Why User-Mode Threads Are Often the Right Answer

Ron Pressler
Java Platform Group
14 April 2021
Java Is Made of Threads

- Exceptions
- Thread Locals
- Debugger
- Profiler (JFR)
Threads in Java

- `java.lang.Thread`
- One implementation: OS threads
- OS threads support all languages.
- RAM-heavy — megabyte-scale; page granularity; can’t uncommit.
- Task-switching requires switch to kernel.
- Scheduling is a compromise for all usages. Bad cache locality.
Synchronous

• Easy to read
• Fits well with language (control flow, exceptions)
• Fits well with tooling (debuggers, profilers)

But

• A costly resource
Concurrency

\[ L = \lambda W \]
Reuse with Thread Pools
Reuse with Thread Pools

• Return at end
  • Leaking ThreadLocals
  • Complex cancellation (interruption)
Reuse with Thread Pools

• Return at end
  • Leaking ThreadLocals
  • Complex cancellation (interruption)
• Return at wait
  • Incompatible APIs
  • Lost context
Asynchronous

- Scalable

But

- Hard to read
- Lost context: Very hard to debug and profile
- Intrusive; nearly impossible to migrate
simple
less scalable

scalable,
complex,
non-interoperable,
hard to debug/profile

OR

SYNC
Programmer
OS / Hardware

ASYNC
Programmer
OS / Hardware
Codes Like Sync, Scales Like Async
“We must carefully balance conservation and innovation”
— Mark Reinhold

• **Forward Compatibility**: we want existing code to enjoy new functionality
• We want to **correct past mistakes** and start afresh

“The solutions of *yesterday* are the problems of *today*”
— Brian Goetz
Threads in Java

• The use of `Thread.currentThread()` and `ThreadLocal` is pervasive. Without support, or with changed behaviour, little existing code would run.

• Other parts are superseded by new APIs since Java 5 so their datedness/clunkiness is mostly hidden/ignored.
Threads in Java

- `java.lang.Thread`
- The Java runtime is well positioned to implement threads.
- Resizable stacks (possible b/c we only need to support Java).
- Context-switching in user-mode.
- Pluggable schedulers, default optimised for transaction processing.
Threads in Java

When code in a virtual thread calls an I/O method in the JDK, suspend the virtual thread, start a non-blocking I/O operation in the OS, the scheduler schedules another virtual thread, when I/O completes re-submit waiting thread to scheduler.
Module java.base  
Package java.util.concurrent  

Class ConcurrentHashMap<K,V>  
java.lang.Object  
java.util.AbstractMap<K,V>  
java.util.concurrent.ConcurrentHashMap<K,V>  

Type Parameters:  
K - the type of keys maintained by this map  
V - the type of mapped values  

All Implemented Interfaces:  
Serializable, ConcurrentMap<K,V>, Map<K,V>  

public class ConcurrentHashMap<K,V>  
extends AbstractMap<K,V>  
implements ConcurrentMap<K,V>, Serializable  

A hash table supporting full concurrency of retrievals and high expected concurrency for updates. This class 
obey the same functional specification as Hashtable, and includes versions of methods corresponding to each 
method of Hashtable. However, even though all operations are thread-safe, retrieval operations do not entail 
locking, and there is no support for locking the entire table in a way that prevents all access. This class is 
fully interoperable with Hashtable in programs that rely on its thread safety but not on its synchronization.

Module java.base  
Package java.nio.channels  

Class SocketChannel  
java.lang.Object  
java.nio.channels.SocketChannel  

All Implemented Interfaces:  
Closeable, AutoCloseable, ByteChannel, Channel, GatheringByteChannel, InterruptibleChannel, 
NetworkChannel, ReadableByteChannel, ScatteringByteChannel, WritableByteChannel  

public abstract class SocketChannel  
extends AbstractSelectableChannel  
implements ByteChannel, ScatteringByteChannel, GatheringByteChannel, NetworkChannel  

A selectable channel for stream-oriented connecting sockets.  

A socket channel is created by invoking one of the open methods of this class. It is not possible to 
create a channel for an arbitrary pre-existing socket. A newly-created socket channel is open but 
not yet connected. An attempt to invoke an I/O operation upon an unconnected channel will cause a 
NotYetConnectedException to be thrown. A socket channel can be connected by invoking its 
connect method; once connected, a socket channel remains connected until it is closed. Whether or not a 
socket channel is connected may be determined by invoking its

Module java.base  
Package java.util.concurrent.locks  

Class ReentrantLock  
java.lang.Object  
java.util.concurrent.locks.ReentrantLock  

All Implemented Interfaces:  
Serializable, Lock  

public class ReentrantLock  
extends Object  
implements Lock, Serializable  

A reentrant mutual exclusion Lock with the same basic behavior and semantics as the implicit monitor lock 
across using synchronized methods and statements, but with extended capabilities.  

A ReentrantLock is owned by the thread last successfully locking it, but not yet unlocking it. A thread 
invoking lock will return, successfully acquiring the lock, when the lock is not owned by another thread. 
The method will return immediately if the current thread already owns the lock. This can be 
checked using methods isHoldLock(), isHeldByCurrentThread(), and getHoldCount(). 

The constructor for this class accepts an optional fairness parameter. When set true, under contention, locks 
favor granting access to the longest-waiting thread. Otherwise this lock does not guarantee any particular 
access order. Programs using fair locks accessed by many threads may display lower overall throughput, 
(i.e., are slower)

Module java.base  
Package java.io  

Class InputStream  
java.lang.Object  
java.io.InputStream  

All Implemented Interfaces:  
Closeable, AutoCloseable  

Direct Known Subclasses:  
AudioInputStream, ByteArrayInputstream, FileInputStream, FilterInputStream, ObjectInputStream, 
PipedInputStream, SequenceInputStream, StringBufferInputStream  

public abstract class InputStream  
extends Object  
implements Closeable  

This abstract class is the superclass of all classes representing an input stream of bytes.  

Applications that need to define a subclass of InputStream must always provide a method that returns the 
next byte of input.  

Since: 1.6
virtual threads

“carrier” platform threads managed by a scheduler
async/await

C#
JavaScript
Kotlin
C++/Rust

User-Mode Threads

Erlang
Go
Java

Zig
Algorithm (semantic)
(an abstract description of) *What* the computer does

Expression (syntactic)
*How* the algorithm is written (in a specific programming language/paradigm)
Concurrency: The Algorithmic View

Schedule multiple largely independent tasks to a set of computational resources

Performance: throughput (tasks/time unit)

Parallelism: The Algorithmic View

Speed up a task by splitting it to sub-tasks and exploiting multiple processing units

Performance: latency (time unit)
Concurrency: The Syntactic View

• ; — Sequential composition
  E.g. \( X;Y, \text{await } X;Y, X.\text{andThen}(Y) \)

• | — Parallel composition
  E.g. \( \text{Thread.start}(X), \text{Promise.submit}(X) \)

• \((a|b);c\) — join
  E.g. \( \text{Thread.join}, \text{Future.get} \)

• assignment/channels/locks/IO

\[
a;((b;c)|(d|(e;f));g));h
\]
Process: Unit of Concurrency

E.g. a transaction

- Code (writing/reading)
- Troubleshooting: stack traces, debugger single-stepping
- Profiling
Process

\[ a; b; c; d = (a; b); (c; d) \]

(Nondeterminism https://youtu.be/9vupFNsND6o )
foo() {
    ...
    bar();
    ...
}

bar() {
    ...
    baz();
    ...
}

baz() {
Thread

Call Stack

Thread

foo() {
    ...
    bar();
    ...
}

bar() {
    ...
    baz();
    ...
}

baz() {
    ...
}
async foo() {
  ...
  await bar();
  ...
}

async bar() {
  ...
  await baz();
  ...
}

async baz() {
  ...
}
Thread vs. Async/Await

Scheduling/interleaving points

**Thread**: Everywhere except where explicitly forbidden (with a CS)

**async/await**: Nowhere except where explicitly allowed (with `await`)
Thread vs. Async/Await

Scheduling/interleaving points

**Thread:** Everywhere *except* where explicitly forbidden (with a CS)

**async/await:** Nowhere *except* where explicitly allowed (with `await`)
Thread vs. Async/Await

Implementation

**Thread:** Requires integrating with the implementation of subroutines (control over backend)

**async/await:** Can be implemented in the compiler frontend
Thread vs. Async/Await

Implementation

Thread: Requires integrating with the implementation of subroutines (control over backend)

async/await: Can be implemented in the compiler frontend
Thread vs. Async/Await

Recursion & virtual calls

**Thread**: Yes (requires large/resizable stacks)

**async/await**: Can be excluded
Thread vs. Async/Await

Recursion & virtual calls

Thread: Yes (requires large/resizable stacks)

async/await: Can be excluded

C++/Rust
Resizable Stack

- Transparent allocation
- Efficient allocation
- No internal pointers/tracked pointers (no FFI)
Performance

**Latency** — How long an operation takes (s)

**Throughput** — How many operations complete per time unit (ops/s)

**Impact** — How much a metric would improve with full optimisation (%)
Syntactic Concurrency: Generators et al.

- Updating simulation entities in a frame
- Generators (two processes with an unbuffered channel)

```python
def rev_str(my_str):
    length = len(my_str)
    for i in range(length - 1, -1, -1):
        yield my_str[i]

for char in rev_str("hello"):  
    print(char)
```
Context-Switching Impact: Generators

- Impact: 100%
- Best case latency: ~0ns (monomorphic, fits in cache)
Concurrency: Transactions

\[ L = \lambda W \]
Throughput: \( \lambda = \frac{L}{W} \)

Context-switch impact on throughput: \( \frac{t}{\mu} \)

- \( t \) — Mean context-switch latency
- \( \mu \) — Mean wait (I/O) latency
Context-Switching Impact: Transactions

- Impact: low if blocking for external events
- Best case latency: 60ns (polymorphic, doesn’t fit in cache) (1.5% impact)
- Target latency for ≤5% impact: ≤200ns
Conclusion

• Control over backend
• Rare I/O in FFI
• No internal pointers/pointers tracked
• Efficient and transparent stack resizing
• Threads already in the platform, libraries and tooling
async/await

User-Mode Threads

C#

JavaScript

Kotlin

C++/Rust

Erlang

Go

Java

Zig
Thank you