Survey of Concurrency, Parallelism, and Asynchrony

Your Name

August 29, 2024

Introduction

This document provides a comprehensive overview of key concepts related to concurrency, parallelism, and asynchrony, with a focus on how these terms are used in the context of threading, task management, and the associated hardware and OS support.

Terminology

Concurrency

Definition: Concurrency refers to the ability of a system to handle multiple tasks at once. This does not necessarily mean that the tasks are executed simultaneously; rather, the system switches between tasks, making progress on each over time.

Key Characteristics:

- Concurrency can be achieved on a single-core CPU through context switching, where the operating system rapidly switches between tasks.
- Concurrency is useful in scenarios where tasks involve waiting, such as I/O operations, allowing other tasks to make progress in the meantime.

Examples:

- `std::thread` in C++ allows for concurrent execution of code, where each thread can perform tasks independently.
- Rust's `async/await` syntax, where multiple asynchronous tasks can be executed concurrently within a single thread.

OS/Hardware Support:

- The operating system's scheduler manages the execution of concurrent tasks by rapidly switching between them, giving the illusion of simultaneous execution.

- Concurrency does not require multiple cores, but it benefits from them.

Parallelism

Definition: Parallelism involves executing multiple tasks simultaneously, typically on multiple processors or cores. True parallelism requires hardware support in the form of multi-core CPUs or multiple CPUs.

Key Characteristics:

- Parallelism is ideal for CPU-bound tasks where tasks can be divided into smaller subtasks that can be executed simultaneously.
- Parallelism improves performance by utilizing multiple cores to perform computations at the same time.

Examples:

- In C++, `std::thread` can be used to run threads in parallel if there are enough CPU cores available.
- Rust's `Rayon` library provides parallel iterators that enable data parallelism, allowing operations like `map` and `reduce` to run in parallel.

OS/Hardware Support:

- The operating system schedules threads across multiple cores or processors, enabling true parallel execution.
- Modern CPUs with multiple cores or simultaneous multithreading (SMT) support parallel execution of threads.

Asynchrony

Definition: Asynchrony refers to the ability to initiate tasks that can be executed independently of the main program flow, allowing the program to continue executing without blocking. Asynchronous tasks may be executed concurrently or in parallel, depending on the implementation.

Key Characteristics:

- True asynchrony often requires non-blocking system calls, where the operating system or hardware handles tasks in the background, freeing the main program to perform other operations.
- Asynchrony is crucial for I/O-bound tasks, such as network communication or file I/O, where waiting for operations to complete would otherwise block the program.

Examples:

- In C++, `std::async` can be used to run functions asynchronously, with different launch policies (`std::launch::async` or `std::launch::deferred`).

- In Rust, asynchronous tasks are created using `async/await`, with executors like `tokio` or `async-std` managing the execution.

OS/Hardware Support:

- Asynchronous I/O operations are supported by the OS through non-blocking I/O APIs, such as POSIX AIO or Windows Overlapped I/O.
- Hardware interrupts can also trigger asynchronous behavior, allowing the CPU to respond to external events (e.g., network packets) without blocking.

Categories of Threading Primitives

Thread-Based

Definition: Thread-based concurrency involves creating and managing threads, where each thread represents a separate path of execution within a process.

Examples:

- `std::thread` in C++ and `std::thread::spawn` in Rust are used to create new threads.
- These threads can run concurrently or in parallel, depending on the number of available CPU cores.

Task-Based

Definition: Task-based concurrency abstracts the notion of threads, allowing developers to focus on defining tasks that are executed by an underlying thread pool or executor.

Examples:

- `std::async` in C++ launches tasks that are executed by the system, potentially on separate threads.
- In Rust, the `tokio` and `async-std` runtimes manage task execution, allowing tasks to be executed asynchronously.

Atomic Operations

Definition: Atomic operations are low-level operations that are guaranteed to be performed without interruption, ensuring thread safety without the need for locks.

Examples:

- In C++, `std::atomic` provides atomic operations on basic data types, ensuring safe concurrent access.

- Rust's `std::sync::atomic` module provides similar functionality for atomic operations in Rust.

OS/Hardware Support:

- Atomic operations are typically supported by CPU instructions, such as Compare-And-Swap (CAS), which ensure that the operation is completed atomically.

OS and Hardware Support

Operating System Support

Concurrency and Parallelism:

- The OS scheduler manages thread execution, balancing load across available CPU cores.
- The OS provides APIs for creating and managing threads, as well as synchronization primitives like mutexes, semaphores, and condition variables.

Asynchronous I/O:

- The OS supports asynchronous I/O operations through non-blocking syscalls, allowing programs to initiate I/O operations and continue executing.
- Examples include POSIX AIO on Unix-like systems and Overlapped I/O on Windows.

Hardware Support

Parallelism:

- Multi-core CPUs and multi-processor systems provide the hardware support necessary for parallel execution of threads.
- Simultaneous Multithreading (SMT), such as Intel's Hyper-Threading, allows multiple threads to run on the same core.

Atomic Operations:

- Modern CPUs provide atomic instructions, such as Compare-And-Swap (CAS), which are used to implement atomic operations in high-level languages.
- Hardware interrupts and DMA (Direct Memory Access) enable asynchronous behavior at the hardware level, allowing the CPU to perform other tasks while I/O operations complete.