Lab 6.1 : IPC Mechanism using Named Pipe and Shared Memory

November 7, 2016

Objective :

The purpose of this lab is to teach students to use named Unix pipes and introduce to System V IPC through shared memory and message queues.You should refer to *Lecture notes on IPC* and run all examples before proceeding with this lab

- **Named Pipe**

  1. **Background :**
     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 regualar file IO operations on the (read, write, close...).
  2. **Assignment :**
     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 recieves 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:
     ```
     iiita:$ ./pipe_reader
     Creating named pipe: /tmp/mypipe
     Waiting for input...Got it: 'Oh! God'
     Waiting for input...Got it: 'OS lab trouble'
     Waiting for input...Got it: 'exit'
     Exiting
     ```

writer:

```
iiita:$ ./pipe_writer
Opening named pipe: /tmp/mypipe
Enter Input: Oh! God
Writing buffer to pipe...done
Enter Input: OS lab trouble
Writing buffer to pipe...done
Enter Input: exit
Writing buffer to pipe...done
Exiting
```

Note : **pipe_reader** and **pipe_writer** need to be executed in separate shells at the same time. The reader stops at
emphWaiting for input... until it recieves data from the pipe (the read completes).

- **Shared Memory:**

  1. Compile *read-write-1.c* with gcc, and run it a few times. Describe the output you get, and explain briefly how it is produced.
  2. Remove the sleep statement from the child process, rerun *read-write-1.c*, and explain the output produced.
  3. Restore the sleep statement from the previous step, and remove it from the parent process. Again, rerun *read-write-1.c*, and explain the output produced.
  4. 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. (Why must this be done before the fork?)
     - 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.
  5. Then, compile *read-write-2.c*, run them a few times, and review the code to be sure you understand how the programs work.
  6. Explain whether *read-write-2.c* will work when there are multiple readers or when there are multiple writers. In each case, if the code would work, explain why. If the code would not work, give a timing sequence involving the several readers and/or writers showing what might go wrong.