



西北工业大学  
NORTHWESTERN POLYTECHNICAL UNIVERSITY

# Lab 3 Report

**Report Subject: OS Experiment - Lab 3**

**Student ID: 2018380039**

**Student Name: Dikshya Kafle**

**Submission Date: 2021/11/18**

**Computer Operating System Experiment**

## **Laboratory 3**

### **Inter-Process Communication**

#### **1) Objective:**

Internalize a couple of important facts about IPC in Linux. Including:

- ❖ Shared memory
- ❖ Ordinary Pipes
- ❖ Named Pipes

#### **2) Equipment:**

- ❖ VirtualBox with Ubuntu Linux

#### **3) Experiments:**

##### **A. Experiment 1: shared memory communication**

- ❖ The code:

Open  sharedmemory.c ~/Desktop/Lab 3 OS

sharedmemory.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <fcntl.h>
5 #include <sys/types.h>
6 #include <errno.h>
7 #include <stdlib.h>
8 #include <unistd.h>
9 #include <sys/wait.h>
10 #include <sys/ipc.h>
11 #include <sys/shm.h>
12 #include <sys/stat.h>
13 #include <sys/mman.h>
14
15 int main () {
16
17     const int SIZE =4096;//Size of the shared memory in bytes
18     const char *name="SHARED_PC";//Name of the shared memory object
19     int shm_fd;//Shared memory file descriptor
20     void *ptr;//Pointed to the shared memory object
21     const char *greetings_msg="greetings to parents!";
22
23     pid_t pid;
24     int exit_status;
25
26     //Parent process creates a shared memory
27     shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);//Creates shm object
28     ftruncate(shm_fd, SIZE);//Configure size
29     ptr= mmap (0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);//Memory map to the shm object
30
31     //Parent process spawn a child process
32     pid = fork();
33
34     if(pid<0){
35         fprintf (stderr,"Pipe failed");
36         return 1;
37     }else if(pid>0){
38         //parent process
39         printf("I am parent process =%d \n", getpid());
40         wait(&exit_status);//Waiting for child process to exit
41
42         if(exit_status==0) {
43             printf ("Child process exited, exit code: %d, Parent process reading from shared memory... \n",exit_status);
44             printf("Message: ");
45             printf ("%s\n", (char *)ptr);//Read from the object memory
46             /*shm_unlink(name);*/
47         }else { //Child process
48             printf("I am child process %d my parent is process %d \n", getpid(), getppid());
49             shm_fd=shm_open(name, O_RDWR, 0666);//Open the shared memory
50             ptr=mmap (0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);//Memory map
51             sprintf(ptr, "%s", greetings_msg);//Write to the shared memory
52             ptr+= strlen (greetings_msg);//Increase ptr
53             exit(0); }
```

❖ The results:

```
dikshya@dikshya-VirtualBox: ~/Desktop/Lab 3 OS
dikshya@dikshya-VirtualBox:~/Desktop/Lab 3 OS$ gcc -g sharedmemory.c -o sharedmemory -lrt
dikshya@dikshya-VirtualBox:~/Desktop/Lab 3 OS$ ./sharedmemory
I am parent process =30825
I am child process 30826 my parent is process 30825
Child process exited, exit code: 0, Parent process reading from shared memory...
Message: greetings to parents!
dikshya@dikshya-VirtualBox:~/Desktop/Lab 3 OS$
```

#### ❖ Explanation:

The Parent process creates a shared memory then it spawns the child process. The Child process then runs and write a “greeting to parent!” message to shared memory. The Parent process waits child process to terminate, if the exit code == 0, then read the message from shared memory.

As the parent process will only read the shared memory we use PROT\_READ (We can also use PROT\_WRITE cause we opened it using both read and write permissions). As the child process will write the shared memory we use PROT\_WRITE (which implies PROT\_READ as specified in the man page).

❖ **Questions:** On Linux, all shared memory objects can be found in /dev/shm. Can you find the shared memory objects that?

**Answer:** First, as we have used shm\_unlink(name), this will delete the shared memory region given a file descriptor (name). So to answer this question, I commented that line than used the find command to find the shared memory object that we named SHARED\_PC (or I could also run ls -l /dev/shm and try to find it by myself by looking for it)

```
dikshya@dikshya-VirtualBox: ~/Desktop/Lab 3 OS
dikshya@dikshya-VirtualBox:~/Desktop/Lab 3 OS$ gcc -g sharedmemory.c -o sharedmemory -lrt
dikshya@dikshya-VirtualBox:~/Desktop/Lab 3 OS$ ./sharedmemory
I am parent process =30825
I am child process 30826 my parent is process 30825
Child process exited, exit code: 0, Parent process reading from shared memory...
Message: greetings to parents!
dikshya@dikshya-VirtualBox:~/Desktop/Lab 3 OS$ find /dev/shm -name SHARED_PC
/dev/shm/SHARED_PC
dikshya@dikshya-VirtualBox:~/Desktop/Lab 3 OS$ ls -l /dev/shm
total 4
-rw-rw-r-- 1 dikshya dikshya 4096 नवम्बर  16 18:44 SHARED_PC
dikshya@dikshya-VirtualBox:~/Desktop/Lab 3 OS$
```

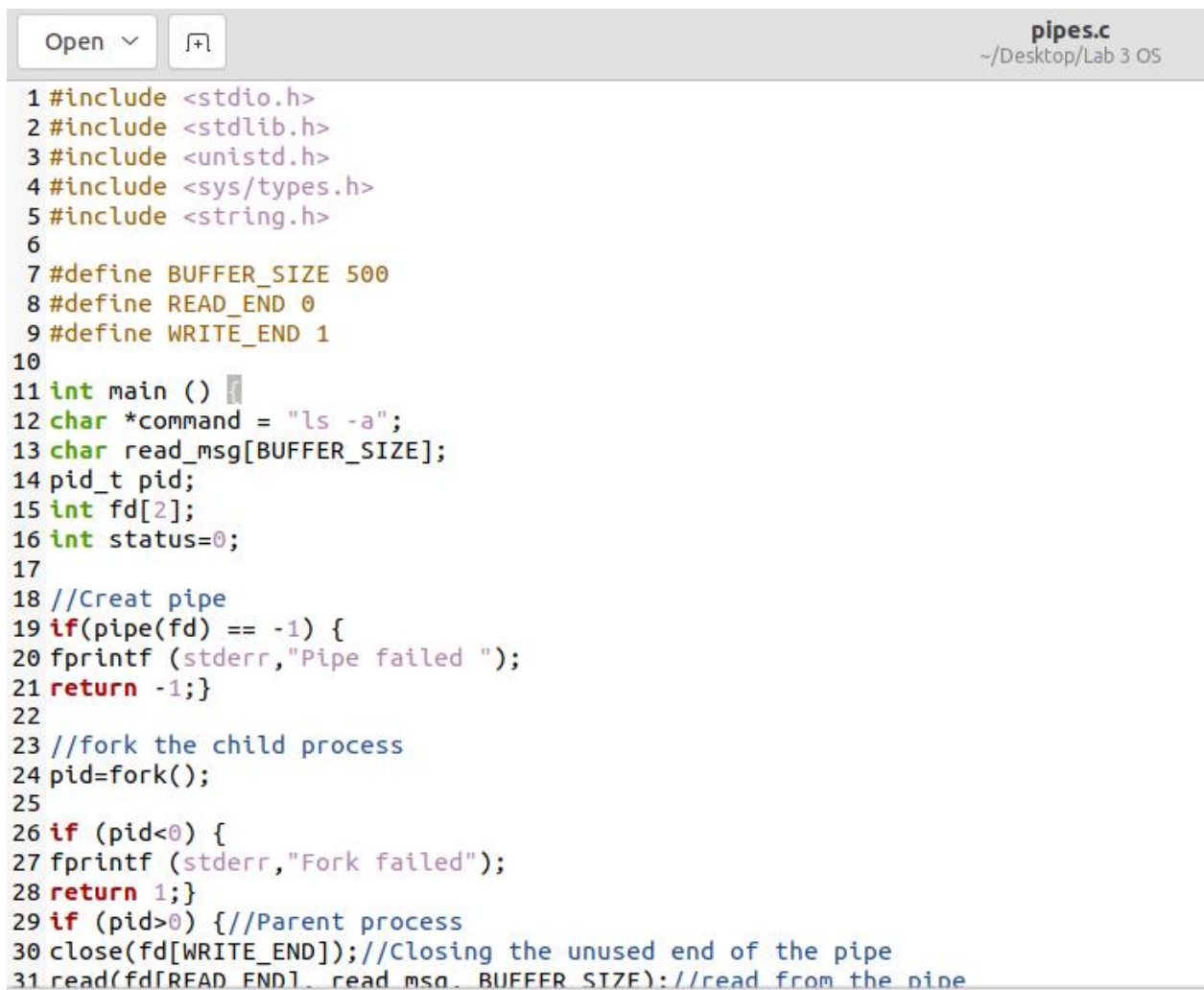
## B. Experiment 2: ordinary pipe communication

## I. Write an ordinary pipe between parent process and child process.

The program first creates a pipe, and then creates a child process of the current process through `fork()`. Then each process closes the file descriptors that are not needed for the read and write pipes. The child process executes the "ls -a" command under the current path, and writes the command execution output to the pipe by copying the pipe write descriptor `fd[WRITE_END]` to standard output; the parent process reads the pipe data and displays it through `fd[READ_END]`.

As the standard output becomes `fd[WRITE_END]`, the output of the command "ls -a" is written in the write end of the ordinary pipe. As we can see as a result, the parent process reads the names of files in the same directory.

❖ The code:



```
Open  [icon] pipes.c
~/Desktop/Lab 3 OS

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/types.h>
5 #include <string.h>
6
7 #define BUFFER_SIZE 500
8 #define READ_END 0
9 #define WRITE_END 1
10
11 int main () {
12     char *command = "ls -a";
13     char read_msg[BUFFER_SIZE];
14     pid_t pid;
15     int fd[2];
16     int status=0;
17
18     //Creat pipe
19     if(pipe(fd) == -1) {
20         fprintf (stderr,"Pipe failed ");
21         return -1;}
22
23     //fork the child process
24     pid=fork();
25
26     if (pid<0) {
27         fprintf (stderr,"Fork failed");
28         return 1;}
29     if (pid>0) { //Parent process
30         close(fd[WRITE_END]); //Closing the unused end of the pipe
31         read(fd[READ_END], read_msg, BUFFER_SIZE); //read from the pipe
```



```

30 close(fd[WRITE_END]); //Closing the unused end of the pipe
31 read(fd[READ_END], read_msg, BUFFER_SIZE); //read from the pipe
32 printf("Parent read:\n%s\n", read_msg);
33 close (fd[READ_END]); //Close the read end of the pipe
34 else{ //Child process
35 close (fd[READ_END]); //Closing the unused end of the pipe
36 if (fd[WRITE_END] != STDOUT_FILENO) {
37 if(dup2(fd[WRITE_END], STDOUT_FILENO) != STDOUT_FILENO)
38 return -1;
39 close(fd[WRITE_END]); //Close the write end of the pipe

```

❖ The result:

As we can see, the content of the directory matches what the parent process has read.

```

dikshya@dikshya-VirtualBox: ~/Desktop/Lab 3 OS
dikshya@dikshya-VirtualBox:~/Desktop/Lab 3 OS$ touch pipes.c
dikshya@dikshya-VirtualBox:~/Desktop/Lab 3 OS$ gcc -g pipes.c -o pipes
dikshya@dikshya-VirtualBox:~/Desktop/Lab 3 OS$ ./pipes
Parent read:
.
..
pipes
pipes.c
sharedmemory
sharedmemory.c

dikshya@dikshya-VirtualBox:~/Desktop/Lab 3 OS$

```

Name	Size	Modified
pipes	19.3 kB	19:05
pipes.c	1.0 kB	19:04
sharedmemory	19.8 kB	18:44
sharedmemory.c	1.6 kB	18:37

## II. Write a two-way ordinary pipe:

❖ The code:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/types.h>
5 #include <string.h>
6 #include <ctype.h>
7
8 #define BUFFER_SIZE 500
9 #define READ_END 0
10 #define WRITE_END 1
11
12 int main (){
13
14     char p_write_msg[BUFFER_SIZE] = "OPeratingSystem!";
15     char p_read_msg[BUFFER_SIZE];
16     int p_to_c[2];
17
18     char c_write_msg[BUFFER_SIZE] = "OPeratingSystem!";
19     char c_read_msg[BUFFER_SIZE];
20     int c_to_p[2];
21
22     pid_t pid;
23
24     //Create pipe 1
25     if(pipe(p_to_c)==-1){
26         fprintf (stderr,"Pipe failed ");
27         return 1;}
28
29     //Create pipe 2
30     if(pipe(c_to_p)==-1){
31         fprintf (stderr,"Pipe failed ");
```

```
31 fprintf (stderr, "Pipe failed ");
32 return 1;}
33
34 //Fork child process
35 pid=fork();
36
37 if(pid<0){
38 fprintf (stderr, "Fork failed ");
39 return 1;}
40
41 if(pid>0){//parent process
42 close(p_to_c[READ_END]); //Closing the read end of the pipe p_to_c
43 close(c_to_p[WRITE_END]); //Closing the write end of the pipe c_to_p
44
45 //Printing the message p_to_c
46 int i=0;
47 while(p_write_msg[i] !='\0'){
48 write(p_to_c[WRITE_END], &p_write_msg[i],1);i++;}
49
50 //Closing the write end of the pipe p_to_c
51 close(p_to_c[WRITE_END]);
52
53 //Printing the message c_to_p
54 int j=0;
55 while(read(c_to_p[READ_END], &p_read_msg[j],1) !='\0'){
56 printf("Parent process read: %c\n", p_read_msg[j]);j++;}
57 printf("\n");
58
59 close(c_to_p[READ_END]);}
60
61 else{//child process
```

```
59 close(c_to_p[READ_END]);}
60
61 else{//child process
62 close(p_to_c[WRITE_END]); //Closing the write end of the pipe p_to_c
63 close(c_to_p[READ_END]); //Closing the read end of the pipe c_to_p
64
65 //Printing the message p_to_c
66 int n=0;
67 while(read(p_to_c[READ_END], &c_read_msg[n],1) !='\0'){
68 printf("Child process read: %c\n", c_read_msg[n]);
69 c_write_msg[n]=toupper(c_read_msg[n]);
70 n++;}
71 printf("\n");
72 c_write_msg[n]='\0';
73
74 //Closing the write end of the pipe p_to_c
75 close(p_to_c[READ_END]);
76
77 //Printing the message c_to_p
78 int m=0;
79 while(c_write_msg[m] !='\0'){
80 write(c_to_p[WRITE_END], &c_write_msg[m],1);m++;}
81 printf("\n");
82
83 close(c_to_p[WRITE_END]);}
```



❖ The results:

```
dikshya@dikshya-VirtualBox: ~/Desktop/Lab 3 OS
dikshya@dikshya-VirtualBox:~/Desktop/Lab 3 OS$ gcc -g upper.c -o upper
dikshya@dikshya-VirtualBox:~/Desktop/Lab 3 OS$ ./upper
Child process read: O
Child process read: P
Child process read: e
Child process read: r
Child process read: a
Child process read: t
Child process read: i
Child process read: n
Child process read: g
Child process read: S
Child process read: y
Child process read: s
Child process read: t
Child process read: e
Child process read: m
Child process read: !

Parent process read: O
Parent process read: P
Parent process read: E
Parent process read: R
Parent process read: A
Parent process read: T
Parent process read: I
Parent process read: N
Parent process read: G
Parent process read: S
Parent process read: Y
Parent process read: S
Parent process read: T
Parent process read: E
Parent process read: M
Parent process read: !

dikshya@dikshya-VirtualBox:~/Desktop/Lab 3 OS$
```

❖ Questions:

1. Does the pipe allow bidirectional communication, or is communication unidirectional in an ordinary pipe?

Answer:

Ordinary pipes are unidirectional, allowing only one-way communication.

2. If two-way communication is allowed, is it half duplex (data can travel only one way at a time) or full duplex (data can travel in both directions at the same time)?

Answer:

Two-way communication is not allowed when working with ordinary pipes.

3. Must a relationship (such as parent–child) exist between the communicating processes in an ordinary pipe?

Answer:

Ordinary pipes require a parent-child relationship between the communicating processes. They cannot be accessed from outside the process that created it. A parent process creates a pipe and uses it to communicate with a child process that it created.

### C. Experiment 3: Named pipe communication

❖ client.c:

```
client.c
~/Desktop/Lab 3 OS

1 #include <stdio.h>
2 #include <string.h>
3 #include <fcntl.h>
4 #include <sys/stat.h>
5 #include <sys/types.h>
6 #include <unistd.h>
7
8 #define MAXSIZE 100
9
10 int main()
11 {
12     int fd;
13     char * myfifo = "/tmp/myfifo"; // FIFO file path
14     int i;
15     int read_bytes;
16     char str[MAXSIZE];
17
18     fd=open(myfifo, O_RDWR);
19
20     while (1)
21     {
22         //Take string from user
23         printf("String> ");
24         fgets(str, MAXSIZE, stdin);
25         i=strlen(str);
26         str[i-1]='\0';
27
28         //Check if end of string
29         if(strcmp(str,"end") !=0){
30
31             write(fd, str, strlen(str)); //Write in FIFO file
32
33             printf("Sent string: %s\n", str);
34             sleep(2); //For not receiving self message
35             read_bytes=read(fd,str,MAXSIZE); //Read from FIFO file
36             str[read_bytes]='\0';
37             printf("Received string: %s\n", str);
38         }else{
39             //Send end to server
40             write(fd,str,strlen(str));
41             printf("Sent string: %s\n",str);
42             sleep(2);
43             close(fd);
44             break;
45         }
46     }
47     return 0;
```

❖ server.c:

```
server.c
~/Desktop/Lab 3 OS

1 #include <stdio.h>
2 #include <string.h>
3 #include <fcntl.h>
4 #include <sys/stat.h>
5 #include <sys/types.h>
6 #include <unistd.h>
7
8 #define MAXSIZE 100
9
10 void reverse_string(char *);
11
12 int main()
13 {
14     int fd;
15     char * myfifo = "/tmp/myfifo"; // FIFO file path
16     int i;
17     char str[MAXSIZE];
18
19     mkfifo(myfifo, 0666);
20     fd=open(myfifo, O_RDWR);
21
22     while (1)
23     {
24         i=read(fd, str, MAXSIZE);
25         str[i-1]='\0';
26         printf("Sent string: %s\n", str);
27
28         //Check if end
29         if(strcmp(str,"end") ==0){
30             close(fd);
31             break;
```

```
31             break;
32         }
33
34         reverse_string(str);
35
36         //write in FIFO
37         printf("Sending string: %s\n", str);
38         write(fd, str, MAXSIZE);
39         sleep(2); //For not receiving self message
40     }
41     return 0;
42 }
43
44 void reverse_string(char *str){
45     int last, limit, first;
46     char temp;
47     last = strlen(str) - 1;
48     limit = last/2;
49     first = 0;
50
51     while (first < last){
52         temp = str[first];
53         str[first] = str[last];
54         str[last] = temp;
55         first++;
56         last--;
57     }
58     return;
59 }
```

❖ Results:

```
dikshya@dikshya-VirtualBox: ~/Desktop/Lab 3 OS
dikshya@dikshya-VirtualBox:~/Desktop/Lab 3 OS$ touch client.c
dikshya@dikshya-VirtualBox:~/Desktop/Lab 3 OS$ gcc -g client.c -o client
dikshya@dikshya-VirtualBox:~/Desktop/Lab 3 OS$ ./client
String> Hello
Sent string: Hello
Received string: Hello
String> Nepal
Sent string: Nepal
Received string: Nepal
String> end
Sent string: end
dikshya@dikshya-VirtualBox:~/Desktop/Lab 3 OS$
```

```
dikshya@dikshya-VirtualBox:~/Desktop/Lab 3 OS$ make
gcc -c client.c -o client.o
gcc -o client client.o
gcc -c server.c -o server.o
gcc -o server server.o
dikshya@dikshya-VirtualBox:~/Desktop/Lab 3 OS$ ./client
```

```
String> Hello
Sent string: Hello
Received string: olleH
String> NPU
Sent string: NPU
Received string: UPN
String> Hello World!
Sent string: Hello World!
Received string: !dlroW olleH
String> end
Sent string: end
```

```
dikshya@dikshya-VirtualBox: ~/Desktop/Lab 3 OS
dikshya@dikshya-VirtualBox:~/Desktop/Lab 3 OS$ make
gcc -c client.c -o client.o
gcc -o client client.o
gcc -c server.c -o server.o
gcc -o server server.o
dikshya@dikshya-VirtualBox:~/Desktop/Lab 3 OS$ ./server
```

```
Received string: Hello
Sending String: olleH
Received string: NPU
Sending String: UPN
Received string: Hello World!
Sending String: !dlroW olleH
Received string: end
```

The server creates a named pipe with permissions of read and write using library function `mkfifo()` with name “fifo” in /tmp directory, if not created. It then Opens the named pipe for read and write purposes and waits infinitely for a message from the client.



If the message received from the client is not “end”, it 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.

The client process opens the named pipe for read and writes purposes. It takes an input string from the user and checks if the user enters “end” or other than “end”. Either way, it sends a message to the server. 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 server and prints the reversed string. This is Repeated infinitely until the user enters the string “end”.

Note, we should take care of the EOF otherwise we get some wrong results. Also if we do not put process in sleep after writing in FIFO file, it will read its own message.

#### ❖ Question:

What are the advantages of using named pipe?

#### Answer:

A Named pipe that can be used for two-way communication and supports bi-directional communication. Named pipes provide a much more powerful communication tool. Communication can be bidirectional, and no parent–child relationship is required.

Also, Multiple users can send requests through the same named pipe and each request is removed from the pipe as it is received. In addition, the message handler procedure can loop indefinitely looking for input because it blocks (waits) until there is something to read. Finally, output through named pipes is more efficient than writing a complete response to an ordinary file, closing the file, and then informing the recipient that the results are available. The receiving process can read the result through a named pipe as soon as it is written.