COPENHAGEN BUSINESS ACADEMY





Concurrency and threads

Jens Egholm Pedersen
<jeep@cphbusiness.dk>

About me

- Political science from Aarhus University
- Software development from IT-University
- 4 years of work experience
  - Mainly in Java

- Generally a nice guy
  - Don’t be afraid to ask questions

About these lectures

- One-way communication
  - You receive information
  - Make sure you understand it!
- Exploit what we prepared for you
  - Bloom's Taxonomy
  - Lecture = Understanding
  - Lecture + Preparation = Analyzing
  - Lecture + Preparation + Exercises = Creating
- When studying for the exam use 'see also'
  - **Not part of the curriculum!**

See also: Something to read, Bloom's taxonomy

What you should know

Goal of todays and tomorrows lectures on threads

- Understand concurrency and develop concurrent applications
- Use mechanisms to synchronise threads
- Understand and identify deadlocks
- Understand and use the publish-subscribe pattern in Java

Litterature: Introduction to Java threads

An instruction

- Basic building block of software
- Written by the programmer, translated into machine code

```
10 + 10    // 20
```

See also: Instruction Set example

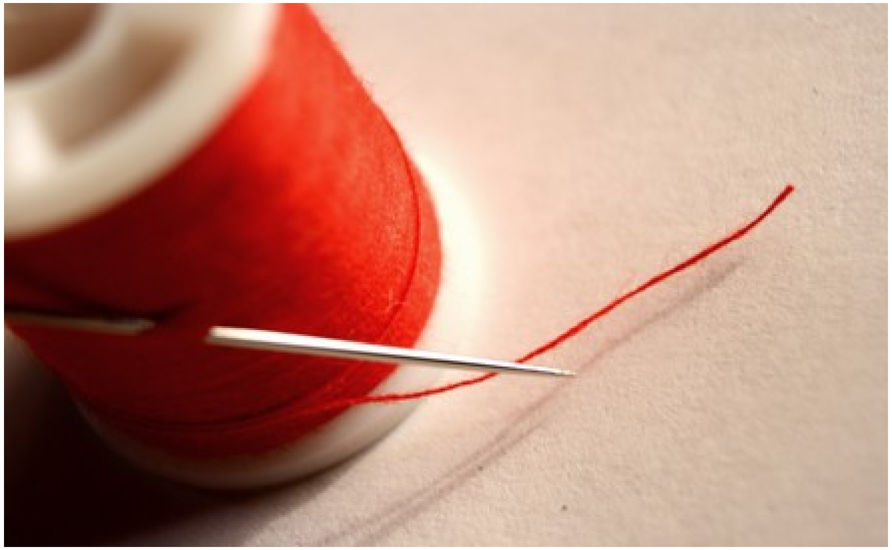
A core

- Processes instructions
- One instruction at the time
  - One core: speed x 1
  - Two cores: speed x 2
  - Four cores: speed x 4
  - … Not entirely true

See also: Central Processing Unit (CPU)

A thread

- A list of many instructions
- Can be assigned to a core
- Can start and stop

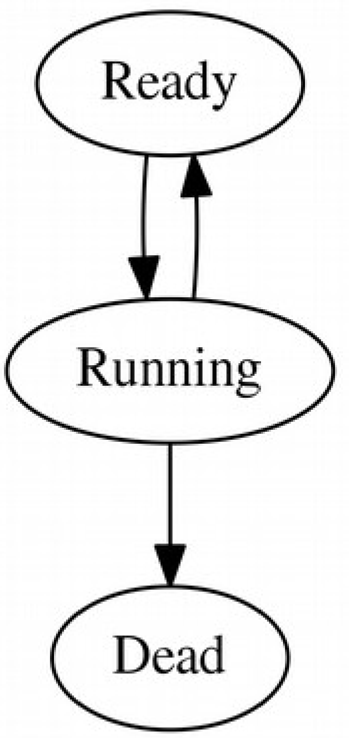


See also: Java API for Thread class

Threads in Java 1/2

- Can be created, started and stopped
- Creating threads
  - With `Thread`
  - With `Runnable`

- Who decides when a thread is run?



Java thread states

See also: Java thread states

A scheduler

- Schedules when threads should be run
  - ‘God mode’ - you have almost no control
- Consequences:
  - Nondeterministic
  - Overhead
    - Scheduling is work
    - More threads, more work
      - Hint: don’t create 1.000.000 threads
        - Unless you have 1.000.000 cores

See also: Definition of ‘scheduling’, Almdahl’s law

Multi-core architecture

- Having many cores in one CPU
  - One core: one thread at once
  - Two cores: two threads at once
  - 1.000.000 cores: 1.000.000 threads at once
  - One core trying to run 1.000.000 threads = disaster
- Many cores but shared hardware
- What are the threads doing?
  - Manipulating memory and interacting with I/O devices

See also: Multicore processors

Recap

- A thread is a list of instructions
- A core can execute one thread at the time
- No one knows *when* threads starts or stops
- Threads share memory

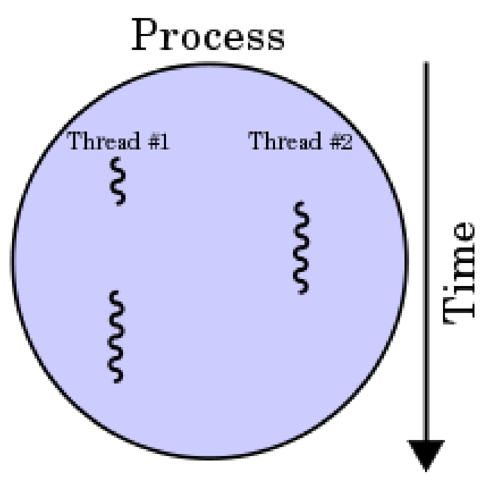
- A process can create and use threads

Coming up

- How and why concurrency ~~can~~ will fail
- More threading in Java
- Synchronising threads
- Deadlock and starvation

Processes and threads

- A process is a running application
- A process can have one or more threads
- Every process has a main thread
  - In Java: `public static void main(String[] args)`



See also: Definition of process

A process in Java

- One thread
- Two threads
- Many threads

Concurrency

- “Happening at the same time”
- Normal program: A – Z
- Concurrent program: G, S, A, D, Y, …

- What could go wrong?
- This is very very very hard

See also: Concepts of Concurrent programming (glossary), Rust Concurrency

Thread unpredictability

- Threads can start and stop **at all times**
  - Even between instructions!
- How many instructions are `count++`?
  - 3: Load, sum, store

See also: java.util.concurrent.atomic package in Java

Problems with concurrency

- Race condition
- Deadlocking
- Starvation

See also: java.util.concurrent.atomic package in Java

Race condition

- Problem with shared memory
  - Who comes first?
- Happens when order of threads matter
- Example: Summing numbers in Java
- Good news: Can be avoided with practice

See also: Java Thread synchronisation tutorial

Deadlocking in Java

- When two threads wait for the same thing
- Example: deadlock in Java
- Bad news: No easy solution
  - Advice from book

    ‘To avoid deadlock, you should ensure that when you acquire multiple locks, you always acquire the locks in the same order in all threads.’

See also: Java deadlock tutorial

Starvation in Java

- When threads cannot progress
  - Greedy threads steals resources
  - Thread slows down
- Hard to reproduce

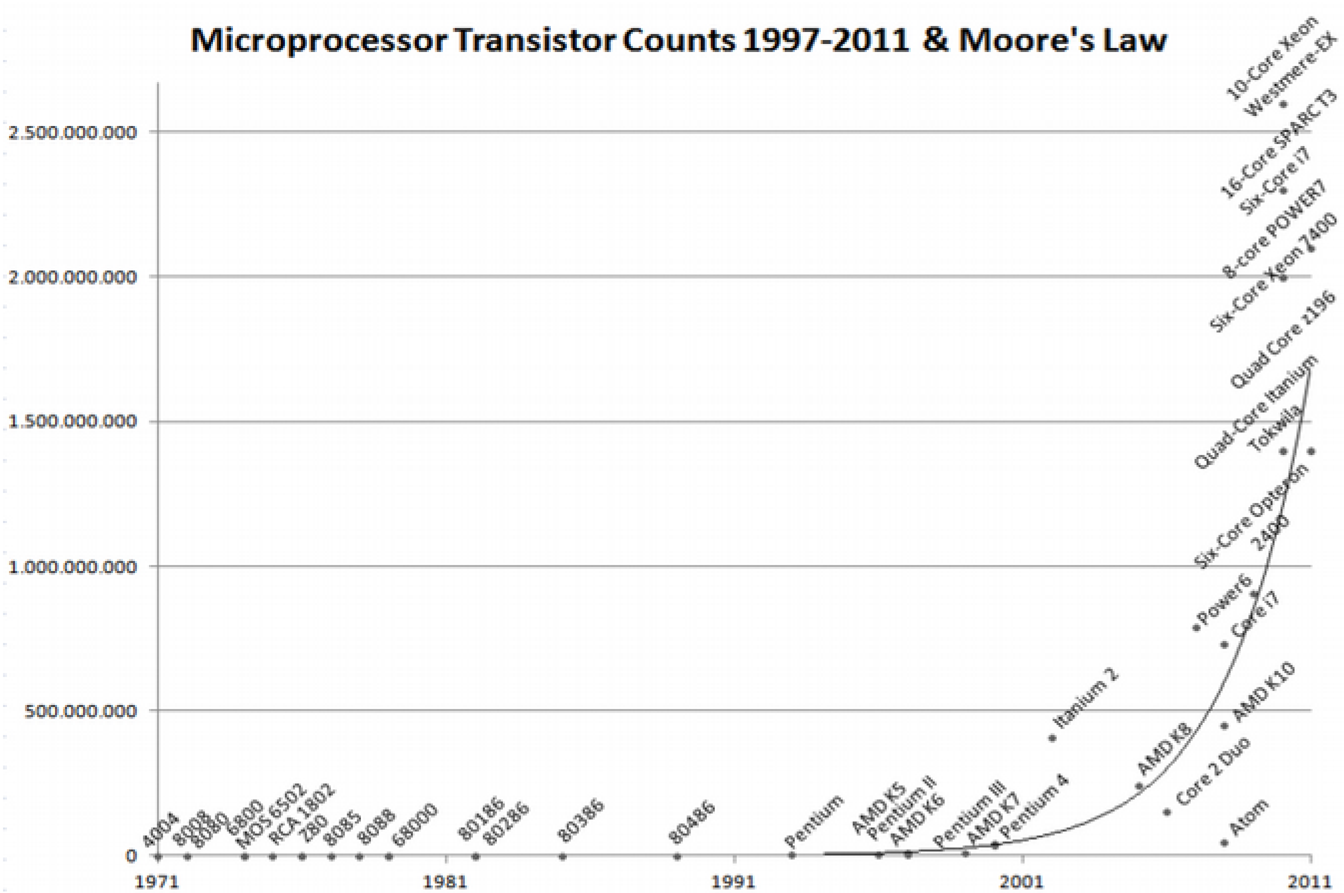
Recap

- A thread is a list of instructions
- Threads run randomly but share memory
- Java processes can create and run threads
- Threads are dangerous!
  - They share memory and risk race conditions
  - They deadlock
  - They can steal resources from others

Coming up

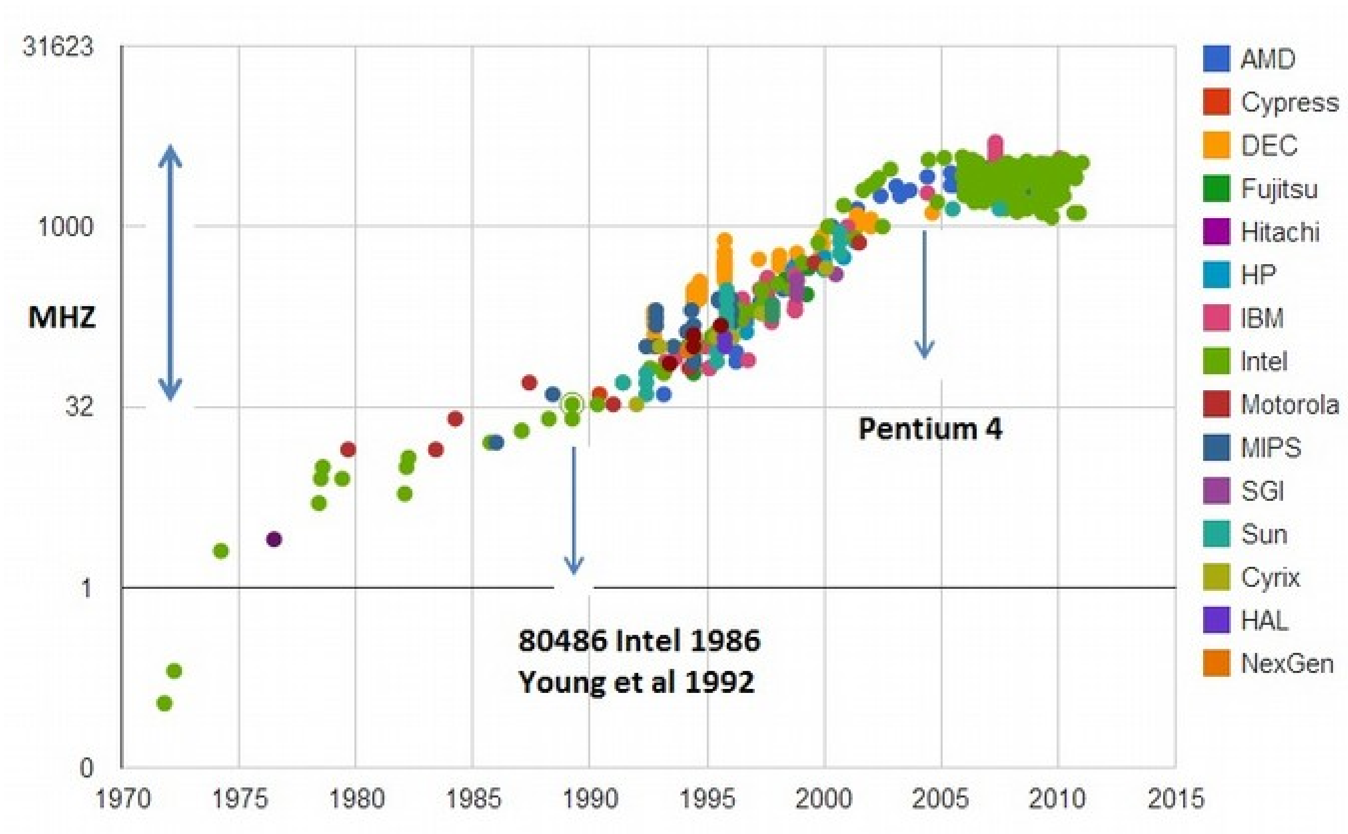
- History of CPU development
- Multi-core processor architecture
- Benefits of multi threaded computing

Moore’s law



See also: Moore’s law

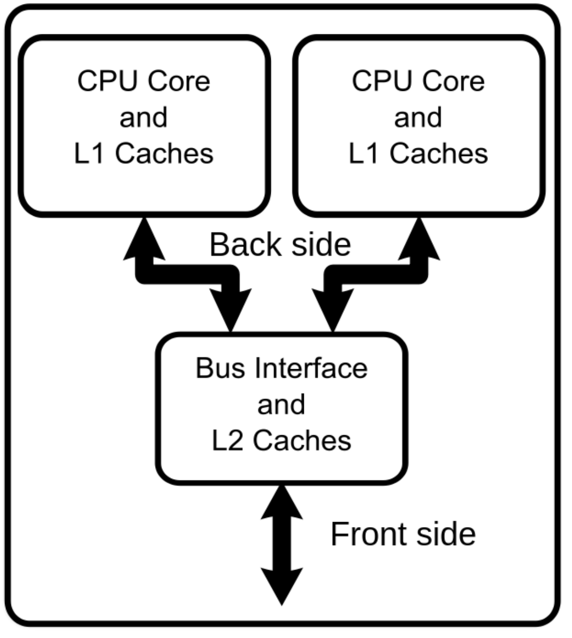
Clock frequency



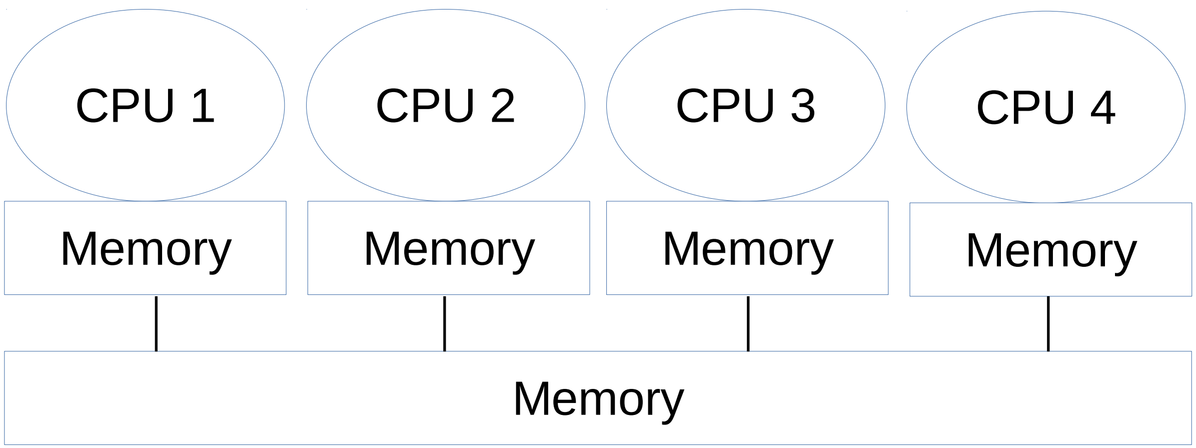
See also: Technology roadmap for semiconductors

Where does the computing power come from?

Multi-core processor

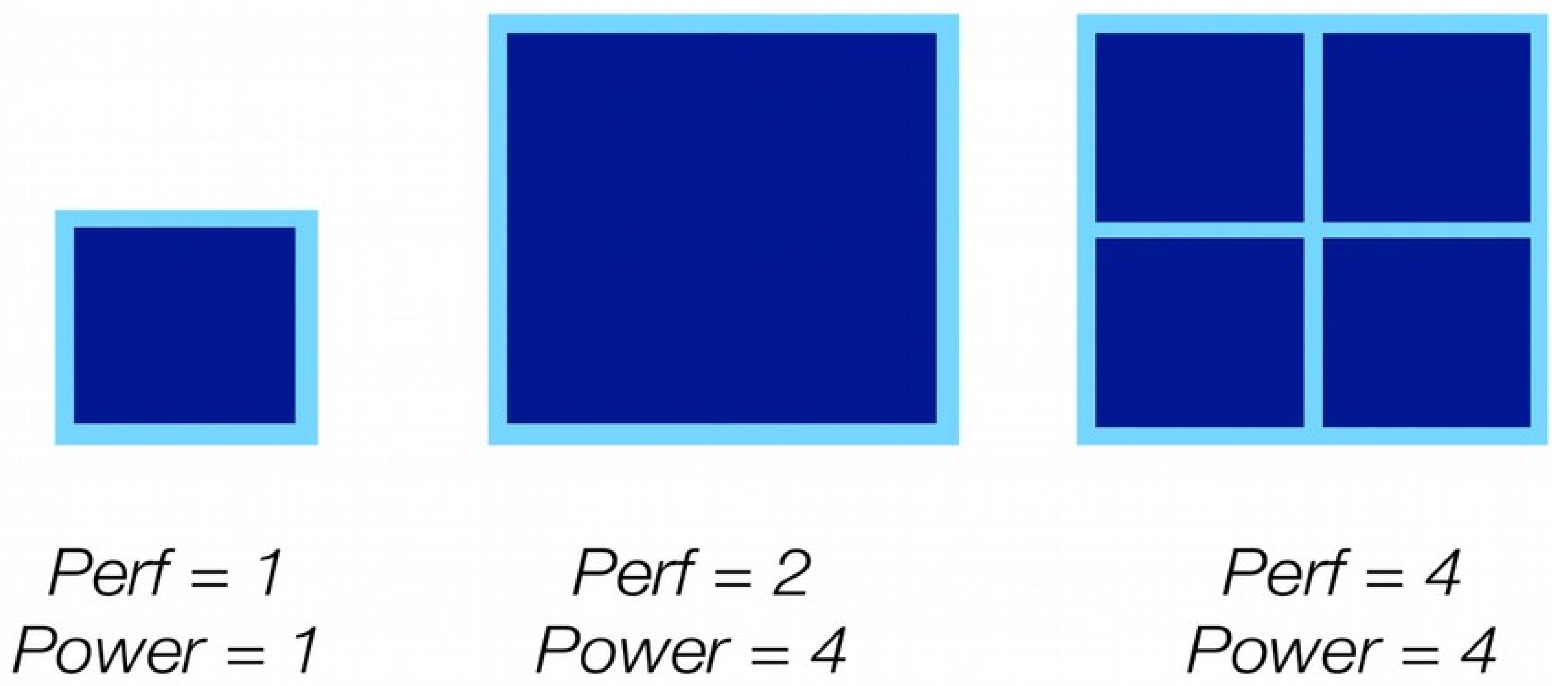


Multi-core processor



Power consumption

- High frequency = high energy consumption
- Low frequency + more cores = efficiency



Parallelisation benefits

- A core can work one instruction at the time
  - 1 core = speed x 1
  - 2 cores = speed x 2
  - 4 cores = speed x 4
- … If you parallelise your software!

Recap

- Number of cores are growing fast!
- Cores in a multi-core processor share memory
- Perfect parallelisation means a speedup factor equal to the number of cores
  - Fast!

Coming up

- Atomic variables
- Critical sections of code
- Locking and synchronisation
- More about threads
- Controlling threads in Java

Critical section

- Critical sections may not run concurrently
  - Example: variable
- Thread safety
  - A guarantee for the same behaviour with many threads
- How to control access to a section?
  - Gatekeeper / locking variable
- Who protects the protector?
  - Good question!

Locking a variable

- Atomic operations cannot be intercepted
  - Also known as thread-safety
  - Lock-free
  - Solves race-conditions for counters
- Javas atomic classes

```
AtomicInteger counter = new AtomicInteger();
counter.get(); // 0
counter.getAndIncrement();
counter.get(); // 1
```

- Why ‘`getAndIncrement`’?
  - Because increments are 3 operations

See also: java.util.concurrent.atomic

Locking a section

- Why not just a simple variable?
  - Because it is not thread safe!

- `volatile` keyword
  - Ensures that reads and writes are atomic
  - Is this good enough?
- No, not always good enough
  - Good for reading (loop condition)
  - Bad for counting (race-condition)

See also: Mutual exclusion (mutex)

Synchronisation

- Only allow one thread to access code at once
- Method synchronisation
- Block synchronisation
  - Synchronising on `this`
  - Synchronizing on object(s)

- Which one is preferred?
  - Block synchronisation. It only locks the critical section and not the whole method

See also: Java synchronization

Synchronisation question

- Is this implementation thread safe?

```
int i = 0;
synchronized void increment() {
    i++;
}
```

- No! `i` is public!
  - Synchronize does not mean you do not have to think
  - Encapsulate, encapsulate, encapsulate

Recap

- Thread is a list of instructions
- Concurrency can go wrong!
- Atomic operations
- Locks
- Synchronisation
- Thread-safety

Thread safety question

- Is this thread-safe?

```
class Singleton {
    private static Singleton instance = null;
    private Singleton() { /* private constructor */ }
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

- No! `getInstance` is not `synchronized`

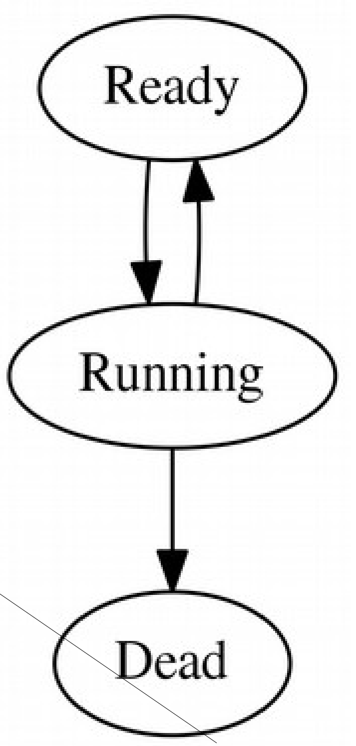
Coming up

- Locking and synchronisation
- More about threads
- Controlling threads in Java
- Semaphores

Threads in Java 1/2

- Can be created, started and stopped
- Creating threads
  - With `Thread`
  - With `Runnable`

I lied!



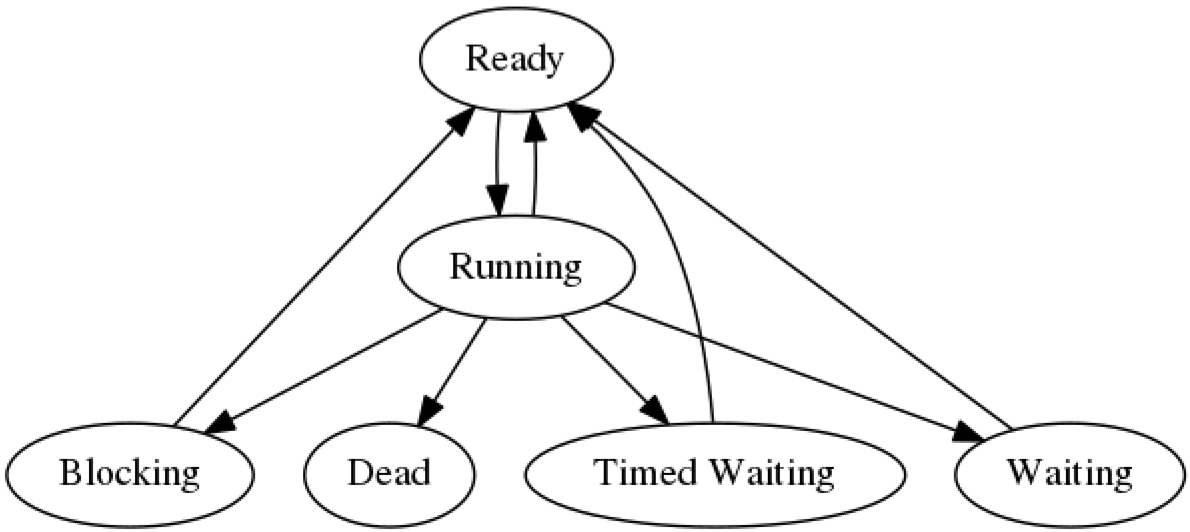
Java thread states

See also: Java thread states

Threads in Java 2/2

- Threads have three more states

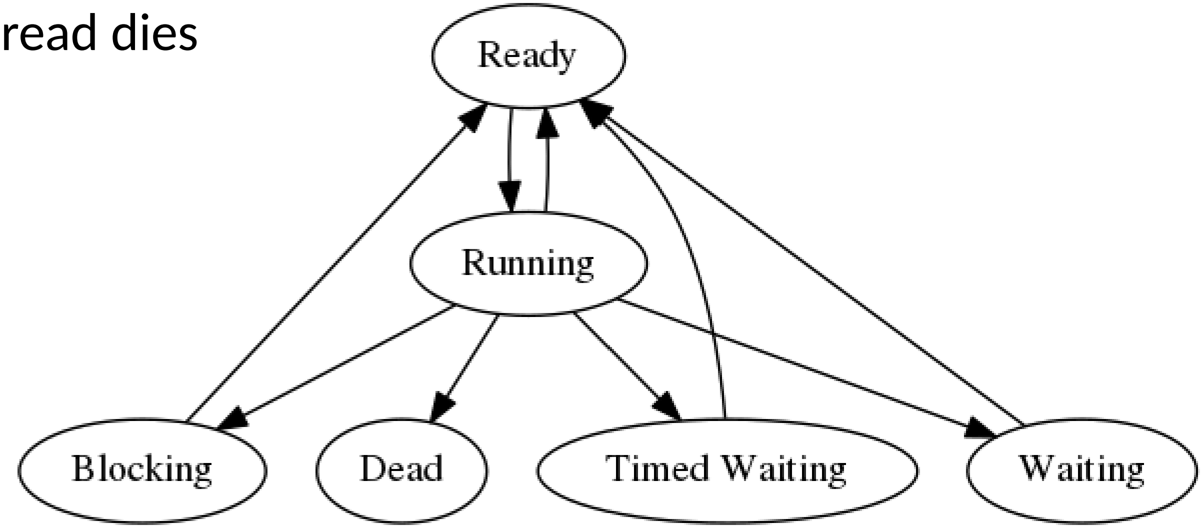
- Blocking
  - Waiting for resource until released !
  - For instance when waiting for access to `synchronized`
- Waiting
  - `Object.wait();`
  - `Thread.join();`
- Waiting with a timeout
  - `Object.wait(...);`
  - `Thread.join(...);`
  - `Thread.sleep(...);`



See also: Java API for Thread class

Controlling threads 1/2

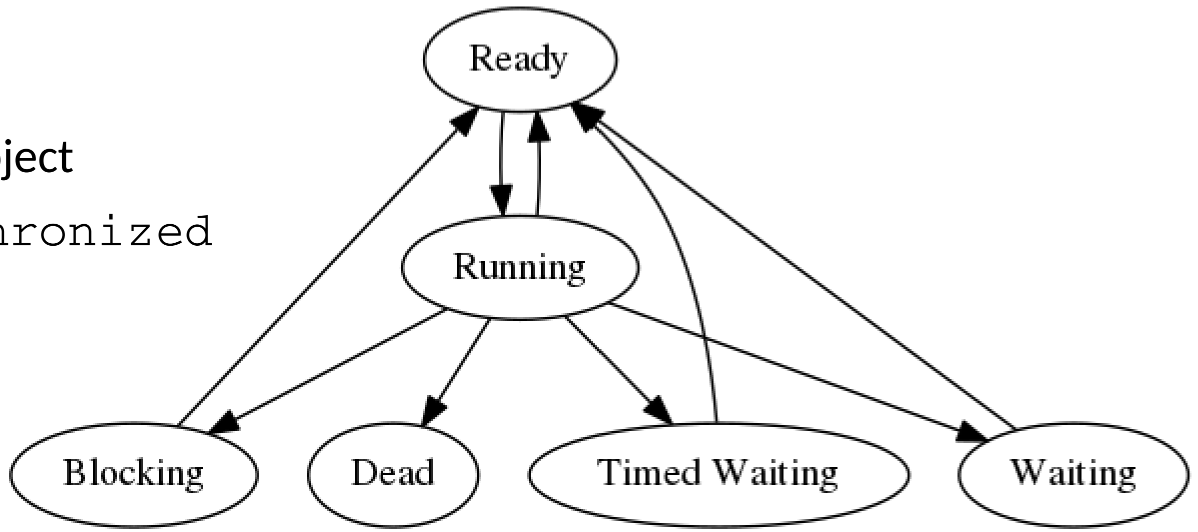
- Threads can be controlled (a little)
  - `interrupt()`
    - Sends a signal to the thread to stop what it's doing
    - Can be ignored by the thread
  - `join()` and `join(...)`
    - Wait until a thread dies
  - `sleep(...)`
    - Do nothing



See also: Java API for Thread class

Controlling threads 2/2

- Threads can also be controlled via `Object`
  - `wait()` and `wait(...)`
    - Blocks current thread until someone calls notify()
  - `notify()`
    - Wakes a single thread
  - `notifyAll()`
    - Wakes all threads
  - Both are tied to an object
    - MUST be `synchronized`



See also: Java API for Thread class

A Semaphore

- Data used to control access for multiple threads
- Useful metaphor: How many units are available?
- Two types of semaphores
  - Binary (1 resource)
  - Counting (2+ resources)
- Is the `synchronized` keyword a semaphore?

See also: Java API for Semaphore

Semaphore examples

- Two operations
  - `acquire()` access to critical region
  - `release()` access permission
- Binary semaphore
  - `Semaphore s = new Semaphore(1);`
- Counting semaphore
  - `Semaphore s = new Semaphore(10);`

See also: Java API for Semaphore

Locks in Java

- Another way to controls access to critical section
- Fairness
  - Longest waiting thread gets the lock
- Lock interface
  - `lock()`
  - `unlock()`
- ReadWriteLock interface
  - `readLock()` - returns a Lock
  - `writeLock()` - returns a Lock
- Remember to release the lock!
  - Use `try-finally`

See also: Java API for Lock, Java API for ReadWriteLock

Locks in Java example

- ReentrantLock
  - Reentrant means that a thread can have more than one lock

```
ReentrantLock lock = new ReentrantLock();
lock.lock();
try { … } finally { lock.unlock(); }
```

- ReentrantReadWriteLock

```
ReentrantReadWriteLock lock = new
ReentrantReadWriteLock();
WriteLock writeLock = lock.writeLock();
writeLock.lock();
try { … } finally { writeLock.unlock(); }
```

See also: ReentrantReadWriteLock versus StampedLock

Synchronisation versus locks

- `Lock` provides more visibility and functionality
  - For instance `tryLock()`
- `Lock` require `try-finally` blocks
  - Synchronization code can be cleaner
- `Lock` can be aquired in one method and released in another
  - Good or bad?
- `Lock` provide fairness
  - The longest waiting threads get the lock first

Recap

- Critical sections consist of code that cannot run concurrently
- Atomic operations cannot be intercepted
- Controls to critical sections
  - `synchronized`
  - Semaphores (can allow one or more threads at once)
  - `Lock`
- Threads can be `join()`'ed and `interrupt()`'ed

Coming up

- Getting data out of threads
- Producer-consumer problem
- Observable
- Thread pools and `ExecutorService` in Java
- Threads in Swing
- Thread priority

Getting data out of threads

- `Threads` and `Runnables` return `void`
- How do we get the data out?!

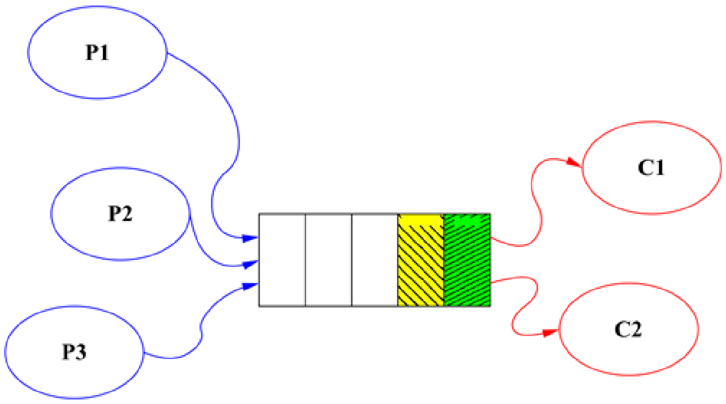
- Use class variables
  - Not thread safe?
- Use collections like lists, queues etc.
  - Producer/consumer problem!
- Use observables
  - Only notified when something happens (reactive)

A note on `java.util.concurrent`

- Great collection library
- Contains the `atomic` package
- Contains the `locks` package
- Contains `CopyOnWriteList`
- Contains `ConcurrentHashMap`

Producer-consumer problem 1/2

- Consider two processes:
  - One process puts elements in a queue
  - One process takes elements from the queue
- The queue has a fixed size
- What happens if
  - The producer slows down?
    - Starvation
  - The consumer slows down?
    - Buffer overflow



Ekstra læsning: Guarded blocks

Producer-consumer problem 2/2

- In modern Java: `ArrayBlockingQueue`
- Example

```
ArrayBlockingQueue q = new ArrayBlockingQueue();
// Thread 1 - Producer
q.put(…);
// Thread 2 – Consumer
q.take();
```

Observer pattern

- Instead of pushing (producing) we can observe

```
interface Observable<T> {
  public void onEvent(T event);
}
```

- Observer versus Observable

1) An Observer is registered in an Observable
2) The Observable will notify the Observer
   - Don’t call us, we’ll call you!

Executors in Java

- Creating threads is not free
  - Reuse them!

- Thread pool in Java

```
Executors
    .newCachedThreadPool()
    .execute(runnable);
```

Ekstra læsning: OSI Model

Threads in Swing

- Swing uses a special event dispatch thread
- All Swing component methods must be invoked from this thread
  - Unless the API states it is thread safe
- Example
  - `SwingUtils.invokeLater(() -> text.setText("hi"));`
- What if you submit a 1000 second task?
  - It blocks!

Ekstra læsning: Swing event dispatch thread

Thread priority

- Thread can have a priority
  - Tells the scheduler how to prioritise
- Ranges from 1 (min) to 10 (max)
  - `Thread.MIN_PRIORITY` and `Thread.MAX_PRIORITY`
- Must be set *before* `start`

```
Thread t = ...;
t.setPriority();
t.start();
```

Detecting deadlocks

```
ThreadMXBean mx =
           ManagementFactory.getThreadMXBean();
long[] deadlocked = mx.findDeadlockedThreads();
```

Ekstra læsning: ThreadMXBean

Recap

- Threads can be prioritised
- Use executors when you have many tasks
- Swing uses one single event thread
- Producer-consumer problem

Todays exercise

1) Getting stuff from the web
2) Waiting for stuff from the web
3) Fibonacci observer
    1) 1 – 1 – 2 – 3 – 5 – 8 - …
4) Fibonacci observer GUI
5) Deadlock detection