Exploration: Inter-Process Communication

Introduction

In this exploration, we will take a tour of the mechanisms provided by Linux for processes to communicate with each other, i.e., functionality for Inter-Process Communication or IPC. We have already studied signals in an earlier module which are used by one process to notify another process, and thus provide a form of (limited) IPC. In another module, we explored mutex locks in the Pthreads API which is a facility that multiple threads in one process can use to synchronize their actions. In this exploration, we will look at the breadth of IPC facilities, and then, in later explorations in this module, we will study some of these facilities in greater depth .

Categories of IPC

At a high level, we can categorize IPC facilities into **communication facilities** which help processes exchange data, and **synchronization facilities** which help processes (or threads) coordinate their actions.

Communication Facilities for IPC

We can further categorize communication facilities into two categories, the ones where processes communicate by transferring data and the ones where processes communicate via shared memory.

▶ C Using Data Transfers

These facilities follow a pattern where one process writes data that is read by another process. In some of these facilities, the data is exchanged as **byte streams**. The writer writes data as a byte stream and a read operation reads an arbitrary number of bytes from the byte stream. In other facilities, data is exchanged as a complete **message**. The writer writes the data as messages. Each read operation reads a complete message and cannot read parts of a message. Data transfer facilities include the following:

Pipes

A pipe is a unidirectional data channel for processes on one machine in which one process writes to the pipe and another process reads from the pipe. A pipe can only be used between processes that are forked by a common ancestor. A pipe can be used, e.g., for a parent process to write to the child process, or vice versa. Pipes exchange data as byte streams. We will study pipes in the next exploration.

FIFO

FIFO stands for First-in, First-out. These are similar to pipes in that one process writes to a FIFO and another process on the same machine reads from it. However, FIFO can be used for communication between two process even if these processes are not related to each other. Unlike pipes, FIFOs have name and they are also called **named pipes**. FIFOs exchange data as byte streams. We will study FIFOs in the next exploration.

Message Queues

Message queues are used to exchange data in the form of messages. Messages in the queue have a type field, a length field and the actual message bytes. Messages can be fetched in first-in, first-out order. But it is also possible to fetch messages in a non-FIFO order using the type field.

Sockets

Sockets are different from the other IPC facilities that we have discussed in that these can be used for IPC by processes running on the same machine as well processes that are on different machines. **Datagram** sockets provide message based communication, whereas **stream** sockets support byte streams. We will study sockets in depth in another module.

IPC Using Shared Memory

As we have discussed, when a process is forked it gets its own memory which is protected from access by other processes. By using IPC facilities for shared memory, special memory segments can be created which are shared among multiple processes. The processes can then exchange information by reading and writing to this shared memory region. A variation on shared memory is **memory mapped file** where a part or whole of a file is mapped to memory using the `mmap()` ps://man7.org/linux/man-pages/man2/mmap.2.html) system call (https://man7.org/linux/man-pages/man2/mmap.2.html). Once the file has been mapped into memory, the file contents are accessed as if they were in a buffer in memory rather than accessing them using `read()` or `write()` system calls.

Comparing Data Transfer & Shared Memory Facilities

- Communication is a lot **faster** with shared memory than with data transfer. Once the shared memory has been set up, processes do not need to make system calls in order to read or write the data. Data transfer facilities typically require transferring data from the memory of the writer process to kernel memory during write, and then a transfer from kernel memory to the memory of the reader process.
- Synchronization of read and write operations is a lot **easier** with data transfer. A call to read data blocks if no data is available, and a call to write blocks if the data channel is full. However, in case of shared memory, it is the responsibility of the processes to use appropriate synchronization facilities, such as semaphores, to synchronize access to the shared memory.

For example, if one process is updating a data structure in shared memory, other processes should be prevented from accessing this data structure.

- With data transfer, reading the data removes it from the communication facilities. This means that only one process can read the data. As opposed to this, multiple processes can read the data when shared memory is used.

Exercises

Communication facilities, such as pipes and FIFOs, allow communication between processes. Do you think it would be important to provide similar communication facilities for multiple threads in the same process? If yes, why? If no, why not?

Answer

Synchronization Facilities

Processes and threads use synchronization facilities to coordinate their action, especially, when modifying or accessing shared resources, such as memory or files. Let us look at different types of synchronization facilities.

Semaphores

A semaphore is an integer whose value is always 0 or more. Processes, or threads in a process, can use semaphores to coordinate their actions. There are two fundamental operations that a process can do on a semaphore - increments its value and decrement its value. If the value is ady 0, then decrement operation is blocked until another process increments the value of the semaphore.

To enforce mutual exclusion, processes can use a **binary semaphore** whose value is either 0 or 1. A process wanting to gain exclusive access to a shared resource will try to decrement the value of the binary semaphore. It will succeed if the value is 1; otherwise it will be blocked.

However, if processes want to access a shared resource with multiple resources, they can use **counting semaphores** where the value of the semaphore can go up to the number of shared resources. We will study semaphore in a later exploration in this module.

Mutexes and Condition Variables (for threads)

We have already studied mutexes and condition variables which can be used to synchronize access by multiple threads in the same process. However, these facilities cannot be used for synchronization by multiple processes.

File Locks

File locks are a specialized synchronization facility to coordinate operations on a file by multiple processes. File locks can be created for complete files or at the finer granularity of portions of a file.

Additional Resources

Here are some references to learn more about the topics we discussed in this exploration.

- Chapter 43 of *The Linux Programming Interface* contains an excellent overview of the different IPC facilities provided in Linux. The categorization of IPC facilities in this exploration is adapted from that overview.

  Kerrisk, M. (2010). *The Linux programming interface : a Linux and UNIX system programming handbook*. San Francisco: No Starch Press.