INDIAN INSTITUTE OF TECHNOLOGY
GOA

OPERATING SYSTEMS LAB (310)

Faculty: Dr. Sharad Sinha
Teaching Assistant: Prachi Kashikar

LAB 04

In this lab, you will be introduced to the concept of UNIX pipes. You will also learn the concept of System V IPC through shared memory and message queues. This lab exercise will be split into two parts.

PART A:

Pipes are a form of Inter-Process Communication (IPC) implemented on Unix and Linux variants. The kernel provides the synchronization between the processes accessing the same pipe. Data stored in the pipe is read on a First-In First-Out (FIFO) basis. The read/write operations are guaranteed to be atomic. The pipe is automatically removed by the OS when all the processes using it terminates. In fact, a pipe is a buffer managed by the kernel. It is a temporary storage of the data to be transferred between participating cooperative processes. The kernel takes care of the process synchronization.

A pair of unrelated processes can use a 'named pipe' to pass information between them. This allows a situation where two processes started in separate shells can communicate with each other through a 'named pipe' on the file system. A named pipe or FIFO, can be created using the mkfifo() function. It can be removed (like any other file on the file system) using the unlink() function. Once a named pipe file exists, programs can open it like they would other files and then use the file descriptor obtained to perform regular file IO operations (read, write, close...).

Some codes related to working with pipes are given along with this manual for your reference.

PART B:

1) You will write two simple programs pipe_reader.c and pipe_writer.c that use a named pipe to communicate. The pipe reader program will set up a named pipe using mkfifo() , open it read only, and read strings from it until it receives the string exit . The writer will open the named pipe file, read strings from the user and write them to the named pipe. When the user enters exit , the program will write the string to the pipe and then exit. Execution should look something like this (note that you must start the reader first):
   Reader:
   mypc:$ ./pipe

Reader
Creating named pipe: /tmp/mypipe
Waiting for input...Got it: 'Thursday'
Waiting for input...Got it: 'OS lab 4
Waiting for input...Got it: 'exit'
Exiting
Writer:
mypc:$ ./pipe
Writer
Opening named pipe: /tmp/mypipe
Enter Input: Thursday
Writing buffer to pipe...done
Enter Input: OS lab 4
Writing buffer to pipe...done
Enter Input: exit
Writing buffer to pipe...done
Exiting

Extend the above exercise to create a chat application between two processes.

2) Compile lab4_6.c with gcc, and run it a few times. Describe the output you get, and explain briefly how it is produced.
3) Remove the sleep statement from the child process, rerun lab4_6.c and explain the output produced.
4) Restore the sleep statement from the previous step, and remove it from the parent process. Again, rerun lab4_6.c , and explain the output produced.
5) Rather than rely upon sleep statements to synchronize the two processes, consider the use of spinlocks. In this approach, the parent will write to shared memory when the memory location contains the value -1, and the child will read when the memory location is not -1.
   - Initialize the shared memory location in main memory to -1 before the fork operation.
   - Replace the sleep statement for the child by a spinlock that checks that the shared memory contains a nonnegative value. At the end of the child's loop, the shared memory location should be reset to -1.
   - Remove the sleep statement from the parent at the end of the loop, and insert a spinlock at the beginning of the loop that checks that memory is -1. When this condition occurs, the parent may write the next nonnegative number to shared memory.
   - Then, compile lab4_6.c , run them a few times, and review the code to be sure you understand how the programs work.