Computer Operating System Experiment

Laboratory 3

Inter-Process Communication

Objective:

In computer science, Inter-process communication (IPC) is a mechanism which allows processes to communicate with each other and synchronize their actions. IPC provides a mechanism to allow the processes to manage shared data. The objective of this lab is to help you internalize a couple of important facts about IPC in Linux.

Including:

- Shared memory
- Pipes

Equipment:

VirtualBox with Ubuntu Linux

Methodology:

Program and answer all the questions in this lab sheet.

1 Inter-Process communication

A process is a program in execution, and each process has its own address space, which comprises the memory locations that the process is allowed to access. An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently, in reality, there are many situations when co-operative nature can be utilized for increasing computational speed, convenience and modularity. Inter process communication (IPC) is a mechanism which allows processes to communicate with each other and synchronize their actions.

Cooperating processes require an inter-process communication (IPC) mechanism that will allow them to exchange data and information. There are two fundamental models of inter-process communication: shared memory and message passing. In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes

can then exchange information by reading and writing data to the shared region. In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes such as pipes, message queue, and sockets.

2 Experiments

2.1 Experiment 1: shared memory communication

Inter-process communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space.

These are steps for shared memory communication:

- Create or open a shared memory object with shm_open(). A file descriptor will be returned if shm_open() creates a shared memory object successfully.
- Set the shared memory object size with ftruncate().
- Map the shared memory object into the current address space with mmap() and MAP_SHARED.
- Read/write the shared memory.
- Unmap the shared memory with munmap().
- Close the shared memory object with close().
- Delete the shared memory object with shm_unlink().
- Control operations on the shared memory segment (shmctl())

POSIX provides five entry points to create, map, synchronize, and undo shared memory segments:

- **`shm_open()`**: Creates a shared memory region or attaches to an existing, named region. This system call returns a file descriptor.
- **`shm_unlink()`**: Deletes a shared memory region given a file descriptor (returned from `shm_open()`). The region is not actually removed until all processes accessing the region exit, much like any file in UNIX. However, once `shm_unlink()` is called (typically by the originating process), no other processes can access the region.
- **`mmap()`**: Maps a shared memory region into the process's memory. This system call requires the file descriptor from `shm_open()` and returns a pointer to memory. (In some cases, you can also map a file descriptor to a plain file or another device into memory. A discussion of those options is beyond the scope of this introduction; consult the `mmap()` documentation for your operating system for specifics.)
- **`munmap()`**: The inverse of `mmap()`.
- **`msync()`**: Used to synchronize a shared memory segment with the file system — a technique useful when mapping a file into memory.

A program shown in text book (page 133) use the producer-consumer model in implementing shared memory. The producer establishes a shared memory object and writes to shared memory, and the consumer reads from shared memory.

Requirements:

You need to write codes that implements a parent and child process that communicates via a shared memory object.

1) Parent process creates a shared memory
2) Parent process spawn child process
3) Child process runs and write a "greeting to parent!" message to shared memory
4) Parent process waits child process to terminate, if the exit code == 0, then read the message from shared memory.

Shared Memory Objects and Linux Kernels
On Linux, all shared memory objects can be found in /dev/shm. You may list them with ls -l /dev/shm. You may also remove a shared memory object with rm /dev/shm/[name]. This is handy when you are debugging your program.

Questions:
1. On Linux, all shared memory objects can be found in /dev/shm. Can you find the shared memory objects that you created?

2.2 Experiment 2: ordinary pipe communication

Ordinary pipes allow two processes to communicate in standard producer– consumer fashion: the producer writes to one end of the pipe (the write-end) and the consumer reads from the other end (the read-end). As a result, ordinary pipes are unidirectional, allowing only one-way communication. If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction. We next illustrate constructing ordinary pipes on Linux systems. In the example, one process writes the message Greetings to the pipe, while the other process reads this message from the pipe. An ordinary pipe cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it creates via fork().

On Linux systems, ordinary pipes are constructed using the function

pipe(int fd[ ])

This function creates a pipe that is accessed through the int fd[] file descriptors: **fd[0]** is the read-end of the pipe, and **fd[1]** is the write-end.

In order to send structured data through a pipe, you have to serialize it to push it from the sending to receiving process as a stream of bytes. For instance, imagine you are sending

a struct type: you have to turn the struct into a sequence of bytes in order to send them over the pipe.

When you send two different instances of serialized data structures over a serial channel such as a Linux pipe, you have to be able to distinguish where one datum ends and the next one begins. What you use as the “marker” or “sentinel value” that distinguishes the two instances is up to you to define.

1) Write an ordinary pipe between parent process and child process.

Copy the pipes-test.c program from the text book (Page 143) to another file named pipes.c. Now, modify your new program so that the parent process read to the pipe content from child process when child process executed commands.

The program first creates a pipe, and then creates a child process of the current process through fork(). Then each process closes the file descriptors that are not needed for the read and write pipes. The child process executes the *"ls -a*" command under the current path, and writes the command execution output to the pipe by copying the pipe write descriptor fd[1] to standard output; the parent process reads the pipe data and displays it through fd[0] .

2) Write a two-way ordinary pipe

Copy your pipes-test.c program to another file named upper.c. Modify your program so that it defines two pipes: one for communication from parent to child and another for communication from child to parent. Make sure to close the correct ends of each pipe. Ultimately, your goal is to have two pipes, one in each direction, so that you have bi-directional communication between the two processes. It helps a lot to use names for the pipes file descriptors that indicate their direction. For instance: p-to-c and c-to-p tell us the processes that the pipes interconnect and the direction of the flow of information.

This new version of the program must behave as follows.

- The parent sends a message to the child (byte-by-byte or line) and when it is done, it enters a loop to read characters from the pipe coming from the child (also byte-by-byte), terminating the loop on the receipt of EOF.
- The child receives the message (also byte-by-byte or line), printing each character to standard out as it arrives. For each character received, the child converts it to uppercase (using toupper(3) function) and sends it back to the parent using your second pipe. As the parent reads the characters received from the child, it prints them to standard out.

Make sure to reason carefully about when the write ends of the two pipes should be closed, so that your processes can terminate their loops gracefully and reach their

termination state when appropriate. Also, make sure to use the wrappers defined previously (Fork, Pipe, Read, and Write).

Questions:

1 Does the pipe allow bidirectional communication, or is communication unidirectional in an ordinary pipe?

2 If two-way communication is allowed, is it half duplex (data can travel only one way at a time) or full duplex (data can travel in both directions at the same time)?

3 Must a relationship (such as parent–child) exist between the communicating processes in an ordinary pipe?

2.3 Experiment 3: Named pipe communication

We used one pipe for one-way communication and two pipes for bi-directional communication. We can use single named pipe that can be used for two-way communication (communication between the server and the client, plus the client and the server at the same time) as Named Pipe supports bi-directional communication. Named pipes provide a much more powerful communication tool. Communication can be bidirectional, and no parent–child relationship is required.

A named pipe, however, can last as long as the system is up, beyond the life of the process. It can be deleted if no longer used. Usually a named pipe appears as a file and generally processes attach to it for inter-process 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.

A FIFO special file is entered into the filesystem 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 simultaneously before you can proceed to do any input or output operations on it.

The following steps outline how to use a named pipe from Linux programs:

- Create a named pipe using the mkfifo() function. Only one of the processes that use the named pipe needs to do this.
- Access the named pipe using the appropriate I/O method such as open().
- Communicate through the pipe with another process using file I/O functions:
  - Write data to the named pipe.
  - Read data from the named pipe.
  - Close the named pipe.

Creating a FIFO file: In order to create a FIFO file, a function calls i.e. mkfifo is used.

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

mkfifo() makes a FIFO special file with name pathname. mode specifies the FIFO's permissions.
Pathname specifies the name of the new FIFO.
mode specifies which access modes the FIFO has when it is created. Refer to the documentation for stat() for a discussion of access mode flags. You can uses 0666 (read and write allowed for everyone)

Requriments:

You need to write two programmers which follow rules below.
Step 1: Create two processes, one is client and another one is server.

Step 2: Server process performs the following:

- Creates a named pipe (using library function mkfifo()) with name "fifo" in /tmp directory, if not created.
- Opens the named pipe for read and write purposes.
- Here, created FIFO with permissions of read and write for Owner.
- Waits infinitely for a message from the client.
- If the message received from the client is not "end", prints the message and reverses the string. The reversed string is sent back to the client. If the message is "end", closes the fifo and ends the process.

Step 3: Client process performs the following:

- Opens the named pipe for read and write purposes.
- Accepts string from the user.
- Checks, if the user enters "end" or other than "end". Either way, it sends a message to the server. However, if the string is "end", this closes the FIFO and also ends the process.
- If the message is sent as not "end", it waits for the message (reversed string) from the client and prints the reversed string.
- Repeats infinitely until the user enters the string "end"

Questions:

1 What are the advantages of using named pipe?