Inter-Process Communication in Linux

If you want to develop software for Linux, a thorough understanding of processes and Inter-Process Communication (IPC) concepts is a must. Let's delve deep into the fundamentals of a process, the importance of IPC mechanisms and explore various IPC techniques used in Linux.

One of the principal roles of an operating system is to act as a resource manager allocating resources for various processes. If a system supports concurrency, then communication between different processes and the sharing of critical resources among them, is one of the fundamental design goals.

Linux, as an operating system, executes many processes simultaneously and, hence, communication between these is a must to its performance. Linux provides a variety of techniques for allowing processes to communicate with each other. Such communications may be to notify about the occurrence of an event, pass data from one process to another or synchronise shared data among processes. However, the broad design goal of Inter-Process Communication (IPC) techniques is the high performance of the system.

Linux used signals for communication between processes in the earlier days but this is not satisfactory in many situations. Later, many flexible, advanced IPC techniques were made available. Initially, pipe and FIFO (First In First Out) were the two mechanisms used to communicate messages between related and unrelated processes correspondingly. Later, System V IPC and POSIX IPC were developed, which are suited to perform communication and synchronisation very efficiently.

If you want to develop software for Linux, a thorough understanding of processes and IPC concepts is a must. This article explains the fundamentals of a process, the importance of IPC mechanisms, and then explores various IPC techniques used in Linux.

The aim is to explore the most important IPC mechanisms, which are used in Linux to communicate between processes. We start with the overview of IPC and the pipe and FIFO mechanisms for communicating messages between related or unrelated processes. The System V IPCs (message queues, shared memory, semaphores) and POSIX IPCs, are explained. We end with the Socket program, which will be used to communicate between different systems.

The article is addressed to those who know the basics of UNIX and have some programming experience in Linux. Programmers, too, have a range of needs that call for different communication and synchronisation mechanisms. Even though the main focus is on IPC mechanisms, whenever necessary, associated topics like fork, execve, mandatory and advisory locking, concurrent and iterative servers, etc, are also explained.

The fundamentals of a process

To achieve some task, we just write a program and compile it. The compilation creates an executable program. When we execute the program, it is copied from the file system to the main memory and becomes a process. A process is the fundamental entity in a modern operating system like Linux. The main function of a kernel (the core part of an operating system) is to create, manage and terminate processes. The Linux kernel is a monolithic one, which means all the kernel subsystems are grouped together as a single entity. Process management is one of the kernel subsystems. Since an operating system is also a resource manager, when the process is being executed, the kernel must allocate system resources like the file descriptor, process space in main memory, processor, etc. It also coordinates and synchronises process activities.

When a process is created, the kernel puts the process in a queue. When its turn comes, the kernel selects the process to schedule it to run. While running, the kernel executes the instructions of the process sequentially, either in the user or a kernel mode, depending on the instruction. Less privileged instructions are executed in user mode but whenever high privileged instructions are to be executed (for example, system call), the processor changes mode from user to kernel, and the system call is executed in a kernel mode. It can access either user or kernel space, or both the spaces, based on the execution context. Less privileged instructions are executed in user mode with process context and can access only the corresponding process space. One process cannot access another process space unless a suitable IPC mechanism is used. System calls and signals are executed in a kernel mode with process context, but can access both the user and kernel space. Some of the system wide tasks, like rescheduling process priorities, are executed in a kernel mode and can access only kernel space since this space doesn't depend on any specific process. These tasks will not access a process space.

When a process is running and needs to wait for a certain event to occur, then it will be put in the sleep state. The kernel saves the context of the process and loads the new process image, which is called context switch. When the awaited event occurs, the kernel wakes up the process and schedules it to run. Each process is identified by its process id (pid) and allocates process space in the main memory. Unless the space is declared as shared, no other process can access it. To perform multiprogramming on a uniprocessor system, each running process is allocated at a fixed time slice (round robin) to run on a processor. Once the time slice is over, it will come back to the scheduler queue to run again in order to execute the remaining code.

In Linux, the processes are dependent. *init* (pid = 1) is the first process, which will spawn a tree of processes, and this can be seen by executing the *pstree* command. Once the process is completed, the exit status should be given to its parent process. If the parent is busy with some other activities, then the kernel de-allocates all the child resources and puts it into the zombie state. The child process is terminated after the parent collects its exit status. Suppose the parent exits before completion of a child process, then the child will become an orphan process and *init* will take care of it. In Linux, the fork *system call* is used to create a child process.

There are two types of processes—independent and co-operating. An independent process execution does not depend on other processes nor shares any data with others. But a cooperating process can influence or be influenced by another running processes in the system. Process cooperation is needed in different environments to achieve different goals. For example, the cooperating processes need to communicate between themselves to perform many inter-related operations.

Importance of IPC

To achieve associated operations in a complex programming environment, multiple cooperating processes are used frequently. These processes have to communicate with each other, to ensure certain events occur, to share resources, exchange information and synchronise between them to access shared data. To perform this, the system must provide some method, which is called IPC.

Let's take the example of a process that has to send data to another process. A typical scenario is to execute the *ls –l | wc –l* command. Here *ls –l* is executed by one process and the output is sent to another process through pipe. Then the second process executes *wc –l*, which will print the total number of lines in the long list of files. So, this explains the importance of pipe. Multiple processes have to share the frequently used common data that is kept in some part of the memory that can be accessed by the concerned processes. Linux uses shared memory to achieve this. Without shared memory, one process cannot access the address space of the other process. In shared memory, if a process modifies the data, the change will be immediately available to other processes. For example, many users in a system work in *vi* editor—not every user has a separate copy of the *vi*. Instead they share the text portion of the code since the data portion is maintained separately for each user to preserve their own data.

To share resources among certain processes, the resources should be synchronised properly to achieve mutual exclusion. For example, if we want to book a train ticket, the train data is in the central database so that many users can share it at the same time. But if everybody is allowed to update a particular train record at the same time, it will end up in chaos. To avoid this, when the first user enters into the record, the other users are blocked until the first user exits. To accomplish this kind of environment, Linux uses semaphores (or record locking). Suppose we want to access data from the server to client, or one system to another system, Linux uses the socket program. Linux provides different types of IPC mechanisms for programmers, which will be seen briefly in the next section.

Types of IPC

How do we communicate from one process to the others? We can use signals but the problem is that apart from the signal number, we can't pass on any more data. In general, signals are used to kill a process, handle errors, notify about events, etc. To achieve different types of IPC, signals are not suitable since they are expensive, have limited bandwidth and pass only short bits of information. Interested readers can get the fundamentals and working procedures of signals with a demo program from "Linux Signals for the Application Programmer" by Dr B. Thangaraju, *Linux Journal*, March 2003, or get it freely online at [http://www.linuxjournal.com/article.php?sid =6483].

Linux provides many flexible advanced mechanisms to exchange information between processes, for example, pipe. We can visualise pipe as an ordinary hollow pipe. At one end we can send data and receive it through the other end. There are two types of pipe, named and unnamed. Pipe is unidirectional and has zero buffering capacity. If we create a process within a process by using the fork system call, then the new one is a child process and the older one is the parent process. When we want to communicate between parent and child or vice versa we can use an unnamed pipe. Unnamed pipe means the pipe doesn't have any name on it. Instead, it has only two descriptors called pipe descriptors for sending and receiving data between processes. The pipe descriptors will persist in the system until explicitly closed by the process, or till the process exits.

If we want to communicate between two processes that are unrelated, for example, processes running in two different sessions—then the unnamed pipe can't be used since the pipe descriptors created by the process are not visible to other unrelated processes. So, we need some technique to communicate between unrelated processes. Linux has different file types, one of which is FIFO, which is a named pipe. We can create FIFO using a shell command or a system call. Once we create a FIFO, it is stored in the file system and any process (if it has access permission) can use it with all ordinary file operations such as open, read, write, close, etc.

Shared memory is one of the IPC mechanisms to store some data into a specified memory segment, and any process that has permission can access the data. By accessing we mean that any process can read, write or modify the data. If more than one process has to read the shared data at the same time, there is absolutely no problem—the kernel can allow it. But in some situations, if more than one process has to modify or write data simultaneously, then it creates a synchronisation issue. Shared memory is one of the fastest mechanisms compared to all other IPC techniques. Since the shared memory is set up by a process, the cooperating processes can access the data in the shared memory without kernel intervention. But the processes that are sharing the memory need some synchronisation tool to avoid unwanted race conditions. The shared resources include some portion of code called critical section, which should be accessed in a mutually exclusive manner.

To synchronise any critical section, one of the most common synchronisation techniques used is semaphore. Semaphore is a synchronisation tool and it maintains an integer value. When any process needs to enter into a critical section, it will first decrease the value, and after the critical code access, the process will increase the semaphore's integer value. So, if a process has to access critical data, it will first check whether the integer value is greater than zero or not. If so, the kernel will put the process to sleep until the other processes increment the value. Apart from semaphore, if we want to synchronise any shared files or records in a file, Linux has very flexible file locking mechanisms to use. There are two types of file locking—mandatory and advisory locking, and they have read and write lock variants too. Generally, read lock will allow many processes to read the file simultaneously, but the write lock will allow only one process to write into the file at any point of time.

There are two types of implementations to work with message queues, shared memories and semaphores—System V IPC and POSIX IPC. Almost all UNIX variants, including Linux, support System V IPC, which is derived from the Unix System V release 4, from At&T Bell Laboratories. The original Linux implementation was created by Krishna Balasubramanian and it has been continually enhanced by several developers. POSIX IPC needs some libraries to work around. POSIX IPC was added by the POSIX real-time standard (1003.1b –1993). LFY

By: Dr B. Thangaraju is working as a tech lead in Embedded and Product Engineering Solutions (E&PE), Wipro Technologies, Bangalore.

Acknowledgement: The author would like to thank S. Jaisankar, Indian Institute of Science, Bangalore for his critical reading of the manuscript.