Xebia

Virtual Threads

What is a Platform Thread ?

- **Operating System-Level Threads**: Platform threads, also known as kernel threads, are threads of execution managed and scheduled by the underlying operating system or platform.
- **Hardware and OS Dependency**: Characteristics and behavior of platform threads depend on the specific hardware architecture and operating system in use.
- **Efficient Multitasking**: Platform threads allow multiple tasks or processes to run concurrently, making efficient use of CPU processing power.
- **Thread Scheduling**: Operating system's scheduler determines which platform thread runs next based on factors like thread priority, time-sharing, and real-time constraints.
- **Concurrency and Parallelism**: Platform threads enable both concurrency (simultaneous execution of multiple threads) and parallelism (simultaneous execution on multiple CPU cores or processors).
- **Thread States**: Platform threads can be in states like "running" (actively executing), "waiting" (suspended until a condition is met), and "blocked" (waiting for a resource or event), managed by the scheduler.
- **User-Level Threads**: In some cases, user-level threads may be mapped to or managed by platform threads, with user-level threads managed by a thread library or runtime.
- **Thread Safety and Synchronization**: Proper synchronization mechanisms are used to ensure safe access to shared resources when working with platform threads.
- **Concurrency Control**: Platform threads are essential for achieving concurrency control in multi-threaded and multi-process applications, allowing parallel execution of code segments.
- **Resource Management**: Platform threads play a crucial role in resource management, including memory, CPU time, and I/O resource allocation and deallocation.

What is a Virtual Thread ?

User-Level Threads: Virtual threads are user-level threads, managed by a runtime library or framework, rather than the operating system's kernel.

Decoupled from Platform Threads: They are decoupled from the underlying platform threads, allowing efficient multiplexing onto a smaller number of platform threads.

Concurrency and Parallelism: Virtual threads provide concurrency and parallelism, enabling multiple threads to execute concurrently.

Cooperative Multitasking: They typically rely on cooperative multitasking, where threads voluntarily yield control to other threads, as opposed to preemptive multitasking used by platform threads.

Low Overhead: Virtual threads have low memory and resource overhead compared to platform threads, making them suitable for scenarios with many threads.

Lightweight Context Switching: Context switching between virtual threads is lightweight, as it doesn't involve the overhead of switching between platform threads.

Language and Framework Support: Virtual threads are often provided as part of programming languages or concurrency frameworks as libraries or language features.

Task Parallelism: They are well-suited for task parallelism, where tasks or functions are executed concurrently and asynchronously without a one-to-one mapping between threads and tasks.

Scalability: Virtual threads can efficiently scale to a large number of concurrent tasks, making them suitable for highly concurrent applications.

Simplified Code: Using virtual threads can simplify code, as developers don't need to manage low-level thread creation, synchronization, and thread pools.

Resource Efficiency: They are efficient in terms of memory usage and resource management, making them suitable for resource-constrained environments.

Use Cases: Virtual threads are suitable for applications with high concurrency, I/O-bound tasks, and scenarios where lightweight thread creation and management are desired.

Why use Virtual Threads

- **Lightweight**: Virtual threads are lightweight compared to platform threads, making them suitable for scenarios where you need to create and manage many concurrent tasks efficiently.
- **Efficient Multiplexing**: Virtual threads can be multiplexed onto a smaller number of platform threads, which is more resource-efficient.
- **Task Parallelism**: They are well-suited for task parallelism, where tasks or functions are executed concurrently and asynchronously, without a one-to-one mapping to platform threads.
- **Simplified Code**: Virtual threads simplify concurrent code development by abstracting away low-level thread management complexities, making code easier to read and maintain.
- **Concurrency Control**: They help achieve efficient concurrency control without the need for manual thread management.
- **Scalability**: Virtual threads can efficiently scale to handle many concurrent tasks, which is crucial for applications with varying workloads.
- **Resource Efficiency**: They are resource-efficient, making them suitable for resource-constrained environments like mobile and IoT devices.
- **I/O-Bound Operations**: Ideal for I/O-bound operations where tasks spend time waiting for external resources, allowing other tasks to run during the waiting period.
- **Enhanced Responsiveness**: Virtual threads can enhance application responsiveness by executing long-running or blocking operations in the background, ensuring the application remains responsive to user interactions.
- **Compatibility**: They can be used alongside traditional platform threads, allowing you to leverage their benefits while maintaining compatibility with existing code and libraries.

Creating & Running a Virtual Thread

Choose a Concurrency Framework: To create and run virtual threads, you typically need to choose a concurrency framework or library that supports this concept. Examples include Kotlin Coroutines, Java's Project Loom, and Python's asyncio.

Create a Virtual Thread: Use the framework's provided syntax or API to create a virtual thread. This often involves defining a block of code that represents the behavior of the virtual thread.

Define Virtual Thread Behavior: Inside the virtual thread, specify the code that should be executed concurrently. This can include tasks, functions, or operations that run asynchronously.

Concurrency Management: The concurrency framework manages the execution of virtual threads, ensuring they run concurrently without the need for manual thread management.

Synchronization (Optional): If needed, you can use synchronization mechanisms provided by the framework to coordinate and synchronize the execution of virtual threads.

Wait for Completion (Optional): Optionally, you can wait for the virtual thread to complete its execution. This ensures that the main thread or other threads wait for the virtual thread to finish before proceeding.

Resource Efficiency: Virtual threads are typically lightweight and resource-efficient, making them suitable for scenarios where many concurrent tasks need to be managed efficiently.

Simplified Code: Virtual threads abstract away low-level thread management complexities, allowing developers to write asynchronous and concurrent code in a more structured and readable manner.

Enhanced Responsiveness: By using virtual threads, applications can remain responsive to user interactions, as long-running or blocking operations can be executed in the background.

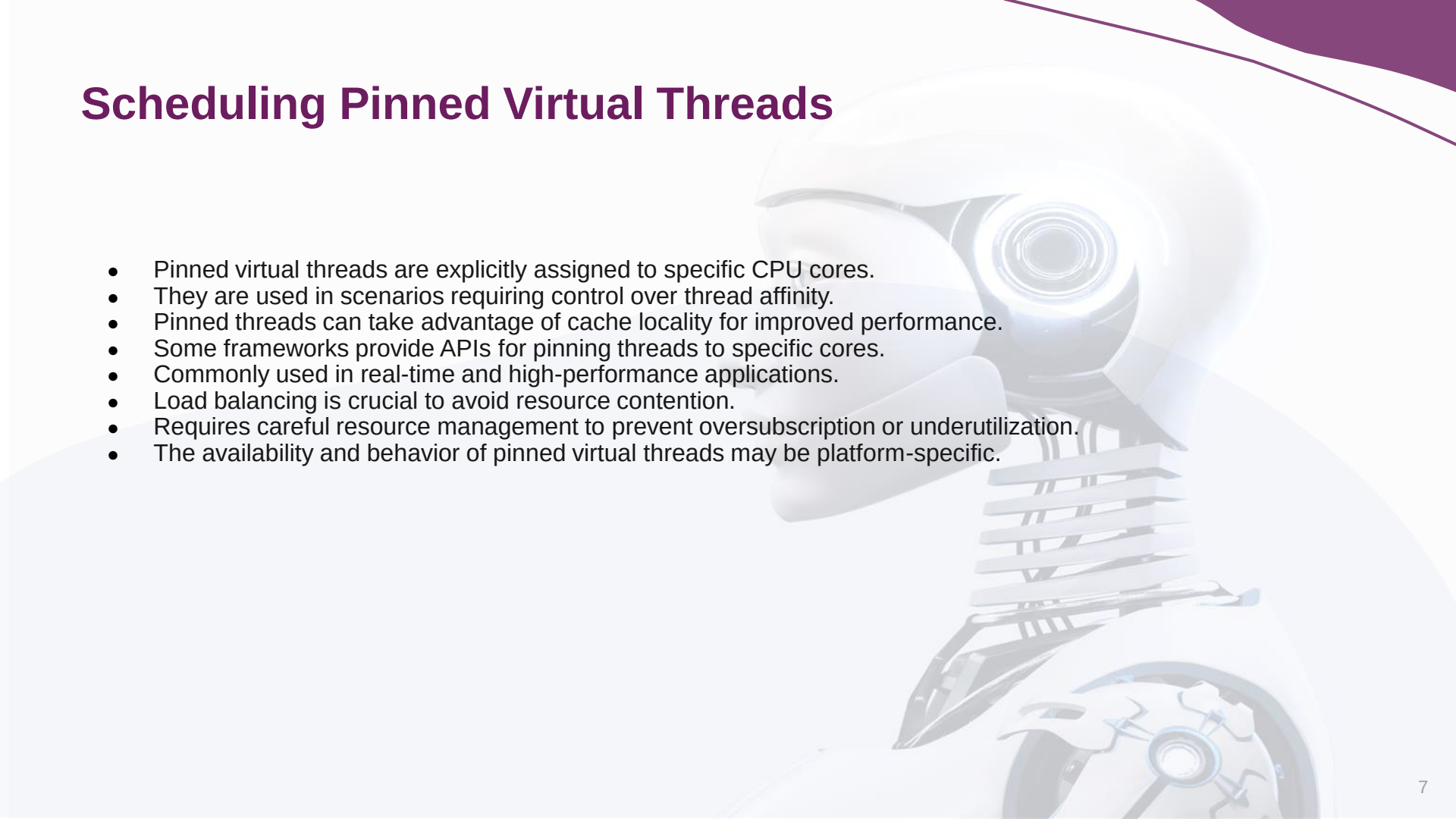
Compatibility: Virtual threads can often be used alongside traditional platform threads, allowing you to leverage their benefits while maintaining compatibility with existing code and libraries.

Scheduling Virtual Threads

- Virtual threads are typically managed by a concurrency framework or library.
- A scheduler determines when and how virtual threads are executed.
- Concurrency control is handled by the scheduler, allowing threads to run concurrently.
- Cooperative multitasking is often used, where threads yield control voluntarily.
- Task prioritization may be available to influence the order of execution.
- Task queues are used to manage and schedule tasks or coroutines.
- Resource-efficient scheduling enables efficient execution of many concurrent tasks.

Scheduling Pinned Virtual Threads

- Pinned virtual threads are explicitly assigned to specific CPU cores.
- They are used in scenarios requiring control over thread affinity.
- Pinned threads can take advantage of cache locality for improved performance.
- Some frameworks provide APIs for pinning threads to specific cores.
- Commonly used in real-time and high-performance applications.
- Load balancing is crucial to avoid resource contention.
- Requires careful resource management to prevent oversubscription or underutilization.
- The availability and behavior of pinned virtual threads may be platform-specific.



xebia.com