System Programming - Linux
Week 9 - Inter-Process Communication (IPC)

Lecturer: Dr. Aggrey Obbo (PhD)

May 30, 2025

Contents

1 Introduction

Understanding how separate processes work together and exchange information is crucial, given that Linux provides an extensive array of tools to enable this communication, with each method having its own advantages and disadvantages. These lecture notes explore the basic principles of inter-process communication, analyze different approaches like pipes, shared memory, message queues, and sockets, and consider their real-world uses in creating strong and efficient multi-process applications.

Objectives

- Understand the fundamental concepts of Inter process Communication (IPC).
- Describe and implement various IPC mechanisms provided by the Linux operating system
- Apply semaphores to effectively control access to shared resources.
- implement synchronization in concurrent Linux processes, preventing race conditions and ensuring data integrity.
- Explain the principles of socket-based communication in Linux.
- Evaluate and compare different IPC mechanisms in Linux.

2 Inter Process Communication (IPC)

Inter-Process Communication (IPC) refers to mechanisms provided by operating systems to allow distinct processes to share data and coordinate their actions[1]. Processes need to communicate with each other in many situations. It helps processes synchronize their activities and share information to avoid conflicts while accessing shared resources.

In Linux, IPC promotes modularity, resource sharing, and effective multitasking. IPC supports the shared use of resources like memory and files among various processes, enhancing system efficiency and preventing redundancy. The ability for processes to synchronize their operations through IPC mechanisms is also vital for avoiding data corruption and ensuring the proper execution sequence in concurrent environments. Linux offers an extensive array of IPC mechanisms, each designed for specific communication requirements and presenting different trade-offs related to performance and complexity.

Why IPC?

IPC is mechanisms enable inter process communication. And processes need to communicate, among others, for the following reasons[2][3];

1. Synchronization: In a collaborative task involving several processes, synchronization of their actions is often essential to maintain the correct sequence of execution and avoid race conditions. Interprocess communication (IPC) methods such as semaphores and mutexes help achieve this synchronization, enabling processes to manage access to shared resources or indicate when particular tasks are finished.

2. Data Sharing: A key reason for IPC is to enable various processes to access and share information. This might include sharing data structures, transmitting computation results, or supplying input to other processes. For example, in a client-server architecture, the server analyzes data and disseminates the results to several client processes.

3. Convenience: In certain instances, it may be easier to design an application as several collaborating processes. This approach can result in a more coherent code organization and improved distinction between different aspects, even if achieving strict parallelism isn't the main objective.

4. Speedup (Parallelism): Complicated tasks can be divided into smaller sub-tasks that multiple processes can execute at the same time. Interprocess communication (IPC) is crucial for managing these parallel processes and integrating their outcomes to reach a quicker total execution time. Consider a video encoding application that employs several processes to encode various frames at once.

5. Modularity: Creating an application as a collection of collaborating processes enhances modularity. Each process can handle a particular task, which simplifies the system's design, implementation, debugging, and maintenance. Modifications in one module (process) are less probable to impact other sections of the system, as long as the IPC interfaces stay consistent.

6. Resource Sharing: Processes may need to utilize hardware or software resources that are governed by the operating system, including printers, network connections, or files. Interprocess communication (IPC) methods can facilitate the management of access to these shared resources, averting conflicts and promoting equitable usage. For instance, several processes might need to input data into the same log file in a regulated way.

IPC thus binds together would be independent processes, enabling them to work collectively to achieve more complex efficient computing tasks within the Linux operating system environment.

3 Relationships Between Processes

Independent vs. Cooperating processes

An independent process functions separately, meaning its execution does not impact or gets impacted by other processes in the system. It does not share any resources such as memory or data with other processes. The result of an independent process is predictable and relies solely on its input.

However, a cooperating process can be affected by or can affect other processes. These processes share resources or data, resulting in an unpredictable outcome in which the final result is influenced by the order of execution and interactions with other processes.

Parent vs Child Processes

When a pre-existing process, referred to as the parent process, spawns a new process, that new entity is termed the child process. This creates a hierarchical structure. The parent can generate multiple child processes, resulting in a tree-like configuration.

Usually, the child process inherits specific resources from its parent, such as environment variables and open files. Nevertheless, the child process also receives its own distinct process ID (PID) and memory space.

The parent and child processes can run simultaneously. The parent may wait for its child processes to finish before resuming its own execution, or it may carry on running independently. The operating system offers methods for the parent to oversee and manage its children, including the option to terminate them if needed. This parent-child relationship is essential for organizing and managing intricate applications and system functionalities.

4 Challenges in IPC

Inter-Process Communication (IPC) is crucial for allowing processes to collaborate by sharing information and synchronizing their actions, yet it comes with several notable challenges.

1. Data Consistency: One major issue is maintaining data consistency. When multiple processes concurrently access shared resources, keeping the data accurate and coherent becomes complicated. Race conditions, where the program's result hinges on the unpredictable order of execution among different code sections, can result in data corruption or erratic behavior. Effective synchronization tools, including locks, semaphores, and monitors, are vital for controlling access to shared resources and averting these inconsistencies, but implementing and managing them can be complex.

2. Synchronization: Another significant challenge is synchronization itself. Aligning the activities of several processes to achieve a unified goal necessitates careful oversight of their execution sequence. Processes may need to pause for certain events or conditions to be met by others. Establishing proper synchronization can be challenging, and mistakes can result in deadlocks, where two or more processes become indefinitely blocked, waiting for one another to release resources. Preventing deadlocks demands meticulous resource allocation strategies and the application of deadlock detection or prevention algorithms.

3. Complexity: The complexity of implementation also presents a critical issue. Various IPC methods, such as shared memory, message passing, pipes, and sockets, come with distinct programming approaches and varying degrees of complexity. Selecting the right method hinges on the specific needs of the cooperating processes, including the volume and frequency of data exchange, the necessity for synchronization, and the processes' locations (whether on the same machine or across a network). Properly implementing these methods, managing potential errors, and ensuring effective communication can be quite challenging for developers.

4. Performance overhead: Additionally, the performance overhead is a significant factor. IPC by nature involves overhead due to system calls, data duplication (in certain methods), context switching between processes, and the enforcement of synchronization mechanisms. Inefficient IPC can create a bottleneck, negatively affecting the overall system performance. Fine-tuning IPC methods and reducing overhead is essential for applications requiring high performance.

5. Security: Finally, security introduces another layer of challenges. When processes share resources or communicate, there is a risk of unauthorized access or disruption. Safeguarding the integrity and confidentiality of the data exchanged through IPC methods demands careful design and security measures, particularly when managing communication between processes with varying privilege levels or across network boundaries. This may include implementing access control measures, encryption, and secure communication protocols. Addressing these diverse challenges is vital for developing robust and dependable concurrent systems.

5 Categories of IPC mechanisms in Linux

In Linux, Inter-Process Communication (IPC) methods are classified based on their properties and how they enable data sharing and synchronization among processes. The following are the known categories of IPC mechanisms in Linux.

1. Data Transmission Mechanisms: These IPC techniques mainly concentrate on the transfer of data between processes.

- Pipes: Offer a one-way data stream between related processes (usually a parent process and its child). Anonymous pipes are temporary and last only while the associated processes are active, whereas named pipes (FIFOs) are enduring, file-like entities that permit communication between unrelated processes.

  A. Anonymous Pipes:

  An anonymous pipe serves as a temporary, one-way communication channel for two processes.

  Creation: Anonymous pipe are created using the pipe() system call. This function requires an array of two integers as its argument. If executed successfully, this array will contain two newly created file descriptors. If the array is called fd, then fd[0] serves as the reading end of the pipe, while fd[1] acts as the writing end.

  Characteristics:

  Unidirectional: Data only moves in one direction – from the writing end to the reading end.

  Temporary: Anonymous pipes last only while the processes that utilize them are active. Once those processes end, the pipe ceases to exist.

  Related Processes: They are generally used for communication between processes that have a relationship, such as a parent process and its child. The parent sets up the pipe before forking, and both processes receive the file descriptors.

  Example: A parent process might send data to fd[1], allowing the child process to read that data from fd[0]. This setup enables them to collaborate or exchange information with each other.

  B. Named Pipes (FIFOs):

  Named pipes, or FIFOs (First-In, First-Out), function like a permanent mailbox in a shared area, allowing different processes to exchange data.

  Creation: Unlike anonymous pipes, you must explicitly create a named pipe within the file system using the mkfifo command in the terminal or the mkfifo() system call in C. After assigning it a name, it manifests as a special file. For instance, running mkfifo mypipe generates a named pipe called mypipe.

  Access in File System: After creation, the mypipe file resides in the file system. Any process with the right permissions can open it for reading or writing, similar to how a regular file operates.

  Characteristics of named pipes:

  Bidirectional (in theory): While data still flows in a single direction from the writer to the reader for each pipe opening, multiple processes can access the same named pipe, enabling more intricate communication patterns that may seem bidirectional. Persistent: Named pipes remain present in the file system even after the processes employing them have ended, until they are explicitly removed. Unrelated Processes: A significant advantage is that they facilitate communication between unrelated processes that are running concurrently on the same system. Example: One process might continuously send sensor data to mypipe, while another completely independent process collects and logs that information. They do not have to be in a parent-child relationship.

- Message Queues: Facilitate the exchange of messages between processes, which are generally structured data. The kernel oversees these queues, allowing processes to send and receive messages asynchronously. Various types of message queues are available, including System V and POSIX message queues.

  Message queues function as organized mailboxes that enable processes to exchange data chunks known as messages asynchronously[4].

  Creation: On POSIX systems, a message queue can be established by utilizing functions such as mq_open(), where you define a name and access options. This results in the creation of a kernel object recognized by that name.

  Structure: Each message within a queue usually includes a priority level and the actual data content. The queue keeps track of the order of messages, typically based on priority or the time of arrival (FIFO).

  Sending and Receiving: Messages are transmitted to a queue through mq_send(). It specifies the queue descriptor, message content, and its priority. And to extract messages, processes use mq_receive(), which retrieves the message with the highest priority (or the oldest one) from the queue.

  Limitations: Traditional System V message queues often had fixed limits on maximum message sizes and queue capacities, and messages would remain in the kernel even when not used by any processes.

  POSIX Message Queues: POSIX message queues provide benefits such as more adaptable message characteristics and, by default, messages are not stored persistently across system reboots unless specifically set up to be. This results in improved control and resource management compared to the older System V model. They also include more advanced methods for notifying processes when messages are received.

- Shared Memory: Allows multiple processes to access a common memory area. This provides a very effective way to share large volumes of data since it eliminates the need for data copying between processes. Synchronization methods are often required to manage simultaneous access to shared memory. Sockets: Though primarily utilized for network communication, Unix domain sockets (also referred to as local sockets) create a byte-stream or datagram communication pathway between processes on the same machine. They tend to be more adaptable than pipes for unrelated processes.

  System V Shared Memory is a mechanism that allows processes gain access to a shared memory segment, enabling them to read from and write to the same memory area directly.

  Creation: A process utilizes shmget() to establish a shared memory segment, indicating a key (for identification), size, and permissions. This results in a shared memory identifier being returned.

  Access: Other processes can then use the identical key with shmget() to retrieve the identifier and subsequently attach the segment to their address space through shmat(), obtaining a pointer to the shared memory.

2. Synchronization Methods: These IPC techniques are primarily used to synchronize the actions of multiple processes and manage access to shared resources.

   - Semaphores: Integer variables that regulate access to shared resources among several processes. They support two atomic actions: wait (decrementing) and signal (incrementing), which can help implement mutual exclusion and condition synchronization. Both System V and POSIX semaphores are available. Let's look into how Linux provides various types of semaphores for process synchronization, which is essential for managing access to shared resources like the aforementioned "whiteboard."

     System V Semaphores: These are created through semget() and exist as kernel objects identified by a key. The operations involve semop() for increasing (releasing) and decreasing (acquiring) semaphore values. They can handle sets of semaphores, making them useful for intricate resource management. Benefit: Persistency in the kernel. Drawback: Less adaptable, with a more complex API.

     POSIX Named Semaphores: These are established using sem_open() with a specified name, appearing as objects in the file system. The operations are sem_post() (to increment) and sem_wait() (to decrement). Benefit: Greater flexibility, simpler API, and they can be utilized between unrelated processes. Drawback: Generally not persistent in the kernel by default.

     POSIX Unnamed Semaphores: These are found in shared memory and are initialized using sem_init(). They are employed for synchronization between threads within a process or among related processes that share memory. The operations are also sem_post() and sem_wait(). Benefit: Lightweight and efficient for use with shared memory. Drawback: Restricted to processes that share the same memory space.

     Use Case: Consider several processes attempting to write to a common log file (a shared resource). Semaphores can ensure that only one process writes at a time, thus avoiding data corruption. A semaphore set to 1 (acting as a binary semaphore or mutex) can be obtained before writing and released afterward, ensuring exclusive access.

   - Mutexes (Mutual Exclusion Locks): Pthreads (POSIX Threads) offer robust synchronization tools for handling concurrent execution within a process.

     Mutexes (Mutual Exclusion): You can think of a mutex as a protective lock for a critical section of code or a shared resource. The function pthread_mutex_init() is used to initialize a mutex. Before a thread accesses the protected resource, it will call pthread_mutex_lock(). If the mutex is available, the thread takes the lock. If another thread currently holds it, the requesting thread will block until the lock is released. The function pthread_mutex_unlock() releases the lock, making it available for another waiting thread to acquire. Lastly, pthread_mutex_destroy() cleans up the mutex when it is no longer required. Mutexes guarantee that only one thread can access the shared resource at any time, thereby preventing race conditions.

     Condition Variables (Signaling): Condition variables work alongside mutexes to enable threads to wait for certain conditions to be met. The function pthread_cond_init() is used to initialize a condition variable. A thread that needs to wait for a specific condition first locks the associated mutex and then calls pthread_cond_wait(). This operation atomically releases the mutex and puts the thread to sleep. When another thread alters the condition, it acquires the same mutex and invokes pthread_cond_signal() (which wakes up one waiting thread) or pthread_cond_broadcast() (which wakes up all waiting threads). The awakened thread then reacquires the mutex and checks if the condition is true. The function pthread_cond_destroy() cleans up the condition variable. These mechanisms allow for efficient waiting without resorting to busy-waiting

3. Signaling Methods: These techniques entail sending alerts between processes.

   Signals: Asynchronous alerts dispatched to a process to signal the occurrence of a certain event (such as a termination request or an illegal instruction). Signals represent a limited form of IPC since they convey minimal information. It is essential to recognize that some IPC methods can fulfill both data transmission and synchronization roles. For example, shared memory allows for efficient data transfer but typically necessitates additional synchronization methods (like semaphores or mutexes) to maintain data integrity. The selection of an IPC method relies on factors such as the volume of data to be transferred, the relationship between the processes, the necessity for synchronization, and performance criteria.

   Linux signals serve as a means of asynchronous notification, enabling the kernel or other processes to inform a process about particular events. The kill() system call is the main method used to send a signal to a process, which is identified by its process ID (PID) and a specific signal number. Standard signals include SIGHUP (hangup) through to SIGKILL (forced termination).

   When a process receives a signal, it can either execute a default action (usually termination) or run a user-defined signal handler. These handlers are specific functions that the process registers to manage certain signals, allowing it to respond in custom ways to events such as user interruptions (SIGINT), the termination of child processes (SIGCHLD), or the expiration of timers (SIGALRM).

   Signals do have some limitations; they can only convey a small amount of information—typically just the signal number. They can also be unreliable in certain situations, particularly with real-time signals or during periods of high system load, where signals may be lost or delivered in a non-sequential manner. Additionally, it is not practical to transfer complex data using signals. Despite these drawbacks, signals continue to be a key and lightweight method for basic asynchronous communication and event notification in Linux, especially for managing process lifecycles and simple event handling.

4. Sockets (Unix Domain) Unix Domain Sockets (UDS), often referred to as local sockets, serve as an effective means of inter-process communication on the same Unix-like operating systems, such as Linux. Unlike network sockets that rely on IP addresses and ports, UDS transmit data using file paths in the file system.

   You can think of them as enhanced named pipes. They enable both reliable byte-stream communication (SOCK_STREAM, akin to TCP) and unreliable datagram communication (SOCK_DGRAM, similar to UDP). This versatility allows developers to select the communication method that best fits their requirements.

   To form a connection, processes refer to the same socket file. One process takes the role of the server by binding to a designated file path, while other processes function as clients that connect to that path. Once the connection is made, data can flow both ways.

   UDS typically exhibit greater efficiency for local IPC compared to network sockets, as they bypass the network stack's overhead. They also provide a more organized communication framework than basic pipes or FIFOs, making them ideal for intricate interactions among local processes.

6 Memory-mapped files (mmap())

Memory mapping, or mmap, in Linux is a powerful system call that creates a direct mapping between a file (or a special anonymous memory region) and a process's virtual address space. Instead of using traditional read/write system calls that involve copying data between the kernel buffer and user space, mmap allows a process to access the file's content directly as if it were in memory.

When a process maps a file, a range of its virtual memory is associated with a portion of the file on disk. Subsequent reads and writes to this memory region are, in essence, direct operations on the file data. The kernel handles the underlying details of loading pages from the file into physical memory as needed (demand paging). Similarly, modifications made to the mapped memory region can be written back to the file, either explicitly using msync or implicitly when the mapping is unmapped.

There are two primary types of mappings: shared mappings and private mappings. With a shared mapping (MAP_SHARED), changes made by one process to the mapped region are visible to all other processes mapping the same file and are eventually written back to the underlying file. This is a highly efficient way for multiple processes to share data.

In contrast, a private mapping (MAP_PRIVATE) creates a copy-on-write mapping. When a process modifies a page in a private mapping for the first time, the kernel creates a private copy of that page for the process. Subsequent modifications are made to this private copy and are not visible to other processes or written back to the original file.

This is useful when a process needs to work with a file's content without affecting other processes or the original file.

Memory mapping offers several advantages. It can significantly improve I/O performance, especially for large files, by eliminating the need for explicit buffer copies. It simplifies file manipulation as processes can use regular memory access instructions. Furthermore, it provides an efficient mechanism for inter-process communication through shared mappings. However, developers need to be mindful of potential issues like cache coherency (in shared mappings) and the need for explicit synchronization when multiple processes are involved.

7 Choosing the Right IPC Mechanism

1. Selecting the appropriate Inter-Process Communication (IPC) method in Linux requires a careful consideration of several important factors based on the specific requirements of the processes involved.

2. Data Size and Structure: For smaller, straightforward messages, using signals or anonymous pipes may be adequate. However, for larger or more complex data, it is usually necessary to utilize message queues, shared memory, or named pipes.

3. Type of Communication: Is the communication one-directional (pipe), two-directional (sockets), or multi-directional (message queues)? Do the processes need to coordinate their actions (semaphores, mutexes, condition variables) or simply share data asynchronously (signals, message queues)?

4. Process Relationships: Are the processes connected (parent-child) or independent? Anonymous pipes are generally suited for related processes, while named pipes, message queues, shared memory, and sockets are more appropriate for unrelated processes.

5. Performance Considerations: When dealing with large data transfers, shared memory provides the quickest communication since it avoids duplication of data. In contrast, utilizing message queues and pipes requires kernel intervention and data copying, which can affect performance. Signals incur minimal overhead for basic notifications.

6. Synchronization Requirements: If there is a need to safeguard shared resources, mutexes, semaphores, or condition variables become essential, often implemented alongside shared memory.

7. Longevity: Is there a need for the communication channels to outlast the processes involved? Named pipes and message queues can persist, while anonymous pipes are ephemeral. System V IPC mechanisms often maintain persistence within the kernel, which may not be favorable in certain scenarios.

8. Implementation Complexity: Some methods, like pipes and signals, are relatively straightforward to implement, whereas others, such as shared memory requiring explicit synchronization, demand more meticulous programming. The decision frequently involves weighing the balance between user-friendliness and the advanced capabilities offered.

8 Conclusion

Linux Inter-Process Communication (IPC) comprises various methods that enable processes to share data and coordinate their actions. The main categories are:

IPC challenges include maintaining data integrity, overseeing process synchronization to prevent deadlocks and race conditions, complexities in implementation, performance costs, and security. Effective IPC is vital for developing intricate, concurrent applications in Linux.

List of References

1. Silberschatz, A., & Galvin, P. B. (2013). Operatings System Concepts.

2. Stevens, W. R., & Rago, S. A. (2013). Advanced programming in the UNIX environment. Addison-Wesley.

3. Love, R. (2013). Linux system programming: talking directly to the kernel and C library. " O'Reilly Media, Inc.".

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

Some example codes on IPC

Two separate processes ls and grep communicating via the pipe.
Lists files and directories and filter only lines containing ".txt".
ls -l | grep ".txt"

Process interacting with the file system, which can then be read by another
echo "Communication between processes" > tutorialsarc.txt
cat < tutorialsarc.txt
The first command writes to tutorialsarc.txt. The second command reads from the file tutorialsarc.txt.

Processes Communication via pipes - Named Pipes using mkfifo:
Named pipes, or FIFOs (First-In, First-Out), are more explicit. You create a file that acts as a conduit for data.
mkfifo my_fifo
Open two terminals
In the first terminal: echo "Data to FIFO" > my_fifo
In the second terminal: cat < my_fifo
The cat command in the second terminal blocks until the echo command in the first terminal sends data.