OSL Assignment 7(a)

Name: Kumar Deshmukh

Roll No: 33143

Batch: M9

Title:

Full duplex communication between two independent processes using named pipes (FIFO)

Problem Statement:

First process accepts sentences and writes on one pipe to be read by second process and second process counts number of characters, number of words and number of lines in accepted sentences, writes this output in a text file and writes the contents of the file on second pipe to be read by first process and displays on standard output.

Theory:

What is Inter-process Communication?

Inter-process communication is the mechanism provided by the operating system that allows processes to communicate with each other. This communication could involve a process letting another process know that some event has occurred or the transferring of data from one process to another.

Synchronization in Inter-process Communication

Synchronization is a necessary part of inter-process communication. It is either provided by the inter-process control mechanism or handled by the communicating processes. Some of the methods to provide synchronization are as follows:

- ***Semaphore***
  A semaphore is a variable that controls the access to a common resource by multiple processes. The two types of semaphores are binary semaphores and counting semaphores.

- ***Mutual Exclusion***

  Mutual exclusion requires that only one process thread can enter the critical section at a time. This is useful for synchronization and also prevents race conditions.
- ***Barrier***

  A barrier does not allow individual processes to proceed until all the processes reach it. Many parallel languages and collective routines impose barriers.
- ***Spinlock***

  This is a type of lock. The processes trying to acquire this lock wait in a loop while checking if the lock is available or not. This is known as busy waiting because the process is not doing any useful operation even though it is active.

Approaches to Inter-process Communication

The different approaches to implement inter-process communication are given as follows:

- ***Pipe***

  A pipe is a data channel that is unidirectional. Two pipes can be used to create a two-way data channel between two processes. This uses standard input and output methods. Pipes are used in all POSIX systems as well as Windows operating systems.
- ***Socket***

  The socket is the endpoint for sending or receiving data in a network. This is true for data sent between processes on the same computer or data sent between different computers on the same network. Most of the operating systems use sockets for inter-process communication.
- ***File***

  A file is a data record that may be stored on a disk or acquired on demand by a file server. Multiple processes can access a file as required. All operating systems use files for data storage.
- ***Signal***

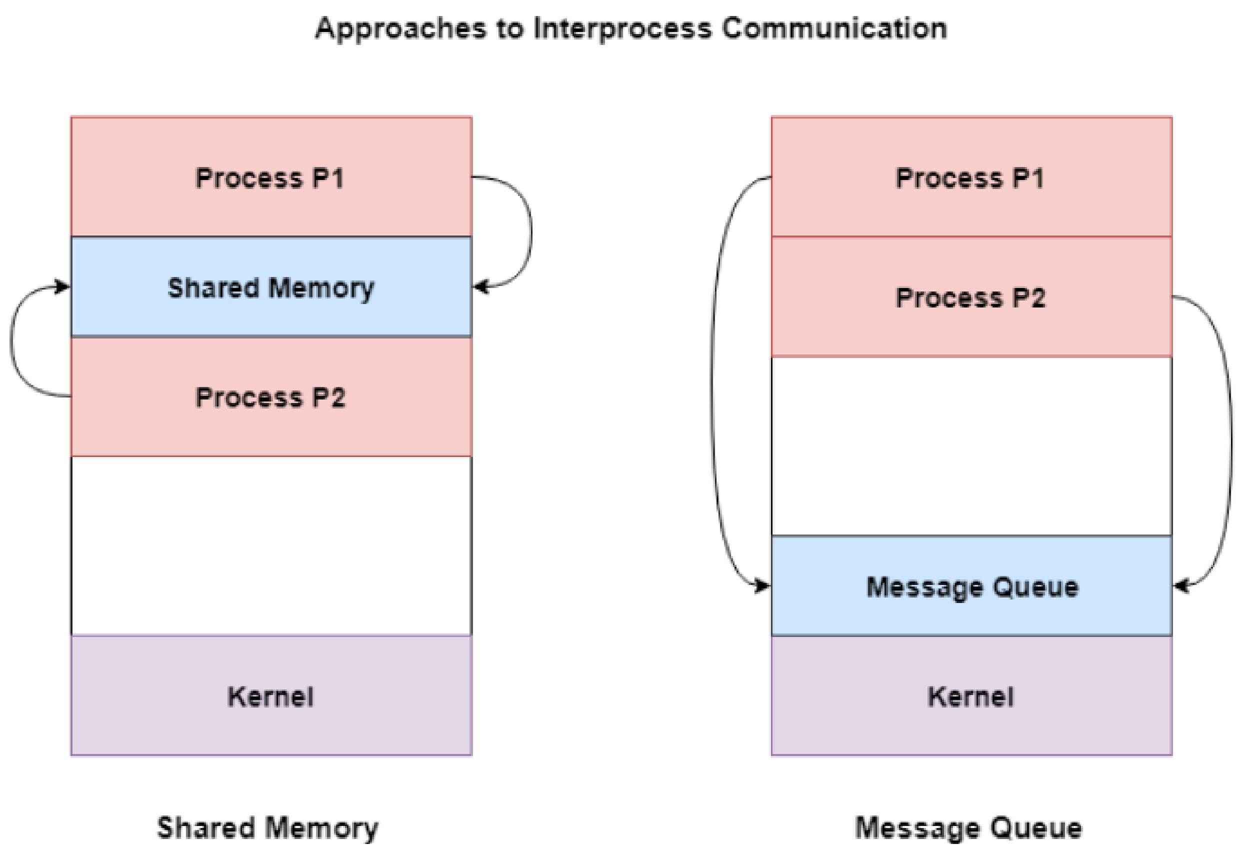
  Signals are useful in inter-process communication in a limited way. They are system messages that are sent from one process to another. Normally, signals are not used to transfer data but are used for remote commands between processes.
- ***Shared Memory***

  Shared memory is the memory that can be simultaneously accessed by multiple processes. This is done so that the processes can communicate with each other. All POSIX systems, as well as Windows operating systems use shared memory.

- ***Message Queue***

  Multiple processes can read and write data to the message queue without being connected to each other. Messages are stored in the queue until their recipient retrieves them. Message queues are quite useful for inter-process communication and are used by most operating systems.

A diagram that demonstrates message queue and shared memory methods of inter-process communication is as follows:



Linux mkfifo command

The mkfifo command basically lets you create FIFOs (a.k.a named pipes). Following is the syntax of the command:

```
mkfifo [OPTION]... NAME...
```

And here's how the tool's man page explains it:

```
Create named pipes (FIFOs) with the given NAMEs.
```

Following are some Q&A-styled examples that should give you a good idea on how mkfifo works.

Q1. What exactly are named pipes?

To understand this, you should first be aware of the concept of basic pipes. You'd have seen commands that contain a vertical bar (|) in them. This bar is called a pipe. What it does is, it creates a channel of communication between the two processes (when the complete command is executed).
For example:

```
ls | grep .txt
```

The command mentioned above consists of two programs: ls and grep. Both these programs are separated by a pipe (|). So what pipe does here is, it creates a channel of communication between these programs - when the aforementioned command is executed, the output of ls is fed as input to grep. So finally, the output that gets displayed on the terminal consists of only those entries that have '.txt' string in them.
So that was a quick refresher of normal pipes. Now comes the concept of named pipes. As the name itself suggests, these are pipes with names. You can create a named pipe using the mkfifo command. For example:

```
mkfifo pipe2
```

So 'pipe2' is now a named pipe. Now comes the question how named pipes are more useful? Well, consider the case where you have a process running in a terminal and producing output, and what you want is to channelize that output on to a different terminal. So here, a named pipe could of great help.
For example, suppose ls is the process running in the first terminal, and you want to see its output in a different terminal.. So here's what you can do:

```
ls > pipe2
```

and here's what you can do in the second terminal:

```
cat < pipe2
```

Q2. How to identify named pipes?

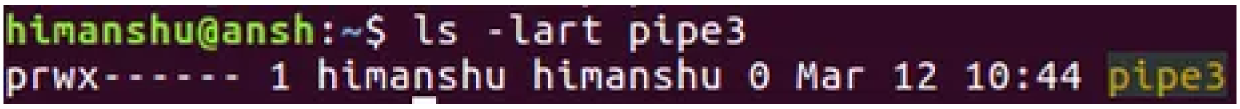
Named pipes can be accessed normally like files. So that means you can use the ls command to access them. If you see the access permissions for a named pipe, you'll see a 'p' in the beginning. This signifies the file in question is a named pipe. Here's an example:

Q3. How to set custom access permissions?

As you can see in the previous Q&A, the default access permissions for named pipes is 'rw', 'rw', and 'r' (for user, group, and others, respectively). However, if you want, you can set custom permissions as well, something which you can do using the -m option.
For example:

```
mkfifo pipe3 -m700
```

The following screenshot confirms custom permissions were set:



```
himanshu@ansh:~$ ls -lart pipe3
prwx------ 1 himanshu himanshu 0 Mar 12 10:44 pipe3
```

CODE:

1.cpp

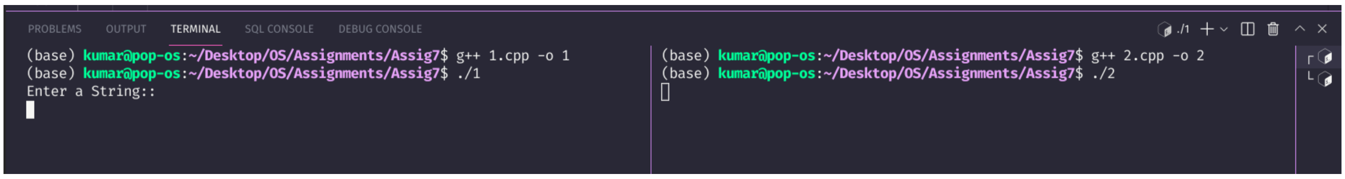
```
#include <bits/stdc++.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

using namespace std;

int main()
{
    int f1, f2;
    char str[100];
    char buffer[100], str1[100];

    mkfifo("fifo1", 0666);

    cout << "Enter a String::"<<endl;
    fgets(str1, 100, stdin);

    f1 = open("fifo1", O_RDWR);
    write(f1, str1, 100);

    f2 = open("fifo2", O_RDWR);
    read(f2, buffer, 100);

    cout << "\n" << buffer << endl;

    close(f1);
    close(f2);

    return 0;
}
```

2.cpp

```
#include <bits/stdc++.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

using namespace std;

int main()
```

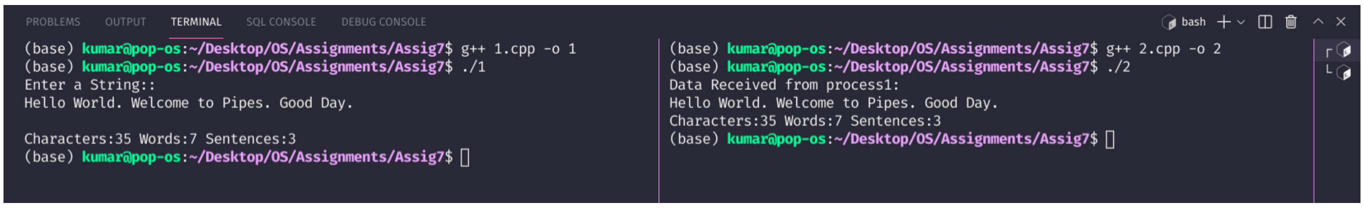
```
{
    int f1, f2;
    int i = 0, space = 0;

    char str[100];
    char buf[100], ch;
    int char_count, word_count, sent_count;
    char a[100], b[100], c[100];

    FILE *file;
    char_count = word_count = sent_count = 0;

    mkfifo("fifo2", 0666);
    f1 = open("fifo1", O_RDWR);
    read(f1, str, 100);

    printf("Data Received from process1:\n%s", str);
    while (str[i] != '\0')
    {
        if (str[i] != ' ' && str[i] != '\n')
            char_count++;

        if (str[i] == ' ')
            word_count++;

        if (str[i] == '.')
            sent_count++;
        i++;
    }
    if (char_count > 0)
    {
        char_count++;
        word_count++;
    }

    cout<<"Characters:"<<char_count<<" Words:"<<word_count<<"
Sentences:"<<sent_count<<endl;

    f2 = open("fifo2", O_RDWR);
    file = fopen("myfile.txt", "w");
    fprintf(file, "Characters:%d Words:%d Sentences:%d", char_count,
word_count, sent_count);
    fclose(file);
    file = fopen("myfile.txt", "r");
    fscanf(file, "%s %s %s", a, b, c);
    fclose(file);
    strcat(a, " ");
    strcat(a, b);
    strcat(a, " ");
    strcat(a, c);
    write(f2, a, 100);
    close(f1);
    close(f2);
    return 0;
}
```

OUTPUT:



```
(base) kumar@pop-os:~/Desktop/OS/Assignments/Assig7$ g++ 1.cpp -o 1
(base) kumar@pop-os:~/Desktop/OS/Assignments/Assig7$ ./1
Enter a String::
```

```
(base) kumar@pop-os:~/Desktop/OS/Assignments/Assig7$ g++ 2.cpp -o 2
(base) kumar@pop-os:~/Desktop/OS/Assignments/Assig7$ ./2
```



```
(base) kumar@pop-os:~/Desktop/OS/Assignments/Assig7$ g++ 1.cpp -o 1
(base) kumar@pop-os:~/Desktop/OS/Assignments/Assig7$ ./1
Enter a String::
Hello World. Welcome to Pipes. Good Day.

Characters:35 Words:7 Sentences:3
(base) kumar@pop-os:~/Desktop/OS/Assignments/Assig7$
```

```
(base) kumar@pop-os:~/Desktop/OS/Assignments/Assig7$ g++ 2.cpp -o 2
(base) kumar@pop-os:~/Desktop/OS/Assignments/Assig7$ ./2
Data Received from process1:
Hello World. Welcome to Pipes. Good Day.
Characters:35 Words:7 Sentences:3
(base) kumar@pop-os:~/Desktop/OS/Assignments/Assig7$
```

CONCLUSION:

We studied and implemented inter-process communication using named pipes via *mkfifo* command.