Week 5 Journal: Inter-Process Communication (IPC)

Course: Operating Systems **Instructors:** Tanaya Bowade & Dr Shabih Fatima **Learning Objective:** LO2 - IPC mechanisms and theoretical foundations

---

Table of Contents

---

1. Introduction to IPC

Definition

Inter-Process Communication (IPC) is a fundamental mechanism within an operating system that enables **COOPERATING processes** to exchange data and coordinate their activities.

Why Processes Need Communication

Information Sharing

Multiple processes accessing shared data need mechanisms to communicate and coordinate access.

Computation Speedup

Breaking tasks into concurrent subtasks allows for parallel processing and improved performance.

Modularity

Enabling modular components to interact promotes better software design and maintainability.

Convenience

Providing a structured way for processes to interact simplifies application development.

Real-world Examples

Web Browsers

Modern web browsers use separate processes for tabs and extensions that communicate to share data, improving stability and security.

Shell Pipelines

Commands can be chained together using pipes (e.g., `ls | grep .txt`) to create powerful data processing workflows.

Client-Server Applications

Client processes send requests to server processes, which process the requests and respond, forming the foundation of networked applications.

2. IPC Models Overview

Inter-Process Communication mechanisms rely on two primary models: **shared memory** and **message passing**. The choice between them depends on application requirements.

Shared Memory Model

Characteristics: - **Speed:** High speed - direct memory access - **Complexity:** Higher - requires synchronization - **Synchronization:** Explicit synchronization needed for data consistency

How it works: Multiple processes access the same physical memory region, allowing for very fast data exchange without involving the kernel after initial setup.

Message Passing Model

Characteristics: - **Speed:** Lower - kernel involvement required - **Complexity:** Lower - simpler to manage - **Synchronization:** Built-in to message semantics

How it works: Processes communicate by exchanging messages through the operating system kernel via system calls like `send()` and `receive()`.

Key Decision Factor

Choose shared memory for performance, message passing for simplicity.

---

3. Shared Memory Systems

Concept

Shared memory allows multiple processes to access the same physical memory region for high-speed data exchange. This is the fastest IPC mechanism as it avoids data copying between address spaces.

Advantages

High Speed

Direct memory access without system calls after initial setup makes this the fastest IPC method.

Efficient for Large Data

No data copying between address spaces means large datasets can be shared efficiently.

Challenges

Complex Synchronization

Requires semaphores, mutexes, or other synchronization primitives to prevent race conditions and ensure data consistency.

Security Risks

Data is vulnerable if not properly managed. Improper access control can lead to security vulnerabilities.

Producer-Consumer Example

A classic example of shared memory usage:

- **Producer** generates data into a shared buffer
- **Consumer** retrieves data from the same buffer
- **Synchronization** ensures:
- Producer doesn't write to a full buffer
- Consumer doesn't read from an empty buffer

This pattern requires careful coordination to maintain data integrity and prevent race conditions.

---

4. Message Passing Systems

Concept

Message passing is an IPC mechanism where processes communicate by exchanging messages. Communication occurs through the operating system kernel via system calls such as `send()` and `receive()`.

Direct Communication

In direct communication, processes explicitly name each other:

- `send(P, message)`: sends a message to process P
- `receive(Q, message)`: receives a message from process Q

This creates tight coupling between processes as they must know each other's identities.

Indirect Communication

Uses mailboxes (also called ports) to facilitate communication:

- Messages are sent to specific mailboxes
- Mailboxes can be shared by multiple processes
- Provides loose coupling between processes

Client-Server Example

A web browser (client) sends a request message to a web server. The server processes the request and sends a response back. This exchange relies on message passing mechanisms and forms the foundation of networked applications.

Flow: 1. Client Process (Browser) ��� Request Message: `send(server, request)` 2. Server Process (Web Server) processes the request 3. Server Process ��� Response Message: `send(client, response)` 4. Client receives and processes response

---

5. Pipes and Redirection in Unix

Unix Philosophy

Unix emphasizes building small, simple, and modular programs that each do one thing well, combining them using mechanisms like pipes for complex tasks. This philosophy enables powerful data processing through composition.

Pipe (|)

Pipes connect the standard output (stdout) of one command to the standard input (stdin) of another:

```
bash ls -l | wc -l
```

This lists files and then counts the lines (number of files/directories).

Redirection

Output Redirection (>)

`bash echo "Hello" > file.txt` Writes "Hello" to file.txt (overwrites if exists)

Input Redirection (<)

`bash sort < names.txt` Sorts lines in names.txt

How Pipes Work

```
Cmd1 ��� stdout (Produces output) ��� Pipe ��� stdin
(Consumes input) ��� Cmd2
```

The pipe creates a unidirectional communication channel where data flows from one process to another in a stream-like fashion.

---

6. Pipes, FIFOs, and Sockets

Pipes vs. FIFOs Comparison

Feature	Pipes (Anonymous)	FIFOs (Named)
Nature	Unidirectional communication	One-way (half-duplex) ��� but can be opened twice for full-duplex communication
Naming	Unnamed; created in memory	Named; appears as a special file in the file system
Relationship	Used between related processes (e.g., parent-child)	Used between unrelated processes
Persistence	Exist only as long as communicating processes are active	Persist until explicitly deleted from the file system
Creation	Created using the `pipe()` system call	Created using the `mkfifo()` system call

Sockets

Sockets serve as an endpoint for communication, enabling processes to exchange data either on the same machine or across a network.

Key Features: - More versatile and robust IPC mechanism compared to pipes and FIFOs - Support various communication protocols (TCP, UDP) - Unix domain sockets allow IPC between processes on the same machine - Network sockets enable communication across different machines

Socket Communication Example:

Client-server model provides a fully bidirectional communication channel where both client and server can send and receive data simultaneously.

---

7. Signals and Process Communication

Definition

Signals are asynchronous notifications in Unix-like systems that indicate events to processes. They provide a lightweight mechanism for process control and event notification.

Common Signals

SIGINT

- Generated by Ctrl+C
- Requests graceful termination
- Can be caught and handled by the process

SIGKILL

- Forceful termination
- Cannot be caught or ignored
- Immediately terminates the process

SIGTERM

- Request for termination
- Can be caught and handled
- Allows cleanup before termination

How Processes Use Signals

Process Management

- Parent processes send signals to children
- Child processes notify parent of events
- Enables hierarchical process control

System Calls

The `kill()` system call is used to send signals to processes: `bash kill [signal] [PID]`

---

8. Synchronization Primitives

Introduction

Synchronization primitives are crucial for managing concurrent access to shared resources in multi-process or multi-threaded environments. They ensure data consistency and prevent race conditions.

Semaphores

Signaling mechanisms for controlling access to resources: - Maintain a count of available resources - Processes can block until a resource is available - Support both binary (mutex-like) and counting operations

Mutexes

Mutual exclusion objects for exclusive access: - Ensures only one process/thread accesses critical section - Process blocks until mutex is available - Provides binary locking mechanism (locked/unlocked)

Monitors

High-level constructs encapsulating shared data: - Provide mutual exclusion for data access - Support condition synchronization - Abstract away low-level locking details

Importance in IPC

- **Prevent race conditions** in shared data access
- **Ensure data consistency** across processes
- **Coordinate process execution order** for dependent operations

---

9. IPC in Practice

Real-world Applications

Databases

Utilize shared memory for efficient cache management, allowing multiple database processes to access and modify shared data structures in memory, reducing disk I/O and improving performance.

Web Servers

Employ multiple processes or threads to handle concurrent client requests. These processes communicate using various IPC mechanisms, such as message queues or sockets, to coordinate tasks and share session information.

OS Components

The kernel and various user-space daemons communicate extensively using IPC. For instance, a system logger might receive messages from different processes via message queues or named pipes.

Microservices

Distributed systems often rely on IPC, particularly network sockets (TCP/UDP), for communication between independent services running on the same or different machines.

Performance Considerations

Kernel Involvement

Message passing has higher overhead due to context switches and data copying, while shared memory allows direct memory access with minimal kernel involvement.

Data Amount

For large data transfers, shared memory is more efficient as it avoids copying data between process address spaces.

Communication Frequency

For frequent communication, the overhead of establishing connections in message passing may be offset by its simplicity and built-in synchronization.

Process Location

Local communication favors shared memory for performance, while networked communication typically requires message passing mechanisms like sockets.

---

10. Laboratory Exercises

Learning Objectives

- Understand inter-process communication mechanisms
- Implement basic IPC examples
- Explore pipes, signals, and file-based communication

Task 1.1: Pipes and Redirection

Understanding Standard Streams

Standard Output: `bash echo "Hello World"`

Redirect Output to File: `bash echo "Hello World" > output.txt cat output.txt`

Append to File: `bash echo "Second line" >> output.txt cat output.txt`

Redirecting Standard Error

```bash

# This command will generate an error

ls /nonexistent 2> error.txt cat error.txt ```

**Redirect both stdout and stderr:** `bash ls /nonexistent /etc 2>&1 > combined.txt cat combined.txt`

**Using Pipes**

**Simple Pipe:** `bash ls -l | grep "txt"`

**Chain Multiple Commands:** `bash ps aux | grep "bash" | wc -l`

**Process Log Files:** `bash sudo cat /var/log/auth.log | grep "Failed" | tail -10`

**Complex Pipeline:** `bash cat /etc/passwd | cut -d: -f1,3 | sort -t: -k2 -n | tail -5`

**Find Top 10 Processes by Memory Usage:** `bash ps aux | sort -k4 -r | head -10`

**Key Insight:** Pipelines enable process cooperation and data flow between programs, implementing the Unix philosophy of combining simple tools to accomplish complex tasks.

## Task 1.2: Named Pipes (FIFOs)

**Creating a Named Pipe**

```
bash mkfifo mypipe ls -l mypipe
```
```

Note the 'p' indicating pipe type.

Writing to the Pipe (Terminal 1)

```
bash echo "Message through pipe" > mypipe
```

This will block until someone reads.

Reading from the Pipe (Terminal 2)

```
bash cat mypipe
```

Bidirectional Communication

Terminal 1: `bash tail -f mypipe`

Terminal 2: `bash echo "First message" > mypipe echo "Second message" > mypipe echo "Third message" > mypipe`

Producer/Consumer Pattern with Named Pipes

Terminal 1 (Producer): `bash for i in {1..10}; do echo "Data item $i" > mypipe sleep 1 done`

Terminal 2 (Consumer): `bash while read line; do echo "Received: $line" done < mypipe`

Cleanup: `bash rm mypipe`

Key Insight: Named pipes (FIFOs) differ from anonymous pipes by having a name in the filesystem, allowing unrelated processes to communicate. They persist until explicitly deleted.

Task 1.3: Process Signals

List Available Signals

```
bash kill -l
```

Basic Signal Operations

Start a process: `bash sleep 300 & PID=$! echo "Process ID: $PID"`

Send SIGTERM (graceful termination): `bash kill $PID`

Force kill with SIGKILL: `bash sleep 300 & PID=$! kill -9 $PID`

Suspend and Resume

```bash sleep 300

## Press Ctrl+Z (sends SIGTSTP - suspend)

jobs bg # Resume in background fg # Bring to foreground ```

**Signal Handling Script**

**signal-demo.sh:** ```bash

## !/bin/bash

## Signal Handling Demonstration

## Function to handle SIGINT (Ctrl+C)

handle_sigint() { echo "" echo "SIGINT received! Cleaning up..." echo "Programme terminated gracefully" exit 0 }
```

Function to handle SIGTERM

handle_sigterm() { echo "" echo "SIGTERM received! Shutting down..." exit 0 }

Trap signals

trap handle_sigint SIGINT trap handle_sigterm SIGTERM

echo "Signal handler running (PID: $$)" echo "Press Ctrl+C to send SIGINT" echo "Or use 'kill $$' from another terminal to send SIGTERM" echo ""

Main loop

counter=0 while true; do echo "Running... (iteration $counter)" sleep 2 ((counter++)) done ```

Make Executable and Test: ```bash chmod +x signal-demo.sh ./signal-demo.sh

Test with Ctrl+C or from another terminal: kill [PID]

```

**Key Insight:** Signal handling is crucial for robust applications. It allows processes to respond gracefully to termination requests, perform cleanup operations, and manage their lifecycle properly.

### Task 1.4: File-Based IPC Example

#### Producer Script
```

producer.sh: ```bash

!/bin/bash

Producer: Writes data to shared file

DATAFILE="shared_data.txt" LOCKFILE="data.lock"

echo "Producer started (PID: $$)"

for i in {1..10}; do # Wait for lock to be available while [-f "$LOCKFILE"]; do sleep 0.1 done

```
# Create lock
touch "$LOCKFILE"

# Write data
timestamp=$(date +"%H:%M:%S")
echo "[$timestamp] Data item $i" >> "$DATAFILE"
echo "Produced: Data item $i"

# Release lock
rm "$LOCKFILE"

sleep 1
```

done

echo "Producer finished" ```

Consumer Script

consumer.sh: ```bash

!/bin/bash

Consumer: Reads data from shared file

DATAFILE="shared_data.txt"

echo "Consumer started (PID: $$)" echo "Waiting for data..."

Wait for file to exist

while [! -f "$DATAFILE"]; do sleep 0.5 done

Monitor file for new data

tail -f "$DATAFILE" & TAIL_PID=$!

Run for 15 seconds

sleep 15

Stop monitoring

kill $TAIL_PID 2>/dev/null

echo "" echo "Consumer finished" ```

Testing

Make Executable: `bash chmod +x producer.sh consumer.sh`

Terminal 1: `bash ./consumer.sh`

Terminal 2: `bash ./producer.sh`

Cleanup: `bash rm shared_data.txt data.lock`

Key Insight: File-based IPC demonstrates the challenges of inter-process synchronization and race conditions. The lockfile mechanism provides simple mutual exclusion, ensuring only one process accesses the shared file at a time.

---

11. Assessment Requirements

Phase 2: Security Planning and Testing Methodology (Week 2)

Design a security baseline and performance testing methodology.

Deliverables (Journal):

1. **Performance Testing Plan**
2. Describe remote monitoring methodology
3. Outline testing approach

4. Define metrics and benchmarks

5. **Security Configuration Checklist**

6. SSH hardening
7. Firewall configuration
8. Mandatory access control
9. Automatic updates
10. User privilege management

11. Network security

12. **Threat Model**

13. Identify at least 3 specific security threats
14. Provide mitigation strategies for each threat
15. Assess risk levels and priorities

---

12. Review and Key Concepts

Key Concepts Summary

Shared Memory

Direct memory access for high-speed data exchange, requiring explicit synchronization to prevent race conditions.

Message Passing

Communication via explicit messages, simpler to manage but with higher overhead due to kernel involvement.

Pipes & FIFOs

- **Anonymous Pipes:** Unidirectional communication channels for related processes (parent-child)
- **Named Pipes (FIFOs):** Named files in the filesystem for communication between unrelated processes

Sockets

Network-enabled IPC for communication between processes on the same or different machines. Support multiple protocols (TCP, UDP).

Signals

Asynchronous notifications for event handling or process control. Enable processes to respond to system events and user actions.

Synchronization

Essential for managing concurrent access to shared resources and preventing data inconsistencies. Includes semaphores, mutexes, and monitors.

Exam Preparation Tips

Understand Fundamental Purpose

Be clear on the basic operational mechanism of each IPC type. Know what problem each mechanism solves.

Know the Trade-offs

Be prepared to articulate advantages and disadvantages between different IPC models: - Shared memory vs. message passing - Pipes vs. FIFOs vs. sockets - Different signal types

Provide Examples

Be ready to give relevant examples for each IPC mechanism from both theory and practical applications.

Compare and Contrast

Understand the differences between: - Shared memory and message passing models - Anonymous and named pipes - Direct and indirect communication - Synchronous and asynchronous communication

Focus on conceptual understanding rather than memorization.

Review Questions

1. **IPC Models:** What are the main differences between shared memory and message passing?

2. **Pipes and FIFOs:** How do pipes and FIFOs differ in terms of naming, persistence, and use cases?

3. **Signals:** What role do signals play in IPC? Name three common signals and their purposes.

4. **Synchronization:** Why is synchronization important in IPC? What are the main synchronization primitives?

Conclusion

Inter-Process Communication is fundamental to modern operating systems, enabling cooperating processes to exchange data and coordinate activities. Understanding the trade-offs between different IPC mechanisms���shared memory for performance, message passing for simplicity, pipes for stream processing, and signals for event notification���is crucial for designing efficient and robust systems.

The practical laboratory exercises demonstrate how these theoretical concepts are applied in real-world Unix/Linux systems, from simple pipelines to complex producer-consumer patterns with proper synchronization.

End of Week 5 Journal

Table of Contents