  35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

There is also has CLI-compatible command `kill` that can send these signals to any running process (addressed by their PID). For more information on the kill command, add the `--help` argument, read the man-page (`man kill`).

Let’s illustrate this with an example:

```c
#include <stdio.h>
#include <unistd.h>

#define DURATION_IN_MINUTES 10

int main(void) {
  int i = DURATION_IN_MINUTES * 60;

  for(;i>=0;i--) {
    printf("TIMER: 00:%02d:%02d\n", (int)((i-i%60)/60), i%60);
    sleep(1);
  }

  return 0;
}
```

This program will emulate an egg timer. Every second it displays how much time is left. Once the process starts running, it takes 10 minutes to complete. This process can be stopped by just pressing CTRL+C. Note that you don’t have to manually do anything for this to work: your program automatically listens for this signal and exits the program when it is received. This is achieved through a combination of the `gcc` compiler (which adds code for signal handling) and the OS executing that code when needed.

```
jvliegen@localhost:~/$ ./egg_timer.bin
TIMER: 00:10:00
TIMER: 00:09:59
```

```
TIMER: 00:09:58
TIMER: 00:09:57
TIMER: 00:09:56
^C
```

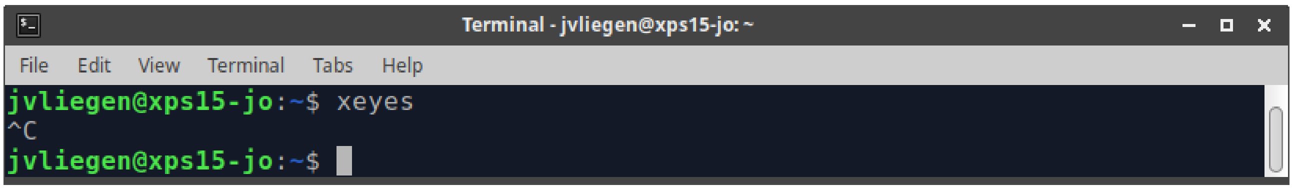
Another way to kill the process would be to explicitly send the signal through the kill command. To use this command, the **PID** is needed as an argument. Through a new CLI-window, this PID has to be searched for first. Note that the type of signal is an argument in the command.

```
jvliegen@localhost:~/$  ps ux | grep timer
jvliegen  5041  0.0  0.0   4504   772 pts/1    S+   06:04   0:00 ./egg_timer.bin
jvliegen  5066  0.0  0.0  21996  1080 pts/2    S+   06:05   0:00 grep --color=auto
timer
jvliegen@localhost:~/$  kill -KILL 5041
```

Try this for yourself. You can also use the `./longhello` program from Section 6.1. Run this program and try to kill it using **both** approaches that were explained above. (Of course this means you shoud run it again, after you killed it the first time 😃)

Although there are numerous uses for sending signals between signals, one more example is interesting to have a closer look at. Above there was already some hinting to **CTRL+Z**.

The CLI is running a shell, as you already know by now. This offers just a single interface. If you were to start a program, that CLI is occupied (you cannot type or execute any commands). Imagine you are working remotely on a server (e.g., through ssh): this would require you to open up a new connection to the server and have a second shell at your disposal every time you executed a longer running command (e.g., starting a web server). A more convenient solution would be to send the running program to the **background**.



```
jvliegen@xps15-jo:~$ xeyes
^C
jvliegen@xps15-jo:~$
```

An example of a program that needs to be killed with CTRL-C

Before you can send processes to the background, the process has to be halted first. This can be done through the CTRL+Z shortcut. With a halted process, the command **bg** sends the halted process to the background. If you do not send it to the background, the process will *freeze*. Once it is in the background it *unfreezes* and continues running. Additionally, this gives you back your shell.

```
jvliegen@localhost:~/$ xeyes
^Z
[1]+  Stopped                 xeyes
jvliegen@localhost:~/$ bg
[1]+ xeyes &
jvliegen@localhost:~/$
```

For the sake of completeness we enumerate a few more usefull aspects about this:

- a process can be started in the background as well. This can be achieved by adding an `&` (ampersand) after the command (e.g., `xeyes &`)
- the command `jobs` gives you an overview of which jobs are running in the background
- through the command `fg <#>` the job with index number <#> will pulled to foreground.

Try this for yourself. If the `xeyes` program is not installed, install it first or use the `longhello` program from before.

Pipes

Another option to achieve message sending is through **pipes**. There are two different types of pipes available:

- anonymous pipes
- named pipes

Anonymous pipes are like waterslides. You can put some data on it on one end (the top of the slide) and it comes out the other (the bottom), but it's not possible to go up the waterslide from the bottom. Put differently: communication is half-duplex (single direction). One process can **write** into the pipe, while the other can **read** from of the pipe. This type of pipe can only be create between two processes that have parent-child relationship. What happens internally is that the **stdout** of the first process is mapped to the **stdin** of the second process. For this, we use the | (pipe) character.

When using the CLI, anonymous pipes are a very powerful tool for chaining different commands. The output of the first command will be the input for the next command. This can be chained multiple times.

```
jvliegen@localhost:~/$ xeyes &
jvliegen@localhost:~/$ ps -ux | grep xeyes | head -1 | tr -s " " | cut -d " " -f 3
5526
jvliegen@localhost:~/$
```

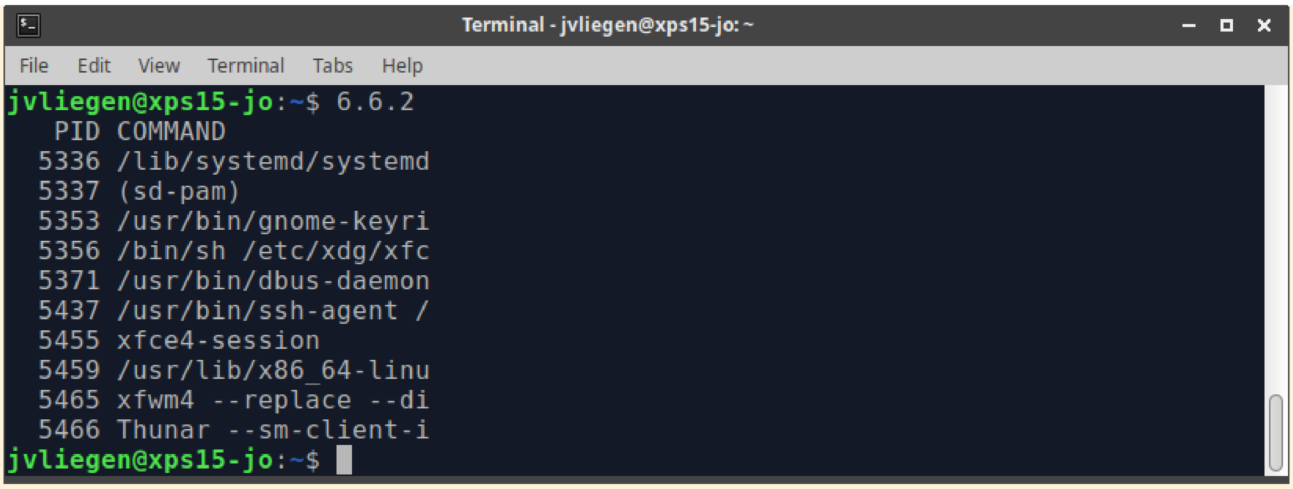
The example above chains the following:

1. give a list of all *my* processes (`ps` = process status)
2. only filter the lines that contain the word *xeyes* (`grep` stands for Global Regular Expression Print)
3. filter only the first line (`head`)
4. squash all spaces `ps` produces together to better prepare for step 5:
5. split the input on a space (" ") and report only the third field (which is the process ID)

Let's try something similar for yourselves:

- Use anonymous pipes to display all the processes of which you are the owner. From these processes only display the PID and the first 10 characters of the

process’s name (the COMMAND column). From this list, only show the first 10 processes.
- Then, add another command to sort the output by descending PID (so the largest PID is on top, the smallest on the bottom)

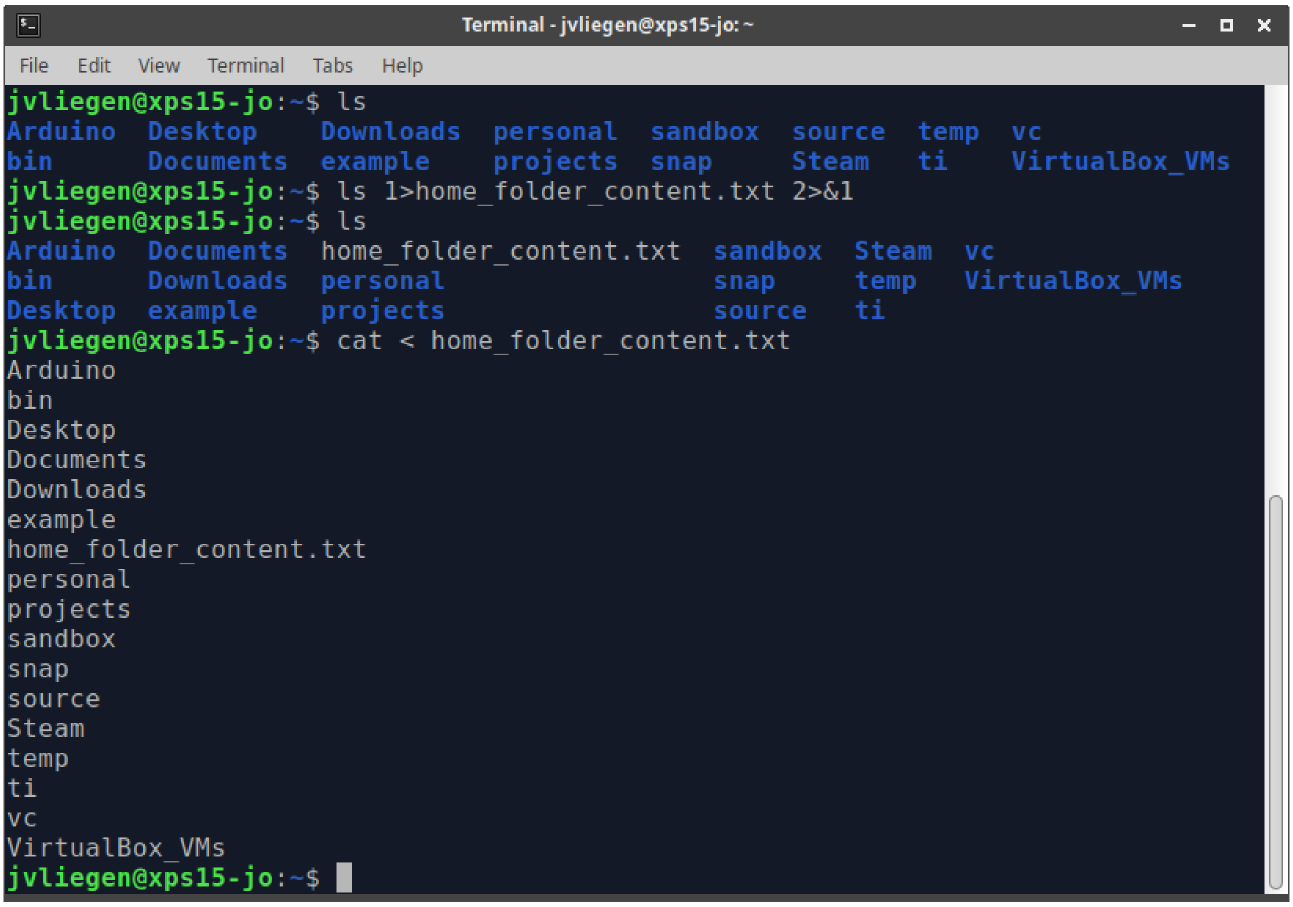

Terminal - jvliegen@xps15-jo: ~
File Edit View Terminal Tabs Help

```
jvliegen@xps15-jo:~$ 6.6.2
   PID COMMAND
  5336 /lib/systemd/systemd
  5337 (sd-pam)
  5353 /usr/bin/gnome-keyri
  5356 /bin/sh /etc/xdg/xfc
  5371 /usr/bin/dbus-daemon
  5437 /usr/bin/ssh-agent /
  5455 xfce4-session
  5459 /usr/lib/x86_64-linu
  5465 xfwm4 --replace --di
  5466 Thunar --sm-client-i
jvliegen@xps15-jo:~$
```

An example output

Do you remember the Process Control Block? This has one field called **list of open files**. We’ve already touched upon stdin, stdout and stderr. Using anonymous pipes will add an entry to this list.

We can also **relink** the 3 default open files to other targets. For example, instead of writing output and errors to the command line, we can redirect them to a file. Similarly, we can read input from a file instead of from the keyboard:


Terminal - jvliegen@xps15-jo: ~
File Edit View Terminal Tabs Help

```
jvliegen@xps15-jo:~$ ls
Arduino  Desktop    Downloads  personal  sandbox  source  temp  vc
bin      Documents  example    projects  snap     Steam   ti    VirtualBox_VMs
jvliegen@xps15-jo:~$ ls 1>home_folder_content.txt 2>&1
jvliegen@xps15-jo:~$ ls
Arduino  Documents  home_folder_content.txt  sandbox  Steam  vc
bin      Downloads  personal                 snap     temp   VirtualBox_VMs
Desktop  example    projects                 source   ti
jvliegen@xps15-jo:~$ cat < home_folder_content.txt
Arduino
bin
Desktop
Documents
Downloads
example
home_folder_content.txt
personal
projects
sandbox
snap
source
Steam
temp
ti
vc
VirtualBox_VMs
jvliegen@xps15-jo:~$ 
```

Redirection of the standard output

The syntax for this is a bit weird though: 1> is meant to redirect data that normally goes to stdout, while 2> is used to relink stderr. You can also point directly to the existing stdout/stderr by using &1 or &2 respectively:

- process 1>{STDOUT} 2>{STDERR}
- process 1>{STDOUT} 2>&1
- process < {STDIN} (read from a file at location {STDIN} rather than from the keyboard)

Note that here we're using the > pipe here instead of | as above. The difference is subtle, but a simple explanation is that > deals with mapping a command to a *file* (or something that pretends to be a file, like stdout/stderr), while | maps a command to *another command*.

Since STDIN is "just a file" in Linux, we can also read from that file to get input into our program.
Look up yourself online how to read data from STDIN and write a small C program that reads (some) data from STDIN and prints it to STDOUT using `printf()`.
Then use a command like this to have data from 1 file pass through your program and into another:
`cat input_file.txt | ./my_program 1> output_file.txt`
Also try to do `input_file.txt > ./my_program 1> output_file.txt` and explain why that doesn't (seem to) work.

Named pipes are the other type of pipes that can be created. The main differences with anonymous pipes are the lifetime of this mechanism and their presence in the file system.

The anonymous pipes above only live for as long as the processes live. Named pipes instead persist and have to be closed explicitly (or are closes automatically at system-shutdown).

Named pipes also have an actual presence in the file system. That is, they show up as files. But unlike most files, they never appear to have contents. Even if you write a lot of data to a named pipe, the file appears to be empty. Making named pipes can be done through the `mkfifo` command.

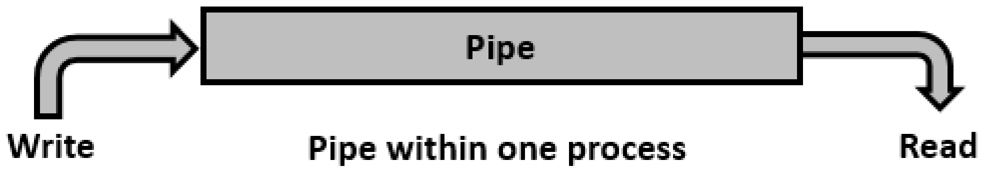
As they are not frequently used, we direct the interested reader to `man` pages.

2.3.

3.

Pipe is a communication medium between two or more related or interrelated processes. It can be either within one process or a communication between the child and the parent processes. Communication can also be multi-level such as communication between the parent, the child and the grand-child, etc. Communication is achieved by one process writing into the pipe and other reading from the pipe. To achieve the pipe system call, create two files, one to write into the file and another to read from the file.

Pipe mechanism can be viewed with a real-time scenario such as filling water with the pipe into some container, say a bucket, and someone retrieving it, say with a mug. The filling process is nothing but writing into the pipe and the reading process is nothing but retrieving from the pipe. This implies that one output (water) is input for the other (bucket).



```
#include<unistd.h>

int pipe(int pipedes[2]);
```

This system call would create a pipe for one-way communication i.e., it creates two descriptors, first one is connected to read from the pipe and other one is connected to write into the pipe.

Descriptor pipedes[0] is for reading and pipedes[1] is for writing. Whatever is written into pipedes[1] can be read from pipedes[0].

This call would return zero on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Even though the basic operations for file are read and write, it is essential to open the file before performing the operations and closing the file after completion of the required operations. Usually, by default, 3 descriptors opened for every process, which are used for input (standard input – stdin), output (standard output – stdout) and error (standard error – stderr) having file descriptors 0, 1 and 2 respectively.

This system call would return a file descriptor used for further file operations of read/write/seek (lseek). Usually file descriptors start from 3 and increase by one number as the number of files open.

The arguments passed to open system call are pathname (relative or absolute path), flags mentioning the purpose of opening file (say, opening for read, O_RDONLY, to write, O_WRONLY, to read and write, O_RDWR, to append to the existing file O_APPEND, to create file, if not exists with O_CREAT and so on) and the required mode providing permissions of read/write/execute for user or owner/group/others. Mode can be mentioned with symbols.

Read – 4, Write – 2 and Execute – 1.

For example: Octal value (starts with 0), 0764 implies owner has read, write and execute permissions, group has read and write permissions, other has read permissions. This can also be represented as S_IRWXU | S_IRGRP | S_IWGRP | S_IROTH, which implies or operation of 0700|0040|0020|0004 → 0764.

This system call, on success, returns the new file descriptor id and -1 in case of error. The cause of error can be identified with errno variable or perror() function.

```
#include<unistd.h>

int close(int fd)
```

The above system call closing already opened file descriptor. This implies the file is no longer in use and resources associated can be reused by any other process. This system call returns zero on success and -1 in case of error. The cause of error can be identified with errno variable or perror() function.

```
#include<unistd.h>

ssize_t read(int fd, void *buf, size_t count)
```

The above system call is to read from the specified file with arguments of file descriptor fd, proper buffer with allocated memory (either static or dynamic) and the size of buffer.

The file descriptor id is to identify the respective file, which is returned after calling open() or pipe() system call. The file needs to be opened before reading from the file. It automatically opens in case of calling pipe() system call.

This call would return the number of bytes read (or zero in case of encountering the end of the file) on success and -1 in case of failure. The return bytes can be smaller than the number of bytes requested, just in case no data is available or file is closed. Proper error number is set in case of failure.

To know the cause of failure, check with errno variable or perror() function.

```
#include<unistd.h>

ssize_t write(int fd, void *buf, size_t count)
```

The above system call is to write to the specified file with arguments of the file descriptor fd, a proper buffer with allocated memory (either static or dynamic) and the size of buffer.

The file descriptor id is to identify the respective file, which is returned after calling open() or pipe() system call.

The file needs to be opened before writing to the file. It automatically opens in case of calling pipe() system call.

This call would return the number of bytes written (or zero in case nothing is written) on success and -1 in case of failure. Proper error number is set in case of failure.

To know the cause of failure, check with errno variable or perror() function.

4. Example Programs

Following are some example programs.

4.1. **Example program 1** – Program to write and read two messages using pipe.

4.2. Algorithm

- ✓ **Step 1** – Create a pipe.
- ✓ **Step 2** – Send a message to the pipe.
- ✓ **Step 3** – Retrieve the message from the pipe and write it to the standard output.
- ✓ **Step 4** – Send another message to the pipe.
- ✓ **Step 5** – Retrieve the message from the pipe and write it to the standard output.

Note – Retrieving messages can also be done after sending all messages.

Source Code: simplepipe.c

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

int main() {
   int pipefds[2];
   int returnstatus;
   char writemessages[2][20]={"Hi", "Hello"};
   char readmessage[20];
   returnstatus = pipe(pipefds);

   if (returnstatus == -1) {
      printf("Unable to create pipe\n");
      return 1;
   }
   printf("Writing to pipe - Message 1 is %s\n",
writemessages[0]);
   write(pipefds[1], writemessages[0],
sizeof(writemessages[0]));
   read(pipefds[0], readmessage, sizeof(readmessage));
   printf("Reading from pipe – Message 1 is %s\n",
readmessage);
   printf("Writing to pipe - Message 2 is %s\n",
writemessages[0]);
   write(pipefds[1], writemessages[1],
sizeof(writemessages[0]));
   read(pipefds[0], readmessage, sizeof(readmessage));
   printf("Reading from pipe – Message 2 is %s\n",
readmessage);
   return 0;
}
```

Note − Ideally, return status needs to be checked for every system call. To simplify the process, checks are not done for all the calls.

4.3. Execution Steps

✓ **Compilation**

gcc -o simplepipe simplepipe.c

✓ **Execution/Output**

```
Writing to pipe - Message 1 is Hi
Reading from pipe – Message 1 is Hi
Writing to pipe - Message 2 is Hi
Reading from pipe – Message 2 is Hell
```

4.4. **Example program 2** − Program to write and read two messages through the pipe using the parent and the child processes.

✓ **Algorithm**
✓ **Step 1** − Create a pipe.
✓ **Step 2** − Create a child process.
✓ **Step 3** − Parent process writes to the pipe.
✓ **Step 4** − Child process retrieves the message from the pipe and writes it to the standard output.
✓ **Step 5** − Repeat step 3 and step 4 once again.
✓ **Source Code: pipewithprocesses.c**

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

int main() {
   int pipefds[2];
   int returnstatus;
   int pid;
   char writemessages[2][20]={"Hi", "Hello"};
   char readmessage[20];
   returnstatus = pipe(pipefds);
   if (returnstatus == -1) {
      printf("Unable to create pipe\n");
      return 1;
   }
   pid = fork();

   // Child process
   if (pid == 0) {
      read(pipefds[0], readmessage, sizeof(readmessage));
      printf("Child Process - Reading from pipe – Message 1
is %s\n", readmessage);
      read(pipefds[0], readmessage, sizeof(readmessage));
      printf("Child Process - Reading from pipe – Message 2
is %s\n", readmessage);
   } else { //Parent process
      printf("Parent Process - Writing to pipe - Message 1
is %s\n", writemessages[0]);
      write(pipefds[1], writemessages[0],
sizeof(writemessages[0]));
      printf("Parent Process - Writing to pipe - Message 2
is %s\n", writemessages[1]);
      write(pipefds[1], writemessages[1],
sizeof(writemessages[1]));
   }
   return 0;
}
```

4.5. Execution Steps

✓ **Compilation**

gcc pipewithprocesses.c –o pipewithprocesses

✓ **Execution**

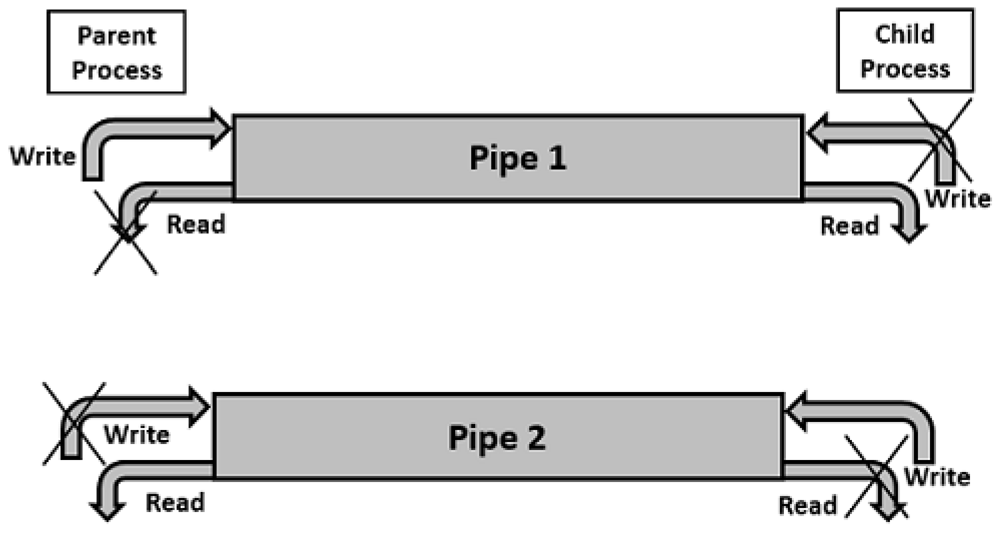
```
Parent Process - Writing to pipe - Message 1 is Hi
Parent Process - Writing to pipe - Message 2 is Hello
Child Process - Reading from pipe – Message 1 is Hi
Child Process - Reading from pipe – Message 2 is Hello
```

5. **Two-way Communication Using Pipes**

Pipe communication is viewed as only one-way communication i.e., either the parent process writes and the child process reads or vice-versa but not both. However, what if both the parent and the child needs to write and read from the pipes simultaneously, the solution is a two-way communication using pipes. Two pipes are required to establish two-way communication.

Following are the steps to achieve two-way communication −

- ✓ **Step 1** − Create two pipes. First one is for the parent to write and child to read, say as pipe1. Second one is for the child to write and parent to read, say as pipe2.
- ✓ **Step 2** − Create a child process.
- ✓ **Step 3** − Close unwanted ends as only one end is needed for each communication.
- ✓ **Step 4** − Close unwanted ends in the parent process, read end of pipe1 and write end of pipe2.
- ✓ **Step 5** − Close the unwanted ends in the child process, write end of pipe1 and read end of pipe2.
- ✓ **Step 6** − Perform the communication as required.



6. **Sample Programs**

6.1. **Sample program 1** − Achieving two-way communication using pipes.

Algorithm

- ✓ **Step 1** – Create pipe1 for the parent process to write and the child process to read.
- ✓ **Step 2** – Create pipe2 for the child process to write and the parent process to read.
- ✓ **Step 3** – Close the unwanted ends of the pipe from the parent and child side.
- ✓ **Step 4** – Parent process to write a message and child process to read and display on the screen.
- ✓ **Step 5** – Child process to write a message and parent process to read and display on the screen.
- ✓ **Source Code: twowayspipe.c**

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

int main() {
   int pipefds1[2], pipefds2[2];
   int returnstatus1, returnstatus2;
   int pid;
   char pipe1writemessage[20] = "Hi";
   char pipe2writemessage[20] = "Hello";
   char readmessage[20];
   returnstatus1 = pipe(pipefds1);

   if (returnstatus1 == -1) {
      printf("Unable to create pipe 1 \n");
      return 1;
   }
   returnstatus2 = pipe(pipefds2);

   if (returnstatus2 == -1) {
      printf("Unable to create pipe 2 \n");
      return 1;
   }
   pid = fork();

   if (pid != 0) // Parent process {
      close(pipefds1[0]); // Close the unwanted pipe1 read
side
      close(pipefds2[1]); // Close the unwanted pipe2 write
side
      printf("In Parent: Writing to pipe 1 – Message is
%s\n", pipe1writemessage);
      write(pipefds1[1], pipe1writemessage,
sizeof(pipe1writemessage));
```

```
      read(pipefds2[0], readmessage, sizeof(readmessage));
      printf("In Parent: Reading from pipe 2 – Message is
%s\n", readmessage);
   } else { //child process
      close(pipefds1[1]); // Close the unwanted pipe1 write
side
      close(pipefds2[0]); // Close the unwanted pipe2 read
side
      read(pipefds1[0], readmessage, sizeof(readmessage));
      printf("In Child: Reading from pipe 1 – Message is
%s\n", readmessage);
      printf("In Child: Writing to pipe 2 – Message is
%s\n", pipe2writemessage);
      write(pipefds2[1], pipe2writemessage,
sizeof(pipe2writemessage));
   }
   return 0;
}
```

7. **Execution Steps**

✓ **Compilation**

```
gcc twowayspipe.c –o twowayspipe
```

✓ **Execution**

```
In Parent: Writing to pipe 1 – Message is Hi
In Child: Reading from pipe 1 – Message is Hi
In Child: Writing to pipe 2 – Message is Hello
In Parent: Reading from pipe 2 – Message is Hello
```

8. Inter Process Communication - Named Pipes

Pipes were meant for communication between related processes. Can we use pipes for unrelated process communication, say, we want to execute client program from one terminal and the server program from another terminal? The answer is No. Then how can we achieve unrelated processes communication, the simple answer is Named Pipes. Even though this works for related processes, it gives no meaning to use the named pipes for related process communication.

We used one pipe for one-way communication and two pipes for bi-directional communication. Does the same condition apply for Named Pipes. The answer is no, we can use single named pipe that can be used for two-way communication (communication between the server and the client, plus the client and the server at the same time) as Named Pipe supports bi-directional communication.

Another name for named pipe is **FIFO (First-In-First-Out)**. Let us see the system call (mknod()) to create a named pipe, which is a kind of a special file.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int mknod(const char *pathname, mode_t mode, dev_t dev);
```

This system call would create a special file or file system node such as ordinary file, device file, or FIFO. The arguments to the system call are pathname, mode and dev. The pathname along with the attributes of mode and device information. The pathname is relative, if the directory is not specified it would be created in the current directory. The mode specified is the mode of file which specifies the file type such as the type of file and the file mode as mentioned in the following tables. The dev field is to specify device information such as major and minor device numbers.

File Type	**Description**	**File Type**	**Description**
S_IFBLK	block special	S_IFREG	Regular file
S_IFCHR	character special	S_IFDIR	Directory
S_IFIFO	FIFO special	S_IFLNK	Symbolic Link
File Mode	**Description**	**File Mode**	**Description**
S_IRWXU	Read, write, execute/search by owner	S_IWGRP	Write permission, group
S_IRUSR	Read permission, owner	S_IXGRP	Execute/search permission, group
S_IWUSR	Write permission, owner	S_IRWXO	Read, write, execute/search by others

S_IXUSR	Execute/search permission, owner	S_IROTH	Read permission, others
S_IRWXG	Read, write, execute/search by group	S_IWOTH	Write permission, others
S_IRGRP	Read permission, group	S_IXOTH	Execute/search permission, others

File mode can also be represented in octal notation such as 0XYZ, where X represents owner, Y represents group, and Z represents others. The value of X, Y or Z can range from 0 to 7. The values for read, write and execute are 4, 2, 1 respectively. If needed in combination of read, write and execute, then add the values accordingly.

Say, if we mention, 0640, then this means read and write (4 + 2 = 6) for owner, read (4) for group and no permissions (0) for others.

This call would return zero on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

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

int mkfifo(const char *pathname, mode_t mode)
```

This library function creates a FIFO special file, which is used for named pipe. The arguments to this function is file name and mode. The file name can be either absolute path or relative path. If full path name (or absolute path) is not given, the file would be created in the current folder of the executing process. The file mode information is as described in mknod() system call.

This call would return zero on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

Let us consider a program of running the server on one terminal and running the client on another terminal. The program would only perform one-way communication. The client accepts the user input and sends the message to the server, the server prints the message on the output. The process is continued until the user enters the string "end".

Let us understand this with an example –

Step 1 – Create two processes, one is fifoserver and another one is fifoclient.

Step 2 – Server process performs the following –

- Creates a named pipe (using system call mknod()) with name "MYFIFO", if not created.
- Opens the named pipe for read only purposes.
- Here, created FIFO with permissions of read and write for Owner. Read for Group and no permissions for Others.
- Waits infinitely for message from the Client.
- If the message received from the client is not "end", prints the message. If the message is "end", closes the fifo and ends the process.

Step 3 – Client process performs the following –

- Opens the named pipe for write only purposes.
- Accepts the string from the user.
- Checks, if the user enters "end" or other than "end". Either way, it sends a message to the server. However, if the string is "end", this closes the FIFO and also ends the process.
- Repeats infinitely until the user enters string "end".

Now let's take a look at the FIFO server file.

```
/* Filename: fifoserver.c */
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define FIFO_FILE "MYFIFO"
int main() {
   int fd;
   char readbuf[80];
   char end[10];
   int to_end;
   int read_bytes;

   /* Create the FIFO if it does not exist */
   mknod(FIFO_FILE, S_IFIFO|0640, 0);
   strcpy(end, "end");
   while(1) {
      fd = open(FIFO_FILE, O_RDONLY);
      read_bytes = read(fd, readbuf, sizeof(readbuf));
      readbuf[read_bytes] = '\0';
      printf("Received string: \"%s\" and length is %d\n",
readbuf, (int)strlen(readbuf));
      to_end = strcmp(readbuf, end);
      if (to_end == 0) {
         close(fd);
         break;
      }
   }
   return 0;
}
```

Compilation and Execution Steps

```
Received string: "this is string 1" and length is 16
Received string: "fifo test" and length is 9
Received string: "fifo client and server" and length is 22
Received string: "end" and length is 3
```

Now, let’s take a look at the FIFO client sample code.

```
/* Filename: fifoclient.c */
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define FIFO_FILE "MYFIFO"
int main() {
   int fd;
   int end_process;
   int stringlen;
   char readbuf[80];
   char end_str[5];
   printf("FIFO_CLIENT: Send messages, infinitely, to end
enter \"end\"\n");
   fd = open(FIFO_FILE, O_CREAT|O_WRONLY);
   strcpy(end_str, "end");

   while (1) {
      printf("Enter string: ");
      fgets(readbuf, sizeof(readbuf), stdin);
      stringlen = strlen(readbuf);
      readbuf[stringlen - 1] = '\0';
      end_process = strcmp(readbuf, end_str);
      //printf("end_process is %d\n", end_process);
      if (end_process != 0) {
         write(fd, readbuf, strlen(readbuf));
         printf("Sent string: \"%s\" and string length is
%d\n", readbuf, (int)strlen(readbuf));
      } else {
         write(fd, readbuf, strlen(readbuf));
         printf("Sent string: \"%s\" and string length is
%d\n", readbuf, (int)strlen(readbuf));
         close(fd);
         break;
      }
   }
   return 0;
}
```

Let's take a at the arriving output.

Compilation and Execution Steps

```
FIFO_CLIENT: Send messages, infinitely, to end enter "end"
Enter string: this is string 1
Sent string: "this is string 1" and string length is 16
Enter string: fifo test
Sent string: "fifo test" and string length is 9
Enter string: fifo client and server
Sent string: "fifo client and server" and string length is 22
Enter string: end
Sent string: "end" and string length is 3
```

9. Two-way Communication Using Named Pipes

The communication between pipes are meant to be unidirectional. Pipes were restricted to one-way communication in general and need at least two pipes for two-way communication. Pipes are meant for inter-related processes only. Pipes can't be used for unrelated processes communication, say, if we want to execute one process from one terminal and another process from another terminal, it is not possible with pipes. Do we have any simple way of communicating between two processes, say unrelated processes in a simple way? The answer is YES.

Named pipe is meant for communication between two or more unrelated processes and can also have bi-directional communication.

Already, we have seen the one-directional communication between named pipes, i.e., the messages from the client to the server. Now, let us take a look at the bi-directional communication i.e., the client sending message to the server and the server receiving the message and sending back another message to the client using the same named pipe.

Following is an example −

Step 1 − Create two processes, one is fifoserver_twoway and another one is fifoclient_twoway.

Step 2 − Server process performs the following −

- Creates a named pipe (using library function mkfifo()) with name "fifo_twoway" in /tmp directory, if not created.
- Opens the named pipe for read and write purposes.
- Here, created FIFO with permissions of read and write for Owner. Read for Group and no permissions for Others.
- Waits infinitely for a message from the client.
- If the message received from the client is not "end", prints the message and reverses the string. The reversed string is sent back to the client. If the message is "end", closes the fifo and ends the process.

Step 3 − Client process performs the following −

- Opens the named pipe for read and write purposes.
- Accepts string from the user.
- Checks, if the user enters "end" or other than "end". Either way, it sends a message to the server. However, if the string is "end", this closes the FIFO and also ends the process.
- If the message is sent as not "end", it waits for the message (reversed string) from the client and prints the reversed string.
- Repeats infinitely until the user enters the string "end".

Now, let's take a look at FIFO server sample code.

```
/* Filename: fifoserver_twoway.c */
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
```

```
#define FIFO_FILE "/tmp/fifo_twoway"
void reverse_string(char *);
int main() {
   int fd;
   char readbuf[80];
   char end[10];
   int to_end;
   int read_bytes;

   /* Create the FIFO if it does not exist */
   mkfifo(FIFO_FILE, S_IFIFO|0640);
   strcpy(end, "end");
   fd = open(FIFO_FILE, O_RDWR);
   while(1) {
      read_bytes = read(fd, readbuf, sizeof(readbuf));
      readbuf[read_bytes] = '\0';
      printf("FIFOSERVER: Received string: \"%s\" and length
is %d\n", readbuf, (int)strlen(readbuf));
      to_end = strcmp(readbuf, end);

      if (to_end == 0) {
         close(fd);
         break;
      }
      reverse_string(readbuf);
      printf("FIFOSERVER: Sending Reversed String: \"%s\" and
length is %d\n", readbuf, (int) strlen(readbuf));
      write(fd, readbuf, strlen(readbuf));
      /*
      sleep - This is to make sure other process reads this,
otherwise this
      process would retrieve the message
      */
      sleep(2);
   }
   return 0;
}

void reverse_string(char *str) {
   int last, limit, first;
   char temp;
   last = strlen(str) - 1;
   limit = last/2;
   first = 0;

   while (first < last) {
      temp = str[first];
      str[first] = str[last];
      str[last] = temp;
      first++;
      last--;
   }
   return;
}
```

Compilation and Execution Steps

```
FIFOSERVER: Received string: "LINUX IPCs" and length is 10
FIFOSERVER: Sending Reversed String: "sCPI XUNIL" and length is 10
FIFOSERVER: Received string: "Inter Process Communication" and length is 27
FIFOSERVER: Sending Reversed String: "noitacinummoC ssecorP retnI" and length is
27
FIFOSERVER: Received string: "end" and length is 3
```

Now, let's take a look at FIFO client sample code.

```
/* Filename: fifoclient_twoway.c */
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define FIFO_FILE "/tmp/fifo_twoway"
int main() {
   int fd;
   int end_process;
   int stringlen;
   int read_bytes;
   char readbuf[80];
   char end_str[5];
   printf("FIFO_CLIENT: Send messages, infinitely, to end
enter \"end\"\n");
   fd = open(FIFO_FILE, O_CREAT|O_RDWR);
   strcpy(end_str, "end");

   while (1) {
      printf("Enter string: ");
      fgets(readbuf, sizeof(readbuf), stdin);
      stringlen = strlen(readbuf);
      readbuf[stringlen - 1] = '\0';
      end_process = strcmp(readbuf, end_str);

      //printf("end_process is %d\n", end_process);
      if (end_process != 0) {
         write(fd, readbuf, strlen(readbuf));
         printf("FIFOCLIENT: Sent string: \"%s\" and string
length is %d\n", readbuf, (int)strlen(readbuf));
         read_bytes = read(fd, readbuf, sizeof(readbuf));
         readbuf[read_bytes] = '\0';
         printf("FIFOCLIENT: Received string: \"%s\" and
length is %d\n", readbuf, (int)strlen(readbuf));
      } else {
         write(fd, readbuf, strlen(readbuf));
         printf("FIFOCLIENT: Sent string: \"%s\" and string
length is %d\n", readbuf, (int)strlen(readbuf));
         close(fd);
         break;
      }
   }
   return 0;
}
```

Compilation and Execution Steps

```
FIFO_CLIENT: Send messages, infinitely, to end enter "end"
Enter string: LINUX IPCs
FIFOCLIENT: Sent string: "LINUX IPCs" and string length is 10
FIFOCLIENT: Received string: "sCPI XUNIL" and length is 10
Enter string: Inter Process Communication
FIFOCLIENT: Sent string: "Inter Process Communication" and string length is 27
FIFOCLIENT: Received string: "noitacinummoC ssecorP retnI" and length is 27
Enter string: end
FIFOCLIENT: Sent string: "end" and string length is 3
```