Chapter 6

Inter-Process Communication

Operating Systems • Process Management

Dr. M.Goudjil _ UDBKM _ 2025-2026

Learning Objectives

- Understand inter-process communication (IPC) concepts
- Learn 6 main IPC mechanisms used in operating systems
- Analyze advantages and disadvantages of each method
- Apply IPC methods to real-world scenarios
- Compare efficiency and use cases for each technique

What is IPC?

Inter-Process Communication (IPC) refers to the methods used by processes to exchange data and coordinate (cooperation – competition) their actions within an operating system.

Why IPC is Important:

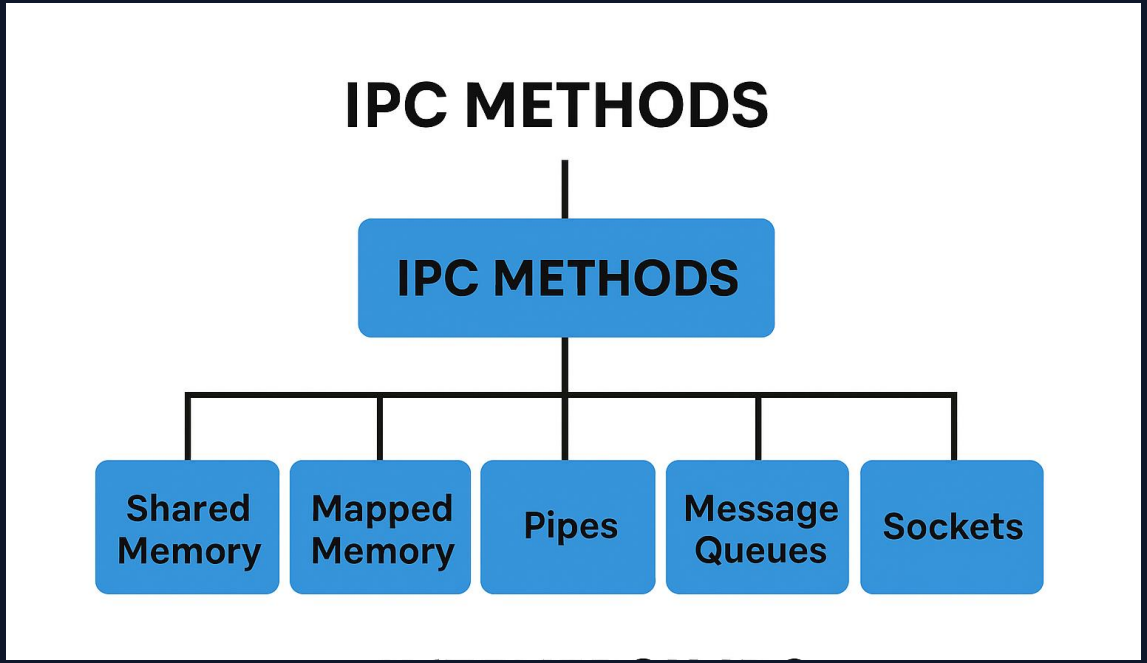
- Enable processes to work together on complex tasks
- Allow data sharing between independent processes
- Synchronize process execution

Criteria Differentiating IPC Types

- IPC methods differ based on several criteria:
  - • Process Relationship Restrictions: Some IPC methods are limited to related processes (e.g., parent-child).
  - • Read/Write Limitations: Some processes may be restricted to only reading or writing.
  - • Number of Communicating Processes: Some IPC methods support only two processes, others support many.
  - • Synchronization Behavior: Some IPC methods (e.g., pipes) provide built-in synchronization.

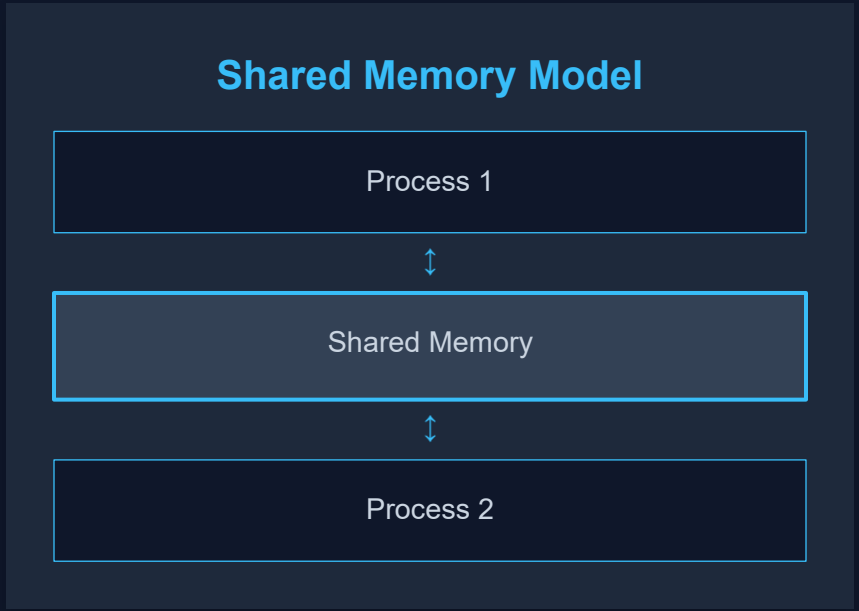
Six IPC Methods in Operating Systems

1. Shared Memory
2. Memory Mapping
3. Pipes
4. Named Pipes (FIFOs)
5. Sockets



1. Shared Memory Communication

Shared memory allows two or more processes to access the same memory area as if they had all called malloc and received pointers to the same space. Any modification made by one process is immediately visible to the others. Because the shared area resides directly in memory, shared memory is the fastest form of interprocess communication, requiring no system calls or kernel involvement once established.



Key Characteristics:

- Fastest IPC method
- No system calls during data transfer
- Direct memory access
- Requires synchronization

Shared Memory - Implementation Steps

To use a shared memory segment, a process must first allocate it. Each process requiring access must then attach the segment to its address space. Once usage is complete, the process detaches the segment. Ultimately, one process is responsible for releasing (remove) the segment to ensure proper resource management.

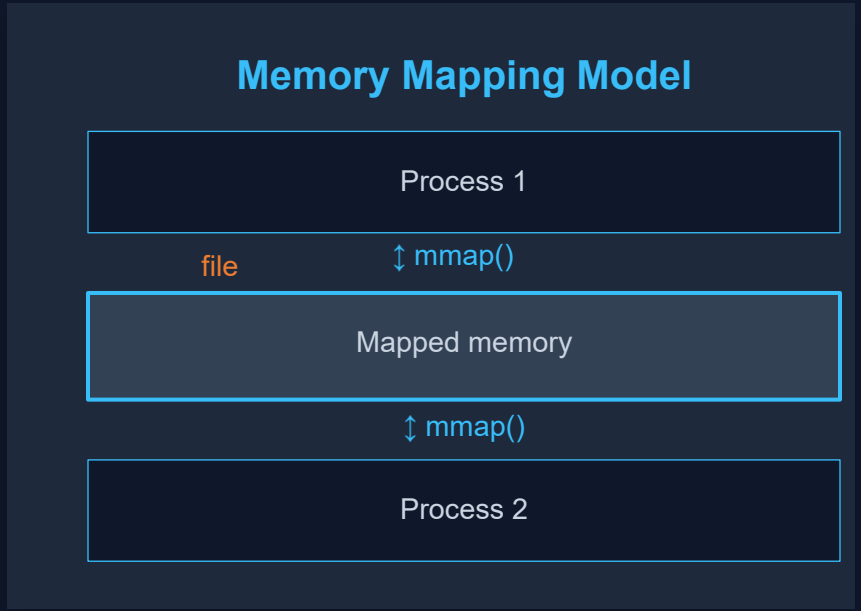
Shared memory segments enable rapid bidirectional communication among multiple processes, allowing each participant to both read and write. However, to prevent concurrent data access conflicts, programs must establish and adhere to a well-defined synchronization protocol.

Shared Memory - Implementation Steps

- **Create:** Use shmget() to allocate a shared memory segment
- **Attach:** Use shmat() to make the segment accessible to the process
- **Use:** Read and write data directly to shared memory
- **Detach:** Use shmdt() when done using the segment
- **Control:** Use shmctl() to manage and remove segments

2. Memory Mapping Communication

Mapped memory enables multiple processes to communicate through a shared file by mapping it into their address space. In Linux, the file is divided into page-sized fragments and loaded into virtual memory pages, making its contents accessible via standard memory operations. This technique allows processes to read file data as if it were regular memory, resulting in fast and efficient access without repeated system calls.



2. Memory Mapping Communication

Definition: Maps a file to process memory for fast access and IPC.

Key Characteristics:

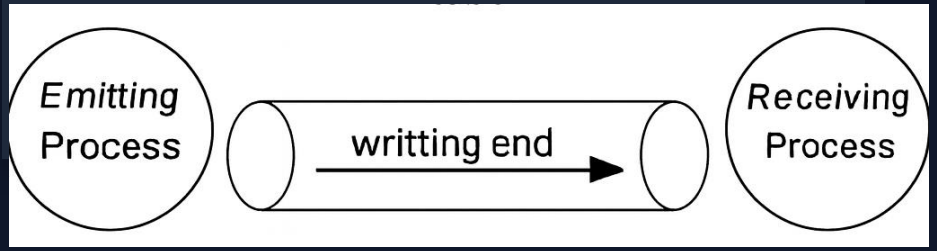
- File-based communication
- Changes visible to all processes
- Faster than traditional file I/O
- Can be persistent or non-persistent

Memory-mapped files allow distinct processes to communicate by mapping regions of memory to a shared file. When the MAP_SHARED flag is used, any write operation to the mapped region is immediately reflected on disk and becomes visible to other processes. As with shared memory, programmers must implement synchronization protocols—such as semaphores—to prevent concurrent access conflicts and ensure data consistency.

3. Pipes (Tubes)

Definition: A pipe is a unidirectional communication mechanism where data written by a sender process at one end is read sequentially by a receiver process at the other. Pipes are typically used between threads of the same process or between parent and child processes. In Linux, they are also employed by the shell to connect commands (e.g., ls | less). Pipes have limited capacity, and if the sender writes faster than the receiver consumes, the sender is blocked until space becomes available. Similarly, if the receiver attempts to read when no data is present, it is blocked until new data arrives. This built-in synchronization makes pipes a reliable method for coordinating process communication.

Pipe Model

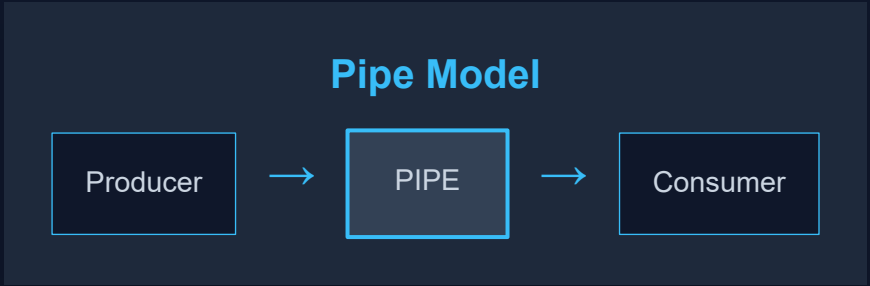


3. Pipes (Tubes)

Definition: One-directional communication channel between processes.

Key Characteristics:

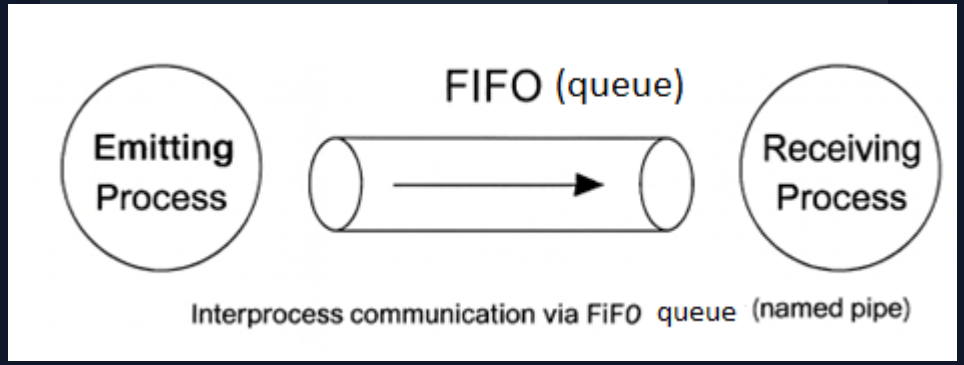
- Unidirectional data flow
- FIFO (First-In-First-Out) order
- Limited capacity
- Synchronization built-in
- Parent-child process link



4. Named Pipes (FIFOs)

Definition: A FIFO queue, also known as a named pipe, is a communication channel that operates on a first-in, first-out (FIFO) basis. Unlike anonymous pipes, FIFO queue are identified by a name in the file system and allow unrelated processes to exchange data. Data written by the sender is read in the same order by the receiver. FIFO queue are created using the mkfifo function, and accessed like regular files using standard I/O operations (e.g., open, write, read, close). Synchronization is inherent, as the sender blocks when the buffer is full and the receiver blocks when no data is available.

FIFO/Named Pipe Model

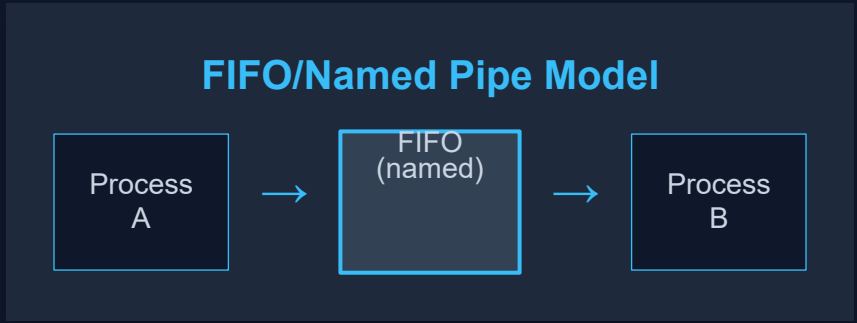


4. Named Pipes (FIFOs)

Definition: Named pipes with a file system name, enabling unrelated process communication.

Key Characteristics:

- Has a name in filesystem
- FIFO principle (First-In-First-Out)
- Connect unrelated processes
- Persists in filesystem
- Unidirectional communication



5. Sockets

Definition: A socket is a bidirectional communication interface that enables data exchange between processes, either on the same machine or across different machines over a network. Sockets are foundational to many Internet-based applications, including Telnet, rlogin, FTP, talk, and the World Wide Web. They provide a flexible and scalable mechanism for both local and remote interprocess communication..

Sockets provide a bidirectional communication interface between processes, whether on the same machine or across a network. Socket creation involves specifying three parameters: communication style, namespace, and protocol.

***The communication style determines how data packets are handled—either reliably with connection-oriented sockets (e.g., TCP) or unreliably with datagram-style sockets (e.g., UDP).**

***The namespace defines how socket addresses are assigned, such as using file paths or IP-port pairs.**

***The protocol governs data transmission, with common examples including TCP/IP, AppleTalk, and UNIX local communication protocols.**

5. Sockets

Definition: Bidirectional communication endpoints for local or network communication.

Key Characteristics:

- Bidirectional communication
- Local or remote processes
- Network capable (TCP/IP)
- Client-server model
- Most flexible IPC method

Sockets: Types & Protocols

Communication Styles

- **Connection-oriented:** Guarantees delivery and order (TCP)
- **Datagram:** No guarantees, best-effort (UDP)

Namespaces

- **Local:** Same machine communication
- **Internet:** Network communication (IP:Port)

IPC Methods Comparison

Shared Memory: Pros & Cons

Advantages ✓

- Fastest IPC method
- No system calls during transfer
- Bidirectional communication
- Multiple processes can access

Disadvantages ✗

- Requires synchronization
- Complex to coordinate
- Risk of data corruption
- Programmer responsibility

Pipes: Pros & Cons

Advantages ✓

- Simple to use
- Automatic synchronization
- Built-in blocking behavior
- Ordered data delivery

Disadvantages ✗

- Unidirectional only
- Limited to related processes
- Limited capacity
- Cannot connect arbitrary processes

Sockets: Pros & Cons

Advantages ✓

- Bidirectional communication
- Network capable
- Works with any processes
- Industry standard (TCP/IP)

Disadvantages ✗

- More complex to implement
- Higher overhead
- Slower than shared memory
- Network latency

When to Use Each Method?

- **Shared Memory:** High-speed data sharing, frequent updates, intensive computations
- **Memory Mapping:** File-based data sharing, persistent data, database caching
- **Pipes:** Producer-consumer patterns, shell commands (command chaining)
- **Named Pipes:** Unrelated process communication, data streams
- **Sockets:** Network communication, client-server architecture, remote services

Key Considerations for IPC Selection

- **Performance:** Shared memory is fastest; Sockets are slower but flexible
- **Scope:** Local only vs. network communication needed?
- **Relationship:** Related processes (pipes) vs. unrelated (sockets, shared memory)
- **Synchronization:** Built-in (pipes) vs. manual (shared memory)
- **Complexity:** Simple to implement vs. more complex but powerful

Chapter 5: Key Takeaways

- IPC enables processes to communicate and coordinate in operating systems
- Each method has different characteristics suited for different scenarios
- Shared Memory offers speed; Sockets offer flexibility and reach
- Pipes provide simplicity with automatic synchronization
- Choice depends on performance needs, process relationships, and scope
- Modern systems use multiple IPC methods for different tasks

Questions?