People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research

Larbi Ben M'hidi University - Oum El Bouaghi
Faculty of Exact Sciences and Natural and Life Sciences
Department of Mathematics and Computer Science

Course Handout

Operating Systems 2

3rd Year Bachelor's Degree in Computer Science

– SI & ISIL -

Presented by:

Dr. Rohallah BENABOUD

DISE (Distributed-Intelligent Systems Engineering) Team
ReLa(CS)2 Laboratory, University of Oum El Bouaghi

Preface

This course is an advanced study of operating system principles, intended for students who have a foundational understanding of the subject. It addresses the critical and complex challenges related to the management of concurrent processes and the resources they share.

The main objective is to produce course support that is both rigorous and clear, ensuring a deep and unambiguous understanding of the concepts covered. The goal is to facilitate mastery of the principles of concurrency and to gather the necessary knowledge for designing and implementing robust, efficient, and correct multi-process and multi-threaded software.

It should be noted that certain prerequisites are necessary to fully grasp the concepts presented. A foundational understanding of operating systems (process and memory management basics), as well as computer architecture, is assumed. Furthermore, strong programming skills, particularly in the C language, are required to understand and implement the practical examples.

This course is organized into four main chapters:

First, Chapter 1 provides a review of fundamental operating system concepts. It re-establishes the role and architecture of an OS and provides a detailed examination of the two core units of execution: processes and threads. This chapter covers their structure, lifecycle, and management, providing the essential foundation for the rest of the course.

Chapter 2 describes the techniques for Process Synchronization. This chapter addresses the core challenge of concurrency: the race condition. It introduces the critical-section problem and details the various solutions, from early software algorithms (like Peterson's) to hardware support (atomic instructions) and powerful OS-level primitives like semaphores and monitors.

The following chapter, Chapter 3, deals with Interprocess Communication (IPC). Once processes are synchronized, they often need to cooperate. This chapter covers the primary mechanisms that allow processes to exchange information, including shared memory, asynchronous notifications via signals, and data streaming using both unnamed and named pipes.

Chapter 4 is entirely dedicated to the problem of Deadlocks. It begins by characterizing the four necessary conditions for a deadlock to occur. It then details the primary strategies for managing them: Deadlock Prevention, by structurally negating one of the conditions; Deadlock Avoidance, using the Banker's Algorithm to maintain a safe state; and finally, Deadlock Detection and Recovery.

Table of Contents

Chapter 1

Operating System Fundamentals: Processes and Threads

Summary

The purpose of this chapter is to build a ground-up understanding of what an operating system is, what it does, and the essential components it manages. We will explore several key areas to achieve this:

- **Fundamental Concepts**: We will start by defining the role and goals of an OS, looking at its historical evolution, and examining the different architectures that form its internal structure.
- **The Process**: We will then introduce the **process**, the basic unit of execution in an operating system. We will cover how processes are created, how they transition between different states (like running and waiting), and how the OS manages them.

- **Concurrency with Threads**: Finally, we will introduce **threads** as a more lightweight mechanism for achieving concurrency within a single process, exploring their advantages and the common models for implementing them.

By the end of this chapter, students will have a solid grasp of these essential concepts, providing the necessary foundation to understand the more advanced topics in process synchronization, process communication.

1. Overview of Operating System Concepts

1.1 The Role of the Operating System

At the heart of every modern computer lies a crucial software component: the **operating system (OS)**. An OS acts as the primary intermediary between the computer's hardware and the user, with the fundamental purpose of providing an environment in which a user can execute programs conveniently and efficiently. As illustrated in Figure 1.1, the operating system is the foundational software layer that rests directly on the hardware, managing all resources and providing a platform for all other application software.

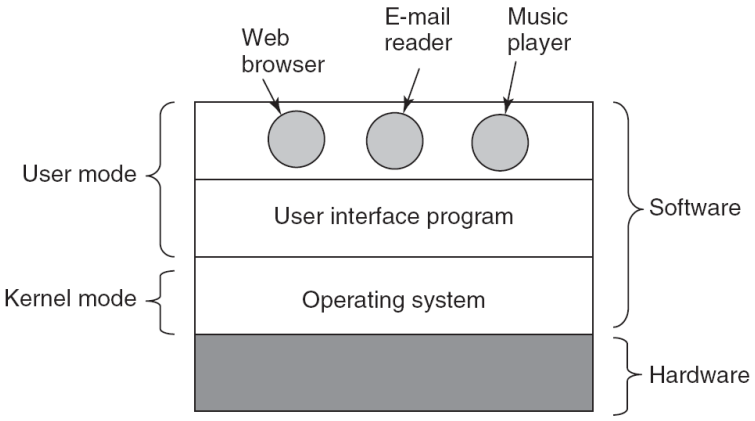


Figure 1.1 The Operating System's Position

The primary goals of an operating system can be summarized as follows:

Execute User Programs: An OS simplifies the process of running applications like web browsers or music players by managing the loading of programs into memory and overseeing their execution. It provides Application Programming Interfaces (APIs) and system calls that allow developers to build software without needing to manage the underlying hardware directly.

Enhance User Convenience: By offering intuitive interfaces—whether graphical (GUI) or command-line (CLI)—the OS abstracts the immense complexity of hardware operations into simple, manageable tasks. It also enables multitasking, allowing users to run multiple applications concurrently.

Maximize Hardware Efficiency: A key responsibility of the OS is to manage system resources like the CPU, memory, and storage devices to maximize their utilization. It achieves this through sophisticated techniques such as CPU scheduling, memory allocation, and optimizing I/O operations to ensure fairness and speed among competing processes.

1.2 The Evolution of Operating Systems

Operating systems have evolved significantly over the decades, driven by advancements in hardware and computing needs. Below is a summarized timeline of key generations in OS development:

Generation	Time Period	Hardware Technology	Operating System Type	Key Characteristics
FIRST	1945-1955	Vacuum Tubes	None (Plugboards)	Programs were loaded manually via hardware switches and punch cards.
SECOND	1955-1965	Transistors	Batch Processing Systems	Jobs were grouped and run sequentially, automating the transition from one job to the next.
THIRD	1965-1980	Integrated Circuits (ICs)	Multiprogramming OS	Multiple programs could reside in memory at once, and time-sharing was introduced to allow for interactive use.
FOURTH	1980-Present	Large-Scale Integration (LSI)	Personal Computer (PC) OS	The rise of PCs brought GUI-based systems like Windows and macOS, along with integrated networking support.

1.3 Modern System Architectures

Modern computing systems are designed in various ways, primarily distinguished by the number of processors they employ.

1.3.1 Single-Processor and Multiprocessor Systems

Single-processor system contains one main CPU that executes user programs. While common in the past, today's systems are predominantly

Multiprocessor systems, which feature two or more CPUs (often as "cores" on a single chip) sharing memory and peripherals. This design offers several advantages:

- Increased Throughput: More tasks can be performed in parallel.
- Economy of Scale: Sharing resources like power supplies and storage is more cost-effective.
- Enhanced Reliability: If one processor fails, the others can continue to operate, a principle known as fault tolerance.

Multiprocessing can be Symmetric (SMP), where all CPUs are peers and perform all OS tasks, or Asymmetric (AMP), where a master CPU assigns tasks to worker CPUs. SMP is the standard in modern operating systems like Windows, Linux, and macOS.

1.3.2 Clustered Systems

Clustered systems are composed of multiple individual computers (or nodes) that are linked together to work as a single, unified system. They are crucial for services that require high availability, as a failure in one node will cause its tasks to be reassigned to other nodes, ensuring continuous operation.

1.4 Operating System Structures

The internal design of an operating system, or its architecture, dictates how its components are organized and interact.

1.4.1 Monolithic Systems: The entire OS is structured as a single, large executable where all components can freely call one another. As shown in Figure 1.2, core functions like file systems, process management, and I/O management are tightly integrated within the kernel. While this design offers high performance, its lack of internal boundaries makes it difficult to debug and maintain.

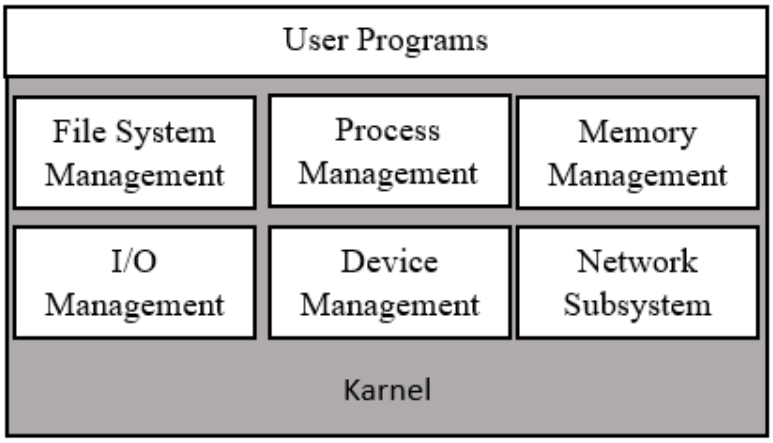


Figure 1.2. Monolithic Systems

1.4.2 Layered Systems: The OS is organized into a hierarchy of layers (Figure 1.3). Each layer is built on top of the one below it and can only invoke services from that lower layer. This modular approach simplifies development and debugging but can introduce performance overhead.

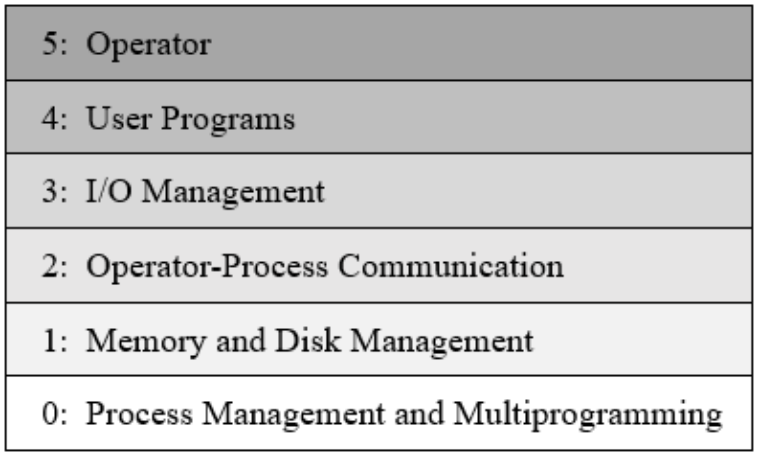


Figure 1.3. Layered Systems

1.4.3 Virtual Machines (VMs): A VM abstracts the physical hardware to create multiple isolated, virtual copies of a complete computer system (Figure 1.4). Each VM can run its own guest OS independently, providing excellent security and flexibility.

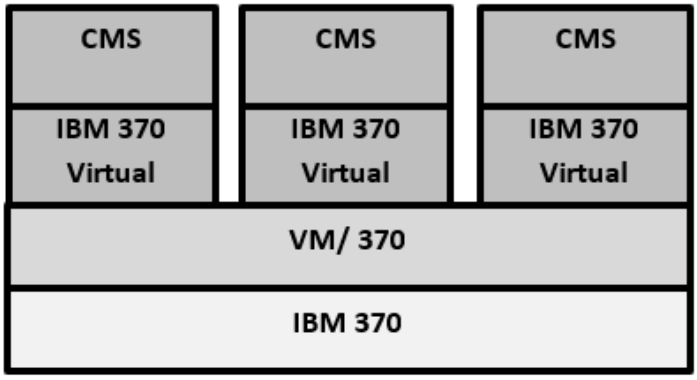


Figure 1.4. Virtual Machines

1.4.4 Client-Server Model (Microkernel): This model streamlines the OS by moving most services (e.g., file systems, networking) out of the core kernel and into user-space processes called servers (Figure 1.5). A minimal microkernel handles only essential tasks like scheduling and basic I/O. This design is highly modular and secure, as a crash in a server process will not bring down the entire system.

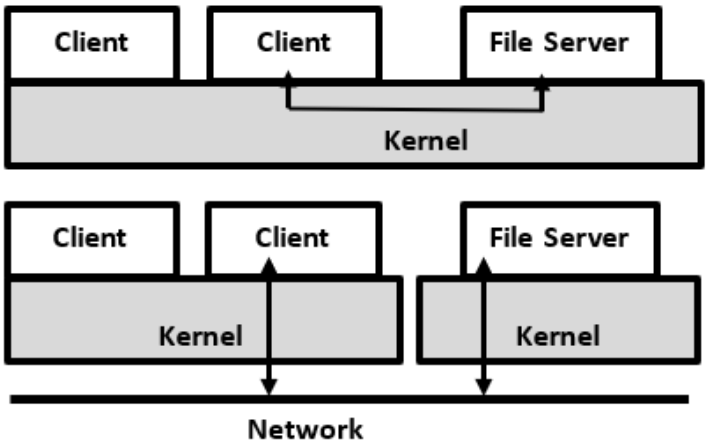


Figure 1.5. Client-Server Model

1.5 Classification of Operating Systems

Operating systems can be classified into several categories based on their architecture, the number of users they support, and their intended application. Understanding these classifications helps to contextualize the design choices behind different systems.

1.5.1 Single-Tasking vs. Multi-Tasking

- **Single-tasking systems** can only execute one process at a time. Once a program begins, it retains full control of the system until it finishes. A classic example is **MS-DOS**.
- **Multi-tasking systems**, which are the standard today, are capable of running multiple processes concurrently. This is achieved through rapid time-sharing, where the OS switches between processes, creating the illusion of parallel execution. This allows a user to, for instance, compile code while editing a document.

1.5.2 Single-User vs. Multi-User

- **Single-user systems** are designed to support only one user at a time and do not provide mechanisms for separating user data or privileges. Early personal computer operating systems fell into this category.
- **Multi-user systems**, such as **UNIX** and **Linux**, are designed to manage the needs of multiple users simultaneously on the same machine. They provide robust mechanisms for user isolation, resource allocation, and security to protect each user's data and processes from one another.

1.5.3 Centralized vs. Distributed

- A **centralized operating system** runs on a single computer and manages only the resources local to that machine.
- A **distributed operating system** manages resources across multiple, independent computers, making them appear to the user as a single, cohesive system. This

architecture is the foundation for modern cloud computing and large-scale data processing frameworks.

1.5.4 Real-Time Systems

A **real-time operating system (RTOS)** is designed to process data as it comes in, typically without buffer delays. The key requirement is that tasks must be completed within a strict, guaranteed timeframe, as a late result can be as bad as a wrong one.

- **Hard Real-Time Systems** have absolute, immovable deadlines. A missed deadline constitutes a total system failure. These are used in mission-critical applications like flight control systems and medical devices.
- **Soft Real-Time Systems** can tolerate occasional delays. While meeting deadlines is important, a missed deadline only degrades the quality of service rather than causing a complete failure. Digital audio and video streaming are common examples.

1.6 A Timeline of Influential Operating Systems

The theoretical concepts of operating systems are best understood by examining the real-world systems that defined each era of computing. The following timeline highlights some of the most influential operating systems and their key contributions.

Early Systems

- **CP/M (Control Program for Microcomputers) - 1974**: Developed by Digital Research, CP/M was the first major operating system for microcomputers. It featured a basic, flat file system with no directories and laid the groundwork for future personal computer operating systems. Its design heavily influenced MS-DOS.
- **UNIX - 1969**: Developed at AT&T Bell Labs, UNIX was the first true multi-user, multi-tasking operating system. Its introduction of a hierarchical file system and its portability (due to being written in the C language) were revolutionary concepts that continue to influence OS design today.

The Personal Computer Era

- **MS-DOS (Microsoft Disk Operating System) - 1981**: As the operating system for the original IBM PC, MS-DOS dominated the personal computer market for over a decade. It was a 16-bit, single-tasking system with no memory protection, meaning a faulty program could crash the entire machine.
- **Classic Mac OS - 1984**: Released with the first Macintosh computer, Apple's System 1 was the first commercially successful OS to feature a graphical user interface (GUI). It introduced concepts like windows, icons, and menus to a wide audience and utilized a cooperative multitasking model.

The Modern GUI Era

- **Microsoft Windows**: Beginning as a graphical shell over MS-DOS with Windows 1.0 (1985), Windows evolved into a dominant force. The **Windows 9x line** (Windows 95, 98) brought features like pre-emptive multitasking to the consumer market, while the more robust **Windows NT line** (Windows 2000, XP) provided the foundation for modern versions like Windows 10 and 11.
- **Linux - 1991**: Started as a hobby project by Linus Torvalds, the Linux kernel, combined with GNU tools, created a powerful, open-source UNIX-like operating system. Its strength lies in its vast ecosystem of distributions (distros) tailored for different needs, such as **Debian**, **Red Hat**, and **Ubuntu**.
- **macOS (formerly OS X) - 2001**: Apple replaced its classic Mac OS with OS X, a modern operating system built on a UNIX foundation (BSD). This move combined the stability and power of UNIX with the user-friendly interface that Apple was known for.

The Mobile and Specialized Era

- **Mobile Operating Systems**: The rise of smartphones brought a new wave of operating systems, including **Symbian** (1997), **BlackBerry OS** (1999), and the two that dominate today: **iOS** (2007) and **Android** (2008).
- **Specialized Operating Systems**: Beyond personal computers, specialized systems like **ChromeOS** (for web-centric computing), **QNX** (for real-time systems in cars and industry), and **FreeRTOS** (for IoT devices) are designed for specific tasks and hardware.

2. Process: The Unit of Execution

The concept of the **process** is fundamental to all modern operating systems. All computer software - whether system-level or user-level - is organized into executable units called processes. A process is a program in execution—the active instance of a program that has been loaded into memory and is being run by the CPU.

2.1 Anatomy of a Process

A process is more than just program code; it is a complete execution environment that includes:

Core Components

- **Execution Context:**
  - Program Counter, which indicates the current instruction.
  - The values of the CPU's registers.
  - The execution stack(s), which store temporary data. A process may have multiple stacks if it contains threads.

- **Memory Segments:**
  - This includes the distinct areas for the program's code, global data, and the heap for dynamic memory allocation.

Attributes and Metadata

- **System Attributes:**
  - A unique Process ID (PID).
  - The process's private memory address space.
  - Its current state (e.g., ready, running, or waiting).
  - A priority level for scheduling purposes.
  - A list of allocated resources, such as open files and devices.
- **Execution Characteristics:**
  - A history of alternating between CPU computation and I/O wait periods.
  - Saved register states and signal handling configurations.
  - Statistics on resource usage.
- **Administrative Metadata:**
  - Its position in the process hierarchy (parent/child relationships).
  - Associated environment variables.
  - User and group ownership credentials.

To manage this information, the OS maintains a **Process Table**, with each process represented by a **Process Control Block (PCB)** that stores its complete context. The relationship between the process table and PCBs is illustrated in Figure 1.6.

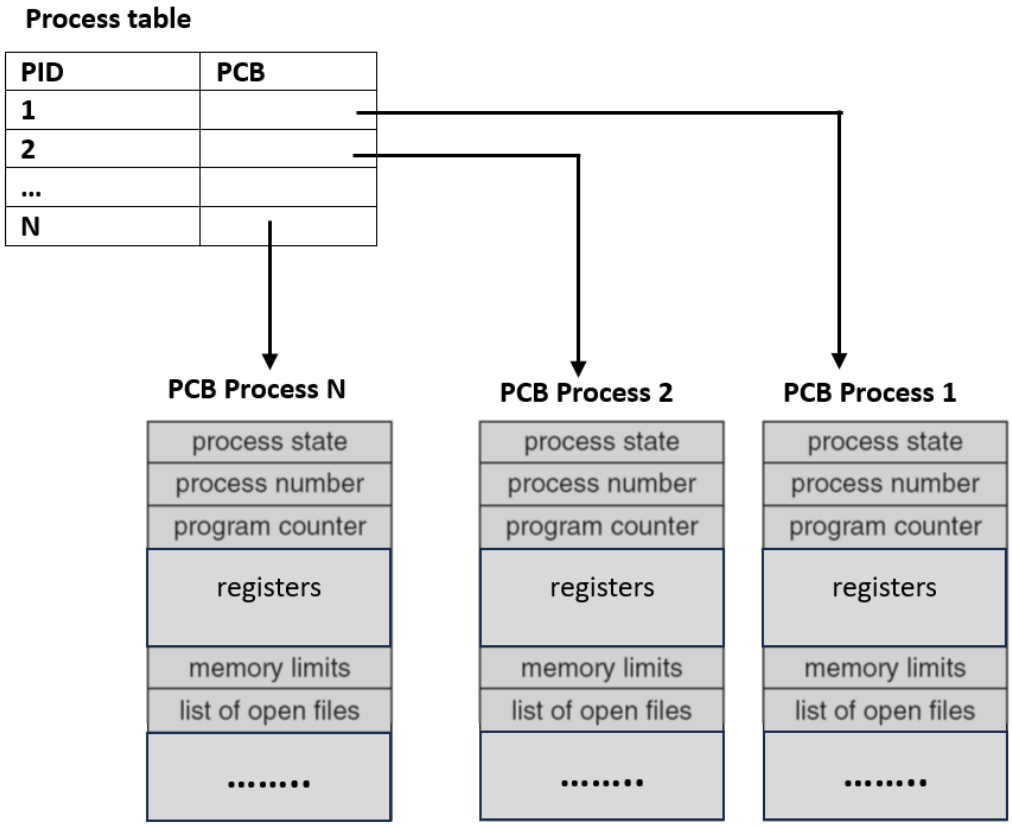


Figure 1.6. Process table and PCBs

2.2 Process States and Context Switching

A process is a dynamic entity, and its status changes throughout its execution. These different statuses are known as process states. The operating system needs to track the state of every process to manage them correctly. A process will be in one of five fundamental states at any given time, with specific events triggering the transitions between them.

As shown in Figure 1.7, the five states of a process are:

1. **New**: This is the initial state where the process is being created. The operating system has acknowledged the request but has not yet fully set it up or admitted it into the pool of executable processes. Resources are being allocated, and the Process Control Block (PCB) is being initialized.
2. **Ready**: In this state, the process has everything it needs to run—it is loaded into memory and is just waiting for the CPU to become available. A "ready queue" often holds all processes that are in this state. The scheduler will pick one of these processes to run next.
3. **Running**: This means the process's instructions are actively being executed by the CPU. A process moves from the Ready state to the Running state when the OS

scheduler dispatches it, giving it control of the processor. On a single-core CPU, only one process can be in the "Running" state at any moment.

4. **Waiting (or Blocked)**: A process enters the Waiting state when it cannot continue without something happening first. For example, it might need to wait for a user to input data, for a file to be read from a disk, or for a network packet to arrive. While waiting, the process cannot use the CPU, even if it were available.
5. **Terminated**: The process has finished its execution or has been explicitly killed. In this final state, the operating system will deallocate all the resources that were assigned to the process and clean up its entry in the process table.

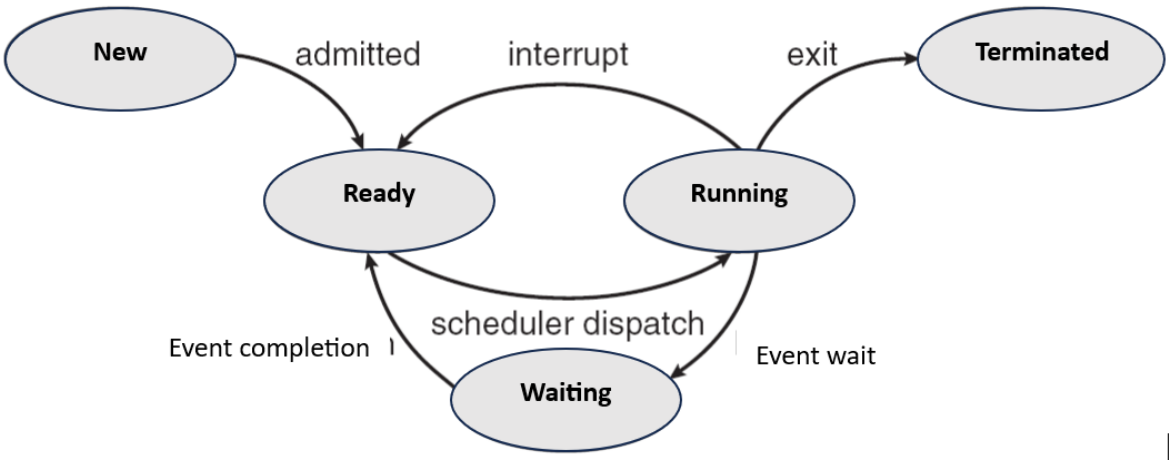


Figure 1.7. Process States

The transitions between the five states can be explained as:

- **New → Ready (Admitted)**: After a process is created, the OS must decide if it has enough resources to let it compete for the CPU. If so, the process is "admitted" and moved into the ready queue.
- **Ready → Running (Scheduler Dispatch)**: The short-term scheduler selects a process from the ready queue and assigns the CPU to it.
- **Running → Ready (Interrupt)**: A running process can be forced to give up the CPU due to an external event, like a timer interrupt for multitasking. Its context is saved, and it is moved back to the ready queue to await its next turn.
- **Running → Waiting (Event Wait)**: If a process needs to perform a long I/O operation or wait for a specific event, it voluntarily relinquishes the CPU and moves to the waiting state.
- **Waiting → Ready (Event Completion)**: Once the event the process was waiting for has occurred (e.g., the disk has finished reading data), the OS moves the process back into the ready queue. It cannot go directly back to "Running" because the CPU may be busy with another process.

- **Running → Terminated (Exit)**: The process finishes its job normally or is terminated by the OS, at which point it moves to the terminated state for deallocation.

Context switching is the core mechanism that enables multitasking in an operating system. It is the process of saving the state of the currently running process and loading the state of the next process that is scheduled to run. This allows a single CPU to handle multiple processes without losing any information.

Why Does a Context Switch Occur?

A context switch is triggered when the CPU needs to stop executing one process and begin executing another. Common triggers include:

- Multitasking: In time-sharing systems, a timer interrupt goes off, forcing the current process to yield the CPU so another process gets its turn.
- I/O Operations: When a process requests an I/O operation, it enters the Waiting state, and the OS switches to another process to keep the CPU busy.
- Interrupt Handling: An external event, like a keypress or network packet arrival, generates an interrupt that may require the OS to switch to a different process to handle it.

Think of a process's context as its complete "workspace"—all the information it needs to pick up exactly where it left off. This workspace is stored in its Process Control Block (PCB).

The steps are as follows:

1. Interrupt or System Call: An event occurs that triggers the switch. The CPU finishes the current instruction and transfers control to the operating system's kernel.
2. Save Context: The OS saves the "workspace" of the current process (Process A). This includes the program counter, all CPU registers, and other status information into Process A's PCB.
3. Update Scheduler: The OS scheduler runs to select the next process to execute from the ready queue (let's say, Process B).
4. Load Context: The OS loads the "workspace" of Process B from its PCB into the CPU's registers.
5. Resume Execution: The CPU is now primed with Process B's information. It resumes executing Process B from the exact point it was last interrupted

Figure 1.8 illustrates the context switch between two processes (Process A and Process B). The currently running process is interrupted, and a scheduler may be called. The scheduler executes in kernel mode to be able to manipulate the PCBs. It's important to note that switching to kernel mode via a system call does not necessarily imply a context switch. Typically, we remain within the same process, except that we gain access to data and instructions that are prohibited in user mode.

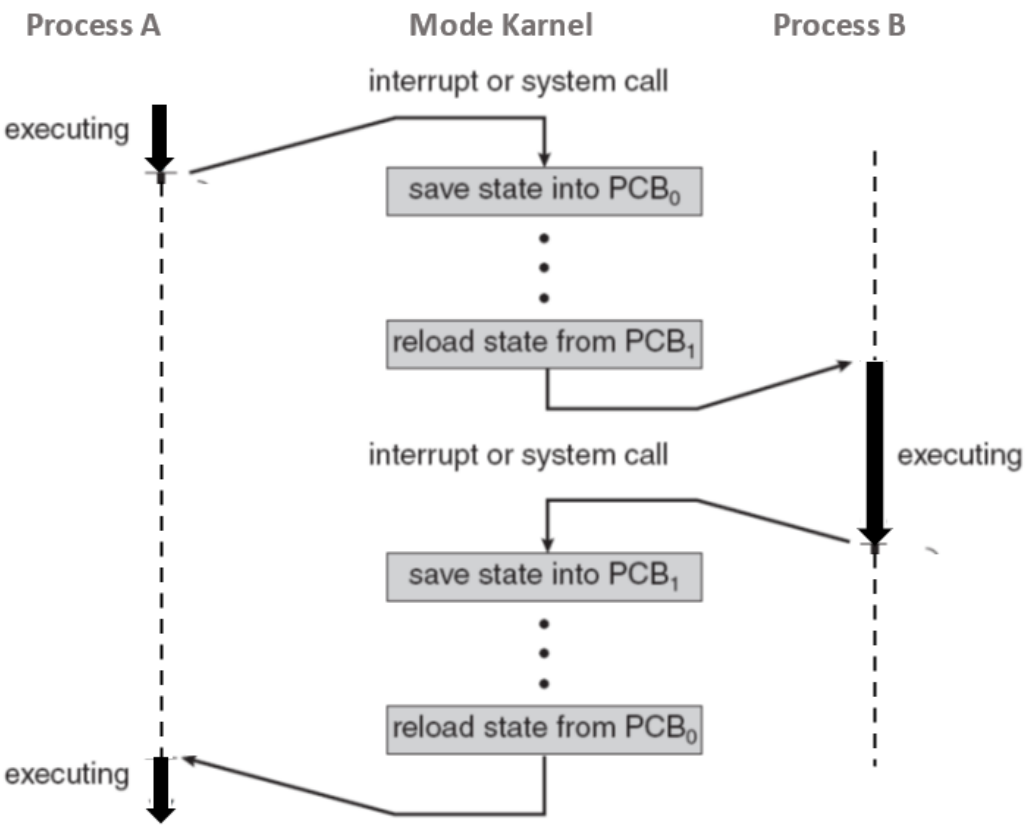
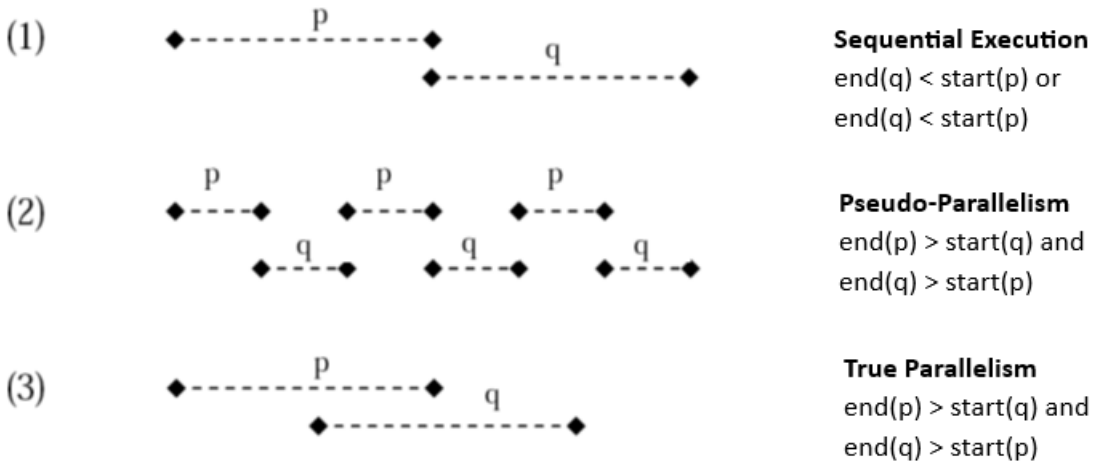


Figure 1.8. Context switch between two processes

2.3 The Process Model: Execution Patterns

When two or more processes, let's call them **P** and **Q**, are set to execute, their interaction can be modeled in three fundamental patterns. These models depend on the capabilities of the operating system and the underlying hardware. We can define these patterns by looking at the start and end times of each process, denoted as **start(x)** and **end(x)**:



Sequential Execution: In this model, one process must complete its entire execution before the other one begins. There is no overlap in their execution times. This is the simplest model and is characteristic of older, single-tasking operating systems.

Pseudo-Parallelism (Concurrent Execution): This pattern creates the illusion that two or more processes are running at the same time, even on a single-CPU system. The operating system achieves this by rapidly switching between the processes (context switching), allowing each to make progress in an interleaved fashion. Their execution periods overlap significantly.

True Parallelism (Simultaneous Execution): True parallelism occurs when processes execute simultaneously, not just in an interleaved manner. This is only possible on a computer with multiple processors or multiple cores. While offering the best performance, this model introduces the complexity of needing synchronization mechanisms, such as locks, to prevent conflicts when processes access shared resources.

2.4 Process Creation

In modern operating systems, processes must be dynamically managed to support concurrent execution - they are frequently created and terminated during system operation. While simple embedded systems (like microwave controllers) may preload all required processes at startup, general-purpose OSs require flexible mechanisms for on-demand process creation and termination. This dynamic process management is essential for handling multiple concurrent tasks efficiently. In this section, we examine these fundamental mechanisms, with specific examples from UNIX systems to illustrate practical implementations.

The Parent-Child Hierarchy

When a process creates a new process, the creator is known as the **parent process**, and the new process is the **child process**. A child process can, in turn, create other processes, forming a hierarchical process tree that represents the relationships between all processes in the system (Figure 1.9). This tree structure is a fundamental organizational concept in operating systems.

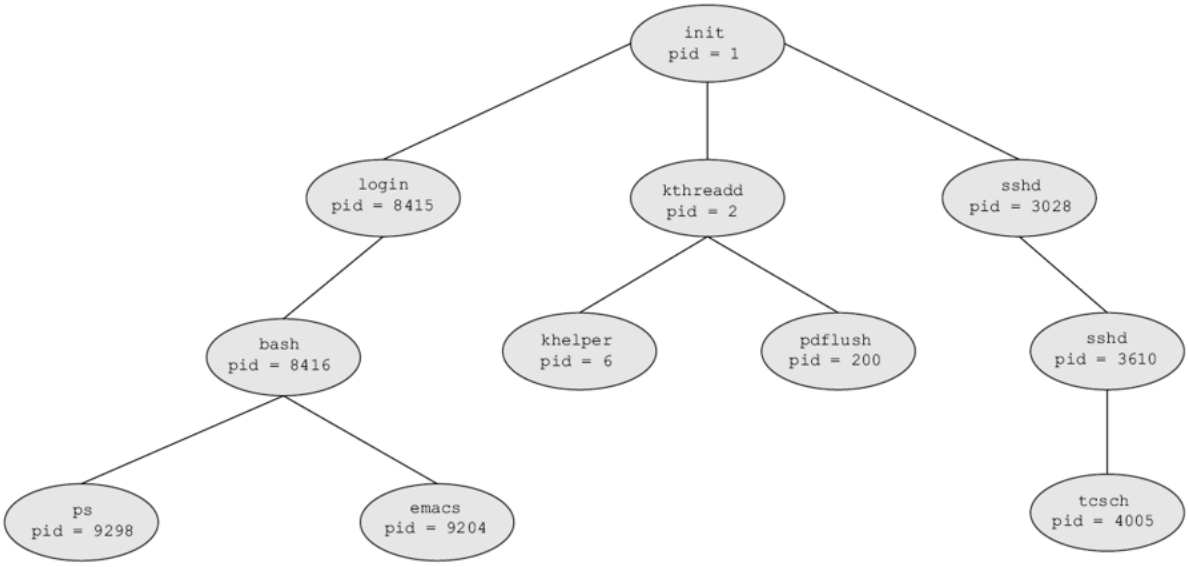


Figure 1.9. A tree of processes

Process Creation in UNIX-like Systems

UNIX-like systems use a small, powerful set of system calls for process management, as defined by the POSIX standard.

- `pid_t fork()`: Child process creation.
- `int execl(), int execlp(), int execvp(), int execle(), int execv()`: The exec() services allow a process to execute a different program (code).
- `pid_t wait()`: Wait for process termination.
- `void exit()`: End process execution.
- `pid_t getpid()`: Returns the process identifier.
- `pid_t getppid()`: Returns the parent process identifier.

In Linux, the pid_t type normally corresponds to a long int.

the fork() system call represents the fundamental mechanism for creating child processes:

```
#include <unistd.h>

int fork();
```

The fork() system call creates a new process by generating an exact duplicate of the calling process. This fundamental UNIX mechanism produces two identical processes - the parent and child - with the following key characteristics:

- **Memory and State Duplication**:
  - ✓ Complete copy of the parent's address space (implemented efficiently via Copy-On-Write)

- ✓ Identical environment strings and open file descriptors
- ✓ Matching execution context (register states, program counter)

– **Post-fork Execution**:

- ✓ Both processes resume execution at the instruction immediately following fork()
- ✓ Differentiation via return value:
  - Child receives 0
  - Parent receives the child's PID (nonzero)

– **Communication Foundation**:

- ✓ Shared open files enable immediate IPC
- ✓ Identical memory layout simplifies debugging
- ✓ COW optimization minimizes initial overhead

The figure illustrates a parent process and its identical child clone immediately after fork(). While both processes share the same initial memory image (including the program counter PC), their execution paths diverge based on fork()'s return value:

– **Variable i Differentiation**:

- ✓ In the **parent process**:
  Receives the child's PID → i = 27 (arbitrary non-zero value shown in the figure).
- ✓ In the **child process**:
  Receives 0 → i = 0

– **Memory Consistency**:

- ✓ At the moment of forking:
  Both processes have identical memory (same variables, same PC value)
- ✓ Post-fork:
  Changes to i (or any variable) become process-specific due to Copy-On-Write

– **Technical Implications**:

- ✓ The divergence in i values is the **first observable difference** between parent and child
- ✓ All other variables initially remain identical until modified
- ✓ The PC's identical starting point ensures both processes execute the **next instruction** after fork()

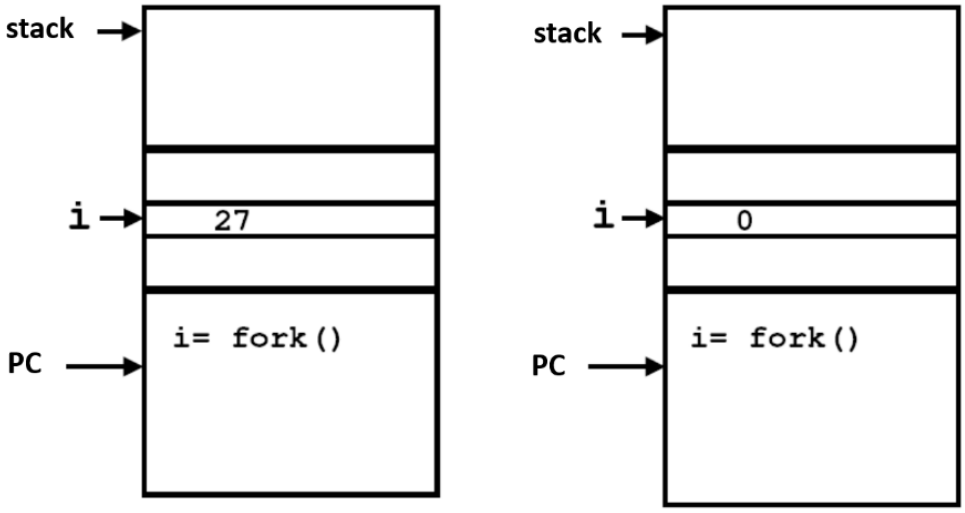


Figure 1.10. fork example

The figure 1.10 demonstrates that after the fork() system call, the pid variable contains:

- 0 in the child process
- The child's process ID (i=27) in the parent process

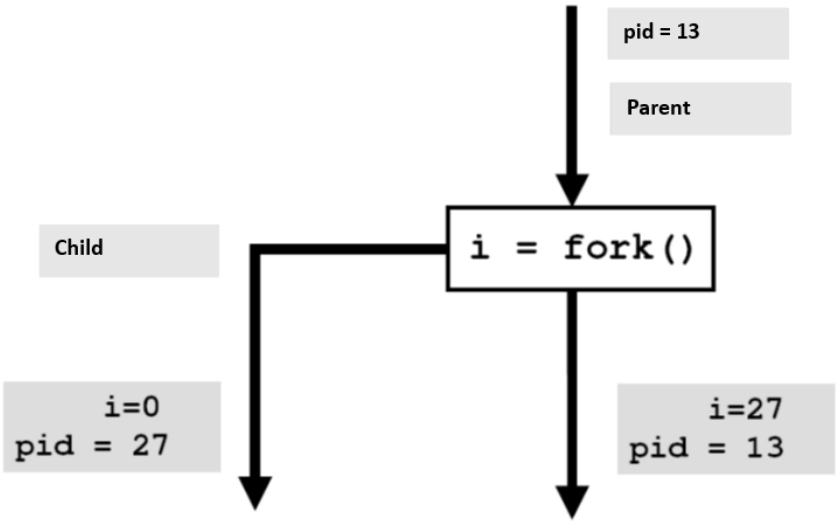


Figure 1.11. After the fork() system call

```c
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>

int main() {
    pid_t child_pid;

    child_pid = fork();

    if (child_pid == 0) {
        // Child process
        printf("I am the child with PID %d\n", getpid());
    }
    else if (child_pid > 0) {
        // Parent process
        printf("I am the parent with PID  %d\n", getpid());
    }
    else {
        // fork() failed
        perror("fork() error");
        return 1;
    }

    return 0;
}
```

Figure 1.12 A C program for process creation

The C program shown in Figure 1.12 illustrates the UNIX system calls previously described. The execution of this program shows the PIDs of the parent and child:

```
$ gcc -o fils fils.c

$ ./fils

I am the parent with PID 5120

$ I am the child with PID 5121
```

2.5 Managing Process Termination

Properly managing the end of a process's life is crucial for a stable operating system. Two special cases, zombie and orphan processes, arise from the parent-child relationship and must be handled correctly.

Zombie Processes

A child process becomes a zombie if it terminates, but its parent process does not "reap" it by calling `wait()` or `waitpid()`.

When a child finishes its execution, its resources like CPU time and memory are released. However, its entry in the system's process table remains because it still holds valuable

information—specifically, its exit status. The parent process needs to read this status to know if the child completed its task successfully.

Until the parent calls wait() to collect this information, the child is in a "zombie" state: it's dead but not yet gone. While a few zombies are harmless, a large number can fill up the process table, preventing new processes from being created.

- Cause: A child process terminates before its parent has called wait().
- State: The process is defunct and does not use CPU or memory, but its entry remains in the process table.
- Solution: The parent process must execute a wait() or waitpid() system call to read the child's exit status, which allows the operating system to finally remove the child from the process table.

Orphan Processes

An orphan process is created when its parent process terminates before the child does. Without a parent, the child would be left running without any process to manage it or reap its exit status when it eventually finishes.

To prevent this, the operating system has a built-in adoption mechanism. All orphaned child processes are automatically adopted by a special system process—traditionally called init (Process ID 1) or, in modern Linux systems, systemd. This "grandparent" process takes over the role of the original parent and is responsible for reaping the orphan's exit status when it terminates, ensuring it does not become a zombie.

3. Threads: The Unit of Concurrency

The process model discussed earlier describes a program with a single flow of execution, defined by one program counter; this is known as a single-threaded process. Modern operating systems, however, support multithreading—the ability for a single process to manage multiple concurrent paths of execution at the same time.

Conceptually, a process acts as a container for shared resources, such as the executable code, data segments, and open files. The threads, sometimes called lightweight processes, are the individual execution flows that operate within this shared environment.

3.1. Anatomy of a Thread

While threads run within the same process, they have a crucial distinction between what they share and what they own privately.

All threads within a process share the same memory space (code and data segments), open files, and process credentials. However, each thread maintains its own private execution context, which includes its own program counter, register set, and execution stack. This model allows

threads to collaborate closely by reading and writing to the same data structures, while still being able to execute independently.

3.2. Advantages of Multithreading

Compared to using multiple separate processes, a multithreaded approach offers significant advantages. First is responsiveness: if one thread in a process blocks (e.g., waiting for a file to load), other threads can continue to run, which is crucial for keeping applications with graphical user interfaces responsive. Second is resource sharing: threads can communicate and share data effortlessly by simply reading from and writing to the shared memory space, which is much simpler and faster than complex Inter-Process Communication (IPC). Third is efficiency: threads are "lightweight," consuming less memory and being much faster to create than full processes. Context switching between threads of the same process is also faster because the OS does not need to switch the memory address space.

3.3. Processes vs. Threads: A Comparison

While both processes and threads provide a way to achieve concurrency, they are designed for different needs.

- In terms of isolation, processes have strong protection because they have separate memory spaces. Threads, however, have weak isolation because they share memory, meaning one faulty thread can corrupt the data of others.
- Regarding resource overhead, processes are "heavyweight," as each requires its own full set of resources. Threads are "lightweight" because they share most resources from their parent process.
- Creation time also differs significantly. Creating a new process is a slow, resource-intensive operation. Creating a new thread is much faster.
- Finally, communication between processes is complex and slow, requiring explicit IPC mechanisms. Communication between threads is simple and fast, achieved directly through their shared memory.

In short, processes are better for tasks that require strong isolation and security, while threads are ideal for tasks within the same application that need to cooperate and share data closely.

Modern operating systems support two fundamental approaches to concurrent execution, as illustrated in Figure 1.13. The traditional process model (Figure1.13 (a)) uses separate processes, each with its own address space and single thread of control, providing strong isolation but higher resource overhead. In contrast, the multithreaded model (Figure1.13(b)) employs multiple threads within a single process that share the same address space while maintaining private stacks, registers, and program counters (Figure1.13 (c)). On single-CPU systems, both approaches create the illusion of parallel execution through rapid context switching between execution contexts, though threads appear particularly efficient as they avoid expensive address space switches. However, this shared-memory architecture means threads can access each other's stacks and global variables, requiring careful synchronization unlike the inherent protection between processes. The multithreaded approach offers superior

performance for cooperative tasks, with threads being significantly faster to create and communicate than full processes, while processes remain preferable when strong isolation is needed.

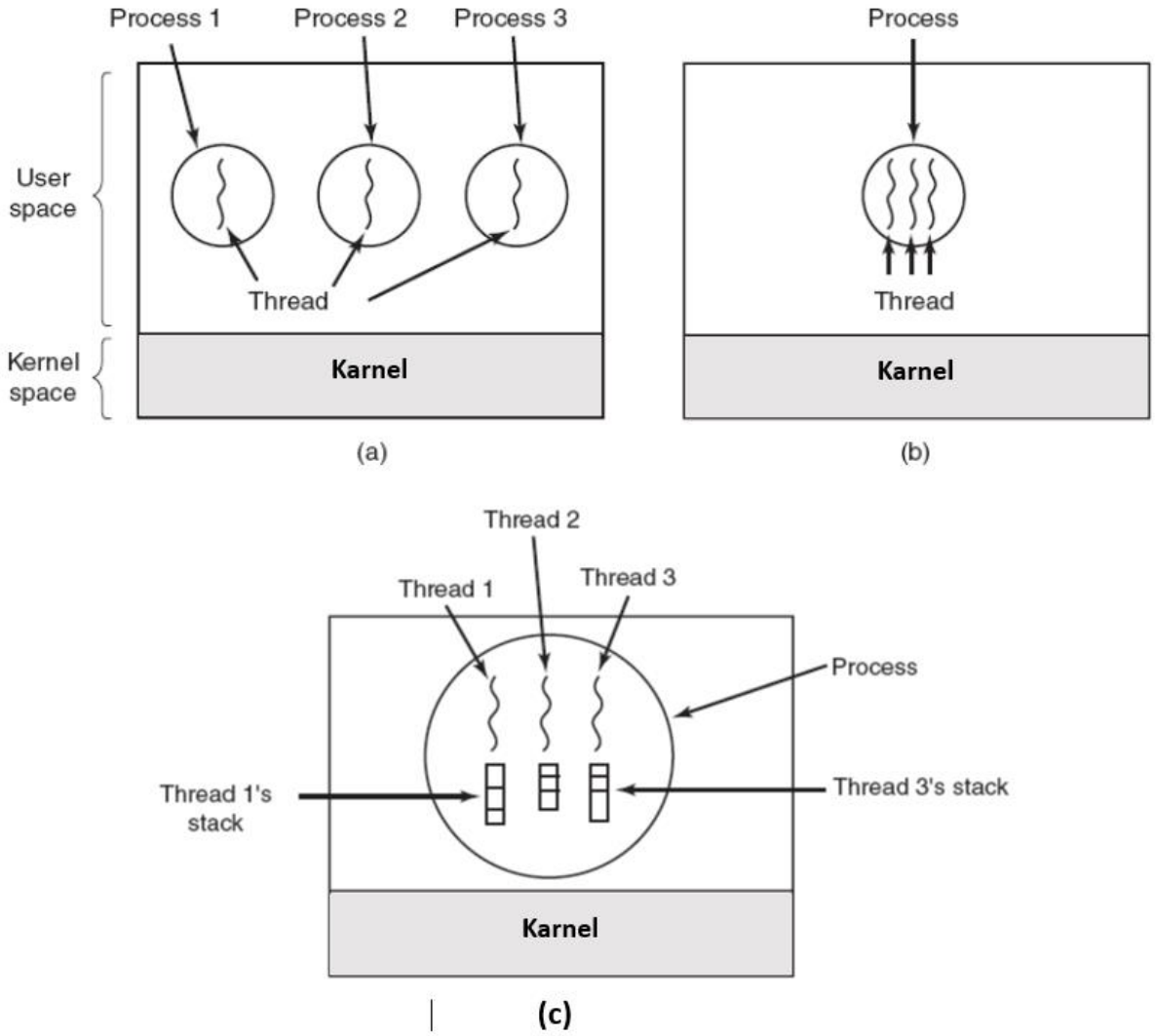


Figure 1.13. Approaches to concurrent execution

3.4.Thread Implementation Models

There are three primary ways to implement threads.

1. User-Level Threads: These threads are managed entirely by a library in user space, and the OS kernel is not aware of them. They are very fast to create and switch between and are highly portable. However, if one thread makes a blocking system call (like reading a file), the entire process blocks. They also cannot take advantage of multiple CPU cores.

2. Kernel-Level Threads: The operating system kernel manages these threads directly. This allows for true parallelism on multi-core systems, and a blocking call in one thread

does not affect the others. However, they are slower to create and switch because it requires a transition to kernel mode and are less portable.

3. Hybrid Threads: This model combines the two, mapping multiple user-level threads to a smaller or equal number of kernel-level threads. It offers the flexibility of user-level threads with the parallelism of kernel-level threads and is the most common approach in modern systems.

3.5.Thread Management: The POSIX pthreads API

The **POSIX (Portable Operating System Interface)** standard defines a thread management API called **pthreads**, widely used in UNIX-like systems (Linux, BSD, macOS). The pthread library (<pthread.h>) provides functions for thread creation (pthread_create), synchronization (mutexes, condition variables), and termination (pthread_join/pthread_exit). Unlike kernel-level threads, pthreads originally operated in **user space**, offering portability and low-overhead context switches. Modern implementations (e.g., Linux's **NPTL**) combine user and kernel threading for parallelism. Essential pthread operations include:

- **`pthread_create`**: Spawns a new thread with specified attributes
- **`pthread_exit`**: Terminates the calling thread while preserving resources for other threads
- **`pthread_join`**: Blocks until a target thread completes (similar to wait() for processes)
- **`pthread_yield`**: Voluntarily relinquishes CPU to other threads (now largely superseded by proper scheduling)
- **`pthread_attr_init/pthread_attr_destroy`**: Manages thread attribute objects for customizing stack size, detach state, etc.

Thread Creation

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg);
```

The pthread_create() function crates a new lightweight thread (LWP) that executes the specified start_routine function with argument arg and thread attributes attr. The attributes structure allows configuration of: Stack size, Scheduling priority, Detach state, Scheduling policy, Other thread characteristics

Thread Suspension/Waiting

```
int pthread_join(pthread_t thid, void **retval);
```

The pthread_join() function blocks the calling thread until the specified thread (identified by thid) terminates.

Example :

```
void *result;                    // Declares pointer to store thread's return value
pthread_join(thread_id, &result);  // Waits for specified thread to complete
printf("Thread returned: %p\n", result); // Prints the returned value
```

Step-by-Step Execution:

- void *result creates a generic pointer to hold the thread's exit status
- The void* type allows threads to return any data type (int, struct, etc.)
- pthread_join(thread_id, &result) pauses the calling thread until the target thread (thread_id) calls pthread_exit()or the target thread returns from its start function
- The second argument (&result) provides the memory address where the return value should be stored.
- When the joined thread terminates its return value (from pthread_exit() or return) is saved in result.

Thread Termination

```
void pthread_exit(void *retval);
```

This function terminates the calling thread and returns retval (a generic pointer) to any thread that joins it via pthread_join().

Example in Practice

The provided sample code (Figure 1.14) demonstrates fundamental thread behavior in Linux through three key aspects: First, it shows thread creation using pthread_create(), where a new execution flow begins running concurrently with the main thread. Second, by printing the same process ID (PID) in both threads via getpid(), it visually confirms that threads share the same process context and memory space - a core characteristic of POSIX threads. Third, the original infinite loops (now replaced with proper synchronization in the improved version) initially demonstrated how threads enable parallel execution within a process, though they lacked proper cleanup. The figure effectively illustrates Linux's threading model where: 1) threads are lightweight compared to processes, 2) they maintain separate execution flows while sharing process resources, and 3) they require explicit management (shown by the addition of

pthread_join in the corrected version). This serves as a foundation for understanding thread operations in Unix-like systems.

```c
#include <unistd.h>      // For getpid(), sleep()
#include <pthread.h>     // For pthread functions
#include <stdio.h>       // For printf()
#include <stdlib.h>      // For exit()

// Thread function that will execute concurrently
void *function(void *arg) {
    printf("Child thread PID = %d\n", (int)getpid());  // Shows shared PID
    printf("Child thread is working...\n");
    sleep(2);  // Simulates 2 seconds of "work"
    printf("Child thread completed its work\n");
    pthread_exit(NULL);  // Proper thread termination
}

int main() {
    pthread_t thread;  // Thread identifier

    printf("Main thread PID = %d\n", (int)getpid());

    // Create thread - error checked
    if (pthread_create(&thread, NULL, function, NULL) != 0) {
        perror("Failed to create thread");
        exit(EXIT_FAILURE);
    }

    printf("Main thread waiting for child to finish...\n");

    // Wait for thread completion - error checked
    if (pthread_join(thread, NULL) != 0) {
        perror("Failed to join thread");
        exit(EXIT_FAILURE);
    }

    printf("Main thread: child thread has terminated\n");
    printf("Program exiting normally\n");

    return 0;
}
```

Sample Output:

```
Main thread PID = 1234

Main thread waiting for child to finish...

Child thread PID = 1234

Child thread is working...

[2 second delay]
```

```
Child thread completed its work

Main thread: child thread has terminated

Program exiting normally
```

Chapter 2

Processes Synchronization

Summary

This chapter explores the problems that arise from concurrent processes accessing shared data and introduces the various synchronization mechanisms required to ensure correct cooperation between them. Specifically, the chapter aims to:

- **Introduce the Challenge of Concurrency**: It explains the issues that occur when multiple processes or threads execute concurrently and need to cooperate by sharing data.
- **Define and Illustrate Race Conditions**: The chapter highlights the core problem of concurrent processing, known as a race condition, where the outcome of an execution depends on the specific order in which instructions are interleaved.
- **Explain the Critical-Section Problem**: It defines the critical-section problem as the challenge of designing a protocol to ensure that when one process is accessing shared data (in its "critical section"), no other process can do the same, a concept known as mutual exclusion.

- **Present Solutions for Synchronization**: The chapter details a wide range of solutions, including algorithmic approaches, hardware-supported solutions, and operating system-level tools like semaphores and monitors.
- **Solve Classic Synchronization Problems**: It applies synchronization tools, particularly semaphores, to solve classic concurrency problems, including the Producer-Consumer problem, the Readers-Writers problem, and the Dining Philosophers problem.

1. Introduction: The Challenge of Concurrency

In a multi-programming environment, it is common for multiple processes or threads to execute **concurrently** (by rapidly switching between them on a single CPU core) or in **parallel** (by running simultaneously on separate cores). While this capability is the foundation of modern computing, it introduces significant challenges, especially when these processes need to cooperate by sharing data.

Independent processes are relatively simple to manage, but cooperating processes that access and manipulate shared data can lead to unexpected and incorrect results if their activities are not carefully coordinated. This chapter explores the problems that arise from concurrent data access and the synchronization mechanisms required to ensure correct cooperation between processes.

The Race Condition

The core problem of concurrent processing arises when multiple processes attempt to access and manipulate the same shared data, and the outcome of the execution depends on the specific order in which their instructions are interleaved. This situation is known as a **race condition**.

An Illustrative Example

Let's consider two processes, P1 and P2, that share a common integer variable named counter, which is initialized to **5**. Process P1 is designed to increment the counter by 2, while Process P2 is designed to decrement it by 1.

The operations are:

- **Process P1**: counter := counter + 2
- **Process P2**: counter := counter - 1

If these processes run sequentially, the final value of counter would be (5 + 2) - 1 = 6 or (5 - 1) + 2 = 6. However, these high-level statements are not **atomic**. At the machine level, they are broken down into a sequence of more basic instructions: loading the variable's value into a register, modifying the register, and writing the new value back to memory.

For **P1 (counter := counter + 2)**:

1. register1 = counter
2. register1 = register1 + 2

3. counter = register1

For **P2 (counter := counter - 1)**:

1. register2 = counter
2. register2 = register2 - 1
3. counter = register2

Now, imagine these two processes run concurrently. The CPU can switch between them at any point, interleaving their low-level instructions. Consider this sequence of events, starting with `counter = 5`:

Time	P1 Execution	P2 Execution	register1	register2	counter
T0	`register1 = counter`		5	-	5
T1	`register1 = register1 + 2`		7	-	5
T2	*--Context Switch--*	`register2 = counter`	7	5	5
T3		`register2 = register2 - 1`	7	4	5
T4		`counter = register2`	7	4	**4**
T5	*--Context Switch--*		7	4	4
T6	`counter = register1`		7	4	**7**

In this scenario, the final value of counter is 7. The expected result was 6, but because P2 overwrote the counter variable before P1 could save its updated value, the work done by P2 was completely lost. If the instructions at T4 and T6 were swapped, the final result would be 4.

This incorrect state occurs because both processes "raced" to update the variable based on an outdated value they had loaded into their private registers. This leads to **data inconsistency** and is a critical bug in concurrent programming.

To prevent race conditions, we need a mechanism to ensure that only one process at a time can access and manipulate the shared counter variable. This requires a form of **process synchronization**. The sections of code that access shared data are known as **critical sections**, and we must protect them to guarantee the integrity of our shared data.

2. The Critical-Section Problem

A Critical Section (CS) is a set of instructions within a program that can lead to unpredictable results when executed simultaneously by different processes. Generally, any part of a program that accesses or modifies shared variables can constitute a critical section.

The core challenge for an operating system, known as the critical-section problem, is to design a protocol that the processes can use to cooperate. When one process is executing in its critical section, no other process is to be allowed to execute in its critical section. This requirement for

exclusive access is called mutual exclusion. It is essential not only for shared data, like a counter variable, but also for non-shareable resources like a printer, where concurrent access would mix the output of multiple processes.

Structure of a Process with a Critical Section

To solve this problem, a process's code is typically divided into four sections (Figure 2.1):

1. Entry Section: The code a process must execute to request permission to enter its critical section.
2. Critical Section: The segment of code where the process accesses shared variables, updates a table, or writes a file. Only one process can be in this section at a time.
3. Exit Section: The code executed after the critical section, which may release the resource or signal to other waiting processes.
4. Remainder Section: The rest of the process's code, where it is not accessing the shared resource.

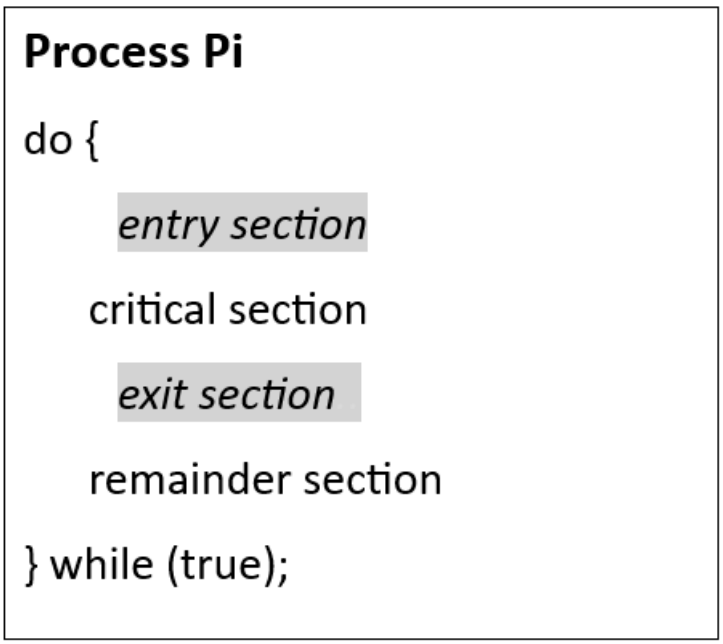


Figure 2.1. Protocol for critical section

Requirements for a Valid Solution

Any solution to the critical-section problem must satisfy three essential requirements:

- **Mutual Exclusion**: If a process is executing in its critical section, then no other processes can be executing in their critical sections. This is the primary goal and prevents race conditions.

- **Progress**: If no process is executing in its critical section and some processes wish to enter, then only those processes that are not in their remainder sections can participate in deciding which will enter next, and this selection cannot be postponed indefinitely.
- **Bounded Waiting**: There must be a limit on the number of times other processes are allowed to enter their critical sections after a process has made a request to enter its own and before that request is granted. This ensures that a waiting process will not wait indefinitely.

Handling Critical Sections in Operating System Kernels

Race conditions are also a major concern within the OS kernel itself, as kernel data structures for managing open files, memory allocation, and process lists are vulnerable.

Operating systems use two general approaches to manage this:

1. Nonpreemptive Kernels: A process running in kernel mode cannot be preempted. This design is largely free from race conditions on kernel data because only one process is active in the kernel at a time.
2. Preemptive Kernels: A process can be preempted even while running in kernel mode. While this can make a system more responsive, it requires careful design to protect shared kernel data from race conditions, especially on multi-processor systems.

3. Algorithmic approach to Critical Section Implementation

Before hardware and operating system-level support became common, the critical-section problem was addressed through purely algorithmic, software-based solutions. These algorithms for two or more processes use shared variables and complex logical checks to enforce mutual exclusion. While many of these are no longer practical on modern computer architectures, they provide an excellent description of the complexities involved in process synchronization.

3.1 Attempts for Two Processes

Solving the critical-section problem, even for just two processes (P0 and P1), is surprisingly difficult. Early attempts highlight the common pitfalls.

Attempt 1: Strict Alternation

A simple approach is to use a single shared integer variable, turn, to track whose turn it is to enter the critical section. For example, if turn == 0, P0 can enter; if turn == 1, P1 can enter.

Process P0	Process P1

do {
while (turn != 0)
{ // do nothing (busy-wait) }
critical section
turn := 1
remainder section
} while (true);
	do {
	while (turn != 1)
	{ // do nothing (busy-wait) }
	critical section
	turn := 0
	remainder section
	} while (true);

Figure 2.2. Strict Alternation Solution

This solution guarantees mutual exclusion. However, it fails the progress requirement. If P0 finishes its critical section and sets turn = 1, it cannot enter its critical section again until P1 has executed and set turn back to 0. If P1 is in its remainder section and has no need to enter its critical section, P0 is blocked by a process that is not even trying to enter. This strict alternation is inefficient if the processes run at different speeds.

Attempt 2: Using State Flags

This approach uses a shared boolean flag for each process to indicate its state. A flag value of true can signify that the process is in its remainder section, while false signifies it wishes to enter or is in its critical section.

A process will wait in a loop as long as the other process's flag is not true.

```
// boolean flag [2] = {false, false};
```

Process P0	Process P1
do {	do {
while (flag[1] == true)	while (flag[0] == true)
{ // do nothing (busy-wait) }	{ // do nothing (busy-wait) }
flag[0] = true;	flag[1] = true;
critical section	critical section
flag[0] = false;	flag[1] = false;
remainder section	remainder section
} while (true);	} while (true);

Figure 2.3. State Flags Solution

This algorithm solves the "progress" issue of the first attempt but introduces a new, more severe problem: deadlock. The flaw occurs if the processes are interleaved in a specific, unfortunate sequence:

1. Process P0 executes its while loop. It sees flag[1] is false, so it exits the loop.
2. P0 is about to execute flag[0] = true, but a context switch occurs.
3. Process P1 now runs. It executes its while loop. It sees flag[0] is false, so it also exits its loop.
4. P1 executes flag[1] = true.
5. Another context switch occurs, and P0 resumes.
6. P0 executes flag[0] = true.

At this point, both flag[0] and flag[1] are true. When P0 tries to enter its critical section again, it will be stuck in the while loop waiting for flag[1] to become false. Likewise, P1 will be stuck waiting for flag[0] to become false. Both processes are waiting for each other, and neither can proceed.

3.2. Peterson's Solution

In 1981, G. L. Peterson developed a classic and remarkably simple software-based solution for two processes that correctly satisfies all three requirements for solving the critical-section

problem. Peterson's solution combines the ideas of a turn variable and state flags. It uses two shared data items:

- int turn: Indicates whose turn it is to enter the critical section.
- boolean flag[2]: An array where flag[i] = true indicates that process Pi is ready to enter its critical section.

Process P0	**Process P1**

do {
flag[0]:= true;
turn := 1;
while (flag[1] and turn == 1)
{ // do nothing (busy-wait) }
critical section
flag[0] := false;
remainder section
} while (true);
	do {
	flag[1]:= true;
	turn := 0;
	while (flag[0] and turn == 0)
	{ // do nothing (busy-wait) }
	critical section
	flag[1] := false;
	remainder section
	} while (true);

Figure 2.4. Peterson's Solution

How Peterson's Solution Works

1. Entry Section: When process Pi wants to enter its critical section, it first sets its own flag (flag[i] = true) to signal its interest. It then generously sets turn = j, giving the other process (Pj) a chance to enter. Pi then enters a while loop, waiting only if the other process is also interested (flag[j] == true) AND it is the other process's turn (turn == j).
2. Guaranteeing Mutual Exclusion: Both processes cannot be in their critical sections at the same time. If both set their flags to true, the turn variable will be set to both i and j almost simultaneously, but only the final assignment counts. If turn == j, then Pi will be stuck in its while loop, allowing Pj to enter. Pj will only enter if flag[i] is false or turn is j. This intricate check ensures that only one process can proceed.
3. Exit Section: When Pi leaves the critical section, it sets flag[i] = false, indicating it is no longer interested. This allows the other process (if it is waiting) to exit its while loop and enter its own critical section.

This algorithm correctly ensures mutual exclusion, progress, and bounded waiting for two processes. However, it is important to note that on modern computer architectures, due to how instructions like load and store are performed, there is no guarantee that software-only solutions like this will work correctly without underlying hardware support.

3.3. An Initial Approach for Multiple Processes: A Lock Variable

A simple, software-based attempt to solve the critical-section problem for multiple processes is to use a single shared variable, often called a lock. The logic is straightforward:

- The lock variable is initialized to false, signifying that the critical section is free.
- When a process wants to enter the critical section, it first tests the lock.
- If lock is false, the process sets it to true (indicating "busy") and enters the critical section.
- If lock is already true, the process must wait until it becomes false again.

The intended code structure for a process would look something like this:

```
// Shared variable: boolean lock = false;

Process Pi
do {
    while (lock == true)
      { // do nothing (busy-wait) }
    lock := true;
    critical section
    lock := false;
   remainder section
} while (true);
```

Figure 2.5. A Lock Variable Solution

Why This Fails: The Race Condition

Unfortunately, this simple solution has a fatal flaw because the actions of testing the lock and setting the lock are not performed as a single, uninterruptible operation. This opens the door for a classic race condition:

1. Assume lock is initially false.
2. Process P0 wants to enter. It executes the while (lock == true) check. It sees that lock is false and proceeds past the loop.
3. Crucially, before P0 can execute lock = true, a context switch occurs, and the CPU is given to Process P1.
4. Process P1 now runs. It also wants to enter its critical section, so it executes while (lock == true). It also sees that lock is still false and proceeds.
5. P1 executes lock = true and enters its critical section.
6. Eventually, a context switch occurs, and P0 resumes execution. Since P0 had already passed its while check, its next instruction is lock = true. It sets the lock (which is already true) and also enters the critical section.

The result is that two processes are in their critical regions at the same time, a clear violation of mutual exclusion. Adding more checks before setting the lock does not solve the problem, as a context switch could always occur between the final check and the act of setting the lock.

3.4. The Bakery Algorithm for Multiple Processes

For situations with more than two processes, a more robust algorithmic solution is needed. The Bakery Algorithm, developed by Leslie Lamport, provides a solution to the critical-section problem for N processes.

The algorithm is inspired by how customers are served in a bakery. Upon entering, each customer takes a numbered ticket. The baker then serves the customer with the lowest ticket number.

If two customers happen to receive the same number (a possibility in the algorithm), they are served in alphabetical order (or in the case of processes, by their pre-determined process ID).

Data Structures and Logic

The algorithm uses two shared global arrays, initialized to false and 0 respectively: boolean Choosing[N] and int Number[N]. Choosing[i] = true means that process i is currently in the process of picking its ticket number. Number[N] stores the ticket number for each process.

The algorithm uses a special comparison rule to decide which process goes next. A process Pi has priority over process Pj if it has a lower ticket number. If their numbers are the same, the one with the lower process ID goes first. This is expressed as (Number[i], i) < (Number[j], j). This condition is true if Number[i] < Number[j], or if Number[i] == Number[j] and i < j.

The Algorithm's Structure

```
Process Pi
do {
    Choosing[i] = true;
    Number[i] = max(Number[0], ..., Number[N-1]) + 1; Choosing[i] = false;
    for (j = 0; j < N; j++) {
         // Wait while process j is choosing its number
         while (Choosing[j] == true) {
                // Do nothing }
        // Wait while process j has a smaller number (or the same
        // number and a smaller process ID)
           while ((Number[j] != 0) and ((Number[j], j) < (Number[i], i))) {
                  // Do nothing  }
        }
 critical Section
Number[i] = 0;
 remainder section
} while (true);
```

Figure 2.6. The Bakery Algorithm

How it Guarantees Mutual Exclusion

The Bakery Algorithm successfully ensures mutual exclusion. If a process Pi is in its critical section, any other process Pk will be blocked in the second while loop. When Pk compares itself to Pi (when j in its for loop equals i), it will find that Number[i] is not zero and that (Number[i], i) is less than (Number[k], k).

Therefore, Pk will be forced to wait in the loop until Pi finishes its critical section and resets Number[i] to 0. This ensures that only one process can be in its critical section at a time.

Limitations of Algorithmic Approaches

Purely algorithmic solutions are complex and become exponentially more difficult to design and verify as the number of processes increases. It is not feasible for application programmers to solve this problem for every application. For this reason, modern operating systems provide hardware and OS-level support for synchronization, which will be discussed in the following sections.

4. Hardware Support for Synchronization

As we've seen, purely software-based solutions to the critical-section problem, like Peterson's Algorithm, are complex and are not guaranteed to work correctly on modern multiprocessor systems. This is because modern CPUs can reorder instructions, making it impossible to rely on the specific sequence of load and store operations.

To overcome these limitations, computer systems provide hardware support for synchronization. These features can make synchronization tasks easier, more efficient, and, most importantly, reliable. All modern solutions to the critical-section problem are based on the premise of locking—protecting critical regions through the use of locks supported by hardware.

4.1. Interrupt Disabling

On a uniprocessor system (a computer with a single CPU core), there is a very direct way to enforce mutual exclusion: disabling interrupts.

```
Process Pi
do {
    disable_interrupts();
    critical section
    enable_interrupts();
   remainder section
} while (true);
```

Figure 2.7. Interrupt Disabling Solution

A process on a single-core CPU will continue to run without interruption until it either invokes an operating system service or a hardware interrupt (like a timer) occurs. If a process disables all interrupts right before entering its critical section and re-enables them immediately upon exiting (figure 2.7), it can guarantee that it will not be preempted. With no context switches possible, the sequence of instructions within the critical section is guaranteed to execute atomically.

While effective, this approach has serious disadvantages:

- Failure on Multiprocessor Systems: This technique is useless on multiprocessor systems. Disabling interrupts only affects the single CPU core that executed the instruction. Other cores can continue running and can access the shared data, leading to race conditions.
- Performance Degradation: The efficiency of the entire system can be noticeably degraded because the processor cannot interleave processes while interrupts are off.
- Dangerous for User Processes: It is unwise to give user-level processes the power to disable interrupts. A malicious or buggy program could disable interrupts and never re-enable them, effectively crashing the entire system.

Because of these limitations, disabling interrupts is typically only used by the operating system kernel itself for very short, critical operations.

4.2. Atomic Hardware Instructions

The standard solution for multiprocessor systems is to use special atomic hardware instructions. An operation is atomic if it is performed as a single, uninterruptible unit. When one of these instructions executes, the hardware guarantees that no other processor can access the same memory location until the instruction is complete.

These instructions allow us to test and modify the content of a memory location in a single step, which is the key to preventing the race conditions we saw in software-only attempts.

The Test-and-Set Instruction

The Test-and-Set instruction (often called TSL) atomically reads the value of a memory location, sets its value to true, and returns the original value.

Its behavior can be defined as:

```
boolean test_and_set(boolean target) {
        boolean original_value := target;
        target := true;
        return original_value; }
```

We can use this to create a simple lock, where a shared boolean lock is initialized to false.

```
Process Pi
do {
    while (test_and_set(lock) == true) {
      // do nothing (busy-wait) }
    critical section
    lock = false;
   remainder section
} while (true);
```

Figure 2.8. TSL based solution

The first process to call test_and_set() will receive false and enter the critical section. Any other process that calls it will receive true (the value set by the first process) and will be stuck in the while loop until the lock is released.

The Compare-and-Swap Instruction

Swap algorithm is a lot like the test_and_set algorithm. Instead of directly setting lock to true in the swap function, key is set to true and then swapped with lock. First process will be executed, and in while(key), since key=true , swap will take place and hence lock=true and key=false. Again next iteration takes place while(key) but key=false , so while loop breaks and first process will enter in critical section. Now another process will try to enter in Critical section, so again key=true and hence while(key) loop will run and swap takes place so, lock=true and key=true (since lock=true in first process). Again on next iteration while(key) is true so this will keep on executing and another process will not be able to enter in critical section. Therefore Mutual exclusion is ensured. Again, out of the critical section, lock is changed to false, so any process finding it gets t enter the critical section. Progress is ensured. However, again bounded waiting is not ensured for the very same reason.

```
void swap(boolean a, boolean b){
   boolean temp = a;
   a = b;
   b = temp;
}
```

```
// Shared variable lock initialized to false
// and individual key initialized to false;

Process Pi
do {
    key := true;
      while(key)
            swap(lock, key);
    critical section
    lock := false;
   remainder section
} while (true);
```

Figure 2.8. swap based solution

Properties and Drawbacks of Hardware Solutions

These machine-instruction approaches have several advantages:

- They are applicable to any number of processes on multiprocessor systems.
- They are simple to understand and verify.
- They can be used to protect multiple different critical sections, each with its own lock variable.

However, these simple implementations share serious disadvantages:

- Busy-Waiting: The process continuously executes a loop while waiting for the lock, consuming CPU time without doing useful work. This is also known as a spinlock.
- Starvation is Possible: When a lock is released, the selection of the next process to acquire it is arbitrary. A process could theoretically be denied access indefinitely.

An Advanced Hardware Solution: Bounded Waiting

The simple hardware locks using Test-and-Set or Compare-and-Swap successfully provide mutual exclusion but fail to guarantee **bounded waiting**. When a lock is released, any waiting process might grab it, meaning a process could theoretically wait forever (starvation).

To solve this, a more sophisticated algorithm can be built using the same atomic instructions combined with additional shared variables to enforce a fair, ordered entry into the critical section.

This solution uses a shared boolean lock (like a simple spinlock) but adds a shared boolean array, waiting[n], where n is the number of processes.

- boolean lock: A global lock, initially false.
- boolean waiting[n]: An array where waiting[i] = true indicates that process Pi is waiting to enter its critical section. Each element is initialized to false.

```
// Shared variable lock initialized to false
// and individual key initialized to false

boolean lock;
Individual key;
Individual waiting[i];

Process Pi

do {

    waiting[i] := true;
    key := true;
     while(waiting[i] and key)
          key := test_and_set(lock);
       waiting[i] := false;

   critical section

   j := (i+1) % n;
  while(j != i and waiting[j] == false)
     j := (j+1) % n;
     if(j == i)
        lock := false;
     else
     waiting[j] := false;

  remainder section

} while (true);
```

Figure 2.9. An Advanced Hardware Solution

The algorithm works as follows (Figure 2.9):

Entering the Critical Section A process Pi that wants to enter its critical section first signals its intent by setting waiting[i] = true. It then enters a spinlock loop, repeatedly calling test_and_set on the global lock variable until it acquires the lock. Once it has the lock, it clears its own waiting[i] flag to false and enters the critical section.

Exiting the Critical Section This is where the algorithm's cleverness lies. Instead of just setting lock = false for any process to grab, the exiting process intelligently hands off control.

- It searches for the next process in line that is waiting. It does this by checking the waiting array in a circular order (starting from j = (i+1) % n).
- If another process is found waiting (e.g., waiting[j] is true), the exiting process "unlocks the door" specifically for that next process by setting waiting[j] = false. This allows process Pj to break out of its spinlock loop and enter the critical section. The global lock remains true, already claimed by Pj.

- If no other process is waiting, the exiting process simply sets the global lock = false, making the critical section available for the next process that arrives.

This approach ensures that processes are served in a circular, first-come-first-served order, thus satisfying the bounded-waiting requirement and preventing starvation.

5. Operating System Support

The hardware-based solutions we've discussed, while functional, have a major drawback: they rely on **busy-waiting**. A process stuck in a spinlock loop consumes CPU cycles without doing any productive work. To overcome this inefficiency, operating systems provide more sophisticated synchronization tools.

5.1. Semaphores

The first major advance in this area was the **semaphore**, a concept introduced by Edsger Dijkstra in 1965. A semaphore is a powerful tool that not only provides mutual exclusion but also allows for more complex synchronization scenarios.

What is a Semaphore?

At its core, a semaphore is an integer variable, let's call it S, that is accessed only through two standard, **atomic** operations:

- **wait(S)**: This operation decrements the semaphore's value. If the value becomes negative, the process calling wait() is blocked. This was originally called P() (from the Dutch *proberen*, "to test").
- **signal(S)**: This operation increments the semaphore's value. If there are any processes blocked on this semaphore, one of them is unblocked. This was originally called V() (from the Dutch *verhogen*, "to increment").

All modifications to the integer value of the semaphore in the wait() and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

Think of a semaphore as a guard controlling access to a resource. A process must ask the guard for permission (wait) before entering its critical section. When it leaves, it tells the guard that it's done (signal), allowing another process to enter.

Semaphore Implementation

The key to a semaphore's efficiency is how it handles a process that must wait. Instead of letting the process spin in a loop (busy-waiting), the semaphore implementation can block the process.

- Blocking a Process: When a process calls wait(S) and finds S is not positive, the OS moves the process from the Running state to the Waiting state and places it in a waiting queue associated with that semaphore. The CPU scheduler is then free to run another process.

- Waking Up a Process: When another process calls signal(S), the OS checks the semaphore's waiting queue. If there are processes waiting, it moves one of them from the Waiting state to the Ready state. The awakened process can then be scheduled to run.

To support this, a semaphore is implemented as a structure containing both its integer value and a list (or queue) of waiting processes (Figure 2.10).

```
Struct semaphore {
    int value;
    Liste_of_process list;
};
wait(semaphore S) {
    S.value := S.value - 1;
    if (S.value < 0) {
        add this process to S.list;
        block();} }
signal(semaphore S) {
    S.value := S.value  + 1;
    if (S.value <= 0) {
        remove a process P from S.list;
        wakeup(P); } }
```

Figure 2.11 Semaphore Implementation

NB/ In this implementation, if the semaphore's value is negative, its magnitude represents the number of processes currently waiting in the queue.

Types of Semaphores

Operating systems typically distinguish between two types of semaphores:

1. **Counting Semaphore**: A counting semaphore can have any non-negative integer value. It is used to control access to a pool of a finite number of resources.

- The semaphore is initialized to the number of available resources (e.g., s = 5 for 5 available resources).
- A process calls wait(s) to request a resource, decrementing the count.
- When the process is done, it calls signal(s) to release the resource, incrementing the count.

  When the semaphore's value reaches 0, it means all resources are in use. Any process that calls wait(s) will block until another process releases a resource by calling signal(s).

2. **Binary Semaphore**: A binary semaphore can only have a value of 0 or 1. It functions essentially as a mutex lock and is an excellent tool for providing simple mutual exclusion for a single resource.

The General Semaphore Solution for Critical Section

Regardless of whether you are using a binary or a counting semaphore, the fundamental pattern for solving the critical-section problem is the same. The semaphore acts as a gatekeeper, and each process must follow a simple protocol: **wait before entering, signal after leaving**.

To protect a critical section, a semaphore s is used. Each cooperating process must wrap its critical section with wait() and signal() calls as follows (Figure 2.12):

```
// Semaphore 's' is initialized to a specific value
Process Pi
do {
    wait(s); // "Request" to enter or "Acquire" the lock
    critical section
    signal(s); // "Release" the lock or resource
     remainder section
} while (true);
```

Figure 2.12. Semaphore Solution for Critical Section

How it Works:

1. **The wait(s) Call (Entering)**: When a process wants to enter its critical section, it first performs a wait() operation. This is like asking for permission to proceed. The wait() call decrements the semaphore's value. If the resulting value is positive or zero, the process is allowed to enter. If the value becomes negative, it means the critical section is currently occupied (or all resources are in use), and the process is **blocked** and placed in a waiting queue.

2. **The signal(s) Call (Exiting)**: When a process leaves its critical section, it performs a signal() operation. This is like telling the system that it is finished. The signal() call increments the semaphore's value. If there are any processes blocked in the waiting queue, the OS wakes one of them up, allowing it to finally enter the critical section.

Applying the Solution: Binary vs. Counting

This single algorithm works for both types of semaphores. The only difference is the **initial value** of the semaphore and what that value **represents**.

- **For Mutual Exclusion (Binary Semaphore)**: If you need to ensure only **one process** can enter the critical section at a time, you initialize the semaphore to **1**. This '1' represents a single lock that is either available (s=1) or taken (s=0).
- **For a Resource Pool (Counting Semaphore)**: If you have a pool of **N** identical resources and need to allow up to **N** processes to access them simultaneously, you initialize the semaphore to **N**. The value of the semaphore represents the number of available resources.

By understanding this single, unified pattern, we can solve a wide variety of synchronization problems simply by choosing the correct initial value for the semaphore.

5.2. Monitors (High-Level Synchronization)

While semaphores are effective, they are prone to programming errors that can lead to severe issues. For example, a programmer might accidentally interchange wait() and signal() operations, omit one, or even replace a signal() with a wait(). Such errors can violate mutual exclusion or cause deadlocks that are difficult to detect and reproduce.

To address these dangers, high-level language constructs like the monitor were developed. A monitor provides a structured and less error-prone way to manage synchronization.

What is a Monitor?

A monitor is a software module with a crucial property: **Only one process can be active within the monitor at any given time**.

This rule is automatically enforced by the compiler, which means the programmer doesn't have to manually code for mutual exclusion. By turning critical regions into monitor procedures, the risk of race conditions is significantly reduced.

The key characteristics of a monitor are:

- Its local data variables are private and can only be accessed by its own procedures.
- A process enters the monitor by calling one of its procedures.
- The monitor construct ensures that only one process is executing inside it at a time. Any other process that tries to call a monitor procedure will be blocked until the monitor is free.

Monitor Structure and Usage

A monitor is an Abstract Data Type (ADT) that encapsulates shared variables and a set of programmer-defined operations (functions or procedures) that can act on those variables. This structure ensures that the shared data is protected from unstructured access.

The syntax of a monitor includes (Figure 2.13):

1. Shared variable declarations.
2. A set of functions or procedures that operate on those variables.
3. An initialization code block.

```
Type <monitor_name> = Monitor
// Shared variable declarations
Procedure entry P1(...) Begin
   End;
Procedure entry P2(...) Begin
   End;
....
Procedure entry Pn(...) Begin
   End;
Begin
... //Initialization_code
End.
```

Figure 2.13. Syntax of a monitor.

The representation of a monitor type cannot be used directly by the various processes. Thus, a function defined within a monitor can access only those variables declared locally within the monitor and its formal parameters. Similarly, the local variables of a monitor can be accessed

by only the local functions. The monitor construct ensures that only one process at a time is active within the monitor.

To handle situations where a process needs to wait for a certain condition to be met, monitors use condition variables (Figure 2.14). A programmer can declare one or more variables of type condition within the monitor.

condition x, y;

These variables support only two operations:

- x.wait(): The process that invokes this operation is suspended until another process invokes x.signal(). When a process is suspended, it releases the monitor, allowing another process to enter.
- x.signal(): This operation resumes exactly one of the processes that was suspended by an x.wait() call on the same condition variable.

A critical difference from semaphores is that if x.signal() is called when no process is suspended on condition x, the signal has no effect and is lost forever. The state of x is the same as if the operation had never been executed. This contrasts with a semaphore's signal() operation, which always affects the state of the semaphore.

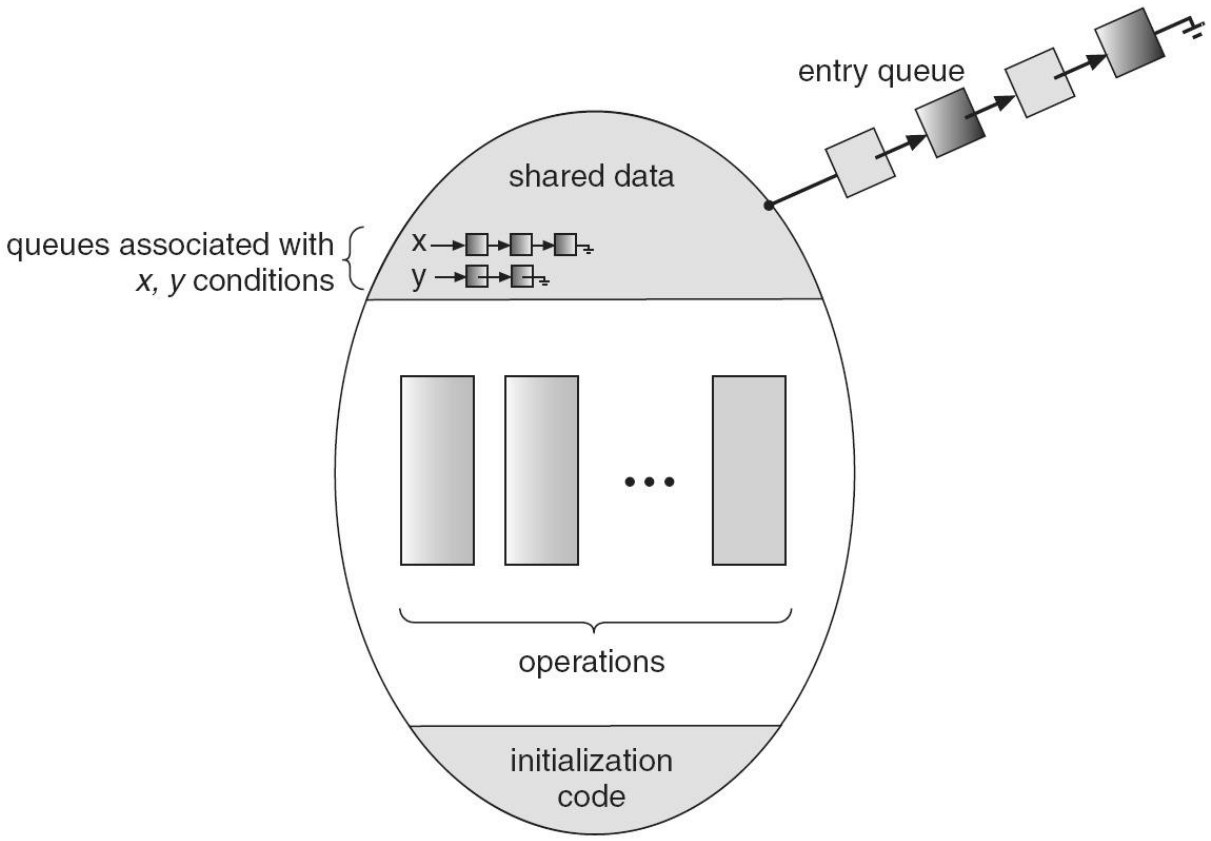


Figure 2.14. Structure of a Monitor

Solving the Critical-Section Problem with a Monitor

A monitor provides a clean and robust solution for controlling access to a single, shared resource among multiple concurrent processes, thereby solving the critical-section problem.

Let's design a monitor called CS_Control to manage this.

Monitor Design and Components

The monitor will contain:

- A boolean variable, occupied, to track if the critical section is currently in use.
- A condition variable, csAvailable, for processes to wait on when the CS is busy.

The monitor will be initialized with occupied = false.

Pseudocode for the Monitor

```
Type CS_Control  = Monitor
// Shared variable declarations
boolean occupied;
condition csAvailable;
Procedure entry AcquireAccess ()
  Begin
  if (occupied == true) { // If resource is busy, wait on the condition
       csAvailable.wait(); }
   occupied := true;
  End;
Procedure entry ReleaseAccess()
  Begin
  // Mark the CS as free
  occupied := false;
  // Wake up one waiting process, if any
 csAvailable.signal();
  End;
Begin
  // --- Initialization ---
 // This code runs once when the monitor is created
  occupied := false;
End.
```

Figure 2.15. Structure of Section Critic Monitor

How a Process Uses the Monitor

Each process that needs to access the shared data will use the monitor's procedures to safely enter and exit its critical section.

```
Process Pi
do {
    // Call the entry procedure to request access
    CS_Control.AcquireAccess();
    critical section
    // Call the exit procedure to release access
    CS_Control.ReleaseAccess();
    remainder section
} while (true);
```

Figure 2.16. Structure of a process Pi using monitor

How It Guarantees Mutual Exclusion

1. **First Process**: The first process to call AcquireAccess() finds occupied is false. It sets occupied to true and immediately enters its critical section.
2. **Second Process**: If a second process calls AcquireAccess() while the first is still in its critical section, it will find occupied is true. It then calls csAvailable.wait(), which puts it to sleep and releases the monitor's lock.
3. **Exiting**: When the first process finishes, it calls ReleaseAccess(). This sets occupied back to false and calls csAvailable.signal(), which wakes up the sleeping second process. The awakened process can then claim the critical section.

Because the monitor guarantees that only one process can be executing its procedures (AcquireAccess or ReleaseAccess) at any given time, a race condition between checking occupied and setting it is impossible.

5.3. Critical Regions (High-Level Synchronization)

Another high-level language construct designed to simplify synchronization and avoid the errors common with semaphores is the **critical region**. This feature provides a structured and more automated way to enforce mutual exclusion.

What is a Critical Region?

A **critical region** is a programming language feature that explicitly associates a block of code with a specific shared variable. The language's compiler is then able to enforce mutual exclusion for that variable automatically.

The basic syntax involves declaring a shared variable and then using a region keyword to define the critical section where that variable can be accessed (Figure 2.17). This construct ensures that only one process can be executing inside the region S block at any given time.

```
// Declare a shared variable 'S'
shared <type> S = <initial_value>;
// In a process's code
region S do {
 // --- Critical Section ---
 // Access and modify the shared variable S here.
        }
```

Figure 2.17. Critical region

Enhancement: The Conditional Critical Region

The basic critical region construct can lead to inefficient busy-waiting if a process needs to wait for a specific condition before proceeding. To solve this, an enhanced version called the **conditional critical region (CCR)** was developed (Figure 1.18).

A CCR adds an await statement, which allows a process to block until a certain boolean condition is met.

```
// Declare a shared variable 'S'
shared <type> S = <initial_value>;
// In a process's code
region S do {
 await(boolean_condition);
 // --- Critical Section ---
 // Access and modify the shared variable S here.
        }
```

Figure 1.18. Conditional critical region

How it Works:

1. When a process enters the region, it evaluates the await condition.
2. If the condition is **true**, the process proceeds into its critical section.
3. If the condition is **false**, the process is **blocked** and temporarily releases its exclusive access to the region.
4. Whenever a process exits the region, the conditions of all waiting processes are re-evaluated.

Solving the Critical-Section Problem with CCR

A conditional critical region provides a very clear and direct solution for ensuring mutual exclusion. We can use a shared boolean variable to track if the critical section is occupied.

```
// Shared variable to control access
shared boolean occupied = false;
Process Pi
do {
  region occupied do {
    // Wait until the critical section is NOT occupied
    await(!occupied);
    // Claim the critical section
    occupied = true;
    Critical Section --- // (Mutual exclusion is guaranteed here)
    // Release the critical section upon exiting the region
    occupied = false;
    }
  remainder section
} while (true);
```

Figure 2.19. Critical-Section Problem with CCR

This solution is highly readable and robust. The await(!occupied) statement explicitly defines the condition for entry, and the region construct guarantees that the check and the subsequent modification of the occupied variable happen without the risk of a race condition.

5.4. Path Expressions (High-Level Synchronization)

Another high-level construct designed to provide a structured approach to synchronization is the **Path Expression**. Like Critical Regions and Monitors, its goal is to make concurrent programming safer and more understandable by embedding synchronization rules directly into the definition of a shared object.

What is a Path Expression?

A **Path Expression** is a synchronization mechanism that uses a syntax similar to regular expressions to define the legal sequence of operations that can be performed on a shared object. This expression is part of the object's definition and is automatically enforced by the system.

The core idea is to separate the synchronization logic (the "path") from the implementation of the operations themselves. The programmer defines the rules of interaction once, and the system ensures they are never violated.

The syntax includes:

- **Sequencing (,)**: Enforces that one operation must follow another.
- **Choice (|)**: Allows one of several operations to be chosen.
- **Repetition (*)**: Allows an operation or group of operations to be performed zero or more times.

Solving the Critical-Section Problem

A Path Expression can provide a very clear solution for ensuring mutual exclusion. We can define a path that forces processes to execute an enter procedure before their critical section and an exit procedure afterward, allowing only one process through at a time.

// Synchronization rule for a shared object

path enter, exit end

How it Works:

1. **Initial State**: The path requires the enter operation to be executed first. The system will only allow one process to successfully complete this operation. Any other process that calls enter will be automatically blocked.
2. **Entering the Critical Section**: After a process successfully calls enter, the path advances. That specific process can now execute its critical section code.
3. **Exiting**: The process must then call the exit procedure. Once it does, the path is completed and resets to the beginning, allowing another blocked process (if any) to successfully complete its enter call.

The actual implementation of the critical section code is separate from this rule. A process would interact with this synchronized object as follows:

```
Process Pi
do {
    SharedObject.enter();
    Critical Section
   SharedObject.exit();
    Remainder Section
} while (true);
```

Figure 2.20. Critical-Section Problem with Path Expression

6. Solution of Classic Problems of Synchronization using Semaphores

There are some classic synchronization problems in computer science. Programmers will face these synchronization problems during development of any application. The semaphore is the solution for all of them.

6.1. The Producer-Consumer (or Bounded-Buffer) Problem

The Producer-Consumer problem describes a scenario where two types of processes share a common, fixed-size buffer (Figure 2.21):

- The **producer**'s job is to generate data (an "item") and put it into the buffer.
- The **consumer**'s job is to take an item out of the buffer and process it.

This pattern appears everywhere. A web server *produces* web pages that a web browser *consumes*. A compiler *produces* object code that an assembler *consumes*.

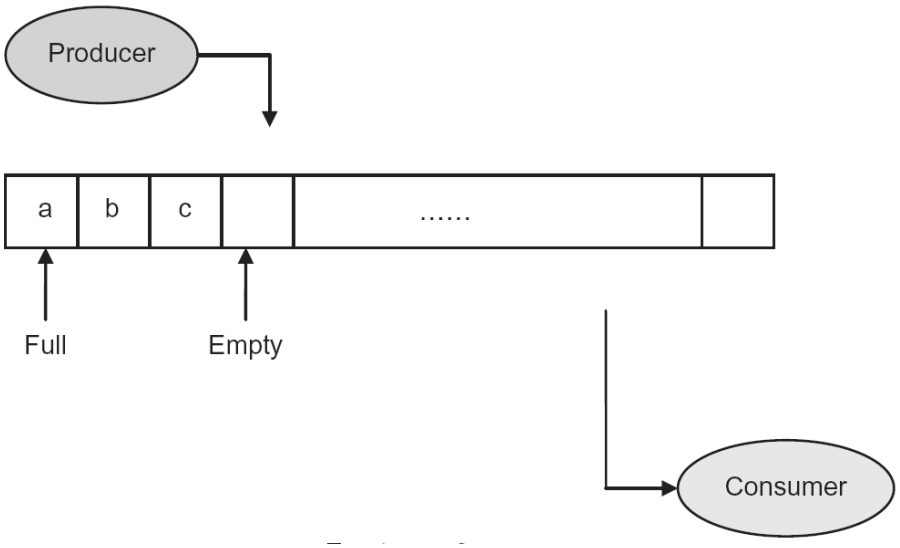


Figure 2.21 Producer–consumer problem's

The Synchronization Challenges

We need to design a system that meets the following requirements:

1. The producer must not try to add an item to the buffer if it is full. If the buffer is full, the producer must block until a slot becomes available.
2. The consumer must not try to remove an item from the buffer if it is empty. If the buffer is empty, the consumer must block until an item is produced.
3. The producer and consumer must not access the buffer at the same time. This is a mutual exclusion requirement to prevent race conditions.

A simple approach using a shared count variable will fail. A "lost wakeup" problem can easily occur, where one process sends a wakeup signal to the other before it has gone to sleep, causing both processes to sleep forever. Semaphores are designed to solve this perfectly.

The Semaphore Solution

The classic solution uses three semaphores to coordinate the producer and consumer (Figure 2.22):

- **mutex**: A semaphore used to enforce mutual exclusion when accessing the buffer. It is initialized to **1**.
- **full**: A semaphore used to count the number of full slots in the buffer. It is initialized to **0**. The consumer will wait on this semaphore.
- **empty**: A semaphore used to count the number of empty slots in the buffer. It is initialized to **N** (the size of the buffer). The producer will wait on this semaphore.

The Producer	The Consumer

do {
// Produce an item
produce_item();
// Wait for an empty slot to become available
wait(empty);
// Acquire exclusive access to the buffer
wait(mutex);
// Add the item to the buffer
add_item_to_buffer();
// Release exclusive access
signal(mutex);
// Signal that another slot is now full
signal(full);
} while (true);
	do {
	// Wait for a full slot to become available
	wait(full);
	// Acquire exclusive access to the buffer
	wait(mutex);
	// Remove an item from the buffer
	remove_item_from_buffer();
	// Release exclusive access
	signal(mutex);
	// Signal that another slot is now empty
	signal(empty);
	// Consume the item
	consume_item();
	} while (true);

Figure 2.21 Producer–consumer problem's solution with semaphores

The producer first waits for an empty slot (wait(empty)). If empty is zero, it blocks. Once a slot is available, it acquires the mutex lock, adds its item, releases the lock, and then signals that the buffer is one item fuller (signal(full)).

The consumer's logic is the mirror image. It first waits for a full slot (wait(full)). If full is zero, it blocks. Once an item is available, it acquires the mutex lock, removes the item, releases the lock, and finally signals that the buffer is one item emptier (signal(empty)).

6.2. The Readers-Writers Problem

The Readers-Writers problem models a situation where a shared data area, such as a database or a file, is accessed by multiple processes. The processes are categorized into two types:

- **Readers**: Processes that only read the shared data. They do not perform any updates.
- **Writers**: Processes that update (read and write) the shared data.

To ensure data integrity, we must enforce a set of rules:

1. Any number of readers may access the shared data simultaneously.
2. Only one writer can access the shared data at a time.
3. If a writer is accessing the data, no reader may access it.

A simple mutual exclusion lock on the entire data area is too restrictive, as it would unnecessarily prevent multiple readers from accessing the data at the same time. The challenge is to create a synchronization scheme that follows these rules.

The Semaphore Solution

To solve the first readers-writers problem, we use two semaphores and an integer counter:

- **rw_mutex**: A semaphore that provides mutual exclusion for the writers. It is also used by the *first* reader who arrives and the *last* reader who leaves. It is initialized to **1**.
- **mutex**: A semaphore that protects the read_count variable, ensuring that only one reader at a time can modify the counter. It is initialized to **1**.
- **read_count**: An integer that keeps track of how many readers are currently accessing the data. It is initialized to **0**.

As illustrated in Figure 2.22, the writer's logic is very simple. It just needs to acquire the rw_mutex lock to gain exclusive access. While the reader's logic is more complex because it has to manage the read_count to control the writer's lock.

The Writer

```
do {

    wait(rw_mutex);

    // Writing is performed here
    signal(rw_mutex);

    } while (true);
```

The Reader

```
do {

    wait(mutex);  // Lock to protect read_count
    read_count++;
    if (read_count == 1) {
    wait(rw_mutex); // First reader in locks out writers }

    signal(mutex); // Unlock read_count

    // Reading is performed here

    wait(mutex); // Lock to protect read_count
    read_count--;
    if (read_count == 0) {
    signal(rw_mutex); // Last reader out allows writers in }

     signal(mutex); // Unlock read_count

    } while (true);
```

Figure 2.22. A solution to the readers and writers problem.

How it Works:

- The First Reader: When the first reader arrives (read_count == 1), it is responsible for locking out any writers by calling wait(rw_mutex).
- Subsequent Readers: As other readers arrive, they simply increment read_count but do not need to call wait(rw_mutex) because it is already held by the first reader.
- The Last Reader: When the last reader leaves (read_count == 0), it is responsible for signaling rw_mutex, which allows a waiting writer (if any) to finally access the data.

- The mutex semaphore simply ensures that the read_count variable itself is updated atomically by the readers.

6.3. The Dining Philosophers Problem

The Dining Philosophers problem is a classic synchronization puzzle that illustrates the challenges of allocating limited resources among competing processes in a way that avoids **deadlock** and **starvation.**

Imagine five philosophers sitting around a circular table (Figure 2.23). Their life consists of two alternating activities: **thinking** and **eating**.

- In the center of the table is a large bowl of rice.
- Between each pair of philosophers is a single chopstick. In total, there are five chopsticks.
- To eat, a philosopher needs **two chopsticks**: the one on their left and the one on their right.
- A philosopher can only pick up one chopstick at a time.

The problem is to design a protocol (an algorithm) that allows the philosophers to eat and think without getting stuck in a state where no one can make progress.

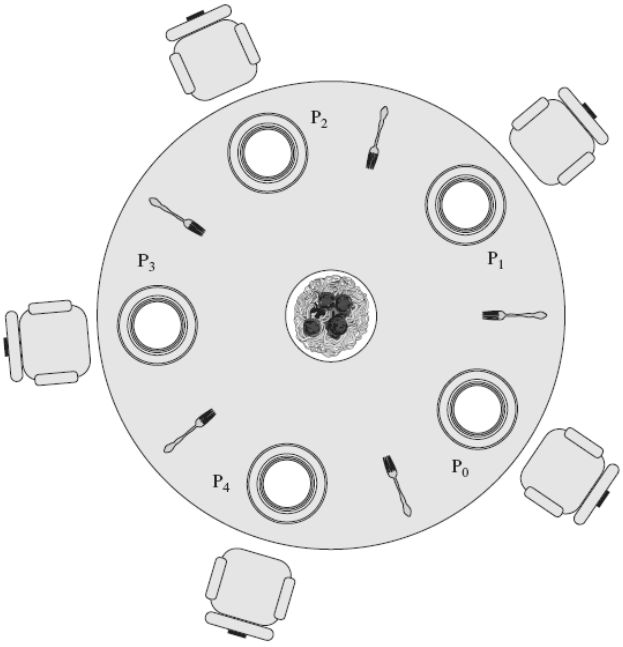


Figure 2.23. Dining Arrangement for Philosophers

The most straightforward solution is to represent each chopstick as a semaphore. Each semaphore is initialized to **1**. A philosopher must perform a wait() on a chopstick to pick it up and a signal() to put it down.

```
Semaphore chopstick[5] = {1, 1, 1, 1, 1};

The Philosopher i

do {

    // Pick up left chopstick
    wait(chopstick[i]);

    // Pick up right chopstick
    wait(chopstick[(i + 1) % 5]);

    // --- Critical Section: Eat ---

    // Put down right chopstick
    signal(chopstick[(i + 1) % 5]);

    // Put down left chopstick
    signal(chopstick[i]);

    // --- Remainder Section: Think ---}
    while (true);
```

Figure 2.24. The structure of philosopher *i*.

This solution can lead to a deadlock. Imagine all five philosophers get hungry at the exact same time. Each philosopher successfully executes wait(chopstick[i]) and picks up their **left** chopstick.

At this point, all chopsticks are held, and all semaphore values are 0. Now, every philosopher tries to execute wait() on their **right** chopstick, but that chopstick is held by their neighbor. Every philosopher will be blocked, waiting for a chopstick that will never be released. This is a classic deadlock.

NB/ A detailed discussion of deadlocks and various strategies to prevent, avoid, and resolve them will be covered in Chapter 4.

6.4 POSIX Semaphore Services

The POSIX standard, which is implemented by Linux and other UNIX-like systems, provides a standard library for creating and managing semaphores. To use these services, the <semaphore.h> header file must be included in the C code.

The POSIX API provides several functions for managing the lifecycle of a semaphore.

Initialization Before a semaphore can be used, it must be initialized. This is done with the sem_init() function.

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- sem: A pointer to the sem_t variable to be initialized.
- pshared: A flag to indicate the level of sharing. A value of 0 means the semaphore is shared between threads of the same process. A non-zero value means it can be shared between different processes.
- value: The initial integer value of the semaphore. For a mutex, this would be 1.

Wait and Signal Operations The core semaphore operations are named sem_wait() and sem_post().

- int sem_wait(sem_t *sem);: This function performs the wait (or P) operation. It decrements the semaphore's value. If the value is zero, the calling thread will block until another thread calls sem_post().
- int sem_post(sem_t *sem);: This function performs the signal (or V) operation. It increments the semaphore's value. If any threads are blocked waiting on the semaphore, one of them will be unblocked.

Destruction When a semaphore is no longer needed, it is important to destroy it to free up system resources. This is done with sem_destroy().

```
int sem_destroy(sem_t *sem);
```

Example: Protecting a Critical Section in C

This example shows how to use these functions to protect a global variable that is modified by two different threads. This is a classic critical-section problem.

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

sem_t mutex; // The shared semaphore
int global_var = 0; // The shared variable

void* thread_function(void* arg) {
    // Wait for permission to enter
    sem_wait(&mutex);

    // --- Critical Section ---
    printf("Thread entered...\n");
    global_var++; // Modify the shared variable
    printf("Global variable is now: %d\n", global_var);
    printf("Thread exiting...\n");

    // Release the lock
    sem_post(&mutex);
    return NULL;
}
```

```c
int main() {
    pthread_t t1, t2;

    // Initialize the semaphore to 1 (acting as a mutex)
    sem_init(&mutex, 0, 1);

    // Create two threads that will run the same function
    pthread_create(&t1, NULL, thread_function, NULL);
    pthread_create(&t2, NULL, thread_function, NULL);

    // Wait for both threads to complete
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    // Clean up the semaphore
    sem_destroy(&mutex);

    printf("Final value of global variable: %d\n", global_var);
    return 0;
}
```

How it Works

1. In main, the semaphore mutex is initialized to a value of 1.
2. Two threads, t1 and t2, are created, both set to run thread_function.
3. The first thread to run will call sem_wait(&mutex), succeed (decrementing the value to 0), and enter its critical section.
4. If the second thread tries to call sem_wait() while the first is inside, it will block, because the semaphore's value is 0.
5. Only after the first thread calls sem_post(&mutex) will the semaphore's value be incremented back to 1, allowing the second, waiting thread to proceed.

This ensures that the global_var++ operation is performed atomically, and the final value will be correct.

Chapter 3

Interprocess Communication

Summary

The purpose of this chapter is to introduce the fundamental mechanisms that allow separate, concurrently running processes to cooperate by exchanging data and information.

By default, processes operate in isolated memory spaces. This chapter explains the essential tools and protocols, known as Interprocess Communication (IPC), that enable them to bridge this isolation. Ultimately, the goal is to provide a foundational understanding of the different IPC methods, their trade-offs, and how they are used in practical programming scenarios on UNIX-like systems.

1. Introduction

Processes executing concurrently within an operating system can be categorized into two types: **independent processes** and **cooperating processes**. An independent process is one that cannot affect or be affected by other processes running in the system; this is the case for any process that does not share data with others. Conversely, a cooperating process is one that can influence or be influenced by other processes, typically by sharing data. Processes often cooperate to solve a common problem and can run in parallel on a single computer or on different machines.

There are several compelling reasons to create an environment that facilitates process cooperation:

- **Information Sharing**: It allows multiple users or processes to concurrently access the same piece of information, like a shared file.
- **Computation Speedup**: A task can be divided into subtasks that run in parallel on multiple processing cores, making the overall task run faster.
- **Modularity**: System functions can be built in a modular fashion, separating them into distinct processes or threads.
- **Convenience**: A single user may perform multiple tasks simultaneously, such as editing a document, listening to music, and compiling code.

For processes to cooperate, they require a mechanism for **interprocess communication (IPC)** to exchange data and information. This chapter introduces the fundamental mechanisms that enable this cooperation. IPC provides a set of tools and protocols that allow processes to bridge their isolated worlds and work together effectively.

We will explore several key IPC mechanisms provided by UNIX/Linux systems, starting with the most basic and moving to more structured methods:

- Shared Memory: We will begin by examining the most direct method, where processes share common variables or files, and discuss the primary challenge of concurrent access that this creates.
- Signals: Next, we will cover signals, a mechanism for sending simple, asynchronous notifications to alert processes of important events.
- Pipes: We will then dive into pipes, a powerful message-passing facility for streaming data between processes. We will cover both unnamed pipes for related processes and named pipes (FIFOs) for unrelated ones.
- Sockets: Finally, we will briefly introduce sockets as the mechanism used for communication between processes on different machines across a network.

By the end of this chapter, an understanding of the principles behind these different IPC methods.

2. Communication via shared memory

The most direct way for processes to communicate is to share a common space for data. This shared memory approach is a powerful and efficient Interprocess Communication (IPC) mechanism where cooperating processes exchange information by reading from and writing to a shared region of memory.

Shared memory communication can take two primary forms:

- **Common Variables:** Processes share a region of main memory (RAM) where common variables or data segments are stored. This is very fast as it avoids the overhead of involving the operating system kernel for each data exchange.
- **Common Files**: Processes share a file stored on a disk. One process writes data to the file, and another reads it. This is slower than using RAM but is useful for sharing large amounts of data or for communication between processes that may not be running at the same time.

While shared memory is efficient, it introduces a significant challenge when multiple processes attempt to access the common data space at the same time.

If two or more processes read or write shared data, the final result depends on the specific scheduling order of the processes. This situation is known as a race condition. As we saw in detail in Chapter 2, this can lead to data inconsistency because the high-level operations (like incrementing a variable) are not atomic. To prevent such problems, concurrent access must be controlled using synchronization techniques.

Shared memory is the underlying mechanism for many classic synchronization challenges, several of which were introduced in Chapter 2.

- The Producer-Consumer Problem, for instance, is a common paradigm for cooperating processes that use shared memory. In this problem, a producer process creates data that is consumed by a consumer process. They communicate via a shared buffer which resides in a region of memory accessible to both.
- Similarly, the Readers-Writers Problem addresses how to manage concurrent access to a shared database or file. It allows multiple "readers" to access the data simultaneously while ensuring a "writer" has exclusive access to make updates.

In all such cases, a region of memory is shared between processes, and access to it must be carefully controlled with synchronization primitives to prevent the race conditions we've discussed.

3. Communication via Signals

While shared memory is excellent for transferring data, operating systems provide a different mechanism for simple notifications, error handling, and minimal communication: signals.

3.1 What Are Signals?

A signal is a form of software interrupt used by the operating system to notify a process that an important event has occurred. They are a lightweight and fundamental Interprocess Communication (IPC) mechanism in UNIX and Linux systems.

Signals can be generated for various reasons:

- Hardware Errors: Events like a division by zero or an invalid memory access are detected by the hardware and converted by the OS into a signal sent to the offending process.
- External Events: A user pressing Ctrl-C in the terminal sends a SIGINT (interrupt) signal to the foreground process.
- Process-to-Process Communication: One process can explicitly send a signal to another process to notify it of an event.

The key advantage of signals is that they are asynchronous. A process can react to an event without having to constantly check for it.

3.2 Signal Handling

When a process receives a signal, it can choose one of three ways to respond:

1. Ignore the signal: The process simply discards the signal and continues its execution.
2. Use the default action: The process can let the operating system perform the default action for that signal. For most signals, the default action is to terminate the process. In some cases, this also creates a core file, a snapshot of the process's memory that can be used for debugging.
3. Provide a custom handler: A process can specify a custom function, called a signal handler, to be executed when the signal is received. When the signal arrives, the process's normal execution is paused, the handler function is executed, and afterward, the process resumes from where it was interrupted.

Some signals, like SIGKILL (terminate immediately) and SIGSTOP (stop execution), are special and cannot be ignored or have a custom handler.

Each signal is identified by a positive integer. The table below describes some of the most important ones.

Signal Name	Number	Default Action	Description
SIGKILL	9	Terminate	The "kill" signal. Cannot be caught or ignored.
SIGSTOP	19	Stop Process	Pauses the process. Cannot be caught or ignored.
SIGCONT	18	Continue	Resumes a stopped process.
SIGTERM	15	Terminate	The standard termination signal (can be caught).
SIGINT	2	Terminate	Interrupt signal, usually sent by Ctrl-C.
SIGUSR1	10	Terminate	User-defined signal 1, for custom communication.
SIGUSR2	12	Terminate	User-defined signal 2, for custom communication.

3.3 POSIX Signal Services

The POSIX standard defines the C library <signal.h> for managing signals.

Sending Signals with kill() A process can send a signal to another process using the kill() system call. The sending process must either have the same owner as the receiving process or be the super-user (root).

```
int kill(pid_t pid, int signal);
```

- pid: The Process ID of the target process.
- signal: The integer number of the signal to send.

Waiting for Signals with pause() and sleep()

Instead of wasting CPU cycles in an empty loop (a "busy-wait"), a process can use more efficient system calls to wait for a signal:

- int pause(void);: This call suspends the process until any signal is received. The process is blocked and does not consume CPU time.
- unsigned int sleep(unsigned int seconds);: This call suspends the process for a specified number of seconds.

3.4 C Code Examples

Example 1: Catching the Termination Signal

This program shows how a process can intercept a "kill" request (SIGTERM) and execute a custom function instead of dying.

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

void sighandler(int signum);

int main(void) {
    if (signal(SIGTERM, sighandler) == SIG_ERR) {
        printf("Cannot handle the signal\n");
        exit(1);
    }
    while (1) {
        char buffer[256];
        fgets(buffer, sizeof(buffer), stdin);
        printf("Input: %s", buffer);
    }
    return 0;
}

void sighandler(int signum) {
    printf("Intercepting the SIGTERM signal\n");
}
```

Figure 3.1. C Code : Catching the Termination Signal

The main function registers sighandler as the handler for the SIGTERM signal. The program then enters an infinite loop. From another terminal, sending kill <PID> sends a SIGTERM signal. Instead of terminating, the program catches it and prints a message. The process only dies when sent an uncatchable signal like SIGKILL.

Example 2: Parent-Child Signal Exchange

This example shows a parent and child process communicating using custom signals SIGUSR1 and SIGUSR2.

```cpp
#include <signal.h>
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>

static void action(int sig);

int main() {
    int pid, etat;
    signal(SIGUSR1, action);
    signal(SIGUSR2, action);

    if ((pid = fork()) == 0) { // Child Process
        kill(getppid(), SIGUSR1);
        for (;;)
            pause();
    } else { // Parent Process
        kill(pid, SIGUSR2);
        std::cout << "Parent: terminating the child" << std::endl;
        kill(pid, SIGTERM);
        wait(&etat);
        std::cout << "Parent: child terminated" << std::endl;
    }
    return 0;
}

static void action(int sig) {
    switch (sig) {
        case SIGUSR1:
            std::cout << "Parent: received signal SIGUSR1" << std::endl;
            break;
        case SIGUSR2:
            std::cout << "Child: received signal SIGUSR2" << std::endl;
            break;
    }
}
```

Figure 3.2. C Code: Parent-Child Signal Exchange

The child process sends SIGUSR1 to its parent, and the parent sends SIGUSR2 to its child. Because of unpredictable scheduling, the order of events can vary, leading to different outputs. This shows that signals are not reliable for enforcing a strict sequence of events without other synchronization.

Example 3: Efficient Waiting with pause()

This example contrasts an inefficient "busy-wait" with an efficient pause() call to wait for 5 SIGINT signals.

Inefficient Code: while (num_signal < 5);

Efficient Code: while (num_signal < 5) pause();

The first version consumes 100% CPU in a loop. The second version uses pause() to suspend the process, consuming no CPU time until a signal arrives. This is the correct and efficient way to wait.

Example 4: Ignoring Signals

This program shows a child process ignoring SIGINT and SIGQUIT, but being terminated by SIGKILL.

```c
#include <signal.h>
#include <stdlib.h>

void trait_sigint() {
    printf("Received SIGINT, but I will ignore it\n");
}

void trait_sigquit() {
    printf("Received SIGQUIT, but I will ignore it\n")
}

int main() {
    int pid;
    pid = fork();

    if (pid == 0) { // Child process
        // Associate SIGINT with the trait_sigint hand
        signal(SIGINT, trait_sigint);
        // Associate SIGQUIT with the trait_sigquit ha
        signal(SIGQUIT, trait_sigquit);

        printf("Child: I am alive\n");
        sleep(5);
        printf("Child: first awakening\n");
        sleep(5);
        printf("Child: second awakening\n");

    } else { // Parent process
        sleep(1);
        printf("Parent: sending SIGINT\n");
        kill(pid, SIGINT);
        sleep(2);
        printf("Parent: sending SIGQUIT\n");
        kill(pid, SIGQUIT);
        sleep(5);
        printf("Parent: sending SIGKILL\n");
        kill(pid, SIGKILL);
    }
    return 0;
}
```

Figure 3.3. C Code: Ignoring Signals

Execution Flow:

– The parent creates a child process.

- The child sets up custom handlers for SIGINT and SIGQUIT that simply print a message.
- The parent waits a bit and then sends SIGINT. The child catches it, prints its message, and continues running.
- The parent then sends SIGQUIT. The child catches it, prints its message, and continues running.
- Finally, the parent sends SIGKILL. This signal cannot be caught or ignored. The child process is terminated immediately by the OS.

Example 5: Synchronized "Ping-Pong" with Signals

This program shows a more complex, synchronized communication where a parent and child process send signals back and forth a set number of times.

```c
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>

void handlerSignal(int sig);
void sendSignal(int pid, int sig);

int Proc1, cptr, limite;

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("format: %s <number>\n", argv[0]);
        exit(1);
    }

    limite = atoi(argv[1]);
    cptr = 0;

    Proc1 = fork();

    switch (Proc1) {
        case 0: // Child Process
            signal(SIGUSR2, handlerSignal);
            printf("Child => handler installed, PID=%d\n", getpid());
            pause();
            while (cptr < limite - 1) {
                cptr++;
                sendSignal(getppid(), SIGUSR1);
                pause();
            }
            cptr++;
            sendSignal(getppid(), SIGUSR1);
            exit(0);
            break;
```

```
            default: // Parent Process
                signal(SIGUSR1, handlerSignal);
                printf("Parent => handler installed, PID=%d\n", getpid());
                cptr++;
                sendSignal(Proc1, SIGUSR2);
                pause();
                while (cptr < limite) {
                    cptr++;
                    sendSignal(Proc1, SIGUSR2);
                    pause();
                }
                break;
        }
        wait(NULL); // Parent waits for child to finish
        return 0;
    }

    void handlerSignal(int sig) {
        printf("\t\t[%d] handler %d => signal caught\n", getpid(), sig);
        fflush(stdout);
    }

    void sendSignal(int pid, int sig) {
        sleep(1);
        if (kill(pid, sig) == -1) {
            printf("ERROR kill PID=%d\n", pid);
            fflush(stdout);
            exit(1);
        }
        printf("#%d [%d] signal %d sent to %d\n", cptr, getpid(), sig, pid);
    }
```

Figure 3.4. C Code: Synchronized "Ping-Pong" with Signals

Execution Flow:

- The parent and child each set up a handler for a specific user signal (SIGUSR1 for parent, SIGUSR2 for child).
- The parent "serves" by sending the first signal (SIGUSR2) to the child and then immediately calls pause(), blocking itself to wait for a response.
- The child, which was also blocked in pause(), wakes up, handles the signal, sends a "return" signal (SIGUSR1) back to the parent, and then calls pause() again.
- This back-and-forth continues until the specified limit is reached. The sleep(1) call is added to slow down the exchange, making the turn-by-turn output easier to observe.

Limitations of Signals

While useful, signals have several important limitations that students should be aware of:

- **They are resource-intensive**: Delivering a signal is not free. It involves a system call, which interrupts the receiving process, modifies its execution stack to run the handler, and then restores the stack afterward. This creates significant overhead.
- **They are limited in number**: The number of available signals is small (around 30 on many systems), and only two of these (SIGUSR1 and SIGUSR2) are specifically reserved for programmer use.
- **They only carry notifications**: Signals are strictly for event notification. They do not carry any data or information with them, not even the identity of the process that sent the signal.

4. Communication via messages

An alternative to the shared-memory model is message passing. This facility, provided by the operating system, allows processes to communicate and synchronize their actions without sharing the same address space.

This is a crucial concept because it's not limited to a single computer. Message passing is particularly useful in a distributed environment, where processes may reside on different machines connected by a network. Think of an internet chat program—you and your friends are separate processes on different computers, exchanging information by sending and receiving messages.

4.1 Principles of Message Passing

A message-passing facility provides at least two fundamental operations:

- send(message)
- receive(message)

The messages themselves can be of a fixed or variable size. Fixed-size messages are simpler for the OS to manage, but variable-size messages are often easier for the programmer to use.

For two processes, P and Q, to communicate, a communication link must exist between them. We are not concerned with the physical implementation of this link (like a hardware bus or network) but with its logical properties. Let's explore the key design choices for these logical links.

There are three main aspects to consider when designing a message-passing system:

1. Naming: How do processes refer to each other?
2. Synchronization: Do the sender or receiver block when exchanging a message?
3. Buffering: How many messages can be stored on the link?

A. Naming: Direct vs. Indirect Communication

Processes need a way to identify the sender or recipient of a message.

Direct Communication In this model, each process must explicitly name the other. The primitives are defined as:

- send(P, message): Send a message to a specific process P.
- receive(Q, message): Receive a message from a specific process Q.

This approach automatically establishes a link between exactly two processes. However, it has a significant disadvantage: if you change the name or ID of a process, you have to find and update that name in every other process that communicates with it, which is not very modular.

Indirect Communication A more flexible approach is indirect communication, where messages are sent to and received from mailboxes (also called ports). A mailbox is an object into which processes can place messages and from which messages can be removed.

- send(A, message): Send a message to mailbox A.
- receive(A, message): Receive a message from mailbox A.

The key advantage here is that processes can communicate without knowing each other's identity; they only need to share a common mailbox. This decouples the processes and makes the system more modular. A link can be associated with more than two processes, who can take turns receiving messages from the mailbox.

B. Synchronization: Blocking vs. Nonblocking

The send() and receive() operations can be either blocking (synchronous) or nonblocking (asynchronous).

- Blocking Send: The sending process is blocked (suspended) until the message is received by the recipient or the mailbox.
- Nonblocking Send: The sending process sends the message and continues its execution immediately.
- Blocking Receive: The receiving process is blocked until a message is available.
- Nonblocking Receive: The receiver retrieves a message if one is available; otherwise, it retrieves a null value and continues execution.

When both send() and receive() are blocking, it's called a rendezvous. This provides strong synchronization, as the sender knows for sure that the message has been received before it continues.

C. Buffering

Messages exchanged between processes reside in a temporary queue. The capacity of this queue is a key design choice.

- Zero Capacity: The queue has no space. The sender must block until the receiver is ready to accept the message. This enforces a rendezvous.

- Bounded Capacity: The queue has a finite size (e.g., 10 messages). The sender can send messages without blocking as long as the queue is not full. If the queue is full, the sender must block until space becomes available.
- Unbounded Capacity: The queue has a potentially infinite length. The sender never blocks.

These different options allow system designers to build IPC mechanisms tailored to specific needs, balancing flexibility, performance, and synchronization requirements.

4.2. UNIX Interprocess Communication Mechanisms

UNIX and Linux offer several powerful mechanisms for processes to communicate by sending messages. The most traditional of these are **pipes**.

4.2.2 Communication via Pipes

A **pipe** is a communication channel that allows two or more processes to exchange information. It acts as a conduit where one process can write a stream of data, and another process can read it from the other end. Data is handled in a **First-In, First-Out (FIFO)** manner.

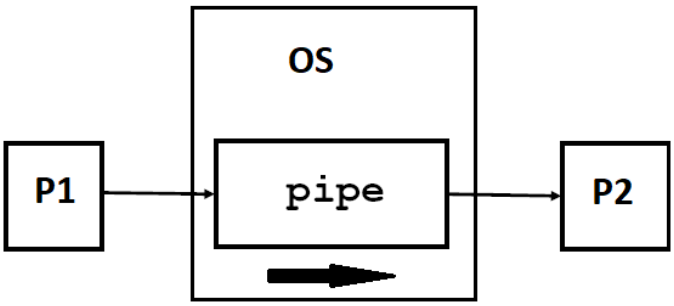


Figure 3.5. Communication via Pipe

UNIX provides two types of pipes:

- **Unnamed Pipes**: For communication between related processes (e.g., parent and child).
- **Named Pipes (FIFOs)**: For communication between any two processes on the same machine.

Unnamed Pipes

An **unnamed pipe** is a **unidirectional** communication link with a limited capacity (often around 4 KB). Data flows in only one direction. An unnamed pipe is created in memory and does not exist as a file on the disk.

Creating Unnamed Pipes in the Shell

The most common way to use an unnamed pipe is in the command-line shell with the | (pipe) operator. This operator connects the **standard output** of the command on its left to the **standard input** of the command on its right.

Example: who | wc -l

1. **who**: This command lists all users currently logged into the system, writing one line per user to its standard output.
2. |: The shell creates an unnamed pipe.
3. **wc -l**: This command counts the number of lines it receives on its standard input.

In this example, the output of who is sent directly into the pipe. The wc -l process reads from the pipe and counts the lines, effectively telling you how many users are logged in. The OS automatically handles synchronization: if the pipe gets full, who is paused; if the pipe gets empty, wc -l is paused.

Creating Unnamed Pipes Programmatically with pipe()

An unnamed pipe can be created within a C program using the pipe() system call.

int pipe(int descriptor[2]);

This function takes an array of two integers. If successful, it creates a pipe and places two **file descriptors** into the array:

- descriptor[0]: The file descriptor for the **read** end of the pipe.
- descriptor[1]: The file descriptor for the **write** end of the pipe.

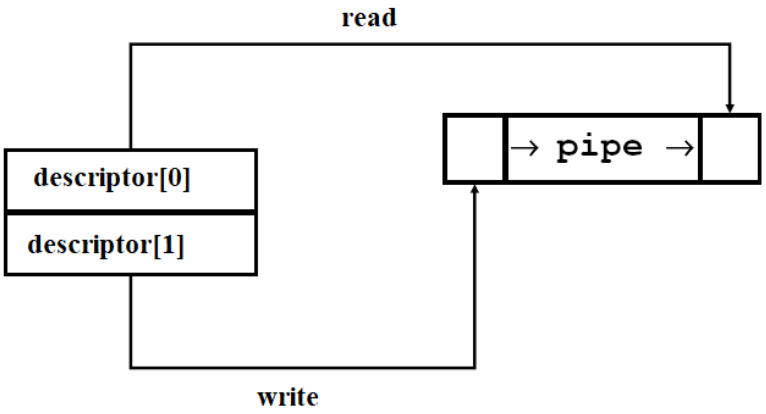


Figure 3.6. Read and Write via Pipe

Access to the pipe happens through these descriptors, just like with ordinary files. Several key points about this call:

- If the system cannot create the pipe (e.g., due to lack of space), the pipe() call returns **-1**. Otherwise, it returns **0**.
- Only the process that created the pipe and its descendants (children created via fork()) can access the pipe. This is because the child process's file descriptor table is a duplicate of the parent's at the time of the fork().
- Because of this inheritance model, unnamed pipes are generally used for communication between a parent and its child processes, with one process acting as the writer and the other as the reader.

Parent-Child Communication with Unnamed Pipes

The typical workflow for communication between a parent and child is as follows (Figure:

1. The parent process creates a pipe using pipe().
2. The parent creates a child process using fork().
3. The writing process closes the unused read end of the pipe.
4. The reading process closes the unused write end of the pipe.
5. The processes communicate using the standard write() and read() system calls on their respective file descriptors.
6. When communication is finished, each process closes its end of the pipe.

Code Example:

This program shows a child process sending a message to its parent through an unnamed pipe.

When compiled and run, the output is: Read 35 bytes: message sent to the parent by the child

The child process writes the string into its end of the pipe (fd[W]), and the parent reads it from its end (fd[R]). The + 1 in strlen(phrase) + 1 is important as it includes the null terminator \0 in the message, ensuring it's treated as a proper C string by the parent.

The child process included the null character in the message sent to the parent process. The parent can easily remove it. If the writer process sends multiple variable-length messages on the pipe, it is necessary to establish rules that allow the reader process to determine the end of a message—in other words, to establish a communication protocol. For example, the writer process can precede each message with its length or terminate each message with a special character like a carriage return or the null character.

```c
#include <unistd.h>
#include <stdio.h>
#include <string.h>

#define R 0 // Index for the read end
#define W 1 // Index for the write end

int main() {
    int fd[2];
    char message[100]; // Buffer to receive the message
    int nboctets;
    char *phrase = "message sent to the parent by the child";

    // Create an unnamed pipe
    pipe(fd);

    // Create a child process
    if (fork() == 0) { // Child Process (Writer)
        // The child closes the unused read descriptor
        close(fd[R]);

        // Write the message into the pipe
        write(fd[W], phrase, strlen(phrase) + 1);

        // Close the write descriptor
        close(fd[W]);

    } else { // Parent Process (Reader)
        // The parent closes the unused write descriptor
        close(fd[W]);

        // Read the message from the pipe
        nboctets = read(fd[R], message, 100);
        printf("Read %d bytes: %s\n", nboctets, message);

        // Close the read descriptor
        close(fd[R]);
    }
    return 0;
}
```

Figure 3.7. A child process sending a message to its parent

Figure 3.8 illustrates how a shell executes a command like ls | wc. First, the pipe is created with the pipe() system call. Then, a child process is created. This child process will inherit the input and output access to the pipe. Next, the parent process closes its access to the pipe's output and redirects STDOUT to the pipe's input. At the same time, the child process closes its access to the pipe's input and redirects STDIN to the pipe's output. Finally, both processes are replaced by the ls and wc programs.

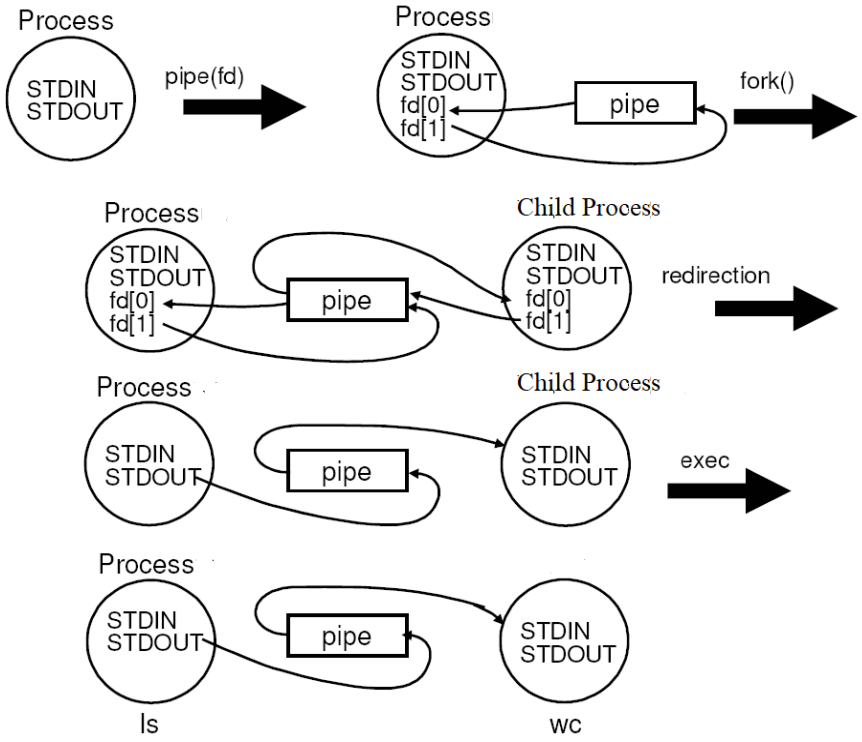


Figure 3.8. Using file descriptors during unidirectional pipe communication between two processes.

Bidirectional Communication

To achieve two-way communication, two separate pipes must be created: one for the parent-to-child flow and another for the child-to-parent flow.

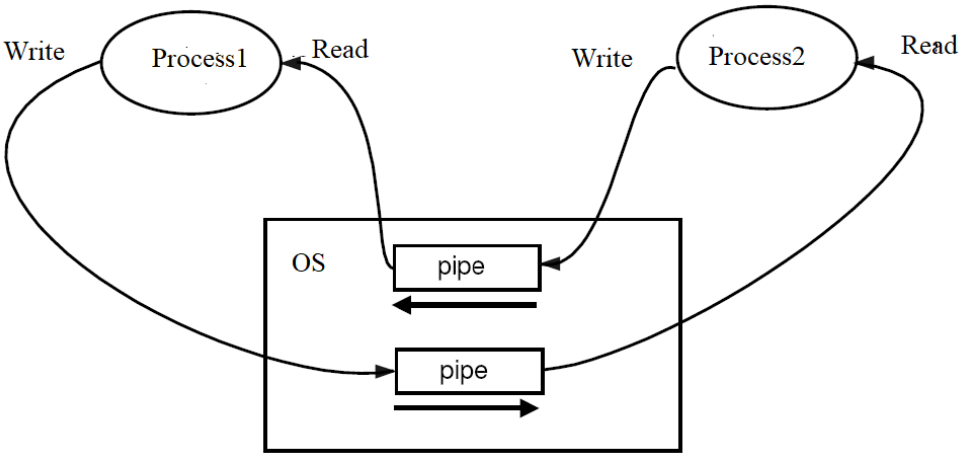


Figure. 3.9. Bidirectional communication via pipes

Named Pipes (FIFOs)

Unnamed pipes are limited to related processes. **Named pipes** (also called **FIFOs**) are a more powerful version that overcomes this.

Named pipes are superior to unnamed pipes for several reasons:

- They have a **name** and exist as a special file in the filesystem.
- They can be used by **any two processes** on the same machine, even if they are unrelated.
- They persist until **explicitly deleted**.
- They often have a larger capacity (e.g., around 40 KB).

Creating Named Pipes with mkfifo()

A named pipe can be created from the command line with mkfifo or programmatically with the mkfifo() system call:

int mkfifo(char *name, mode_t mode);

- name: The filename for the pipe.
- mode: The file permissions (e.g., 0666 for read/write access).

Once created, processes can open() this special file for reading or writing, just like a regular file.

Code Example: writer.c and reader.c

This example uses two separate programs to communicate via a named pipe called mypipe.

```c
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>

int main(void) {
    int fd;
    char message[100];

    sprintf(message, "hello from writer [%d]\n", getpid());

    // Open the named pipe 'mypipe' in write-only mode
    fd = open("mypipe", O_WRONLY);
    printf("here is writer [%d]\n", getpid());

    if (fd != -1) {
        write(fd, message, strlen(message));
    } else {
        printf("sorry, the pipe is not available\n");
    }

    close(fd);
    return 0;
}
```

Figure 3.10. C Code Example: writer.c

```c
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>

int main() {
    int fd, n;
    char input;

    // Open the named pipe 'mypipe' in read-only mode
    fd = open("mypipe", O_RDONLY);
    printf("here is reader [%d]\n", getpid());

    if (fd != -1) {
        printf("Received by the reader:\n");
        while ((n = read(fd, &input, 1)) > 0) {
            printf("%c", input);
        }
        printf("The reader is finishing!\n");
    } else {
        printf("sorry, the pipe is not available\n");
    }

    close(fd);
    return 0;
}
```

Figure 3.11. C Code Example: reader.c

Execution and Explanation

1. First, create the pipe in the terminal: mkfifo mypipe
2. Compile the programs: gcc -o writer writer.c and gcc -o reader reader.c
3. Run them simultaneously in the background: ./writer & ./reader &

The writer process opens mypipe and sends a message. The reader process opens the same pipe, reads the message character by character, and prints it. This demonstrates communication between two completely independent processes.

4.2.3 Sockets: Communication Across Machines

The Interprocess Communication (IPC) mechanisms we've discussed so far, like pipes, are powerful but share a fundamental limitation: they only work between processes running on the **same machine**. To communicate between processes on different computers connected by a network, the UNIX/Linux system provides a mechanism called **sockets**.

What is a Socket?

A **socket** is an endpoint for sending or receiving data across a computer network. Think of it like a telephone. For two people to talk, each needs their own phone, and one must "call" the other's specific number. Similarly, for two processes to communicate over a network, each needs a socket, and they connect using network addresses (an IP address and a port number).

Sockets provide a standard way for processes to establish a communication link, regardless of where they are located on the network.

Why Sockets are Essential

Sockets are the foundation of virtually all modern network communication. They enable processes on different machines to exchange information, making them the traditional and essential IPC mechanism for distributed applications.

Common examples of utilities and services built using sockets include:

- **Remote Login**: Utilities like rlogin or ssh allow a user on one machine to connect to and control another machine.
- **File Transfer**: Applications like FTP (File Transfer Protocol) use sockets to transfer a file from one machine to another.
- **Web Browsing**: When you visit a website, your web browser (a process) opens a socket to communicate with the web server (another process) to request and receive the webpage data.
- **Remote Printing**: Sockets are used to send a file to a printer located on a different machine on the network.

Chapter 4

Deadlocks

Summary

This chapter aim to provide a comprehensive exploration of deadlocks, one of the most critical problems that can arise in concurrent systems where multiple processes compete for finite resources. It explains what a deadlock is, the precise conditions under which one can occur, and the various strategies for managing them.

1. Introduction: The Problem of Permanent Standoff

In any multiprogramming system, processes must compete for a finite number of resources, such as printers, scanners, database records, or internal system tables. An operating system provides mechanisms to grant a process exclusive access to a resource. However, a problem arises when a process needs exclusive access not just to one resource, but to several.

This can lead to a situation where a waiting process is never able to change its state because the resources it has requested are held by other waiting processes. This permanent standoff is called a deadlock.

Imagine two processes, P1 and P2, that both need to use a scanner (R1) and a Blu-ray recorder (R2) to complete a task (Figure 3.1).

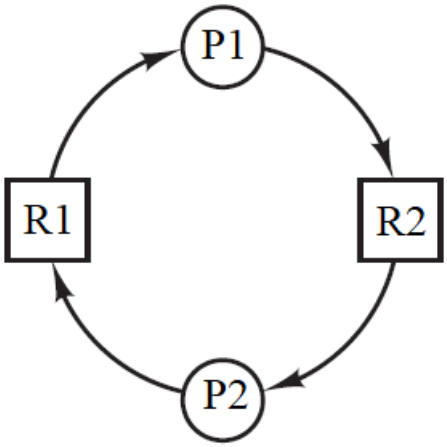


Figure 3.1. Deadlock Situation

1. Process P1 requests and is granted the scanner (R1).
2. Process P2 requests and is granted the Blu-ray recorder (R2).
3. Now, Process P1 requests the Blu-ray recorder (R2), but it's held by P2, so P1 is blocked.
4. Instead of releasing the recorder, Process P2 requests the scanner (R1), but it's held by P1, so P2 is also blocked.

At this point, both processes are blocked, each waiting for a resource held by the other. Neither can proceed, and they will remain in this state forever. This is a deadlock. This issue isn't limited to hardware; it can easily happen with software resources, like when two processes in a database system each lock a record and then try to acquire the other's lock.

Deadlocks are a critical problem in concurrent systems. Operating systems do not typically provide automatic deadlock-prevention facilities, so it is the responsibility of programmers and system designers to write deadlock-free code. In this chapter, we will explore deadlocks in detail. We will cover:

- The necessary conditions that must hold for a deadlock to occur.

- Deadlock Prevention methods, which ensure that at least one of these conditions cannot hold.
- Deadlock Avoidance algorithms, which use advance information to ensure the system never enters an unsafe state.
- Deadlock Detection and Recovery methods for systems that cannot prevent or avoid deadlocks.

2. Processes and Resources

To discuss deadlocks in a general way, we define a resource as anything that must be acquired, used, and then released by a process over the course of its execution. A resource can be a physical device or a piece of information.

Resources are partitioned into types or classes. Each type may have one or more identical instances.

- **Hardware Resources**: Examples include CPU cycles, memory, printers, scanners, and Blu-ray drives. If a system has three identical printers, the resource type Printer has three instances.
- **Software Resources**: Examples include files, database records, and synchronization tools like mutex locks and semaphores.

Under normal operation, a process must utilize a resource in the following three-step sequence:

Request → Use → Release

1. Request: The process requests the resource. If the resource is not immediately available (because it's held by another process), the requesting process must wait. The OS typically blocks the process and places it in a waiting queue for that resource.
2. Use: Once the resource is granted, the process can operate on it. For example, it can print to a printer or write to a file.
3. Release: When the process is finished with the resource, it releases it, making it available for another waiting process.

These steps are often implemented as system calls, like open()/close() for files or wait()/signal() for semaphores.

A critical distinction for understanding deadlocks is the difference between preemptable and non-preemptable resources.

- **Preemptable Resource**: A preemptable resource is one that can be taken away from the process currently holding it without causing any harm. The most common example is **memory**. The OS can safely take memory from one process (by swapping it to disk), give it to another, and later restore it to the original process. Deadlocks involving preemptable resources can often be resolved by simply reallocating the resource.

- **Non-preemptable Resource**: A non-preemptable resource is one that cannot be taken away from its current owner without causing a failure. For example, if a process has started writing to a printer or burning a Blu-ray disc, you cannot take that device away and give it to another process without corrupting the output.

Deadlocks almost always involve non-preemptable resources. Our study of deadlocks will therefore focus on these types of resources.

3. Deadlock Characterization

To deal with deadlocks, we first need to understand precisely what they are and the conditions that cause them.

3.1 Deadlock Situation Definition

A deadlock is a situation in which a set of processes are blocked because each process is holding a resource and waiting for another resource that is held by another process in the same set.

Formally, we can define it as follows: A set of processes is in a deadlocked state when every process in the set is waiting for an event that can only be caused by another process in the set.

Because all the processes are in a waiting state, none of them can run to cause the event that would unblock another process in the set. This results in all processes waiting forever in a circular chain of dependencies.

The most common type of event that processes wait for is the release of a resource, leading to what is called a resource deadlock. These resources can be physical (like printers or DVD drives) or logical (like files, semaphores, or mutex locks).

Examples of a Deadlocked State

- Same Resource Type: Imagine a system with three CD-RW drives, and three processes (P1, P2, P3) each hold one drive. If P1 then requests a second drive (held by P2 or P3), P2 requests a second drive, and P3 requests a second drive, all three will be blocked. Each is waiting for a CD-RW drive that can only be released by one of the other waiting processes.
- Different Resource Types: Consider a system with one printer and one DVD drive.
    1. Process Pi holds the DVD drive and requests the printer.
    2. Process Pj holds the printer and requests the DVD drive. A deadlock occurs because each process holds what the other needs, and neither can proceed.

3.2 Conditions for Resource Deadlocks

A resource deadlock can arise if, and only if, four specific conditions hold simultaneously in a system. If even one of these conditions is not met, a deadlock is not possible. These are known as the Coffman conditions.

1. Mutual Exclusion Condition

- What it is: At least one resource must be held in a non-shareable mode. This means that only one process can use the resource at any given time. If another process requests that resource, it must wait until the resource has been released.
- Simple Explanation: Think of a printer. Only one person can print a document at a time. The printer is an exclusive-use resource.

2. Hold and Wait Condition

- What it is: A process must be holding at least one resource while it is waiting to acquire additional resources that are currently held by other processes.
- Simple Explanation: This is like someone at a buffet holding a plate (one resource) but waiting in line for the main course (another resource). They are holding something while waiting for something else.

3. No Preemption Condition

- What it is: A resource cannot be forcibly taken away from the process holding it. A resource can only be released voluntarily by the process that is holding it, usually after it has completed its task.
- Simple Explanation: If you are in the middle of using the printer, no one can just come and yank it away from you. You have to finish your print job and release it yourself.

4. Circular Wait Condition

- What it is: There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain. For example, a set of waiting processes {P0, P1, ..., Pn} must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ..., and Pn is waiting for a resource held by P0.
- Simple Explanation: This is the "who has what I need" circle. Process A is waiting for a resource that Process B has, and Process B is waiting for a resource that Process A has.

All four of these conditions must be met for a deadlock to occur. By attacking any one of them, we can design strategies to prevent deadlocks.

A real-world example of a deadlock is a traffic jam at an intersection (Figure 3.2), where traffic becomes impossible.

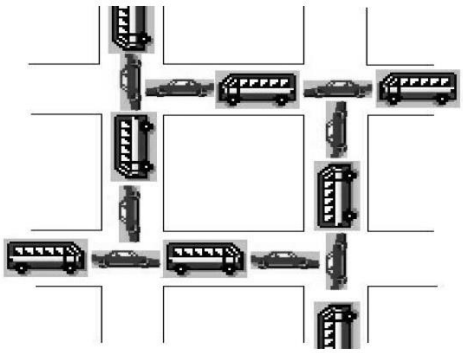


Figure 3.2. Traffic circulation problem.

We can see that the four conditions for deadlock are met:

- **Mutual Exclusion**: Only one car can occupy a particular spot on the road at any given moment.
- **Hold and Wait**: Each car is holding its current spot on the road while waiting for the spot ahead to become free.
- **No Preemption**: A car cannot be forcibly removed from its position on the road.
- **Circular Wait**: Each group of cars at a corner is waiting for the cars at the next corner to move, which are in turn waiting for the next corner, forming a circular dependency.

3.3 Deadlock Scenarios

While deadlocks often involve physical resources like printers or disk drives, they can also be caused by the logical resources used to manage them, including semaphores and messages.

3.3.1 Deadlock with Semaphores

A semaphore is a tool used to guard a resource and provide mutually exclusive access. However, when a process needs to acquire locks on multiple resources, the semaphore itself can become part of a deadlock.

Let's consider a system with two processes, A and B, and two resources: a CD drive and a printer. Each resource is protected by its own semaphore, sem_CD and sem_Printer.

```
Process A
{
    1 wait(sem_CD);
    2 wait(sem_Printer);
    3 use the resources;   //CS
    4 signal(sem_Printer);
    5 signal(sem_CD);
}
```

```
Process B
{
    1' wait(sem_CD);
    2' wait(sem_Printer);
    3' use the resources;   //CS
    4' signal(sem_Printer);
    5' signal(sem_CD);
}
```

Figure 3.3. Guarding two resources with two semaphores (Deadlock-Free)

If both processes request the resources in the same order (Figure 3.3), no deadlock can occur. In this case, both Process A and Process B first request the CD drive and then the printer. If Process A starts first and acquires the lock on the CD drive, Process B will simply be blocked when it tries to acquire the same lock. Process A will then be free to acquire the printer, do its work, and release both resources.

Now, let's look at what happens if there is a small change in the code for Process B, where it requests the resources in the opposite order (Figure 3.4).

```
Process A
{
    1 wait(sem_CD);
    2 wait(sem_Printer);
    3 use the resources;   //CS
    4 signal(sem_Printer);
    5 signal(sem_CD);
}
```

```
Process B
{
    1' wait(sem_Printer);
    2' wait(sem_CD);
    3' use the resources;   //CS
    4' signal(sem_CD);
    5' signal(sem_Printer);
}
```

Figure. 3.4 Guarding two resources with two semaphores (Deadlock)

Here, Process B first requests the printer and then the CD drive. This conflicting order of requests opens the door for a deadlock with the following sequence of events:

1. Process A executes wait(sem_CD) and successfully acquires the lock for the CD drive.
2. A context switch occurs, and Process B runs.
3. Process B executes wait(sem_Printer) and successfully acquires the lock for the printer.
4. Process A resumes and tries to execute wait(sem_Printer). It blocks, because the printer is held by Process B.
5. Process B resumes and tries to execute wait(sem_CD). It blocks, because the CD drive is held by Process A.

At this point, both processes are in a deadlock. Process A holds the CD and is waiting for the printer, while Process B holds the printer and is waiting for the CD. Neither can proceed.

3.3.2 Deadlock with Messages

Deadlocks are not limited to non-consumable resources. They can also occur with consumable resources like messages. A deadlock can occur if two processes both enter a state where they are waiting for a message from the other (Figure 3.5).

```
Process A
{
   …
Receive_msg(B);
   …
Send_msg(B, message);
}
```

```
Process B
{
   …
Receive_msg(A);
   …
Send_msg(A, message);
}
```

Figure 3.5 Deadlock using Messages

In this scenario:

1. Process A executes Receive_msg(B), a blocking call to wait for a message from Process B.
2. Process B executes Receive_msg(A), a blocking call to wait for a message from Process A.

Both processes are now blocked, waiting for a message that will never be sent, because the other process is also blocked waiting. This is another form of deadlock.

3.4 Modelling Deadlocks

When a system has multiple processes and resources, it can be difficult to see if a deadlock exists just by looking at the code. **A Resource-Allocation Graph (RAG)** is a formal, visual tool that provides a precise way to represent the state of resource allocation in a system.

- A Resource-Allocation Graph is a directed graph with two types of nodes and two types of edges:
- Process Nodes: Each process in the system is represented by a circle.
- Resource Type Nodes: Each resource type (e.g., Printer, Scanner) is represented by a rectangle.
- Resource Instances: Dots inside a resource rectangle represent the number of instances of that resource type. For example, a Printer resource with two dots means there are two identical printers available.
- Request Edge: A directed edge from a process circle to a resource rectangle (Pi → Rj) means that process Pi has requested an instance of resource Rj and is currently waiting for it.
- Assignment Edge: A directed edge from a resource instance (a dot) to a process circle (Rj → Pi) means that an instance of resource Rj has been allocated to process Pi.

Example: Let's analyze a specific Resource-Allocation Graph to understand how to interpret it. This graph (Figure 3.3) depicts the following situation:

- Graph Components
  - The set of processes is P = {P1, P2, P3}.
  - The set of resource types is R = {R1, R2, R3, R4}.

  - The set of edges is E = {P1 → R1, P2 → R3, R1 → P2, R2 → P2, R2 → P1, R3 → P3}.

- Resource Instances
  - R1 has one instance.
  - R2 has two instances.
  - R3 has one instance.
  - R4 has three instances.
- Process States
  - Process P1 is holding an instance of R2 and is waiting for an instance of R1.
  - Process P2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.
  - Process P3 is holding an instance of R3.

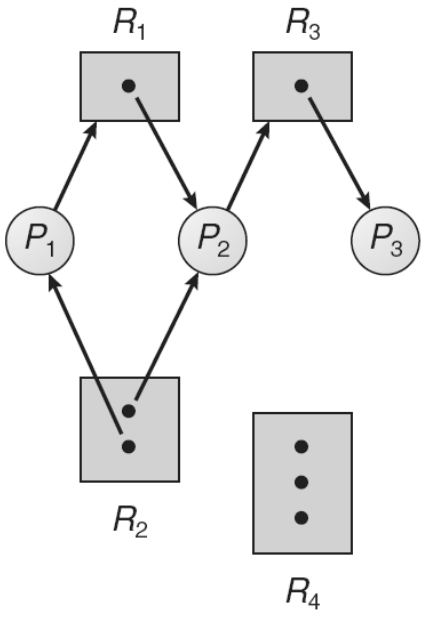


Figure 3.3. Resource-allocation graph

Cycles and Deadlocks: Interpreting the Graph

Example 1: No Cycle, No Deadlock

Let's start with the previous graph (Figure 3.3), which represents a system that can proceed without issue.

P3 is not waiting for any resource held by P1 or P2. The chain of requests ends here. Since there is no path that leads back to P1, the graph is **cycle-free**.

Because there is no cycle, there is **no deadlock**. The system can resolve itself. When P3 finishes, it will release R3, which can then be granted to P2. When P2 finishes, it will release R1, which can then be granted to P1.

Example 2: Cycle with Single-Instance Resources (Deadlock)

Now, let's examine a graph where a cycle guarantees a deadlock (Figure 3.4).

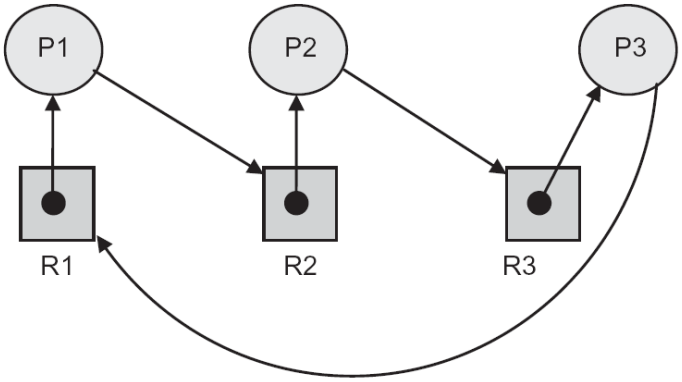


Figure. 3.4 RAG with deadlock

1. **Trace the Paths**: Here, we can clearly trace a circular dependency:
    - P1 is waiting for R2, which is held by P2.
    - P2 is waiting for R3, which is held by P3.
    - P3 is waiting for R1, which is held by P1.
2. **Check for a Cycle**: This forms the cycle P1 → R2 → P2 → R3 → P3 → R1 → P1.
3. **Conclusion**: Because this cycle involves only resource types that have a **single instance** (one dot each), a **deadlock exists**. Each process in the cycle is blocked waiting for another process in the same cycle. A cycle is a necessary and sufficient condition for deadlock in this case.

Example 3: Cycle with Multiple-Instance Resources (No Deadlock)

Finally, let's look at a case where a cycle is present, but the system is not deadlocked (Figure 3.5).

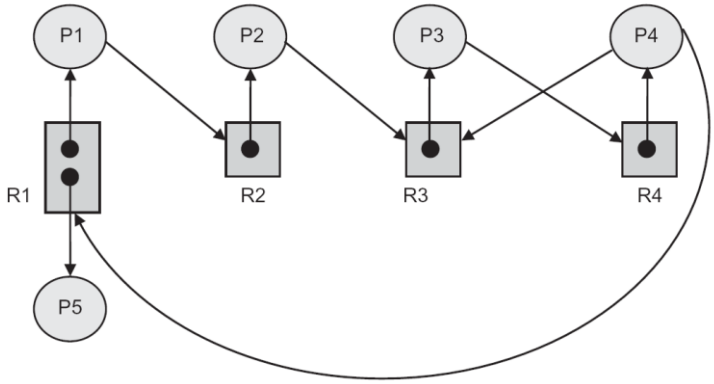


Figure. 3.5 RAG with cycle but without deadlock

1. **Trace the Paths**: This graph contains a cycle: P1 → R2 → P3 → R4 → P5 → R1 → P1.
2. **Check Resource Instances**: Look closely at resource R2. It has **two instances**. One is held by P1 and one is held by P4.
3. **Check for Resolution Path**: Process P2 is holding an instance of R1 and R3, but it is **not waiting** for any other resource. This means P2 is not part of any cycle and is not blocked.
4. **Conclusion**: Even though a cycle exists, there is **no deadlock**. The system can proceed because P2 can finish its work and release its resources (R1 and R3). Once R1 is released, it can be granted to the waiting process P5, which breaks the cycle. The presence of a cycle in a graph with multiple-instance resources does not guarantee a deadlock if there is a path to resolution.

Summary

The key to identifying deadlocks with a RAG is to look for cycles.

- If the graph contains no cycles, then there is no deadlock in the system.
- If the graph does contain a cycle, then a deadlock might exist.

The rule for determining if a cycle means a definite deadlock is based on the number of instances of the resources involved:

- If a cycle involves only resource types with a single instance each, then a deadlock has occurred. A cycle is a necessary and sufficient condition for deadlock in this case.
- If a cycle involves resource types with multiple instances, a cycle does not necessarily mean a deadlock has occurred. In this case, a cycle is a necessary but not a sufficient condition for deadlock.

4. Methods for Handling Deadlocks

After characterizing the deadlocks, the question is how to deal with them. Every system may be prone to deadlocks in a multi-programming environment, with few dedicated resources.

Generally speaking, there are four approaches an operating system can take to deal with deadlocks:

Deadlock Prevention

This is the most direct and restrictive approach. Deadlock prevention works by ensuring that at least one of the four necessary conditions for deadlock (mutual exclusion, hold and wait, no preemption, and circular wait) can never occur. This is achieved by constraining how processes can request resources.

- Analogy: This is like preventing traffic jams by designing a city with no four-way intersections, only overpasses and underpasses. By eliminating a necessary condition, you prevent the problem from ever happening.
- Trade-off: While effective, this method can lead to low resource utilization and may not always be practical.

Deadlock Avoidance

This approach is more flexible than prevention. With deadlock avoidance, the operating system requires advance information about the resources a process will request during its lifetime. For each resource request, the OS analyzes the current state of resource allocation and decides if granting the request could lead to a future deadlock. If the request is deemed "unsafe," the process is forced to wait, even if the resource is currently available.

- Analogy: This is like a smart traffic control system that sees a car approaching an intersection and, based on traffic flow data, predicts that letting it enter will cause a jam. The system holds the car at a red light until the risk has passed.
- Trade-off: This requires knowing all future resource requests in advance, which is often impossible.

Deadlock Detection and Recovery

This strategy assumes that deadlocks can and will occur. Instead of trying to prevent them, the system allows them to happen and then deals with the consequences. This method requires two algorithms:

1. An algorithm that regularly examines the state of the system to determine if a deadlock has occurred.
2. An algorithm to recover from the deadlock, for example by terminating one or more of the deadlocked processes or preempting resources.

- Analogy: This is like having a traffic helicopter that spots a gridlocked intersection (detection) and then sends in a tow truck to move one of the cars out of the way (recovery).
- Trade-off: There is overhead in constantly checking for deadlocks, and the recovery process (like killing a process) can be disruptive.

Ignoring the Problem

The fourth approach is to simply ignore the problem altogether and pretend that deadlocks will never happen in the system.

- Analogy: This is the "ostrich strategy"—burying your head in the sand and hoping for the best.
- Why it's used: This might seem irresponsible, but it is the most common approach used by operating systems like Windows and Linux. The reasoning is that deadlocks are relatively rare, and the performance overhead of prevention or detection is a constant cost that may not be worth paying. In this model, it becomes the application developer's responsibility to write deadlock-free programs.

4.1. Deadlock Prevention

Deadlock prevention is a set of methods for ensuring that at least one of the four necessary conditions for deadlock can never hold. These methods work by constraining how processes can request resources. Let's examine how we can "attack" each of the four conditions.

4.1.1. Attacking the Mutual Exclusion Condition

The first condition states that at least one resource must be non-shareable. To negate this, we would have to make all resources shareable.

- **Strategy**: Avoid assigning a resource exclusively unless absolutely necessary.
- **Feasibility**: This is often impossible. Some resources are intrinsically non-shareable. For example, a printer cannot be used by two processes at the same time without producing garbled output. While we can make some resources shareable (like read-only files), we cannot prevent mutual exclusion for all resource types.
- **Partial Solution (Spooling)**: For a device like a printer, we can use a technique called **spooling**. Processes don't write directly to the printer; they write to a dedicated, shareable area on the disk (the spool). A single system process, the printer daemon, is the only process that ever requests the physical printer. Since the daemon never requests any other resources, it can never cause a deadlock.

4.1.2. Attacking the Hold and Wait Condition

This condition states that a process holds at least one resource while waiting for another. To negate this, we must guarantee that a process requesting a resource does not hold any other resources.

There are two main protocols to achieve this:

- **Protocol 1: Request all resources at once.** A process must request and be allocated all of its resources before its execution begins.
    - **Drawback**: This leads to very low resource utilization. A process that needs a printer only at the very end of its execution would have to hold it for the entire time, preventing other processes from using it.
- **Protocol 2: Release resources before requesting new ones.** A process is allowed to request resources only when it has none. If it holds some resources and needs another, it must first release all currently held resources and then re-request them all at once, along with the new one.

- **Drawback**: This can be inefficient. A process might lose all its work if it has to release a resource and then wait a long time to get it back. Starvation is also possible if a process repeatedly has its resources taken.

4.1.3. Attacking the No Preemption Condition

This condition states that resources cannot be forcibly taken away from a process. To negate this, we would need to allow preemption.

- **Strategy**: If a process is holding resources and requests another that cannot be immediately allocated, the OS can preempt (take away) all the resources the process is currently holding. These preempted resources are added to the list of resources the process is waiting for. The process will only be restarted when it can regain its old resources, as well as the new ones it is requesting.
- **Feasibility**: This is only practical for resources whose state can be easily saved and restored, like CPU registers or memory. It is not feasible for resources like printers or tape drives, where preemption would corrupt the ongoing task.

4.1.4. Attacking the Circular Wait Condition

This is often the most practical condition to attack. The circular wait condition can be prevented by imposing a **total ordering** of all resource types.

- **Strategy**: We assign a unique integer number to each resource type (e.g., Scanner=1, Disk Drive=5, Printer=12). The protocol is simple: **processes can only request resources in an increasing order of enumeration.**
- **How it Works**: A process can request a resource Ri at any time. However, if it wants to request another resource Rj, it can only do so if the number assigned to Rj is greater than the number assigned to Ri (F(Rj) > F(Ri)).
  - For example, a process can request the Disk Drive (5) and then the Printer (12), but it cannot request the Printer (12) and then the Disk Drive (5).
- **Why it Prevents Deadlock**: This rule makes a circular wait impossible. A cycle requires that P0 waits for P1's resource, P1 waits for P2's resource, and so on, until Pn waits for P0's resource. If resource requests are always in increasing order, this implies F(R0) < F(R1) < ... < F(Rn) < F(R0). This is a mathematical contradiction, so a circular wait cannot form.

This approach is effective, but it is up to the application developers to follow the established ordering.

Example 1: Resource Ordering

Let's consider a system with three resource types assigned IDs as shown in the figure 3.6: Hard disk (ID=1), CD drive (ID=5), and Printer (ID=7).

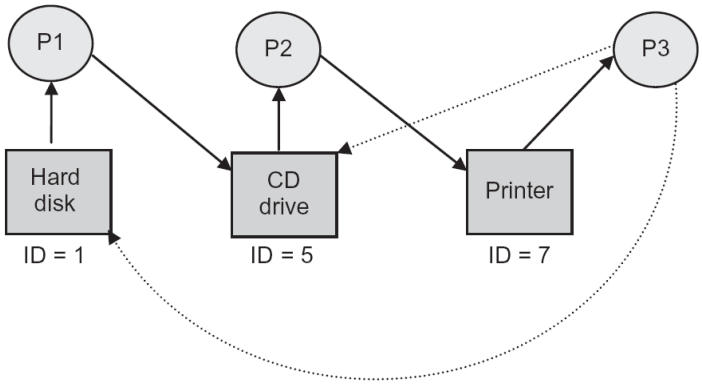


Figure 3.6 Preventing circular wait

The processes in this system must follow the resource ordering protocol.

- Process P1 can hold the Hard disk (1) and request the CD drive (5).
- Process P2 can hold the CD drive (5) and request the Printer (7).

Now, consider Process P3, which holds the Printer (7). If P3 were to request the Hard disk (1), as shown by the dotted line, this request would be **denied** by the system. The request is invalid because the ID of the requested resource (1) is not greater than the ID of the resource it currently holds (7).

By denying this request, the system prevents the possibility of a circular wait (P1 waiting for P2, P2 waiting for P3, and P3 waiting for P1), thus making a deadlock impossible.

Example 2: Dining Philosophers (A Deadlock-Free Solution)

In Chapter 3, we introduced a semaphore-based solution to the Dining Philosophers problem where each philosopher tried to pick up their left chopstick and then their right. We saw that this simple approach could lead to a deadlock.

To correct this, we will apply the strategy of attacking the circular wait condition. We will do this by assigning a unique number (an ID) to each resource (the chopsticks) and forcing all processes (the philosophers) to request the resources in increasing numerical order (Figure 3.7).

The deadlock in the original solution occurs because the chain of requests forms a circle (P0 waits for P1's chopstick, who waits for P2's, ..., who waits for P0's). We can break this circle by creating a strict rule: a philosopher must always pick up the lower-numbered chopstick before the higher-numbered one.

First, we number the chopsticks from 0 to 4. Chopstick i is the one to the left of philosopher i.

// N is the number of philosophers (5)

Semaphore chopstick[N] = {1, 1, 1, 1, 1};

// Chopstick IDs correspond to their index: 0, 1, 2, 3, 4

The Philosopher i

```
do {

    if (i != 4) {  // Pick up left chopstick then right chopstick.
    wait(chopstick[i]);
    wait(chopstick[(i + 1) % 5]);

    } else {  // Pick up right chopstick then left chopstick.
    wait(chopstick[(i + 1) % 5]);
    wait(chopstick[i]); }

    // --- Critical Section: Eat ---

    / --- Exit Section: Put down chopsticks ---
    // The order of release does not matter
    signal(chopstick[(i + 1) % 5]);
    signal(chopstick[i]);

    // --- Remainder Section: Think ---

    } while (true);
```

Figure 3.7 Dining Philosophers (A Deadlock-Free Solution)

This simple change makes it impossible for all philosophers to hold one chopstick while waiting for another in a circular chain.

Let's trace the worst-case scenario where all 5 philosophers get hungry at the same time:

1. **Philosophers 0, 1, 2, and 3** all run first. They follow the standard rule and successfully pick up their **left** chopsticks (chopsticks 0, 1, 2, and 3, respectively).
2. **Philosopher 4** (the last one) runs. According to the new rule, they must try to pick up their **right** chopstick first, which is chopstick 0.
3. However, chopstick 0 is already held by Philosopher 0. So, **Philosopher 4 waits** and cannot pick up any chopsticks.
4. Now, the chain of dependencies looks like this:
    - P0 has chopstick 0, wants chopstick 1 (held by P1).
    - P1 has chopstick 1, wants chopstick 2 (held by P2).
    - P2 has chopstick 2, wants chopstick 3 (held by P3).
    - P3 has chopstick 3, wants chopstick 4...

5. But chopstick 4 is **free**! Because Philosopher 4 was blocked trying to get chopstick 0, they never got to pick up their left chopstick (chopstick 4).

Since chopstick 4 is available, Philosopher 3 can acquire it, eat, and then release their chopsticks. This allows Philosopher 2 to eat, and so on. The circular wait is broken, and the deadlock is prevented.

4.2. Deadlock Avoidance

Deadlock prevention can be too restrictive and often leads to low resource utilization. An alternative strategy is **deadlock avoidance**. Instead of preventing the conditions for deadlock, this approach allows them to exist but carefully manages resource allocation to ensure a deadlock can never actually occur.

The central idea is that the operating system uses advance information about the resources a process *might* need in the future to decide whether a current request is "safe" to grant. If granting a request could lead to a future deadlock, the process is forced to wait, even if the resource is currently available.

The Concept of a Safe State

A safe state is a fundamental concept in deadlock avoidance. A system is in a safe state if it can allocate all the future resource requests of all processes in some specific order and still avoid a deadlock. The key to proving a state is safe is finding a safe sequence.

A safe sequence is a specific ordering of all processes in the system, let's say $<P_1, P_2, ..., P_n>$, that guarantees a path to completion for everyone.

For a sequence to be considered "safe," it must satisfy the following rule for every process Pi in the sequence: The resources that process P_i still needs (its maximum need minus what it currently holds) can be satisfied by the currently available resources PLUS all the resources held by every process P_j that comes before it in the sequence ($j < i$).

How it Works in Practice This definition means that if process P_i's needs can't be met right away, it can wait. The system can guarantee that all the processes P_j before P_i in the sequence will eventually finish and release their resources. When they do, there will be enough free resources to satisfy P_i's needs, allowing it to complete as well. This continues down the line until all processes are finished.

If no such sequence can be found, the system is in an unsafe state.

Important Distinction: An unsafe state is **not** a deadlocked state. An unsafe state is simply a condition from which a deadlock *might* occur. It is still possible for processes to release resources and for the system to recover. However, from a safe state, the OS can **guarantee** that a deadlock will not happen. The goal of deadlock avoidance is to never leave a safe state.

An Example of Safe and Unsafe States

To understand this, let's consider a system with a total of **12** tape drives and three processes. Each process declares in advance the maximum number of drives it will ever need.

At time t_0, the system is in the following state:

Process	Maximum Need	Currently Holding
P_0	10	5
P_1	4	2
P_2	9	2

Total drives held = 5 + 2 + 2 = 9.

Total drives available = 12 - 9 = **3**.

Is this a Safe State?

Yes. We can find a **safe sequence** that allows all processes to finish.

- **Process P1 runs first**: P1 needs 4 - 2 = 2 more drives. The system has 3 available, so it can grant P1's request. P1 runs to completion and releases all 4 of its drives.
    - *System now has 3 - 2 + 4 = 5 available drives.*
- **Process P0 runs next**: P0 needs 10 - 5 = 5 more drives. The system has 5 available, so it can grant P0's request. P0 runs to completion and releases all 10 of its drives.
    - *System now has 5 - 5 + 10 = 10 available drives.*
- **Process P2 runs last**: P2 needs 9 - 2 = 7 more drives. The system has 10 available, so it can grant P2's request. P2 runs to completion.
    - *System now has 10 - 7 + 9 = 12 available drives.*

Because we found a safe sequence (<P1, P0, P2>), the initial state was safe.

Moving to an Unsafe State

Now, let's go back to the initial state (3 drives available) and see what happens if P2 requests one more drive.

Process	Maximum Need	Currently Holding
P_0	10	5
P_1	4	2
P_2	9	**3**

Total drives held = 5 + 2 + 3 = 10.

Total drives available = 12 - 10 = **2**.

If the OS grants this request, is the system still in a safe state?

- P_1 needs 4 - 2 = 2 more drives. The system has 2, so P_1 can finish. It releases its 4 drives.
    - *System now has 2 - 2 + 4 = 4 available drives.*
- Now we are stuck.
    - P_0 needs 10 - 5 = 5 more drives, but the system only has 4.
    - P_2 needs 9 - 3 = 6 more drives, but the system only has 4.

There is **no safe sequence**. The system is in an **unsafe state**. A deadlock will occur if both P_0 and P_2 now request their maximum number of drives. Therefore, the deadlock avoidance algorithm would have denied P_2's initial request for one more drive, forcing it to wait until another process (like P_1) finished and released its resources.

4.2.1 Deadlock Avoidance for Single-Instance Resources

When dealing with a system where each resource type has only one instance, a variation of the Resource-Allocation Graph can be used to avoid deadlocks. To do this, a new type of edge is added to the graph.

The Claim Edge

In addition to the request and assignment edges, a claim edge is added. A claim edge, drawn as a dashed line from a process to a resource (Pi ---> Rj), indicates that process Pi may request resource Rj at some point in the future. All claim edges for a process must be declared to the system in advance.

The Avoidance Algorithm

The deadlock avoidance algorithm works by ensuring that a circular wait condition can never be created. The step-by-step process is as follows:

1. When a process Pi requests a resource Rj, the system does not grant it immediately.
2. The system pretends to grant the request by temporarily converting the claim edge to an assignment edge.

3. Next, the system runs a cycle-detection algorithm on this hypothetical new graph.
4. A decision is then made:
    - If the new graph is cycle-free, the allocation is safe and the request is officially granted.
    - If the new graph contains a cycle, granting the request would put the system into an unsafe state. The request is denied, and the process must wait.

By checking for cycles before every allocation, the system can guarantee it never enters an unsafe state.

Example

Let's look at a scenario based on the graph illustrated by Figure 3.8.

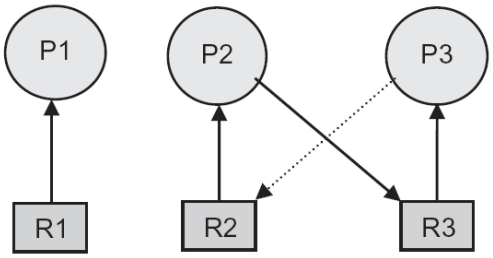


Figure 3.8. RAG with claim edge.

Here’s how the deadlock avoidance algorithm would handle this:

1. Acknowledge Request: The system receives the request from P3 for R2.
2. Pretend to Grant: The algorithm temporarily modifies the graph to reflect this new request. It adds a request edge from P3 to R2. The new, hypothetical state of the system is shown below.
3. Check for Cycles: The system now runs a cycle-detection algorithm on this new graph. It finds the following cycle:
    - P2 is waiting for R3.
    - R3 is held by P3.
    - P3 is now waiting for R2.
    - R2 is held by P2.

    This forms the cycle P2 → R3 → P3 → R2 → P2.
4. Decision: Because granting the request would create a cycle, the system identifies the new state as unsafe. The request from P3 is denied, and P3 is blocked without the request edge being permanently added.

By denying this unsafe request, the system avoids the deadlock and remains in its initial, safe state.

4.2.2 The Banker's Algorithm for Multiple-Instance Resources

When a system has multiple instances of each resource type, the Resource-Allocation Graph method is not sufficient to detect potential deadlocks. For this more complex scenario, we use the **Banker's Algorithm**, which was developed by Edsger Dijkstra.

The name comes from an analogy to a banking system. A bank will not grant a loan to a customer if it knows that it might not have enough cash to fulfill the needs of all its other customers. Similarly, the Banker's Algorithm ensures that the operating system never allocates its resources in such a way that it cannot satisfy the declared needs of all processes.

The algorithm requires that each new process declare in advance the **maximum number of instances** of each resource type that it may ever need.

Data Structures for the Banker's Algorithm

To implement the algorithm, the OS must maintain several data structures that track the state of the system. Let's assume there are n processes and m resource types.

- **Available[m]**: A vector that indicates the number of available instances for each resource type. Available[j] = k means there are k instances of resource type Rj free.
- **Max[n][m]**: A matrix that defines the maximum demand of each process. Max[i][j] = k means process Pi may request at most k instances of resource type Rj.
- **Allocation[n][m]**: A matrix that defines the number of resources currently allocated to each process. Allocation[i][j] = k means Pi is currently holding k instances of Rj.
- **Need[n][m]**: A matrix that indicates the remaining resource need of each process. Need[i][j] = k means Pi may still need k more instances of Rj to complete its task. This is calculated as Need[i][j] = Max[i][j] - Allocation[i][j].

These data structures vary over time in both size and value. To simplify the presentation of the banker's algorithm, we next establish some notation. Let X and Y be vectors of length n. We say that $X \leq Y$ if and only if $X[i] \leq Y[i]$ for all $i = 1, 2, ..., n$. For example, if $X = (1,7,3,2)$ and $Y = (0,3,2,1)$, then $Y \leq X$. In addition, $Y < X$ if $Y \leq X$ and $Y \neq X$.

We can treat each row in the matrices ***Allocation*** and ***Need*** as vectors and refer to them as $\boldsymbol{Allocation}_i$ and $\boldsymbol{Need}_i$ **.** The vector $\boldsymbol{Allocation}_i$ specifies the resources currently allocated to process *Pi*; the vector ***Needi*** specifies the additional resources that process *Pi* may still request to complete its task.

The Safety Algorithm

The first part of the Banker's Algorithm is the **Safety Algorithm**, which checks if the current system state is safe.

The algorithm works by trying to find a safe sequence. It simulates the completion of processes to see if there's a way for everyone to finish.

1. Initialize a temporary Work vector equal to the Available resources. Also, create a boolean vector Finish of length n, and initialize all its elements to false.
2. Find a process P_i such that Finish[i] is false and its $Need_i$ vector is less than or equal to the Work vector ($Need_i \leq Work$). If no such process exists, go to step 4.
3. If a process is found, pretend it finishes its work. Add its allocated resources back to the Work vector ($Work = Work + Allocation_i$) and mark it as finished (Finish[i] = true). Go back to step 2.
4. After checking all processes, if Finish[i] is true for all processes, the system is in a **safe state**. Otherwise, it is unsafe.

The Resource-Request Algorithm

The second part is the algorithm that runs whenever a process makes a request. Let $Request_i$ be the request vector for process P_i.

1. **Check Validity**: First, check if the request is valid. If $Request_i > Need_i$, the process has asked for more than it declared it would need, which is an error.
2. **Check Availability**: Next, check if the resources are available. If $Request_i >$ Available, the process must wait.
3. **Pretend to Allocate**: If the request is valid and the resources are available, the system pretends to grant the request by updating the data structures:
   - Available := Available - $Request_i$
   - $Allocation_i := Allocation_i + Request_i$
   - $Need_i := Need_i - Request_i$
4. **Run Safety Algorithm**: After pretending to grant the request, the system runs the **Safety Algorithm** on this *new*, hypothetical state.
   - If the new state is **safe**, the transaction is officially completed, and the resources are allocated to the process.
   - If the new state is **unsafe**, the request is denied. The original state is restored, and the process must continue to wait.

An Illustrative Example

Consider a system with 5 processes (P_0-P_4) and 3 resource types (A, B, C) with 10, 5, and 7 instances, respectively.

State at time T_0:

	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

First, we calculate the **Need** matrix (Need := Max - Allocation):

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

Is this state safe? Let's run the Safety Algorithm.

Work := Available := [3 3 2].

1. **Check P_1**: Its $Need_1$ [1 2 2] ≤ Work [3 3 2]. Yes. So, we pretend P_1 finishes.
    - Work := [3 3 2] + [2 0 0] = [5 3 2].
2. **Check P_3**: Its $Need_3$ [0 1 1] ≤ Work [5 3 2]. Yes. So, P_3 finishes.
    - Work := [5 3 2] + [2 1 1] = [7 4 3].
3. **Check P_4**: Its $Need_4$ [4 3 1] ≤ Work [7 4 3]. Yes. So, P_4 finishes.
    - Work := [7 4 3] + [0 0 2] = [7 4 5].
4. **Check P_0**: Its $Need_0$ [7 4 3] ≤ Work [7 4 5]. Yes. So, P_0 finishes.

- Work := [7 4 5] + [0 1 0] = [7 5 5].

5. **Check P_2**: Its $Need_2$ [6 0 0] ≤ Work [7 5 5]. Yes. So, P_2 finishes.
   - Work := [7 5 5] + [3 0 2] = [10 5 7].

All processes could finish. The safe sequence is <P_1, P_3, P_4, P_0, P_2>. The state is **safe**.

Scenario 1: P1 requests (1, 0, 2)

Now, suppose Process P1 makes a request for 1 instance of resource A and 2 instances of resource C. So, $Request_1$ = (1, 0, 2).

The Resource-Request Algorithm runs as follows:

1. Check Validity: Is $Request_1 \leq Need_1$? Is (1, 0, 2) ≤ (1, 2, 2)? Yes, it is. The request is valid.
2. Check Availability: Is $Request_1 \leq$ Available? Is (1, 0, 2) ≤ (3, 3, 2)? Yes, it is. The resources are available.
3. Pretend to Allocate: The system calculates the hypothetical new state if the request were granted.
   - Available becomes Available := [3 3 2] - [1 0 2] = [2 3 0].
   - P_1's Allocation becomes $Allocation_1$:= [2 0 0] + [1 0 2] = [3 0 2].
   - P_1's Need becomes $Need_1$:= [1 2 2] - [1 0 2] = [0 2 0].
4. Run Safety Algorithm: The system now checks if this new state is safe. Running the safety algorithm, it finds that the sequence <P_1, P_3, P_4, P_0, P_2> is a safe sequence.
5. Decision: Since the resulting state is safe, the request from P_1 is immediately granted.

Scenario 2: P_4 requests (3, 3, 0)

Now, from this new state (with Available = [2 3 0]), suppose P_4 makes a request for (3, 3, 0).

1. Check Validity: Is $Request_4 \leq Need_4$? Is (3, 3, 0) ≤ (4, 3, 1)? Yes, it is. The request is valid.
2. Check Availability: $Request_4 \leq$ Available? Is (3, 3, 0) ≤ (2, 3, 0)? No, it is not.
3. Decision: The resources are not available, so P_4's request is denied, and P_4 must wait.

Scenario 3: P_0 requests (0, 2, 0)

Finally, let's consider a more subtle case. From the same state (with Available = [2 3 0]), suppose P_0 requests (0, 2, 0).

1. Check Validity: Is (0, 2, 0) ≤ $Need_0$? Is (0, 2, 0) ≤ (7, 4, 3)? Yes.

2. Check Availability: Is (0, 2, 0) ≤ Available? Is (0, 2, 0) ≤ (2, 3, 0)? Yes. The resources are physically available.
3. Pretend to Allocate: The system calculates the hypothetical new state.
    - Available becomes Available := [2 3 0] - [0 2 0] = [2 1 0].
    - P_0's Allocation $Allocation_0$:= [0 1 0] + [0 2 0] = [0 3 0].
    - P_0's Need becomes $Need_0$:= [7 4 3] - [0 2 0] = [7 2 3].
4. Run Safety Algorithm: The system checks if this hypothetical new state (with Available = [2 1 0]) is safe. It will find that there is no process whose Need can be satisfied by the available resources.
5. Decision: Because the resulting state would be unsafe, the request from P_0 is denied, even though the resources were technically available. P_0 must wait.

4.3 Deadlock Detection and Recovery

f a system does not use a deadlock prevention or avoidance strategy, then a deadlock situation may occur. This approach accepts that deadlocks are a possibility. The system must then provide two things:

1. An algorithm that examines the state of the system to **detect** whether a deadlock has occurred.
2. An algorithm to **recover** from the deadlock.

4.3.1 Deadlock Detection

How does the system find out that a deadlock has happened? The method used depends on whether resources have single or multiple instances.

4.3.1.1 Deadlock Detection with One Resource of Each Type

When all resources have only a single instance, we can use a simplified version of the Resource-Allocation Graph called a **wait-for graph**.

A wait-for graph is created by taking a Resource-Allocation Graph, removing the resource nodes, and collapsing the edges. An edge from $P_i \rightarrow P_j$ exists in a wait-for graph if and only if process P_i is waiting for a resource that process P_j holds.

As you can see (Figure 3.9), the resource nodes are removed, and the dependencies are shown directly between processes.

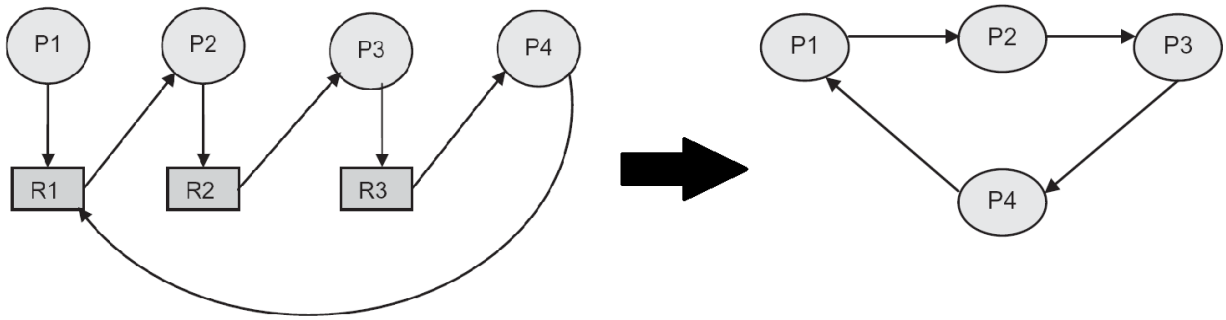


Figure 3.9. From RAG to Wait-for graph

Cycle Detection A deadlock exists in the system if and only if the wait-for graph contains a **cycle**. The system can periodically run a cycle-detection algorithm on the graph. If a cycle is found, a deadlock has been detected, and the processes within the cycle are the deadlocked processes.

While we can often spot a cycle in a simple graph visually, a computer needs a formal algorithm to do so. The method described is a Depth-First Search (DFS) designed to find a cycle. The algorithm works by starting at each node in the graph and trying to trace a path that leads back to itself.

Here is the algorithm, which is performed for each node N in the graph:

1. Initialize L as an empty list (this list will track the current path). Designate all arcs (edges) as "unmarked".
2. Add the current node to the end of L. Check if this node now appears in L twice. If it does, a cycle has been found (the path in L is the cycle), and the algorithm terminates.
3. From the current node, look for any unmarked outgoing arcs.
    - If an unmarked outgoing arc is found: Pick one, mark it, and follow it to the new current node. Then, go back to step 2.
    - If no unmarked outgoing arcs are found: This is a dead end. Remove the current node from L and go back to the previous node, making it the new current node. Then, repeat step 3 from that previous node.
4. If you backtrack all the way to the starting node and there are no more unmarked outgoing arcs to follow, the search from this starting node is complete, and no cycle was found. The algorithm would then start over from the next node in the graph.

Example (Figure 3.9):

Let's trace this algorithm on a simple wait-for graph with the cycle P1 → P2 → P3 → P1.

1. Start at P1:
    - Initialize L = []
    - Add P1. L is now [P1]. No duplicates.
    - From P1, follow the only outgoing arc to P2.
2. Current node is P2:
    - Add P2. L is now [P1, P2]. No duplicates.
    - From P2, follow the only outgoing arc to P3.
3. Current node is P3:
    - Add P3. L is now [P1, P2, P3]. No duplicates.
    - From P3, follow the only outgoing arc to P1.
4. Current node is P1:
    - Add P1. L is now [P1, P2, P3, P1].
    - Check for duplicates: The node P1 now appears twice in the list L.
    - Result: A cycle has been detected. The algorithm terminates and reports a deadlock.

4.3.1.2 Deadlock Detection with Multiple Resources of Each Type

When resource types have multiple instances, the wait-for graph is not sufficient. Instead, a detection algorithm that is very similar in structure to the Banker's Algorithm is used. This algorithm periodically checks the system's state to see if a deadlock has already occurred.

Data Structures Used

The algorithm uses three main data structures to represent the state of the system, where n is the number of processes and m is the number of resource types:

- *Available[m]*: A vector showing how many instances of each resource are currently available.
- *Allocation[n][m]*: A matrix showing how many instances of each resource are currently allocated to each process.
- *Request[n][m]*: A matrix showing the current resource requests of each process. *Request[i][j]* = k means process P_i is currently waiting for k instances of resource type R_j.

The Detection Algorithm

The algorithm works by trying to find an order in which all processes can finish. It optimistically assumes that a process that can finish, will finish and release its resources.

1. Create a temporary Work vector, and initialize it with the Available resources. Also create a boolean vector Finish of length n, and initialize it to false for any process that has resources allocated.
2. Find a process Pi that is not yet finished (Finish[i] = false) and whose current Request can be satisfied by the Work vector ($Request_i \leq Work$).
3. If such a process is found, assume it will finish. Add its Allocation back to the Work vector ($Work := Work + Allocation_i$) and mark it as finished (Finish[i] := true). Go back to step 2.
4. If no such process can be found, go to step 5.
5. After the algorithm completes, if Finish[i] is false for any process P_i, it means that process is deadlocked, and the system is in a deadlocked state.

Example

Consider a system with 5 processes (P_0-P_4) and 3 resource types (A, B, C) with 7, 2, and 6 instances, respectively.

State at time T_0:

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Is the system deadlocked? Let's run the algorithm.

- Work := Available := [0 0 0].
- Finish := [false, false, false, false, false].

1. Check P_0: Request [0 0 0] ≤ Work [0 0 0]. Yes. P_0 can finish.
    - Work := [0 0 0] + [0 1 0] = [0 1 0]. Finish[P_0] := true.
2. Loop again with Work := [0 1 0]: Check P_2: Request [0 0 0] ≤ Work [0 1 0]. Yes. P_2 can finish.
    - Work := [0 1 0] + [3 0 3] = [3 1 3]. Finish[P_2] := true.
3. Loop again with Work := [3 1 3]: Check P_3: Request [1 0 0] ≤ Work [3 1 3]. Yes. P_3 can finish.
    - Work := [3 1 3] + [2 1 1] = [5 2 4]. Finish[P_3] := true.
4. Loop again with Work := [5 2 4]: Check P_1: Request [2 0 2] ≤ Work [5 2 4]. Yes. P_1 can finish.
    - Work := [5 2 4] + [2 0 0] = [7 2 4]. Finish[P_1] = true.
5. Loop again with Work = [7 2 4]: Check P_4: Request [0 0 2] ≤ Work [7 2 4]. Yes. P_4 can finish.
    - Work = [7 2 4] + [0 0 2] = [7 2 6]. Finish[P_4] = true.

Since Finish is true for all processes, the system is not in a deadlocked state.

Now, suppose P_2 makes an additional request for one instance of C, changing $Request_2$ to [0 0 1]. The algorithm would run, but after finishing P_0, the Work vector would be [0 1 0]. At this point, no other process's request can be satisfied. Since Finish would be false for P_1, P_2, P_3, and P_4, the system would correctly detect that it is now in a deadlocked state.

4.3.2 Deadlock Recovery

Once a deadlock detection algorithm determines that a deadlock exists, the system must break it. This can be done manually by a system operator or automatically by the operating system. There are two primary strategies for automatic recovery: terminating processes or preempting resources. Neither one is particularly attractive, but they are necessary to get the system moving again.

1. Recovery through Process Termination

The simplest, most direct way to break a deadlock cycle is to abort one or more of the processes involved. This frees up the resources they hold, allowing other processes to proceed. There are two main approaches:

- Abort All Deadlocked Processes: This method is the brute-force approach. It will definitely break the deadlock, but at a high cost. All the work done by the terminated processes is lost and may have to be recomputed from scratch.
- Abort One Process at a Time: A more nuanced strategy is to terminate one process in the cycle, see if the deadlock is resolved, and if not, terminate another, and so on. This

requires running the deadlock-detection algorithm after each termination, which adds considerable overhead.

Choosing a Victim If we choose to abort processes one by one, the system must decide which process to terminate. This is a difficult policy decision. Should it kill the process with the lowest priority? The one that has run for the shortest amount of time? The one holding the fewest resources? Or perhaps a process that can be easily rerun from the beginning, like a compilation, which is safer to kill than a process that is updating a database.

2. Recovery through Resource Preemption

A less drastic approach than killing a process is to forcibly take a resource away from one process and give it to another. This is called resource preemption.

This strategy introduces three significant challenges:

1. Selecting a Victim: Just like with process termination, the system must decide which resource to preempt from which process. The goal is to minimize the negative impact.
2. Rollback: You can't just take a resource away. If a process was in the middle of using a printer, the system must return that process to a safe state before the resource was acquired. To do this, the system may rely on checkpointing, where the state of a process is periodically saved to a file. To recover, the system can roll back the victim process to a previous checkpoint. All work done since that checkpoint is lost.
3. Starvation: The system must ensure that the same process is not always chosen as the victim, which would prevent it from ever finishing its task.

Recovering through preemption is often difficult or impossible, depending on the nature of the resource.

4.4 A Practical Approach to Deadlock Handling

We have discussed several complex methods for managing deadlocks, but which one is actually used? The practical approach taken by most general-purpose operating systems might surprise you.

The Dominant Strategy: Ignoring the Problem

For most operating systems, including Windows and Linux, the primary strategy for handling deadlocks is to **ignore them**. This may seem irresponsible, but it's a pragmatic decision based on a cost-benefit analysis.

- **Cost**: Implementing prevention or avoidance algorithms is expensive. Prevention imposes restrictive rules on all processes, which can hurt resource utilization. Avoidance requires complex tracking and advance knowledge of resource needs. Detection algorithms must be run periodically, which consumes CPU cycles.
- **Frequency**: In a well-designed, general-purpose system, deadlocks are a relatively rare occurrence.

The conclusion is that the constant performance overhead of a strict prevention or avoidance strategy is often a higher price to pay than the cost of dealing with an occasional, rare deadlock. In this model, the responsibility shifts from the OS to the **application developer** to write careful, deadlock-free code.

When Deadlocks Cannot be Ignored

There are critical environments where ignoring deadlocks is not an option.

- **Real-Time Systems**: Systems that control industrial machinery, medical equipment, or flight systems must run continuously. They cannot afford to freeze or be rebooted, as this could lead to catastrophic data loss or physical danger. These systems often employ strict deadlock prevention or avoidance techniques.
- **Database Systems**: These systems handle a large number of concurrent transactions that lock records. Deadlocks are a frequent and expected problem. Therefore, database systems almost always have built-in deadlock detection and recovery mechanisms, such as two-phase locking.

An Integrated Strategy for Programmers

Since the OS often doesn't solve the problem, programmers should adopt an integrated strategy to minimize the risk of deadlock in their applications:

1. **Minimize Mutual Exclusion**: Use shareable resources where possible. For non-shareable resources, use techniques like spooling (e.g., for a printer) to create a virtual device that reduces contention for the physical hardware.
2. **Apply Resource Preemption Where Possible**: For resources that can be safely preempted (like memory), design the system to take advantage of this.
3. **Use Resource Ordering**: This is one of the most effective prevention techniques. Always establish a linear, hierarchical ordering for locks and resources and ensure all threads/processes acquire them in that increasing order. This breaks the circular wait condition.
4. **Favor Detection over Avoidance**: If prevention is not possible, detection is often a more practical choice than avoidance, as it does not require predicting the future.

Related Issues: Livelock and Starvation

Finally, it's important to distinguish deadlocks from two similar-sounding problems.

- **Livelock**: A livelock occurs when processes are not blocked, but they are stuck in a loop of continuously changing their state in response to each other without making any forward progress. Think of two people trying to pass in a hallway who keep stepping the same way. They are active, but not progressing.
- **Starvation**: This occurs when a process is indefinitely denied access to a resource it needs, even though the resource may become free. This is usually due to a scheduling

policy. For example, if a system always gives a resource to the process with the highest priority, a low-priority process might wait forever. Unlike a deadlock, where everyone is stuck, in starvation, other processes are making progress.

References

1. **Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. C.** (2018). *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books.
2. **Bovet, D. P., & Cesati, M.** (2005). *Understanding the Linux Kernel* (3rd ed.). O'Reilly Media.
3. **Brinch Hansen, P.** (2001). *Classic Operating Systems: From Batch Processing to Distributed Systems*. Springer-Verlag.
4. **Chauhan, N.** (2014). *Principles of Operating Systems*. Oxford University Press.
5. **Coffman, E. G., Elphick, M. J., & Shoshani, A.** (1971). "System deadlocks". *ACM Computing Surveys*, 3(2), 67–78.
6. **Cook, D. J., & Das, S. K.** (2005). *Smart Environments: Technology, Protocols and Applications*. John Wiley & Sons.
7. **Crowley, C.** (1997). *Operating Systems: A Design-Oriented Approach*. Irwin/McGraw-Hill.
8. **Deitel, H. M., Deitel, P. J., & Choffnes, D. R.** (2004). *Operating Systems* (3rd ed.). Pearson/Prentice Hall.
9. **Dijkstra, E. W.** (1965). "Solution of a problem in concurrent programming control". *Communications of the ACM*, 8(9), 569.
10. **Dijkstra, E. W.** (1968). "Co-operating sequential processes". In F. Genuys (Ed.), *Programming Languages*. Academic Press.
11. **Dimitoglou, G.** (1998). "Deadlocks and Methods for Their Detection, Prevention, and Recovery in Modern Operating Systems." *ACM SIGOPS Operating Systems Review*, 32(3), 67–74.
12. **Downey, A. B.** (2016). *The Little Book of Semaphores* (2nd ed.). Green Tea Press.
13. **Gorman, M.** (2004). *Understanding the Linux Virtual Memory Manager*. Prentice Hall.
14. **Herlihy, M. P.** (1991). "Wait-free synchronization". *ACM Transactions on Programming Languages and Systems*, 13(1), 124–149.
15. **Hoare, C. A. R.** (1974). "Monitors: An operating system structuring concept". *Communications of the ACM*, 17(10), 549–557.
16. **Holt, R. C.** (1972). "Some deadlock properties of computer systems". *ACM Computing Surveys*, 4(3), 179–196.
17. **Kernighan, B. W., & Ritchie, D. M.** (1988). *The C Programming Language* (2nd ed.). Prentice Hall.
18. **Kerrisk, M.** (2010). *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press.

19. **Kochan, S. G.** (2014). *Programming in C* (4th ed.). Addison-Wesley Professional
20. **Love, R.** (2010). *Linux Kernel Development* (3rd ed.). Addison-Wesley Professional.
21. **Love, R.** (2013). *Linux System Programming: Talking Directly to the Kernel and C Library* (2nd ed.). O'Reilly Media.
22. **McKusick, M. K., Neville-Neil, G. V., & Watson, R. N.** (2014). *The Design and Implementation of the FreeBSD Operating System* (2nd ed.). Addison-Wesley Professional.
23. **Ritchie, D. M., & Thompson, K.** (1974). "The UNIX time-sharing system". *Communications of the ACM*, 17(7), 365–375.
24. **Rochkind, M. J.** (2004). *Advanced UNIX Programming* (2nd ed.). Addison-Wesley Professional.
25. **Saltzer, J. H., & Kaashoek, M. F.** (2009). *Principles of Computer System Design: An Introduction*. Morgan Kaufmann.
26. **Silberschatz, A., Galvin, P. B., & Gagne, G.** (2018). *Operating System Concepts* (10th ed.). John Wiley & Sons.
27. **Stallings, W.** (2017). *Operating Systems: Internals and Design Principles* (9th ed.). Pearson.
28. **Stevens, W. R., & Rago, S. A.** (2013). *Advanced Programming in the UNIX Environment* (3rd ed.). Addison-Wesley Professional.
29. **Tanenbaum, A. S., & Bos, H.** (2015). *Modern Operating Systems* (4th ed.). Pearson.