Parallel Programming, FS2023

Luzia Kündig

March 6, 2023

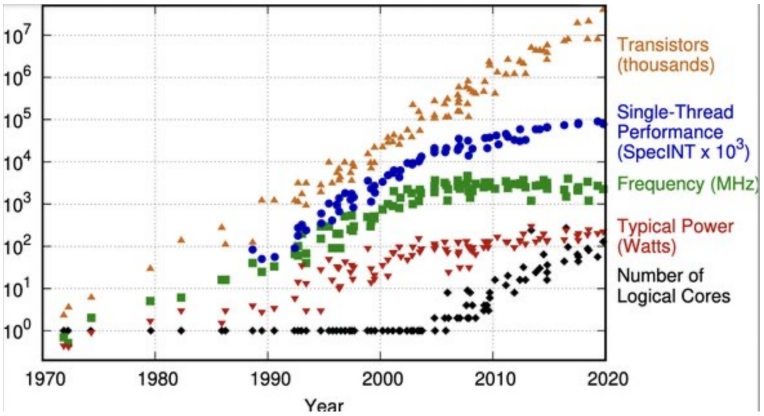

Figure 1: CPU Statistics

1 Week 1 Introduction

Modern hardware does not run on single cores anymore.

Moores Law: The number of transistors that can be packed into a given unit of space will double about every two years. Which means doubling the speed of a computer every two years.

Physical limits of this law have been reached already. New ways of speeding up processors are needed.

1.1 Parallelism

Better CPU utilization, more responsive programs. Divide / modularize tasks of our program. Better software division.

Hyperthreading: two register sets to keep two separate contexts. If one task is busy, switch to other context can happen extremely fast. Creates two logical cores from one physical core. More efficient usage of a single hardware construct, use wait time.

Performance increase is *not* 100% the amount of logical cores.

1.2 Parallel vs Concurrent

Parallel: different Processors, at the same time.
Concurrent: Time-shared, switches between two contexts needed. One actor.

After the end of moores law: parallelization is needed for more speedup.

1.3 OS-Level Parallelism

1.3.1 Process vs. Thread

Process: separates memory completely from other threads. context switches are slow

Threads: Each has their own stack, but all threads inside a process share the same heap. Access to this storage must be synchronized.

1.3.2 User- vs Kernel Threads

Only kernel-level threads let you exploit the parallelism speedup on multiple cores.

User-threads (green threads) only live within Application. No true parallelism.

JVM always launches a kernel-level thread.

1.4 Thread Scheduling

This is concurrency: Interleaved execution.

Each processor can execute one thread at a time. Multiple Threads are managed via scheduling mechanisms.

States: Running, ready, waiting.

Context switches are "lightweight" but still have a performance impact.

synchron (cooperative) waiting for condition, queues itself as waiting
asynchron (preemptive) resource are released after a set amount of time

1.5 JVM Thread Model

intra-process communication

Java Virtual Machine is run as one single process. Main method is called from a startup thread created by the JVM (garbage collectors etc. start their own threads).

JVM process runs as long as all started threads are running. Exception: daemon threads are aborted ad JVM end.

Runtime.Exit()/System.exit() ends jvm abruptly

1.5.1 Java Threading

Interface Runnable
Class Thread (implements Runnable)

Call Thread(Runnable behaviour);
var myThread = new Thread();
myThread.start();
—

Only guarantee possible: statements inside one thread will always be executed in correct order.

Java important: sth.start() starts a new thread and calls the .run() method. if run() is called by the user, the runnable action will be run sequentially.

1.5.2 Non-Determinism

No control: Threads run in any order without any precautionary rules. println in some JVMs is one synchronized (atomar) operation and will not be broken apart. Not always, implementation specific.

1.5.3 Variants of Running Threads

Explicit Runnable Implementation: write named function that implements Runnable Interface. Sub-Class of Thread: new Class with custom implementation of the run function

1.5.4 Thread Methods

Thread Join:
t2.join() blocks t1 from running as long as t2 is still running.

Thread Sleep:
running thread goes into waiting state until the time has passed for it to be ready again.

Thread Yield:
running thread releases the processor but will directly be ready again. not really necessary in time-shared / preemptive scheduling of the scheduler.

Thread Interrupt:

currentThread:

setDaemon:

Quiz

currentThread().Join() creates a deadlock.

Thread marked as daemon can effectively be joined, making it "normal" again.

2 Week 2 Monitor Synchronization

Race Condition: 2 Threads try to update the same shared (on heap) resource. Outcome: either one of the threads can write, the other update gets lost.

`this.balance += amount` gets converted into three separate instructions. They will not be executed atomically.

Synchronization: restriction of concurrency.

Critical section: only one thread at a time can enter and execute this part of the code.

Synchronized Methods with Monitor

Java: Keyword `synchronized` in a method header guarantees that the method body is executed atomically.

Every object has a (Monitor-) Lock, which is acquired when any synchronized method is accessed. `static` methods acquire the class objects lock which there is only one of.

Internal mutual exclusion, only one thread operates at a time. all not-private methodsa re synchronized

Wait & Signal Mechanism: Outer waiting room to acquire Monitor Lock. Inner Waiting Room to wait for condition to be fulfilled.

Two methods: `wait();` for condition, and `notifyAll();` if something in a variable has changed.

Monitor Lock is only freed on method end, not notify/All.

`notifyAll()` if the balance on bank account is increased by a deposit, every withdrawing thread should check their condition again, not just one..

`notify()` is enough, if two conditions are fulfilled:

- Uniform waiting condition (boolean): a change in the condition interests every waiting thread
- One in one out: only a single waiting thread can continue

while loop: Check should happen every time the thread has access to the Monitor, not just the first time. Thread execution continues from where it has left off.

Spurious Wakeup: thread wakes up out of a special reason, maybe woken by the OS.

2.1 Discussion

Advantage: very powerful concept, object oriented

Disadvantage: not always optimal. efficiency and fairness problems

3 Week 3 Synchronization Primitives

3.1 Semaphores

Is in essence a counter which defines the number of free resources. It's coupled with operations that adjust the record *safely*.

General Semaphore (0-n) vs. Binary Semaphore (Mutex, 0 or 1). Fairness can be enforced as FIFO queue:

```
new Semaphore(N, true);
```