National University of Computer and Emerging Sciences

Lab Manual 6

(Operating Systems)

Department of Computer
Science FAST-NU, Lahore

Name Pipes

1. It is an extension to the traditional pipe concept on Unix. A traditional pipe is "un- named" and lasts only as long as the process.
2. A named pipe, however, can last as long as the system is up, beyond the life of the pro- cess. It can be deleted if no longer used.
3. Usually a named pipe appears as a file, and generally processes attach to it for inter-pro- cess communication. A FIFO file is a special kind of file on the local storage which allows two or more processes to communicate with each other by reading/writing to/from this file.
4. A FIFO special file is entered into the file system by calling mkfifo() in C. Once we have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultane- ously before you can proceed to do any input or output operations on it.
5. Reading from or writing to a named pipe occurs just like traditional file reading and writ- ing; except that the data for named pipe is never written to or read from a file in hard disk but memory.

- The standard programming model is that after the pipe has been set up, two (or more) processes can send data using read() and write().
- Pipes opened with open() should be closed with close(fd).

Opening and Creating Named Pipe files

Open system call is used for opening a file.

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

1. pathname is a file name
2. The argument *flags* must include one of the following *access modes*

   O_RDONLY, **O_WRONLY**, or **O_RDWR**. These request opening the file in read-only, write-only, or read/write modes, respectively. Apart from above, flags can also have any of the following:

(A) **O_APPEND (file is opened in append mode)**

(B) **O_CREAT** (If *pathname* does not exist, create it as a regular file.)

(C) **O_EXCL** Ensure that this call creates the file: if this flag is specified in conjunction with **O_CREAT**, and *pathname* already exists, then **open**() fails.

Note: to use two flags at once use bitwise OR operator, i.e., O_WRONLY | O_CREAT

3. **Mode is only required when a new file is created and is used to set permissions on the new file**

```
S_IRWXU  00700 user (file owner) has read, write, and execute
         permission

S_IRUSR  00400 user has read permission

S_IWUSR  00200 user has write permission

S_IXUSR  00100 user has execute permission

S_IRWXG  00070 group has read, write, and execute permission

S_IRGRP  00040 group has read permission

S_IWGRP  00020 group has write permission

S_IXGRP  00010 group has execute permission

S_IRWXO  00007 others have read, write, and execute permission

S_IROTH  00004 others have read permission

S_IWOTH  00002 others have write permission

S_IXOTH  00001 others have execute permission
```

Example Program:

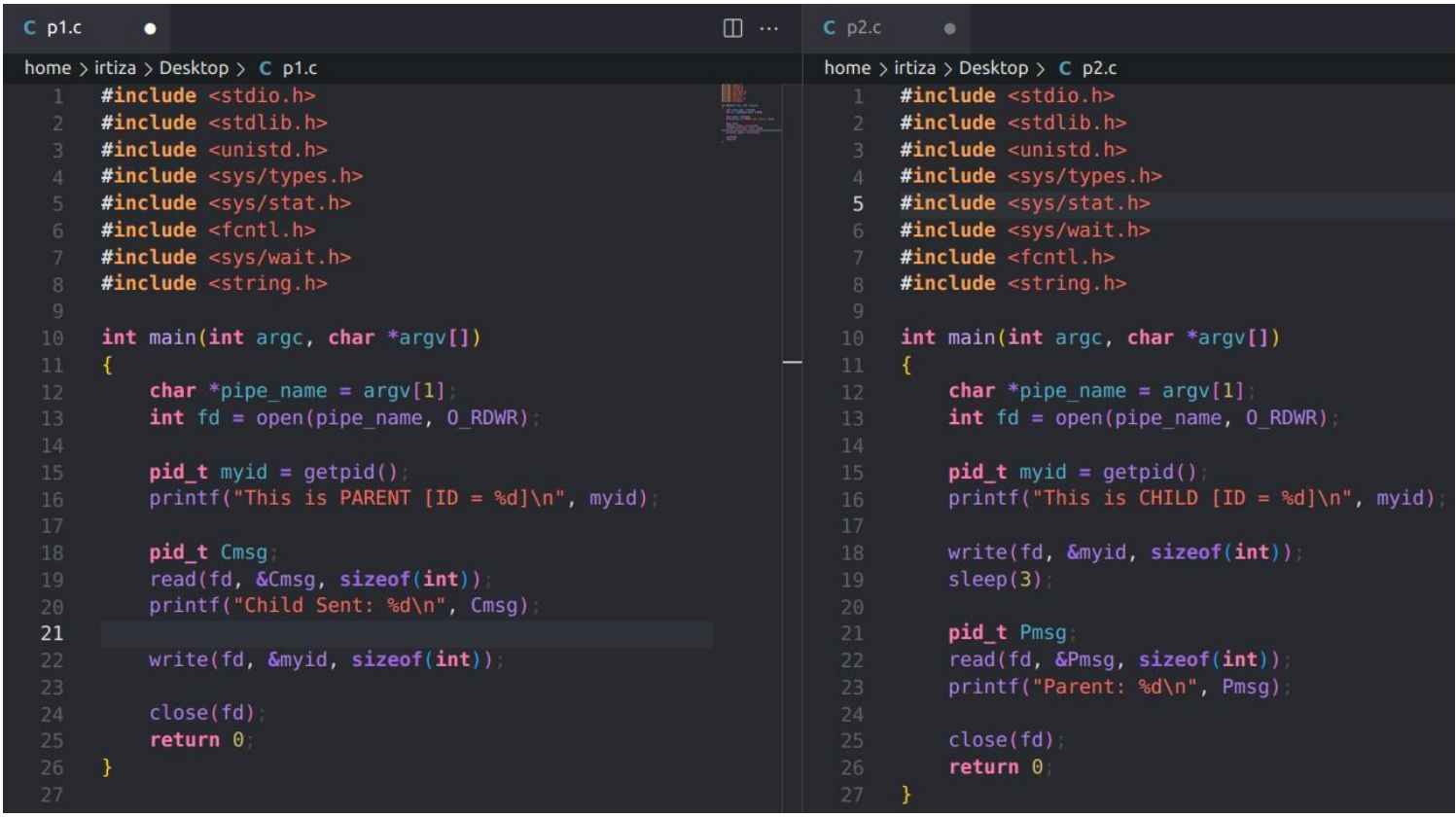
Creating a named pipe:

mkfifo(pipe_name, permissions)

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/wait.h>

#define PIPE_PERM (S_IRUSR | S_IWUSR)

int main(int argc, char *argv[])
{
    char *pipe_name = argv[1];
    int status = mkfifo(pipe_name, PIPE_PERM);
    printf("%d\n", status);
    return 0;
}
```

Sending and Receiving Data:

open(fd,
mode)
close(fd)
read(fd, where_to_read,
size) write(fd,
stuff_to_write, size)



p1.c

home > irtiza > Desktop > C p1.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char *pipe_name = argv[1];
    int fd = open(pipe_name, O_RDWR);

    pid_t myid = getpid();
    printf("This is PARENT [ID = %d]\n", myid);

    pid_t Cmsg;
    read(fd, &Cmsg, sizeof(int));
    printf("Child Sent: %d\n", Cmsg);

    write(fd, &myid, sizeof(int));

    close(fd);
    return 0;
}
```

p2.c

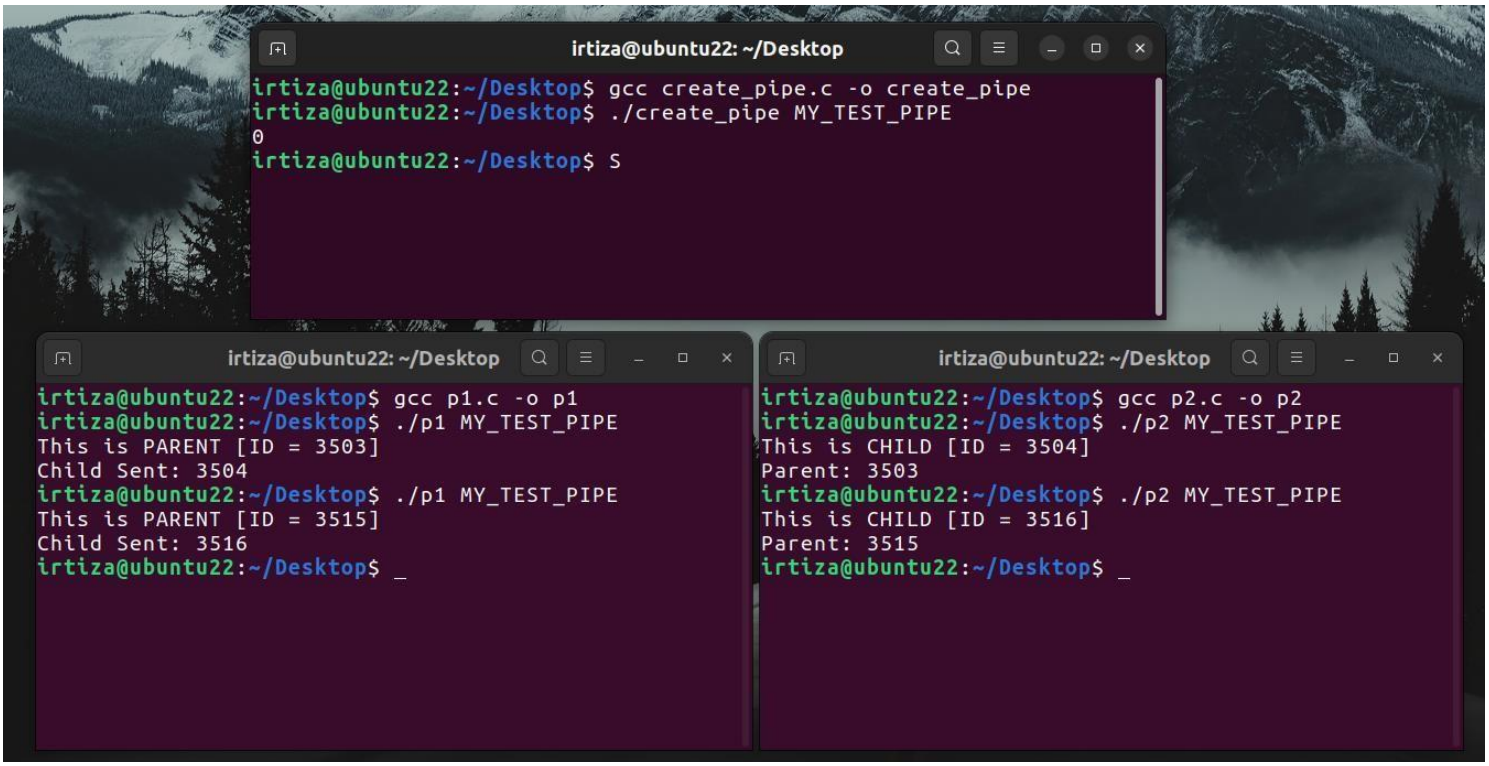
home > irtiza > Desktop > C p2.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char *pipe_name = argv[1];
    int fd = open(pipe_name, O_RDWR);

    pid_t myid = getpid();
    printf("This is CHILD [ID = %d]\n", myid);

    write(fd, &myid, sizeof(int));
    sleep(3);

    pid_t Pmsg;
    read(fd, &Pmsg, sizeof(int));
    printf("Parent: %d\n", Pmsg);

    close(fd);
    return 0;
}
```

- There are 2 processes p1 and p2.
- p2 prints its ID on screen and then sends it to the p1 and p1 prints it on the screen. Then p1 prints its ID on the screen and then sends it to p2, which prints it on the screen.
- I also created a 3rd program to create a named pipe.

Note that, p1 was run before p2 because “read” is a blocking system call and write is not. Which means until there is something to read in the pipe, p1is blocked and it keeps waiting for any data. Therefore, once the p2 writes data to the pipe, p1 consumes it and write some more data to the pipe. “Sleep” is used so p2 does not consume its own data from the pipe and wait for p1 to actually write its data.
You will also have to keep this in mind otherwise, you will get junk values or unwanted results.

Output:



Task 1 (2 marks)

Design a program (**handler.c**) which communicates using named pipes. **handler** sends a string message to a second process (**case_changer.c**), and the second process reverses the case of each character in the message and sends it back to the **handler**, that will print it on the screen.
For example, if the first process sends the message **Hi There**, the second process will return **hi tHERE**.

Task 2 (3 marks)

Design a program (**handler.c**) using named pipes in which one process send an integer array to another process (**calculator.c**) and **calculator** calculates its sum and send it back to the **handler**, which displays the result on screen.
This will require synchronizing the processes using **sleep** so the send and receive are timed seamlessly.
Help:

Use int size = sizeof(int) * num_of_elements_in_array;

Syntax:

write(fd, send_array,
size); read(fd,
recv_array, size);

Task 3 (5 marks)

Create 2 independent programs (**sender.c and worker.c**) that perform

communication using named pipes.

One program will be the **worker** program that will wait for **sender** to send some data via a named pipe. The data sent is as follows:

Operator operand1 operand2

The operands can be **+, -, *, /**. The **worker** will then apply the operator on the operands and return the result to **sender** via named pipe. The **sender** will then print the result on the screen

For example, if the following is sent:

+ 4 10

Then the **worker** will calculate 4+10 and return 14 to **sender** via the pipe which will then print it.