Computação em Larga Escala

Concurrency - part II

Eurico Pedrosa

António Rui Borges

Universidade de Aveiro - DETI

2025-03-01

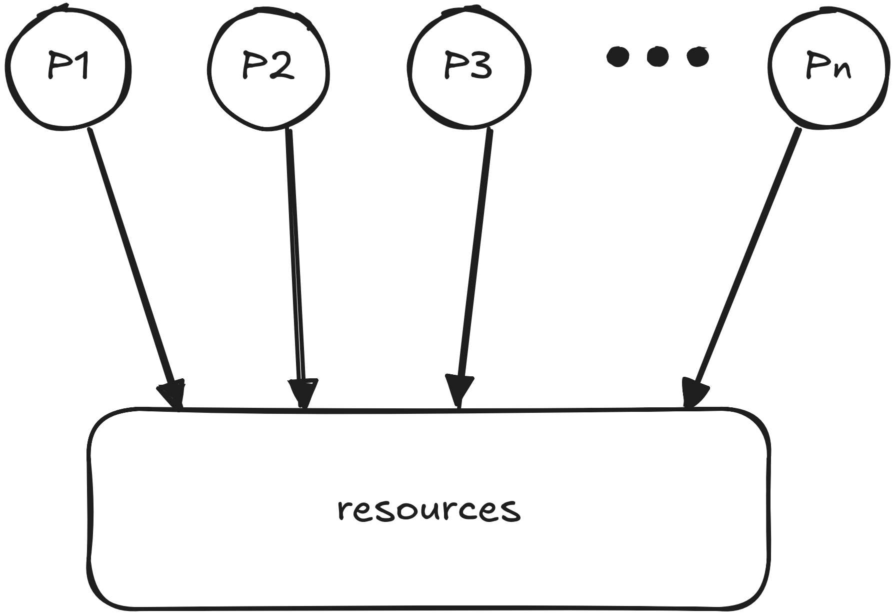
General principles of concurrency

In a **multiprogrammed environment**, coexisting processes may exhibit different types of interaction:

- **Independent Processes** – These processes **do not explicitly interact** with one another. Each process is **created, executed, and terminated independently**, without direct communication or data sharing. However, implicit interaction occurs due to **competition for system resources** (e.g., CPU time, memory, I/O devices).
  - Typically, independent processes are:
    - Created by **different users** or by the **same user** for distinct tasks in an interactive environment.
    - Processes executed separately as part of **batch job processing**.

Process Interaction

Independent Processes

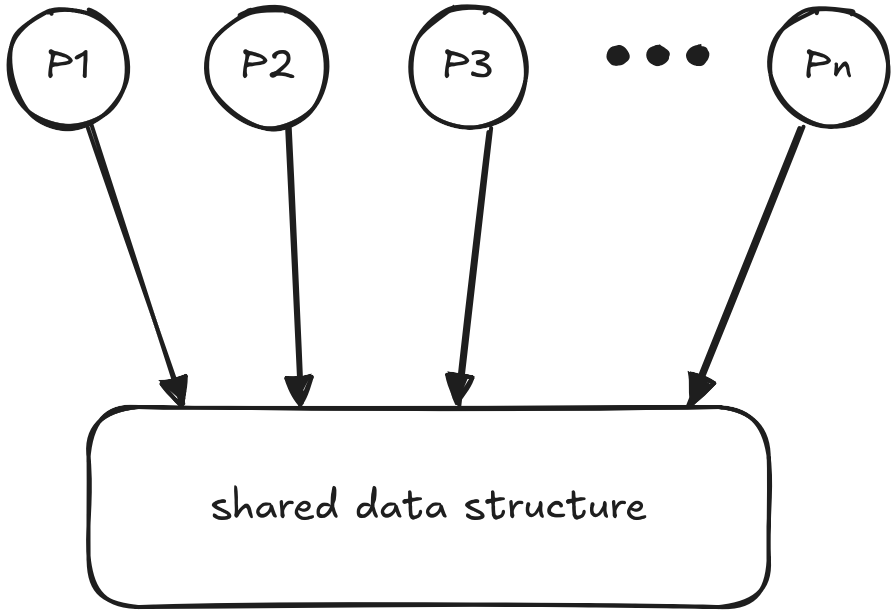


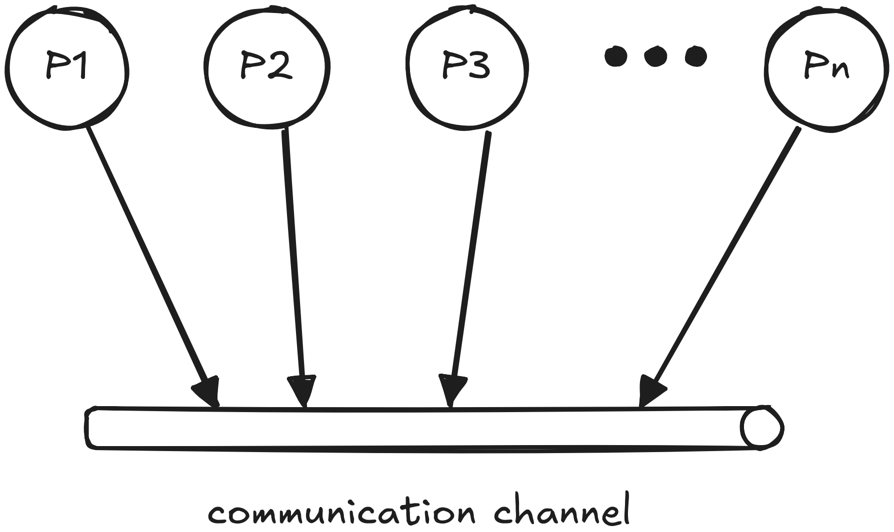
Process Interaction

- **Cooperating Processes** – These processes **share information or explicitly communicate** with each other.
  - **Shared memory** – Cooperating processes share a **common address space**, allowing direct access to shared data.
  - **Inter-process communication (IPC)** – If processes do not share memory, they communicate via **message passing mechanisms** (e.g., pipes, message queues, sockets, or shared memory buffers).

Process Interaction

Cooperating Processes





Process Interaction

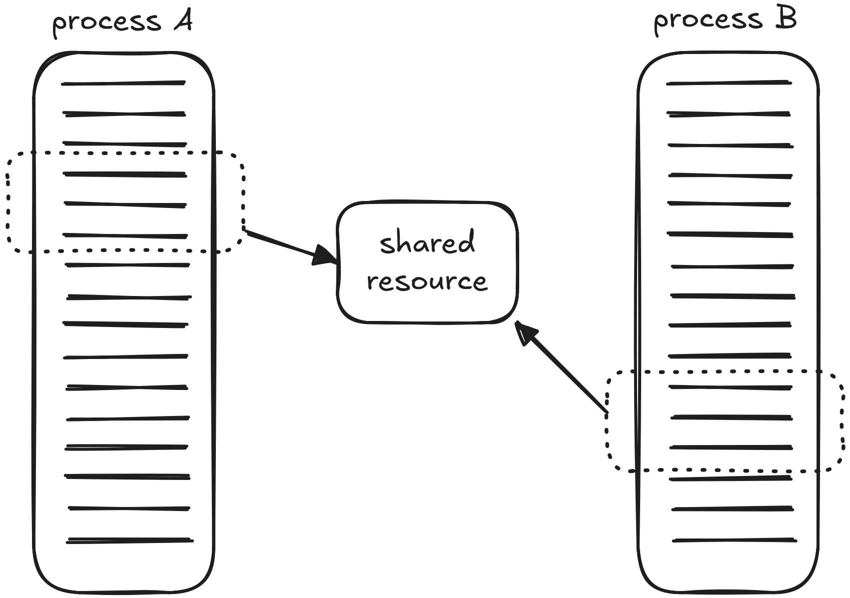
- **Independent processes** that compete for access to a common resource within the computational system.
  - It is the **operating system's responsibility** to ensure that resource allocation is managed in a **controlled manner**, preventing any **loss of information**.
  - This generally requires that only **one process at a time** is granted access to the resource, a principle known as **mutual exclusion**.

Process Interaction

- **Cooperating processes** that share information or communicate with each other.
  - It is the **responsibility of the involved processes** to ensure that access to shared resources is **coordinated and controlled**, preventing information loss.
  - This typically requires that only **one process at a time** accesses the shared resource (**mutual exclusion**).
  - The **communication channel** itself is considered a **computational resource**; therefore, **access to it should be treated as a competition for a shared resource**.

Critial Region

When discussing a process's access to a resource or a shared region, what is actually being referred to is the **processor's execution of the corresponding access code**. This code must be executed in a manner that **prevents race conditions**, which would otherwise lead to **information loss**. To ensure controlled execution, such code is typically referred to as a **critical region**.



Racing Conditions

Imposing **mutual exclusion** on access to a resource or a shared region, due to its restrictive nature, can lead to two undesirable consequences:

- **Deadlock / Livelock** – This occurs when two or more processes remain indefinitely **waiting** (either **blocked** or in **busy waiting**) for access to their respective **critical regions**, depending on events that can be **proven never to occur**. The result is the **complete blocking of operations**.
- **Indefinite Postponement (Starvation)** – This occurs when one or more processes compete for access to a **critical region**, but due to an ongoing influx of new competing processes, access is **repeatedly denied**. As a result, the affected processes are **unable to make progress**, creating a real **obstacle to their execution**.

Racing Conditions

When designing a **multithreaded application**, the goal is to **prevent these pathological conditions** and ensure that the code exhibits the **liveness property**—the guarantee that processes will eventually **make progress** and complete their execution.

Racing Conditions

A **general solution** to the **mutual exclusion problem** must satisfy the following **desirable properties**:

- **Effective enforcement of mutual exclusion** – Access to a **critical region** associated with a given **resource** or **shared region** must be granted to **only one process at a time**, among all processes competing for access **concurrently**.
- **Independence from process execution speed or number** – The correctness of the solution must not rely on any assumptions about **the relative speed of execution** of processes or their **total number**.
- **Non-interference from external processes** – A process **outside** the critical region must **never prevent another process** from entering.
- **No indefinite postponement (Starvation-freedom)** – Any process that **requests access** to a critical region must **eventually be granted entry**, ensuring that access is **not denied indefinitely**.
- **Bounded execution time within a critical region** – A process **inside a critical region** must execute its critical section **within a finite time**, ensuring that **progress is not blocked indefinitely**.

Resources - Definition and Classification

A **resource** is any **component** or **entity** that a process requires for execution. Resources can be classified into two main categories:

1. **Physical resources** – Hardware components of the computational system, such as:
   - **Processors**
   - **Memory regions** (main memory or mass storage)
   - **Input/output devices** (printers, network interfaces, etc.)
2. **Logical resources** – Shared structures managed by the **operating system** or **application-level processes**, such as:
   - **Process control tables**
   - **Interprocess communication channels**
   - **Shared data structures**

Types of Resource Allocation

A critical property of resources is **how they are appropriated by processes**. Based on this, resources are categorized as follows:

- **Preemptable resources** – These **can be forcibly reassigned** from one process to another **without causing malfunction**. Examples include:
  - **Processors** (in multiprogramming environments, where execution can be paused and resumed)
  - **Main memory regions** (used to store a process's address space, which can be swapped in and out)
- **Non-preemptable resources** – These **cannot be forcibly reassigned** without causing errors or inconsistencies. Examples include:
  - **Printers** (a print job cannot be interrupted and resumed without corruption)
  - **Shared data structures** (which require **mutual exclusion** for correct access and manipulation)

Deadlock Characterization

In a **deadlock scenario**, only **non-preemptable resources** are **relevant**, as they **cannot be forcibly reassigned** from one process to another without causing **malfunction or inconsistency**.

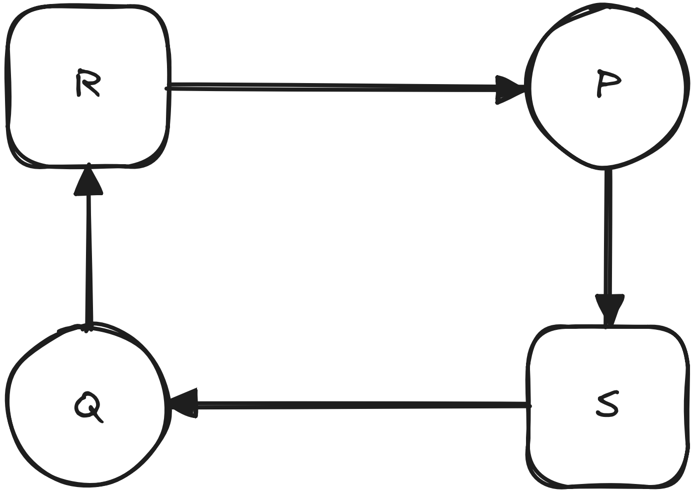
Conversely, **preemptable resources** can always be **reclaimed** and **reallocated** to ensure that other processes can make progress, thereby preventing deadlock.

Using this conceptual framework, it is possible to develop a **graphical notation** that **schematically represents deadlock situations**, illustrating the dependencies and potential circular wait conditions that lead to system stagnation.

Deadlock Characterization

process P holds the resource R

Process P requires the resource S



typical deadlock situation
(the most simplest one)

Necessary Conditions for Deadlock

It can be formally demonstrated that **deadlock** can only occur when all **four necessary conditions** are simultaneously present:

1. **Mutual Exclusion** – Each resource is either **free** or **exclusively assigned** to a single process; resource **holding cannot be shared** among multiple processes.
2. **Hold and Wait (Waiting with Retention)** – A process that **requests a new resource** continues to **hold all previously allocated resources**, instead of releasing them before making new requests.
3. **No Preemption (Non-Liberation)** – A resource that has been **allocated to a process** cannot be **forcibly taken away** by the system; **only the process itself** can decide when to release it.
4. **Circular Wait (Vicious Circle)** – A **circular chain** of processes and resources exists, where **each process in the chain is waiting for a resource currently held by the next process**, forming an inescapable dependency loop.

Deadlock Prevention

The **necessary conditions** for **deadlock occurrence** can be expressed formally as:

$$\begin{aligned}\text{deadlock occurence} \Rightarrow\ & \text{mutual exclusion } \textbf{and}\\ & \text{hold and wait } \textbf{and}\\ & \text{no preeemption } \textbf{and}\\ & \text{circular wait}\end{aligned}$$

This statement is **logically equivalent** to its **contrapositive**:

$$\begin{aligned}\text{no deadlock occurence} \Rightarrow\ & \neg \text{ mutual exclusion } \textbf{or}\\ & \neg \text{ hold and wait } \textbf{or}\\ & \neg \text{ no preeemption } \textbf{or}\\ & \neg \text{ circular wait}\end{aligned}$$

Deadlock Prevention

Thus, **to prevent deadlock**, it is sufficient to **eliminate at least one** of these necessary conditions. Strategies that enforce this principle are known as **deadlock prevention policies**.

The first condition, **mutual exclusion**, is generally **too restrictive to deny** because it applies only to **non-preemptable resources**. If mutual exclusion is **removed for preemptable resources**, **race conditions** are introduced, potentially leading to **data inconsistency** and **information loss**.

Deadlock Prevention

A **typical example** of denying mutual exclusion is allowing **multiple processes** to perform **simultaneous read operations** on a **shared file**. In contrast, it is also common to allow **only a single process to write at a time** while still permitting **concurrent read access**. However, even in this case, **race conditions cannot be entirely ruled out**.

Why? Because if **a read operation and a write operation overlap**, a **reader might access inconsistent or partially modified data**, leading to **data corruption or unexpected behavior**.

For this reason, **deadlock prevention policies** typically focus on **denying one of the last three conditions (hold and wait, no preemption, or circular wait)** rather than **mutual exclusion**.

Deny “hold and wait”

A process must **request all the resources it needs** for execution **at once**. If all required resources are **granted immediately**, the process is **guaranteed to complete** its execution **without further delays**. However, if the resources are **unavailable**, the process must **wait until they become available**.

It is important to note that this strategy does **not inherently prevent indefinite postponement (starvation)**. To ensure that **every process eventually acquires the necessary resources**, the system must implement **fair allocation mechanisms**.

A widely used approach to address **starvation** is the introduction of **aging policies**, which gradually **increase the priority** of a waiting process over time, ensuring that **it will eventually receive the required resources**.

Deny "no preemption"

If a process is **unable to acquire all the resources it requires**, it must **release all currently held resources** and **restart the request procedure from the beginning** once the resources become available.

An alternative approach is to **restrict a process to holding only one resource at a time**; however, this is a **specialized solution** that is **not applicable in most cases**, as many processes require multiple resources simultaneously to proceed.

To avoid **busy waiting**, a process must **block after releasing its resources** rather than continuously **repeating the request-acquire cycle**. It should **only be awakened when the required resources become available**, ensuring that system resources are used efficiently.

Deny “no preemption”

Despite this, **indefinite postponement (starvation) is still possible**, as a process might continuously fail to acquire the required resources due to competition. To prevent this, **fair resource allocation mechanisms** must be implemented. A widely used technique is **aging**, where the **priority of a waiting process gradually increases over time**, ensuring that it will **eventually be granted access to the necessary resources**.

Deny “circular wait”

This approach involves **establishing a strict linear ordering** of resources and enforcing a policy where **processes must request resources in increasing order** based on their assigned numerical values.

By following this **ordered allocation strategy**, the system **prevents the formation of circular wait conditions**, as no process can hold a lower-numbered resource while requesting a higher-numbered one in a way that could lead to a **deadlock cycle**.

Deny "circular wait"

However, it is important to note that **indefinite postponement (starvation) is not inherently prevented** by this method. A process may still **fail to acquire the necessary resources** due to **continuous contention**. To mitigate this issue, **fair allocation policies** must be implemented. A widely adopted solution is **aging**, where the **priority of a waiting process gradually increases over time**, ensuring that **it will eventually gain access to the required resources**.

Suggested Reading

Computer Organization and Architecture: Designing for Performance, Stalling W., 9th Edition, Prentice Hall, 2013

- Chapter 1: Introduction
- Chapter 2: Computer Evolution and Performance

The Art of High Performance Computing, Volume 1, 3rd edition 2022, formatted April 2, 2024

- Chapter 1: Single-processor Computing
- Chapter 2: Parallel Computing