Multiprocessing Introduction

Why Concurrency?

- Writing correct **concurrent** programs is harder than sequential ones, but threads simplify complex asynchronous workflows into clearer, straight-line code.
- Threads exploit **multiprocessor systems**; as core counts rise, effective concurrency matters more.

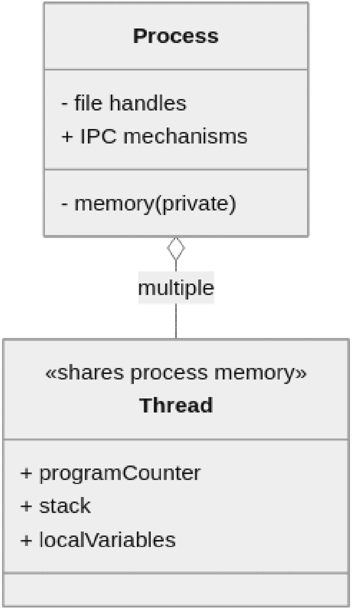
A (Very) Brief History of Concurrency

- Early systems ran a **single program** directly on bare metal—inefficient and hard to develop.
- Operating systems introduced **processes** to improve **resource utilization**, **fairness**, and **convenience**.
- **Threads** inside a process share memory, enabling fine-grained data sharing and **hardware parallelism**.

Processes vs Threads

- **Processes**: isolated address spaces; coarse-grained communication (sockets, shared memory, semaphores, files).
- **Threads**: share address space; each has program counter & stack; easy to schedule on multiple CPUs.
- Without synchronization, shared data access yields **unpredictable interleavings** and **incorrect results**.

Processes vs Threads

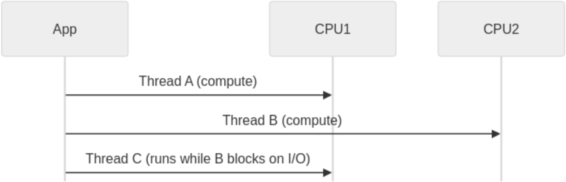


Benefits of Threads

- **Exploit multiple processors**: parallel execution increases throughput; even single-CPU systems benefit by overlapping I/O waits.
- **Simplicity of modeling**: decompose complex async workflows into simpler synchronous ones per thread.
- **Simplified async handling**: thread-per-task/client model often avoids complicated non-blocking I/O.
- **Responsive UIs**: offload long tasks from the event thread.

Exploiting Multiple Processors

- Single-threaded programs use only **one CPU** at a time.
- Overlap **blocking I/O** with useful work in other threads.

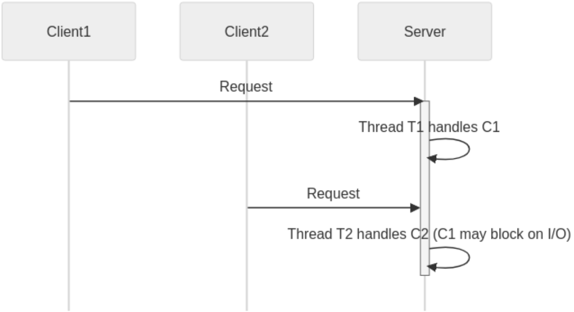


Simplicity of Modeling

- Assign a **thread per task type** (or per simulation element) to insulate domain logic from scheduling and interleaving details.
- Frameworks (e.g., servlets, RMI) let you write **straight-line** request handlers while the framework manages threads & load balancing.

Handling Asynchronous Events

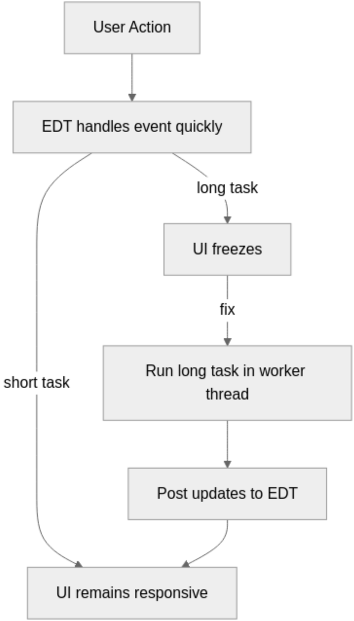
- **Thread per connection** in servers allows synchronous I/O without stalling other requests when one blocks.
- Modern OS support makes **large thread counts** practical in many cases.



More Responsive User Interfaces

- AWT/Swing use an **Event Dispatch Thread (EDT)**; long tasks in the EDT freeze the UI.
- Run long tasks in background threads; post UI updates back to the EDT.

More Responsive User Interfaces



Risks of Threads

- **Safety hazards**: races due to unpredictable interleavings; shared state must be **properly coordinated**.
- **Liveness hazards**: deadlock, starvation, livelock—progress can halt.
- **Performance hazards**: context switching, cache invalidations, synchronization overhead, reduced locality.

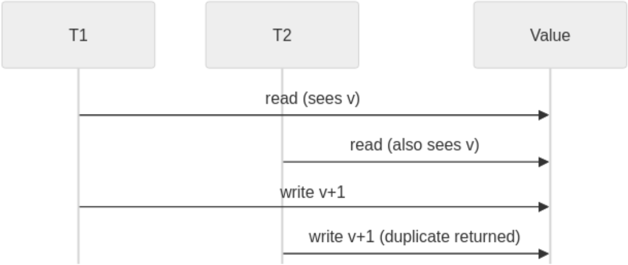
Safety Hazard Example — Race Condition

- `UnsafeSequence` attempts to generate unique integers but is **not thread-safe**; `value++` is read+add+write, which can interleave.

```
@NotThreadSafe
public class UnsafeSequence {
    private int value;

    /** Returns a unique value. */
    public int getNext() {
        return value++;
    }
}
```

Safety Hazard Example — Race Condition



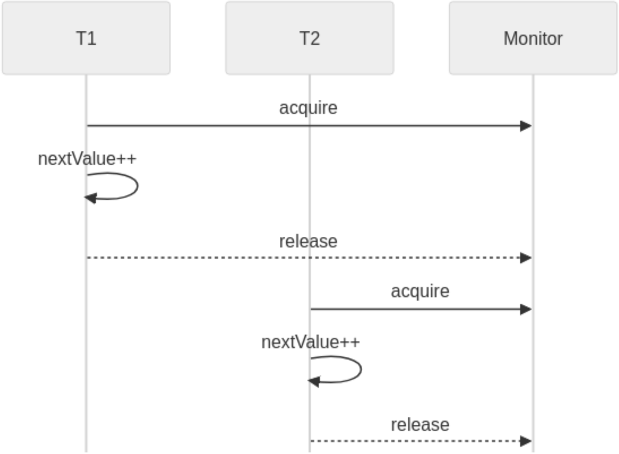
Fixing the Race with Synchronization

- Making `getNext` **synchronized** serializes access; synchronization is essential for correctness & visibility.

```
@ThreadSafe
public class Sequence {
    @GuardedBy("this")
    private int nextValue;

    public synchronized int getNext() {
        return nextValue++;
    }
}
```

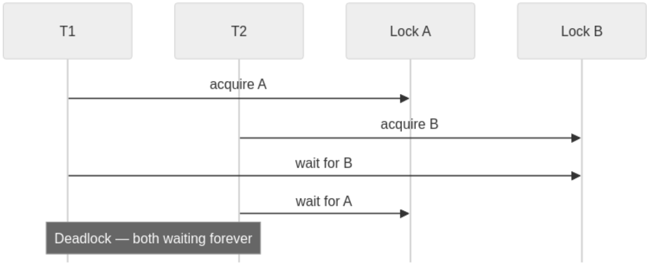
Fixing the Race with Synchronization



Liveness Hazards

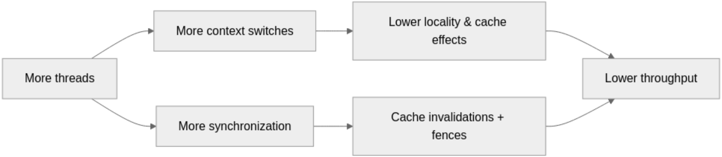
- **Deadlock**: two threads wait on locks held by each other—neither can proceed.
- **Starvation**: a thread never gets CPU or resources due to scheduling or contention.
- **Livelock**: threads keep responding to each other but make no progress.

Deadlock Example



Performance Hazards

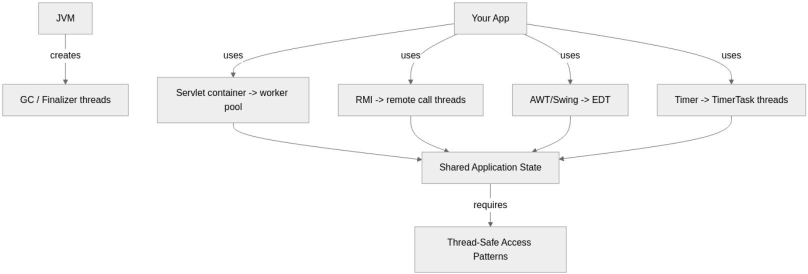
- **Context switches** add overhead (save/restore state), reduce locality, and consume scheduler time.
- Synchronization can **inhibit optimizations**, flush/invalidate caches, and create traffic on shared memory buses.



Threads Are Everywhere

- Frameworks and the JVM create threads: **GC/Finalizer**, **Timer tasks**, **Servlet pools**, **RMI**, **AWT/Swing EDT**.
- Concurrency introduced by frameworks **ripples** through your app; any code accessing shared state must be **thread-safe**.

Threads Are Everywhere



Practical Guidance

- Identify shared state early; **document** synchronization policies (e.g., `@GuardedBy`) and enforce them.
- Prefer **clear ownership** & confinement; use thread-safe utilities and frameworks thoughtfully.
- Keep the **EDT** free; marshal long-running work to background threads and post updates to the EDT.

Key Takeaways

- Threads unlock performance and modeling simplicity—but require **discipline** for safety, liveness, and performance.
- Concurrency is **ubiquitous** in modern Java; frameworks will create threads and call your code—be prepared.
- Use **synchronization** (or higher-level concurrency constructs) to make correctness and visibility explicit.