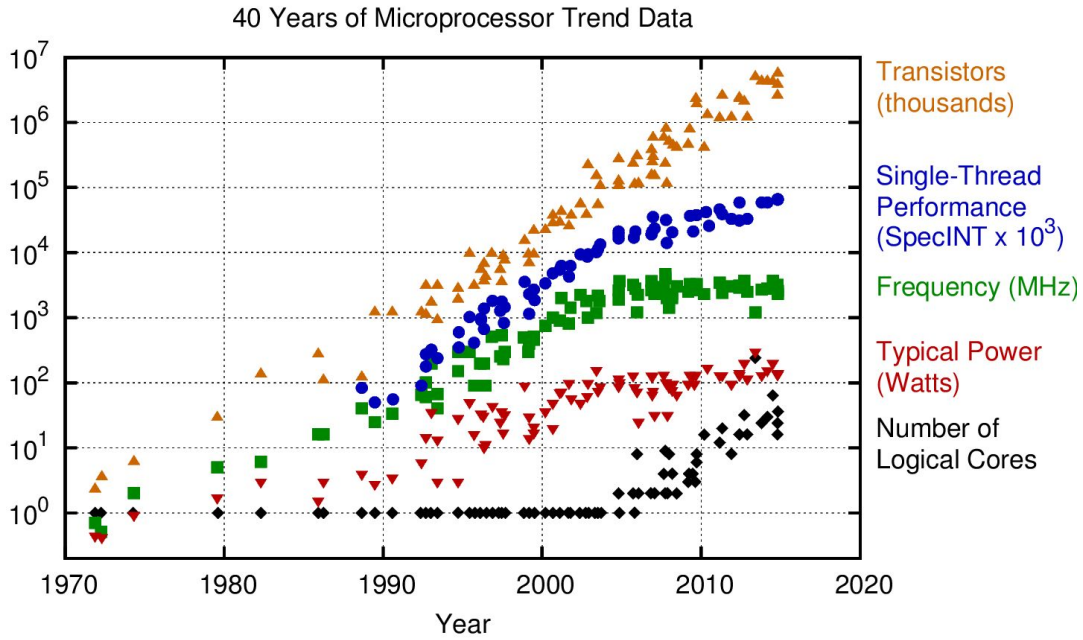
Concurrency Essentials

with Java Samples

Motivation

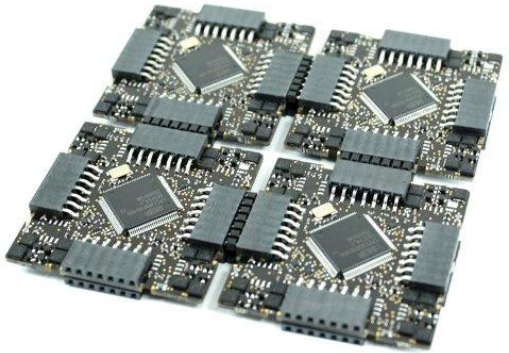
Moore's law still works for the number of transistors, but it doesn’t work for performance of a single thread anymore. So it’s reasonable to scale horizontally increasing number of threads.



Parallelism

A *multiprocessor* consists of multiple hardware processors, each of which executes a sequential program.

A *thread* is a sequential program which represents a unit of parallelism. While a *processor* is a hardware device, a thread is a software construct. A processor can run a thread for a while and then set it aside and run another thread, an event known as a *context switch*.

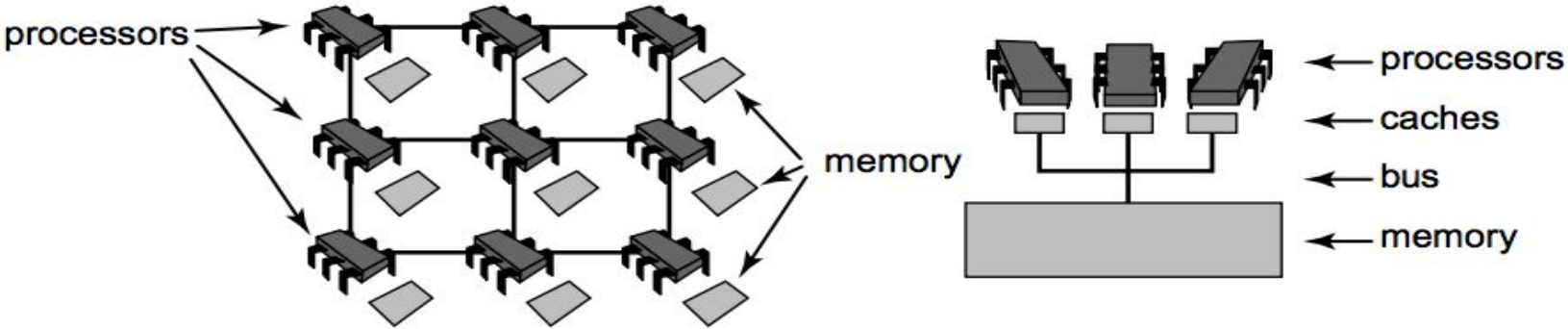


Modern processor architectures combine multi-core with multi-threading, where multiple individually multi-threaded cores may reside on the same chip. The context switches on some multi-core chips are inexpensive and are performed at a very fine granularity, essentially context switching on every instruction.

Cache Coherence

Processors and main memory are far apart. It takes a long time for a processor to read a value from memory. It also takes a long time for a processor to write a value to memory, and longer still for the processor to be sure that value has actually been installed in memory. Accessing memory is more like mailing a letter than making a phone call.

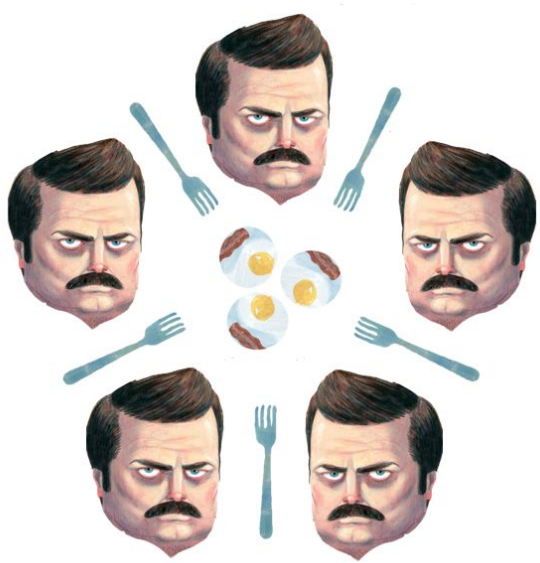
Both processor and memory speed change over time, but their relative performance changes slowly.



Resources Contention

Conflict over access to a shared resources between OS threads.

- I/O
- CPU cycles
- Memory



In Java a class state is a shared resource.
So stateless classes do not contend for resources.

Thread Safety

A thread safe class is one that is no more broken in concurrent environment than in a single-threaded environment.

A class is thread safe if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code.

Stateless objects are always thread safe.

Basic Java Concurrency Primitives

Parallelism

- java.lang.Thread
- java.lang.Runnable
- java.lang.Callable

Resources contention

- synchronized
- volatile
- java.lang.Object.wait()
- java.lang.Object.notify()
- java.lang.Object.notifyAll()
- ThreadLocal

Visibility

To publish an object safely, both the reference to the object and the object's state must be made visible to other threads at the same time. A properly constructed object can be safely published by:

- Initializing an object reference from a static initializer
- Storing a reference to it into a volatile field or AtomicReference
- Storing a reference to it into a final field of a properly constructed object
- Storing a reference to it into a field that is properly guarded by a lock

Atomicity

Locking allows to perform a sequence of actions atomically without interleaving by other threads.

Typical actions which should be performed atomically : read-modify-write, check-then-act. Each java object can be used as a lock. Locking can guarantee both visibility and atomicity; volatile variables can only guarantee visibility. There are synchronized wrappers for major collections which employ locking.

Building Blocks (Collections)

More sophisticated concurrency primitives built on basic primitives.

- ConcurrentHashMap
- CopyOnWriteArrayList
- ArrayBlockingQueue
- PriorityBlockingQueue
- LinkedBlockingQueue
- ConcurrentLinkedQueue

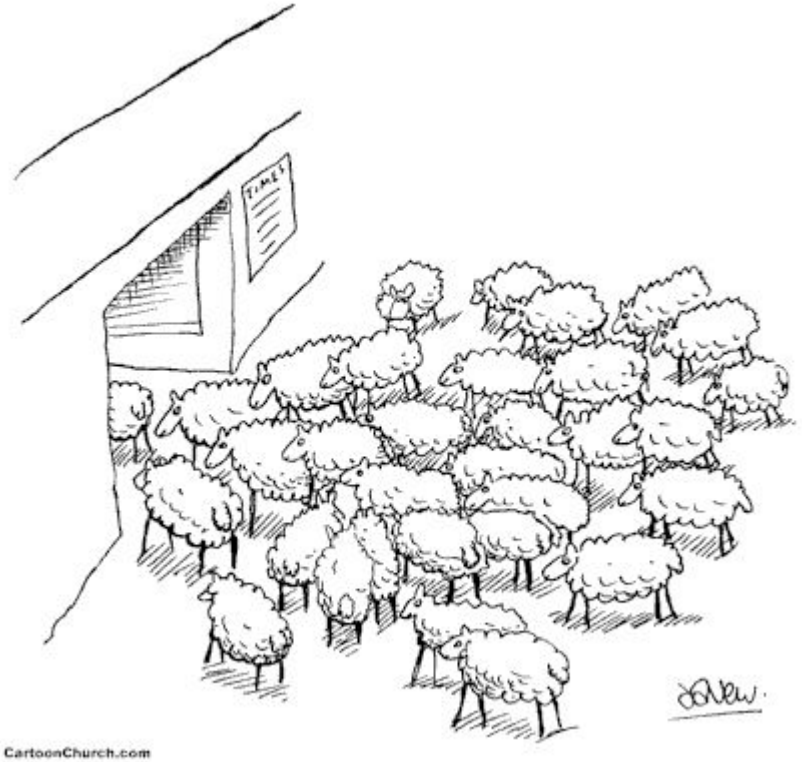
Building Blocks (Synchronizers)

More sophisticated concurrency primitives built on basic primitives.

- CountDownLatch
- CyclicBarrier
- Semaphore
- Phaser

Thread Pools

- Executor
- ExecutorService
- ScheduledExecutorService
- ThreadPoolExecutor
- ForkJoinPool
- Executors
- Future
- RunnableFuture
- FutureTask
- RecursiveTask
- RecursiveAction
- CompletableFuture



Interruption

- Propagate the exception (possibly after some task specific cleanup), making your method an interruptible blocking method, too
- Restore the interruption status so that code higher up on the call stack can deal with it.

- Thread.interrupt
- Thread.isInterrupted
- Thread.interrupted
- InterruptedException
- Future.cancel()
- ExecutorService.shutdownNow

Explicit Locks

- Lock
- ReentrantLock
- ReadWriteLock
- Condition

The rule of thumb

Avoid premature optimization. First make it right, then make it fast if it is not already fast enough.

Many performance optimizations come at the cost of readability or maintainability the more "clever" or non obvious code is, the harder it is to understand and maintain.