Lab 07: Named Pipes

Objective:

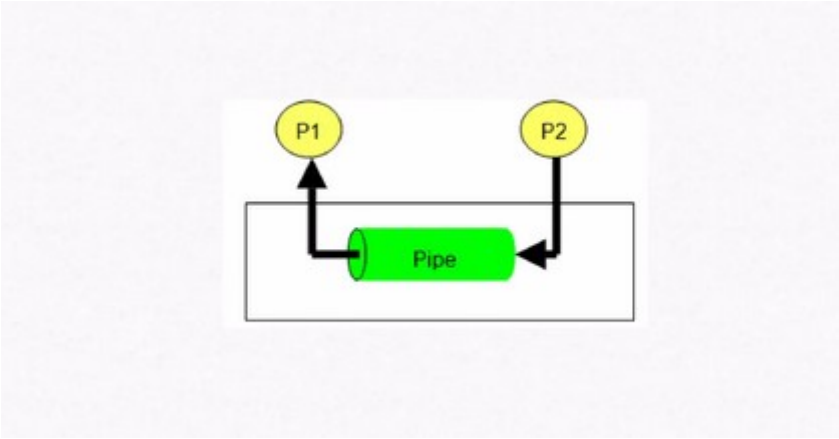
- To understand and implement named pipes for IPC.
- Learn the differences between unnamed and named pipes.
- Implement simple producer-consumer communication using named pipes.

Prerequisites:

- Basic knowledge of shell scripting, Linux file system, and file permissions.
- Familiarity with concepts of processes and IPC in Unix/Linux

Pipes

Pipelines are the oldest form of UNIX inter process communication. They are used to establish communication between processes that share a common ancestor. The process one-way pipe system call is used to create a pipeline. Pipe strictly follow the Inter-process communication.



Types of Pipes:

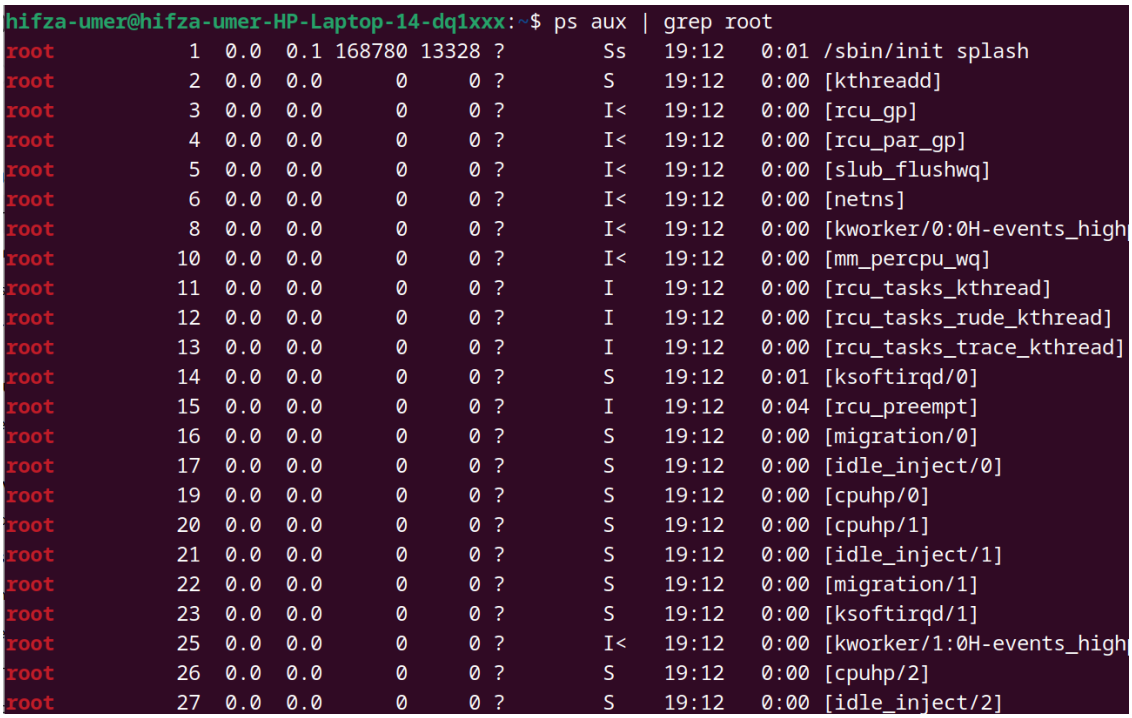
1: Ordinary Pipe

2: Named Pipe

1. Ordinary Pipes (Unnamed Pipes)

An ordinary pipe is a communication channel that is used **only between related processes** (typically parent and child processes). Data written into the pipe by one process can be read by another process. Ordinary pipes do not have a name in the filesystem and exist only as long as the processes are running.

- **Unidirectional**: Data flows in one direction (e.g., from parent to child).
- **Temporary**: The pipe exists only while the processes are running.
- **Used between related processes**: Typically between parent-child processes.



ps aux lists all running processes.

- | is the ordinary pipe, which passes the output of ps aux to the next command.
- grep root filters the output to show only processes related to the user "root".

Implementation of Unnamed Pipe (Parent-Child Communication):

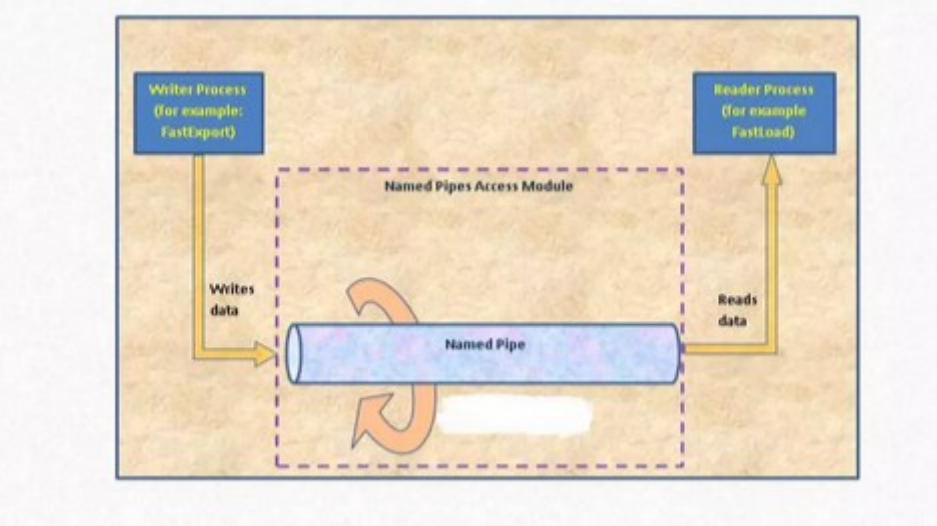
This code demonstrates how a parent process writes data to an unnamed pipe, and the child process reads from it.

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

int main() {
    int pipefd[2];  // Array to hold pipe ends: pipefd[0] for reading, pipefd[1] for writing
    pid_t pid;
    char writeMsg[] = "Hello from parent!";
    char readMsg[100];

    // Create the pipe
    if (pipe(pipefd) == -1) {
        perror("Pipe failed");
        return 1;
    }

    // Fork the process
    pid = fork();

    if (pid < 0) {
        perror("Fork failed");
        return 1;
    }

    if (pid > 0) {  // Parent process
        close(pipefd[0]);  // Close reading end in parent
        write(pipefd[1], writeMsg, sizeof(writeMsg));
        close(pipefd[1]);  // Close writing end after writing
    } else {  // Child process
        close(pipefd[1]);  // Close writing end in child
        read(pipefd[0], readMsg, sizeof(readMsg));
        printf("Child process received: %s\n", readMsg);
        close(pipefd[0]);  // Close reading end after reading
    }

    return 0;
}
```

```
hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx:~$ vim parent.c
hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx:~$ gcc -o parent_child parent.c
hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx:~$ ./parent_cthild
bash: ./parent_cthild: No such file or directory
hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx:~$ ./parent_child
Child process received: Hello from parent!
hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx:~$
```

Named Pipes (FIFOs)

A **named pipe**, also known as a **FIFO (First In, First Out)**, is a pipe that has a name in the file system. Unlike ordinary pipes, named pipes can be used for communication between **unrelated processes** (i.e., processes that don’t have a parent-child relationship). The named pipe exists as a file, and any process that has access to the file can read from or write to the pipe.



- **Bidirectional**: Can allow two-way communication.
- **Persistent**: Named pipes exist as files in the filesystem and persist until manually removed.
- **Used between unrelated processes**: Processes do not need to be parent and child.

Steps:

1: Create a named pipe using the `mkfifo` command

2: Writing to the named pipe: In **Terminal 1**, write data to the named pipe:

```
hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx: ~                    hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx: ~
hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx:~$ mkfifo my_pipe
hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx:~$ echo "Hello from writer" > my_pipe
hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx:~$
```

3: Reading from the named pipe: In **Terminal 2**, read the data from the named pipe

```
hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx: ~                    hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx: ~
hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx:~$ cat < my_pipe
Hello from writer
hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx:~$
```

The output should be: `Hello from writer`.

Check the type of the named pipe with `ls -l`.

```
Hello from writer
hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx:~$ ls -l my_pipe
prw-rw-r-- 1 hifza-umer hifza-umer 0 Sep 19 22:17 my_pipe
hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx:~$
```

Producer-Consumer Problem Using Named Pipes

One process will act as a producer (generating data), while another acts as a consumer (reading and processing the data).

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

int main() {
    int fd;
    char *fifo = "/tmp/my_fifo";  // Path to the FIFO
    char message[100];
    int count = 0;

    // Open the FIFO for writing
    fd = open(fifo, O_WRONLY);
    if (fd == -1) {
        perror("Error opening the FIFO for writing");
        exit(EXIT_FAILURE);
    }

    while (1) {
        // Create a message to send
        snprintf(message, sizeof(message), "Message %d from Producer", ++count);

        // Write the message to the FIFO
        write(fd, message, strlen(message) + 1);

        printf("Producer sent: %s\n", message);

        // Sleep for 2 seconds
        sleep(2);
    }

    // Close the FIFO
    close(fd);
    return 0;
}
```

Write the **Consumer** C program (`consumer.c`). This program will read the messages from the named pipe and print them to the console.

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

int main() {
    int fd;
    char *fifo = "/tmp/my_fifo";  // Path to the FIFO
    char message[100];
    int count = 0;

    // Open the FIFO for writing
    fd = open(fifo, O_WRONLY);
    if (fd == -1) {
        perror("Error opening the FIFO for writing");
        exit(EXIT_FAILURE);
    }

    while (1) {
        // Create a message to send
        snprintf(message, sizeof(message), "Message %d from Producer", ++count);

        // Write the message to the FIFO
        write(fd, message, strlen(message) + 1);

        printf("Producer sent: %s\n", message);

        // Sleep for 2 seconds
        sleep(2);
    }

    // Close the FIFO
    close(fd);
    return 0;
}
```

Compile the Producer and Consumer Programs
Run the Producer and Consumer Programs
Terminal 1 (Consumer):

```
hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx:~$ vim producer.c
hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx:~$ vim consumer.c
hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx:~$ gcc -o producer producer.c
hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx:~$ gcc -o producer consumer.c
hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx:~$ ./consumer
bash: ./consumer: No such file or directory
hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx:~$ gcc -o consumer consumer.c
hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx:~$ gcc -o producer producer.c
hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx:~$ ./consumer
Consumer received: Message 1 from Producer
Consumer received: Message 2 from Producer
Consumer received: Message 3 from Producer
Consumer received: Message 4 from Producer
Consumer received: Message 5 from Producer
Consumer received: Message 6 from Producer
Consumer received: Message 7 from Producer
Consumer received: Message 8 from Producer
Consumer received: Message 9 from Producer
Consumer received: Message 10 from Producer
Consumer received: Message 11 from Producer
Consumer received: Message 12 from Producer
Consumer received: Message 13 from Producer
```

Terminal 2 (Producer):

```
hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx:~$ ./producer
Producer sent: Message 1 from Producer
Producer sent: Message 2 from Producer
Producer sent: Message 3 from Producer
Producer sent: Message 4 from Producer
Producer sent: Message 5 from Producer
Producer sent: Message 6 from Producer
Producer sent: Message 7 from Producer
Producer sent: Message 8 from Producer
Producer sent: Message 9 from Producer
Producer sent: Message 10 from Producer
Producer sent: Message 11 from Producer
Producer sent: Message 12 from Producer
Producer sent: Message 13 from Producer
Producer sent: Message 14 from Producer
```

Clean Up

After you are done, you can remove the named pipe using the following command:

```
hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx:~$ rm /tmp/my_fifo
hifza-umer@hifza-umer-HP-Laptop-14-dq1xxx:~$
```

.

Key Differences Between Named and Unnamed Pipes:

1. **Scope**:
    - Unnamed pipes exist only during the lifetime of processes and work with parent-child relationships.
    - Named pipes exist as files in the file system, allowing communication between any processes.
2. **Access**:
    - Unnamed pipes cannot be accessed by processes other than the ones that created them.
    - Named pipes can be accessed by unrelated processes, provided they have the right permissions.
3. **Persistence**:
    - Unnamed pipes are destroyed when the processes exit.
    - Named pipes remain in the filesystem until they are deleted manually.

Blocking Behavior of Pipes:

Pipes (both ordinary and named) exhibit **blocking behavior** when there is no writer or reader available. This means that the processes using the pipe may be blocked (i.e., paused) until the other side of the pipe is ready to perform its operation (write or read). This behavior can occur in both **unnamed** and **named pipes**.

1. Blocking on Reading (No Writer)

If a process attempts to read from a pipe and there is no writer (i.e., no data available in the pipe), the process will block until:

- Another process writes data to the pipe, or
- The pipe is closed, signaling an **end-of-file (EOF)** condition.

2. Blocking on Writing (No Reader)

If a process tries to write to a pipe and there is no reader to read the data, the process will block until:

- Another process starts reading from the pipe, or
- The pipe is closed, which can cause the write operation to fail.

This blocking mechanism ensures synchronization between the producer and consumer processes.

1: Handling Blocking in Reading

When reading from a pipe, you can check if there is a writer present and handle the absence of data using **non-blocking mode**. This can be achieved by setting the pipe's file descriptor to **non-blocking mode** using the `fcntl()` system call in C.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
int main() {
    int fd;
    char *fifo = "/tmp/my_fifo";
    char buffer[100];
    int flags;
    // Open the FIFO for reading
    fd = open(fifo, O_RDONLY | O_NONBLOCK);
    if (fd == -1) {
        perror("Error opening FIFO");
        exit(EXIT_FAILURE);
    }
    // Try to read without blocking
    while (1) {
        int bytes_read = read(fd, buffer, sizeof(buffer));
        if (bytes_read == -1) {
            perror("No data available (Non-blocking read)");
        } else if (bytes_read > 0) {
            printf("Read: %s\n", buffer);
        } else {
            printf("End of file or no more data.\n");
        }
        sleep(1);  // Sleep to simulate wait time
    }
    close(fd);
    return 0;
}
```

- O_NONBLOCK: This flag is used to set the file descriptor to non-blocking mode, so the process will not block if there is no data in the pipe.
- If there is no data, the program prints an error message without waiting (blocking).

2: Handling Blocking in Writing

Similarly, when writing to a pipe, you can check if there is a reader present and handle the situation gracefully using **non-blocking mode**.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
int main() {
    int fd;
    char *fifo = "/tmp/my_fifo";
    char message[] = "Hello from producer";
    int flags;
    // Open the FIFO for writing
    fd = open(fifo, O_WRONLY | O_NONBLOCK);
    if (fd == -1) {
        perror("Error opening FIFO for writing");
        exit(EXIT_FAILURE);
    }
    // Try to write without blocking
    if (write(fd, message, strlen(message) + 1) == -1) {
        perror("Error writing to FIFO (No reader present)");
    } else {
        printf("Message sent: %s\n", message);
    }
    close(fd);
    return 0;
}
```

- The `O_NONBLOCK` flag is used to ensure the producer does not block when trying to write to the pipe if no reader is available.
- If no reader is present, the `write()` call will return an error, which you can handle using `perror()`.

Handling the EOF Condition

When a process finishes writing or reading from a pipe, it should properly close the pipe to avoid leaving the other process hanging. If the reading process encounters the end of the pipe (no more writers and no data), it will receive an EOF condition.

```
int bytes_read = read(fd, buffer, sizeof(buffer));
if (bytes_read == 0) {
    printf("End of file or writer closed the pipe\n");
    break;
}
```

Tasks:

1: Write a program that creates a named pipe, performs communication between sender and receiver, and then ensures proper cleanup by removing the named pipe after use. **(Estimated time 5 Mins)**

2: Implement a system where multiple producer processes send data to a single consumer process using named pipes. Ensure proper synchronization between the processes and track data flow between the processes.

3: Write a program where a parent process writes multiple messages to a pipe, and the child reads them one by one until the pipe is empty.

(Estimated time 10 Mins)

4: Implement a program where the child process tries to read from an unnamed pipe, but the parent delays writing to it. Experiment with the blocking behavior.

(Estimated time 10 Mins)

5: Write a C++ program that do the following things:

1. Parent Process:

The parent process creates 3 child processes using `fork()`.

2. Child Processes:

Each child generates a random number between 0 and 10. Each child takes guesses from the user (values between 0 and 10) and compares them to the random number. The guessing process continues until the user correctly guesses the random number. Each child counts the number of attempts (turns) the user takes to guess correctly.

3. Inter-Process Communication:

After the user guesses the correct number, each child process sends the number of attempts it took to win back to the parent process using a pipe communication method.

4. Parent Determines the Winner

The parent process receives the number of turns from each child and compares them.

The user with the fewest attempts is declared the overall winner.

Hint: “#include <cstdlib> “ library will be use for random values.

“int randomNum = rand() % 11;” this line generate random number between 0 to 10.

(**Estimated time 20 Mins)**