Introduction to Concurrent Programming

Lecture 1 of TDA384/DIT391

Principles of Concurrent Programming

Nir Piterman

Chalmers University of Technology | University of Gothenburg

SP1 2021/2022

Today’s menu

- A motivating example
- Why concurrency?
- Basic terminology and abstractions
- Java threads
- Traces

A Motivating Example

As simple as counting to two

We illustrate the challenges introduced by concurrent programming on a simple example: a counter modeled by a Java class.

- First, we write a traditional, sequential version.
- Then, we introduce concurrency and…run into trouble!

Sequential counter

```
public class Counter {
  private int counter = 0;

  // increment counter by one
  public void run() {
     int cnt = counter;
     counter = cnt + 1;
  }

  // current value of counter
  public int counter() {
     return counter;
  }
}
```

```
public class SequentialCount {
  public static
  void main(String[] args) {
      Counter counter = new Counter();
      counter.run(); // increment once
      counter.run(); // increment twice
      // print final value of counter
      System.out.println(
          counter.counter());
  }
}
```

- What is printed by running: `java SequentialCount`?
- May the printed value change in different reruns?

Modeling sequential computation

```
5  public void run() {
6      int cnt = counter; ●
7      counter = cnt + 1; ●
8  }                      ●
```

```
● counter.run(); // first call: steps 1-3
● counter.run(); // second call: steps 4-6
```

#	LOCAL STATE		OBJECT STATE
1	pc: 6	cnt: ⊥	counter: 0
2	pc: 7	cnt: 0	counter: 0
3	pc: 8	cnt: 0	counter: 1
4	pc: 6	cnt: ⊥	counter: 1
5	pc: 7	cnt: 1	counter: 1
6	pc: 8	cnt: 1	counter: 2
7	done		counter: 2

Adding concurrency

Now, we revisit the example by introducing concurrency:

Each of the two calls to method `run` can be executed in parallel

In Java, this is achieved by using threads. Do not worry about the details of the syntax for now, we will explain it later.

The idea is just that:

- There are two independent execution units (threads) `t` and `u`
- Each execution unit executes `run` on the same `counter` object
- We have no control over the order of execution of `t` and `u`

Concurrent counter

```
public class CCounter
    extends Counter
    implements Runnable
{
    // threads
    // will execute
    // run()
}
```

```
public class ConcurrentCount {
  public static void main(String[] args) {
     CCounter counter = new CCounter();
     // threads t and u, sharing counter
     Thread t = new Thread(counter);
     Thread u = new Thread(counter);
     t.start(); // increment once
     u.start(); // increment twice
     try { // wait for t and u to terminate
       t.join(); u.join();
     } catch (InterruptedException e) {
       System.out.println("Interrupted!");
     } // print final value of counter
     System.out.println(counter.counter());
} }
```

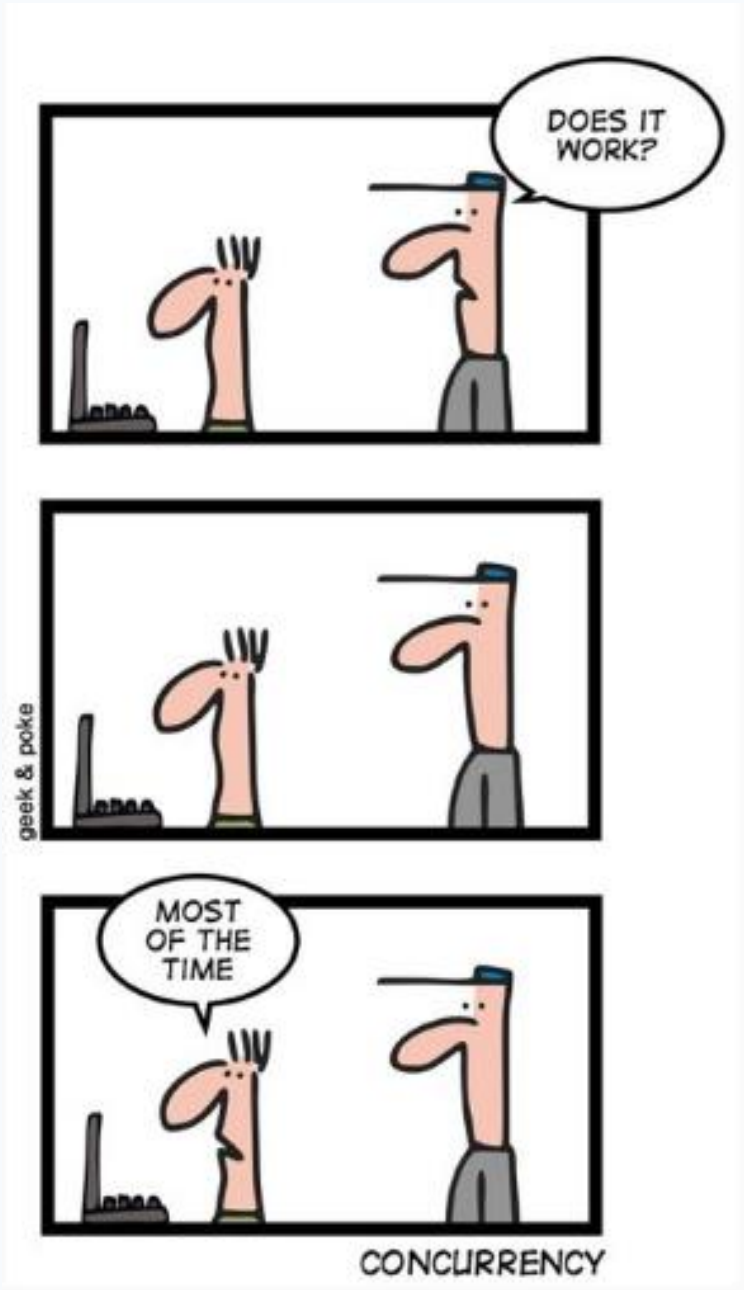
– What is printed by running: `java ConcurrentCount`?
– May the printed value change in different reruns?

What?!

```
$ javac Counter.java CCounter.java ConcurrentCount.java
$ java ConcurrentCount.java
2
$ java ConcurrentCount.java
2
...
$ java ConcurrentCount.java
1
$ java ConcurrentCount.java
2
```

The concurrent version of counter occasionally prints 1 instead of the expected 2. It seems to do so unpredictably.

Welcome to concurrent programming!


DOES IT WORK?
MOST OF THE TIME
geek & poke
CONCURRENCY

Why concurrency?

Reasons for using concurrency

Why do we need concurrent programming in the first place?

- **abstraction:**
  - separating different tasks, without worrying about when to execute them (example: download files from two different websites)
- **responsiveness:**
  - providing a responsive user interface, with different tasks executing independently (example: browse the slides while downloading your email)
- **performance:**
  - splitting complex tasks in multiple units, and assign each unit to a different processor (example: compute all prime numbers up to 1 billion)

Concurrency vs. parallelism

Principles of concurrent programming

vs.

Principer för parallell programmering

Huh?

Concurrency vs. parallelism

In this course we will mostly use concurrency and parallelism as synonyms. However, they refer to similar but different concepts:

- **concurrency:**
  - nondeterministic composition of independently executing units (logical parallelism),
- **parallelism:**
  - efficient execution of fractions of a complex task on multiple processing units (physical parallelism).

- You can have concurrency without physical parallelism: operating systems running on single-processor single-core systems.
- Parallelism is mainly about speeding up computations by taking advantage of redundant hardware.

Concurrency vs. parallelism

Ideal situation

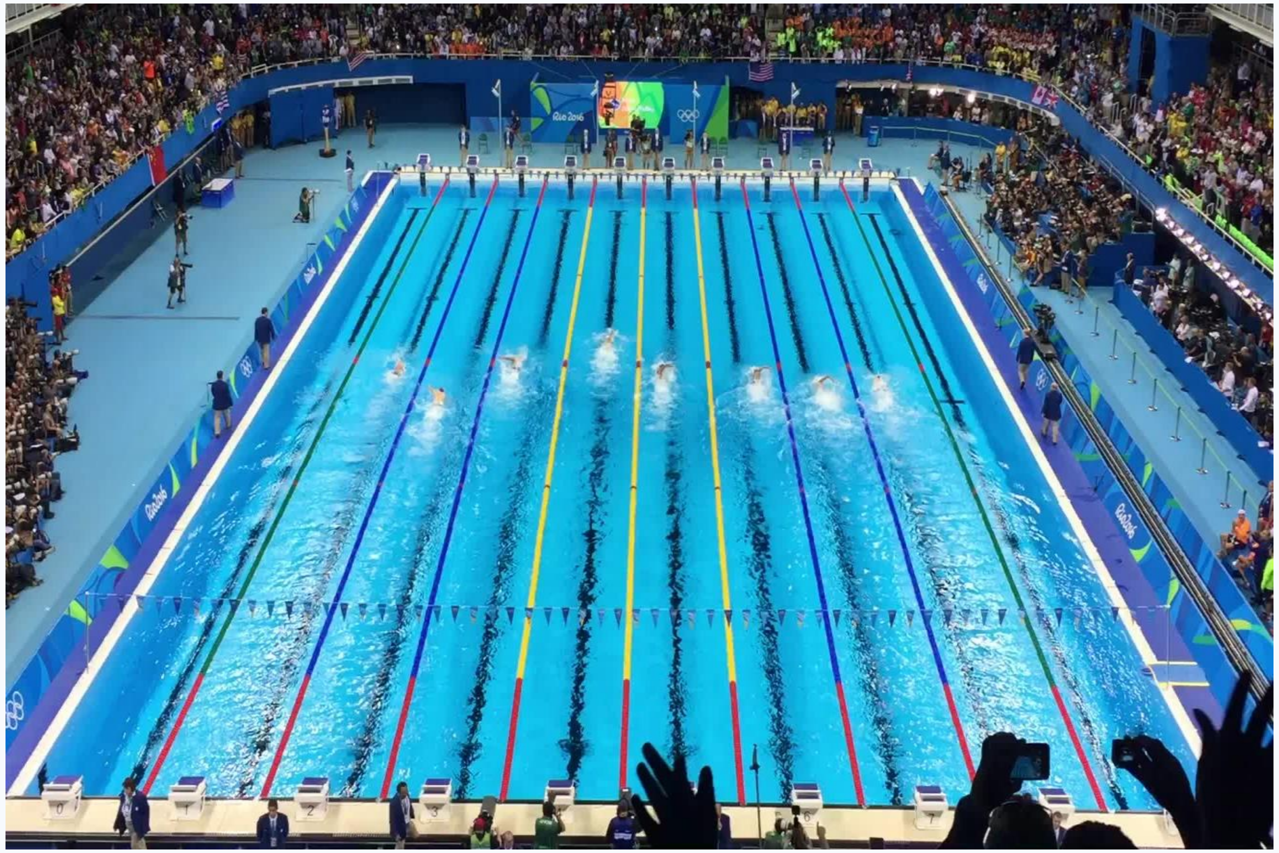


Photo: Summer Olympics 2016, Sander van Ginkel.

Concurrency vs. parallelism

More common situation

Photos: World Cup Nordic ’07, Tomoyoshi Noguchi – Vasaloppet ’06, Steven Hale.

Concurrency vs. parallelism

Real world situation

Photo: Daniel Mott 2009.

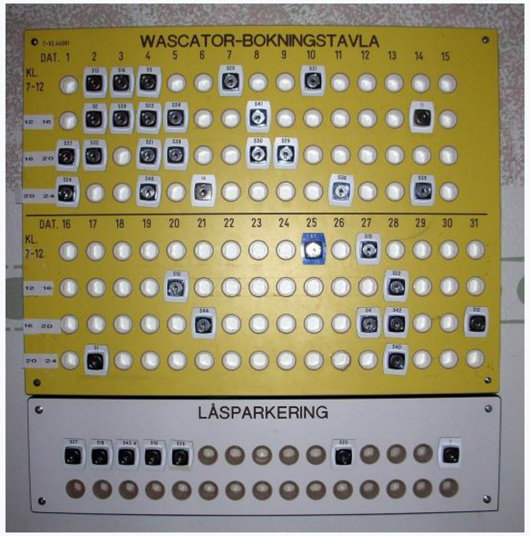


Photo: Wolfgangus Mozart 2010.

Challenges:

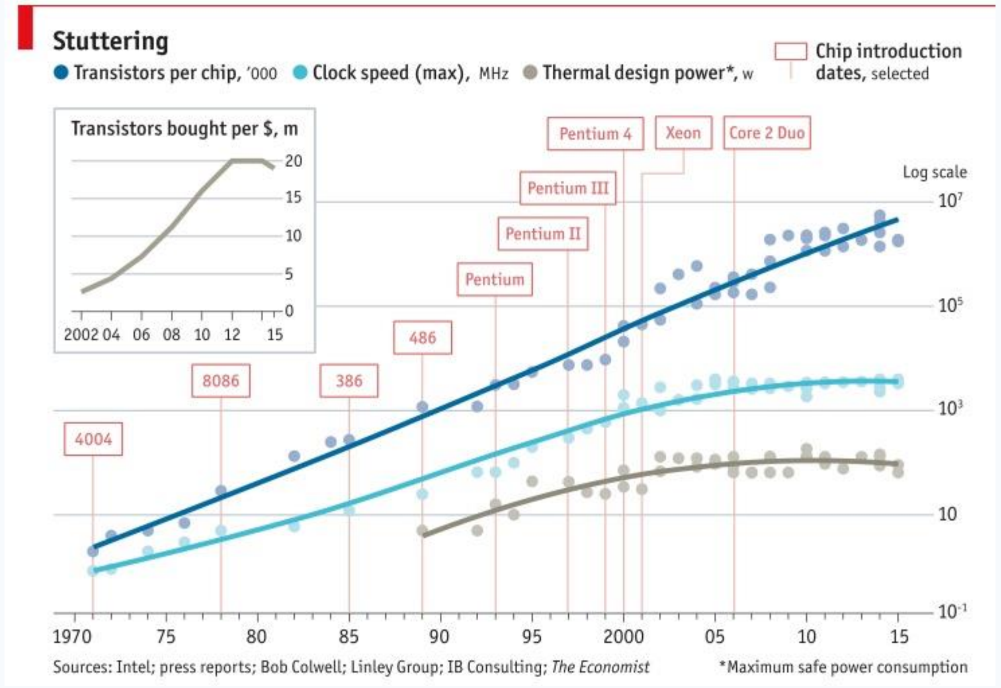
- *concurrency:* everyone gets to do their laundry (fairness), machines are operated by at most one user (mutual exclusion);
- *parallelism:* distribute load evenly over machines/rooms (load balancing).

Solutions: schedules, locks, signs/indicators…

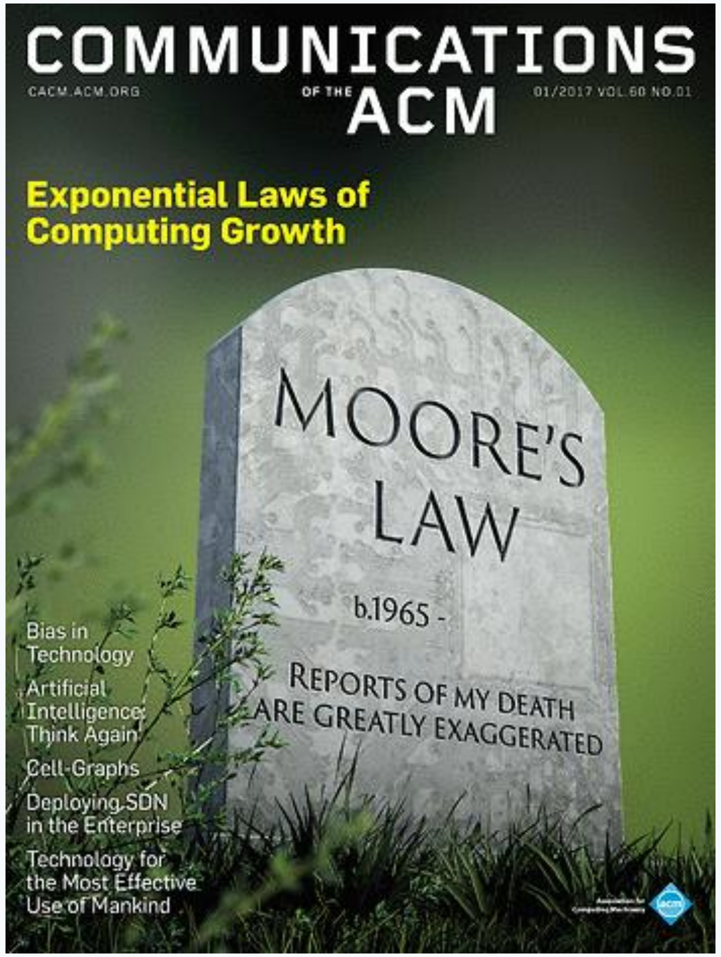
Moore's law and its end (?)

The spectacular advance of computing in the last 60+ years has been driven by Moore's law.

The density of transistors in integrated circuits doubles approximately every 2 years.



Moore's Law in January 2017


COMMUNICATIONS OF THE ACM
CACM.ACM.ORG
01/2017 VOL.60 NO.01
Exponential Laws of Computing Growth
MOORE'S LAW
b.1965 -
REPORTS OF MY DEATH ARE GREATLY EXAGGERATED
Bias in Technology
Artificial Intelligence: Think Again
Cell-Graphs
Deploying SDN in the Enterprise
Technology for the Most Effective Use of Mankind

Concurrency everywhere

Physical restrictions force to change from increasing processing speed to having multiple processing having a major impact on the practice of programming:

- **Before:** CPU speed increases without significant architectural changes.
  - Program as usual, and wait for your program to run faster.
  - Concurrent programming is a niche skill (for operating systems, databases, high-performance computing).
- **Now:** CPU speed remains the same but number of cores increases.
  - Program with concurrency in mind, otherwise your programs remain slow.
  - Concurrent programming is pervasive.

Very different systems all require concurrent programming:

- desktop PCs,
- smart phones,
- video-games consoles,
- embedded systems,
- the Raspberry Pi,
- cloud computing, …

CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Amdahl's law: concurrency is no free lunch

We have n processors that can run in parallel. How much speedup can we achieve?

$$\boldsymbol{speedup = \frac{sequential\ execution\ time}{parallel\ execution\ time}}$$

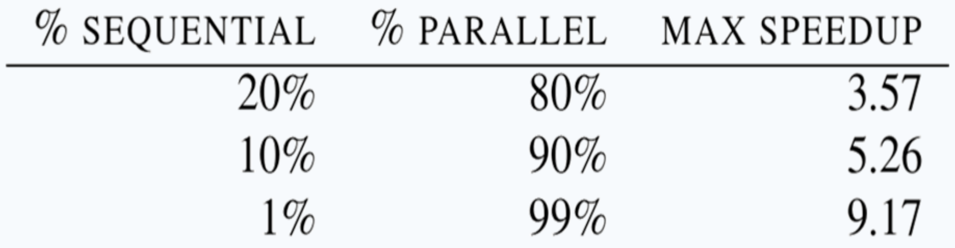
Amdahl's law shows that the impact of introducing parallelism is limited by the fraction p of a program that can be parallelized:

$$\boldsymbol{maximum\ speedup = \frac{1}{\underbrace{(1-p)}_{\text{sequential part}} + \underbrace{p/n}_{\text{parallel part}}}}$$

Amdahl's law: examples

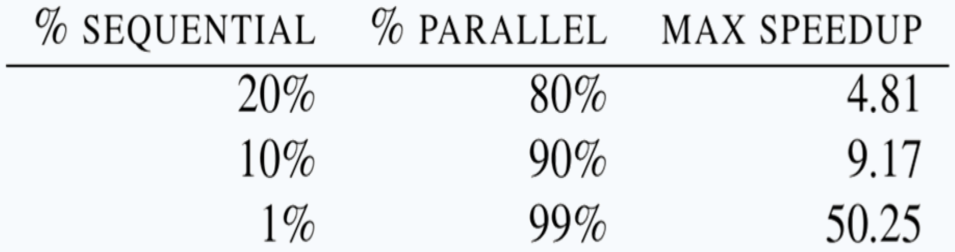
$$\textbf{\textit{maximum speedup}} = \frac{\mathbf{1}}{(\mathbf{1}-\boldsymbol{p})+\boldsymbol{p}/\boldsymbol{n}}$$

With n=10 processors, how close can we get to a 10x speedup?



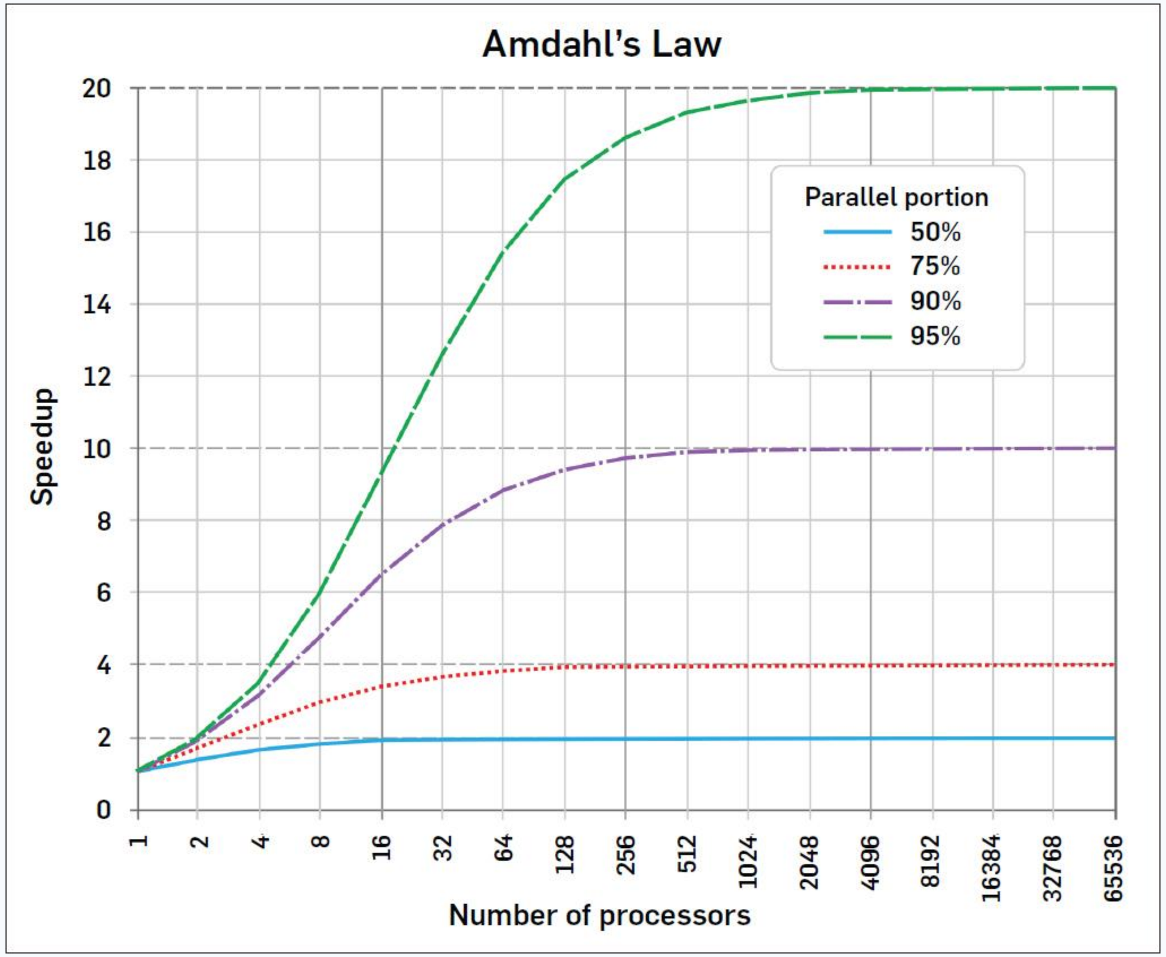
% SEQUENTIAL	% PARALLEL	MAX SPEEDUP
20%	80%	3.57
10%	90%	5.26
1%	99%	9.17

With n=100 processors, how close can we get to a 100x speedup?



% SEQUENTIAL	% PARALLEL	MAX SPEEDUP
20%	80%	4.81
10%	90%	9.17
1%	99%	50.25

Amdahl's law: examples



Source: Communications of the ACM, Dec. 2017

Basic terminology and abstractions

Processes

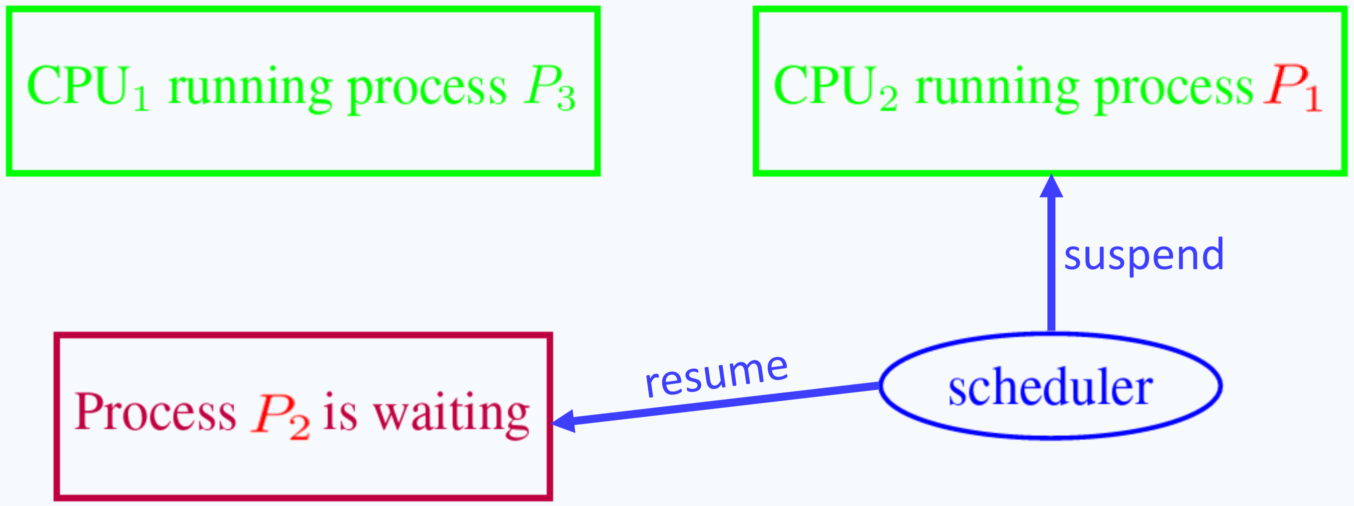
A process is an independent unit of execution – the abstraction of a running sequential program:

– identifier

– program counter

– memory space

The runtime/operating system schedules processes for execution on the available processors:



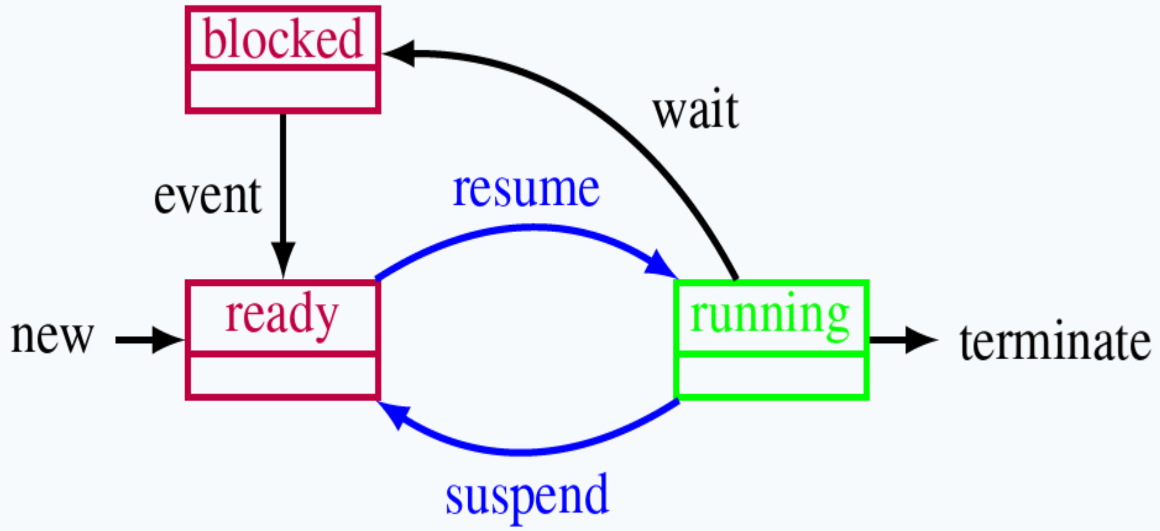
Process states

The scheduler is the system unit in charge of setting process states:

Ready: ready to be executed, but not allocated to any CPU

Blocked: waiting for an event to happen

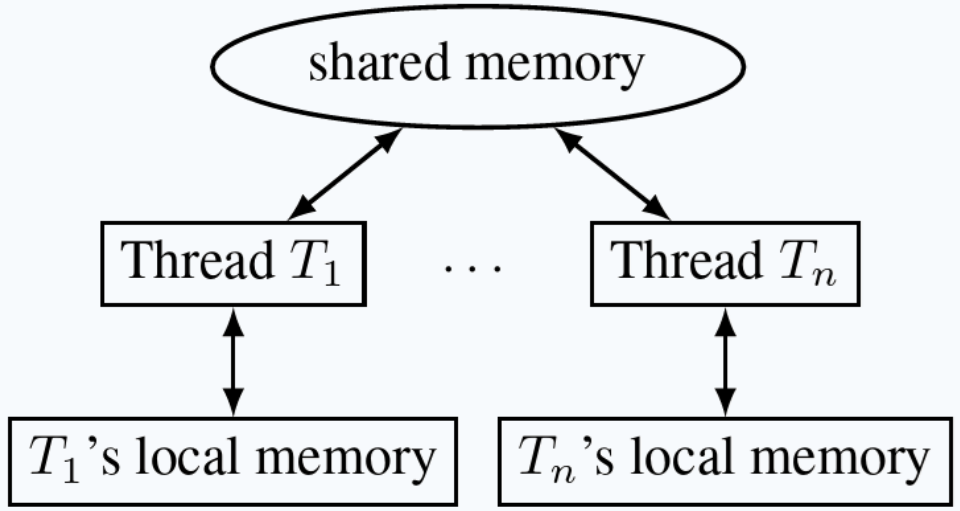
Running: running on some CPU



Threads

A thread is a lightweight process – an independent unit of execution in the same program space:

- identifier
- program counter
- memory
    - local memory, separate for each thread
    - global memory, shared with other threads



In practice, the difference between processes and threads is fuzzy and implementation dependent. Normally in this course:

Processes: executing units that do not share memory (in Erlang)

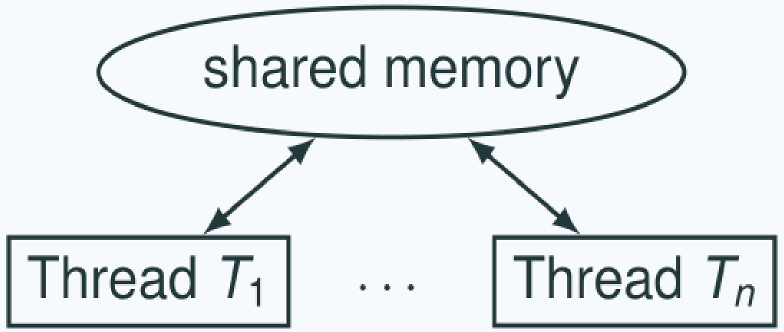
Threads: executing units that share memory (in Java)

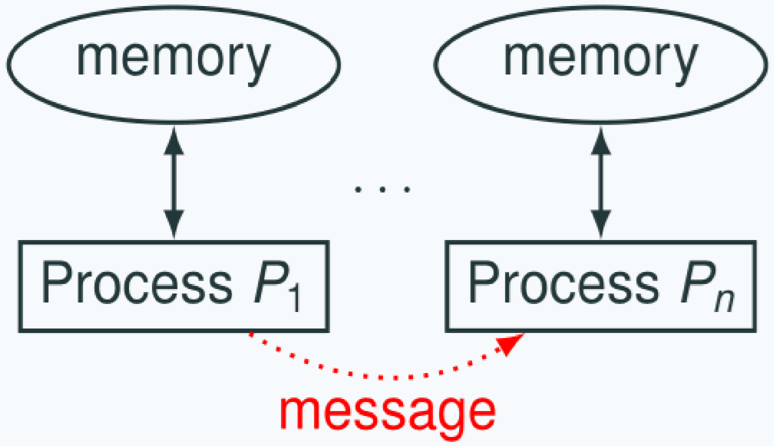
CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Shared memory vs. message passing

Shared memory models:
- communication by writing to shared memory
- e.g., multi-core systems

Distributed memory models:
- communication by message passing
- e.g., distributed systems





Java threads

Java threads

Two ways to build multi-threaded programs in Java:

- inherit from class `Thread`, override method `run`
- implement interface `Runnable`, implement method `run`

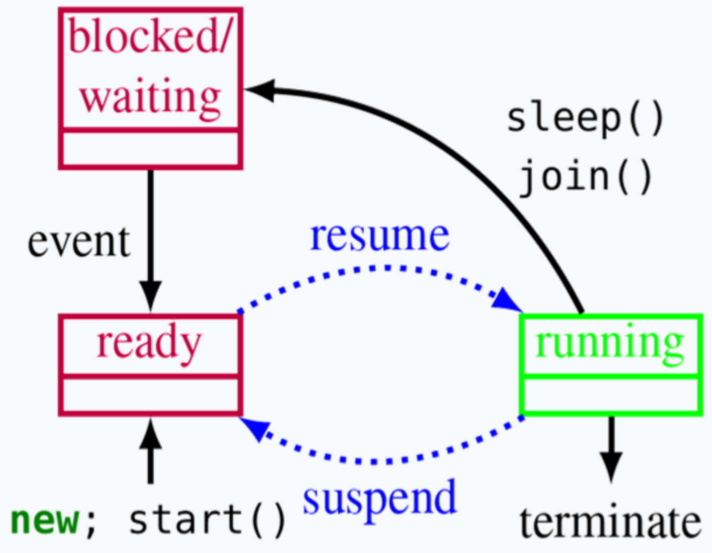
```
public class CCounter
  implements Runnable
{
   // thread's computation:
   public void run() {
     int cnt = counter;
     counter = cnt + 1;
   }
}
```

```
CCounter c = new CCounter();

Thread t = new Thread(c);
Thread u = new Thread(c);

t.start();
u.start();
```

States of a Java thread

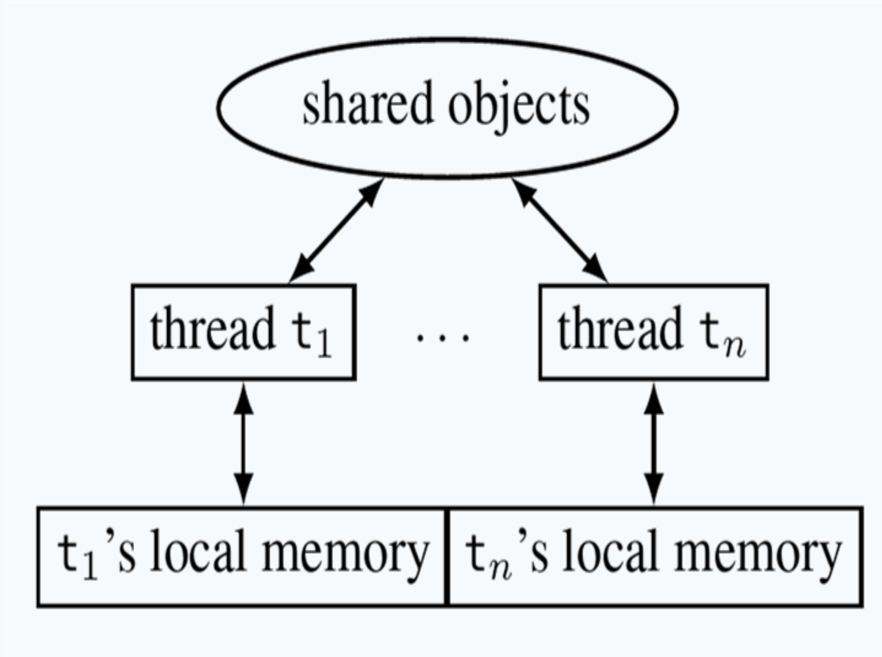


For a `Thread` object `t`:

- `t.start()`: mark the thread `t` ready for execution,
- `Thread.sleep(n)`: block the current thread for `n` milliseconds (correct timing depends on JVM implementation),
- `t.join()`: block the current thread until `t` terminates.

Resuming and suspending is done by the JVM scheduler, outside the program's control.

Thread execution model



Shared vs. thread-local memory:

- shared objects: the objects on which the thread operates, and all reachable objects
- local memory: local variables, and special *thread-local* attributes

Threads proceed asynchronously, so they have to coordinate with other threads accessing the same shared objects.

One possible execution of the concurrent counter

```
1: public class CCounter implements Runnable {
2:     int counter = 0;       // shared object state
3:
4:     // thread's computation:
5:     public void run() {
6:       int cnt = counter; ● ●
7:       counter = cnt + 1; ● ●
8: } }                     ● ●
```

#	t's LOCAL	u's LOCAL	SHARED
1	pc_t : 6 cnt_t : $\perp$	pc_u : 6 cnt_u : $\perp$	counter: 0
2	pc_t : 7 cnt_t : 0	pc_u : 6 cnt_u : $\perp$	counter: 0
3	pc_t : 8 cnt_t : 0	pc_u : 6 cnt_u : $\perp$	counter: 1
4	done	pc_u : 6 cnt_u : $\perp$	counter: 1
5	done	pc_u : 7 cnt_u : 1	counter: 1
6	done	pc_u : 8 cnt_u : 1	counter: 2
7	done	done	counter: 2

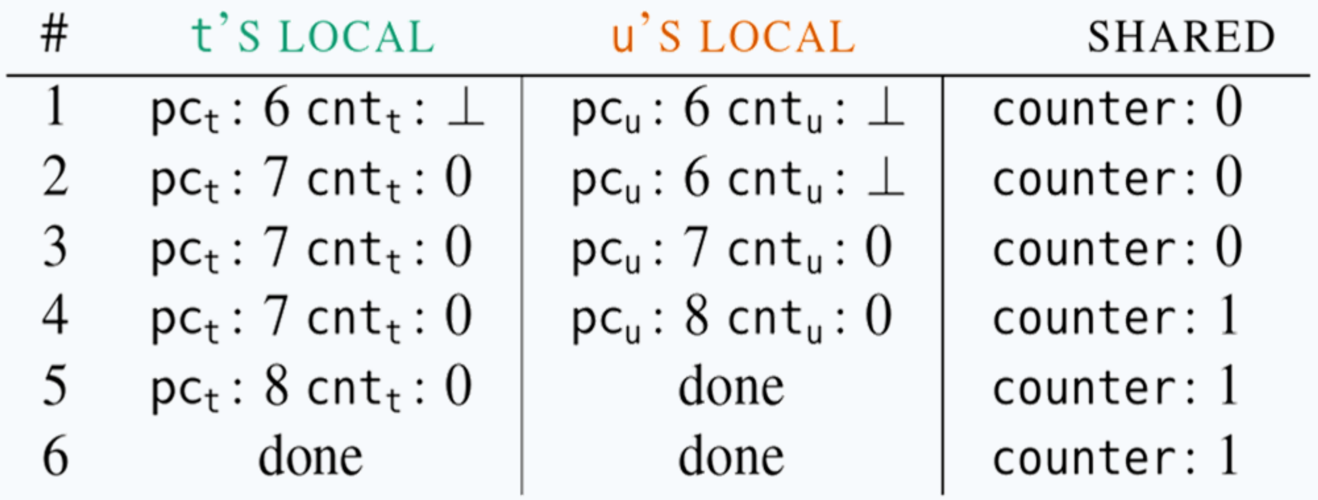
One alternative execution of the concurrent counter

```
1: public class CCounter implements Runnable {
2:      int counter = 0;        // shared object state
3:
4:      // thread's computation:
5:      public void run() {
6:         int cnt = counter; ● ●
7:         counter = cnt + 1; ● ●
8:  } }                       ● ●
```

#	t's LOCAL	u's LOCAL	SHARED
1	$pc_t : 6$ $cnt_t : \perp$	$pc_u : 6$ $cnt_u : \perp$	counter: 0
2	$pc_t : 7$ $cnt_t : 0$	$pc_u : 6$ $cnt_u : \perp$	counter: 0
3	$pc_t : 7$ $cnt_t : 0$	$pc_u : 7$ $cnt_u : 0$	counter: 0
4	$pc_t : 7$ $cnt_t : 0$	$pc_u : 8$ $cnt_u : 0$	counter: 1
5	$pc_t : 8$ $cnt_t : 0$	done	counter: 1
6	done	done	counter: 1

Traces

Traces

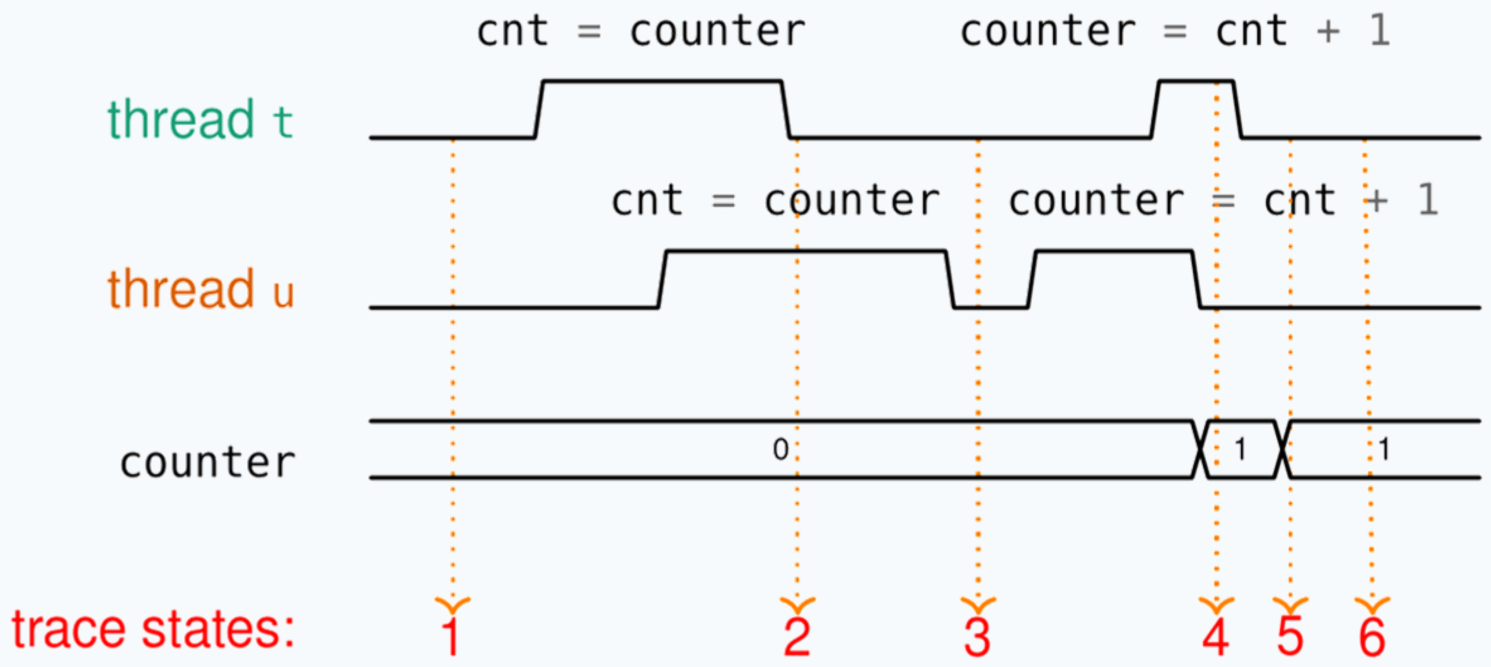


#	t's LOCAL	u's LOCAL	SHARED
1	$pc_t : 6\ cnt_t : \bot$	$pc_u : 6\ cnt_u : \bot$	counter: 0
2	$pc_t : 7\ cnt_t : 0$	$pc_u : 6\ cnt_u : \bot$	counter: 0
3	$pc_t : 7\ cnt_t : 0$	$pc_u : 7\ cnt_u : 0$	counter: 0
4	$pc_t : 7\ cnt_t : 0$	$pc_u : 8\ cnt_u : 0$	counter: 1
5	$pc_t : 8\ cnt_t : 0$	done	counter: 1
6	done	done	counter: 1

The sequence of states gives an execution trace of the concurrent program. A trace is an abstraction of concrete executions:

- atomic/linearized
- complete
- interleaved

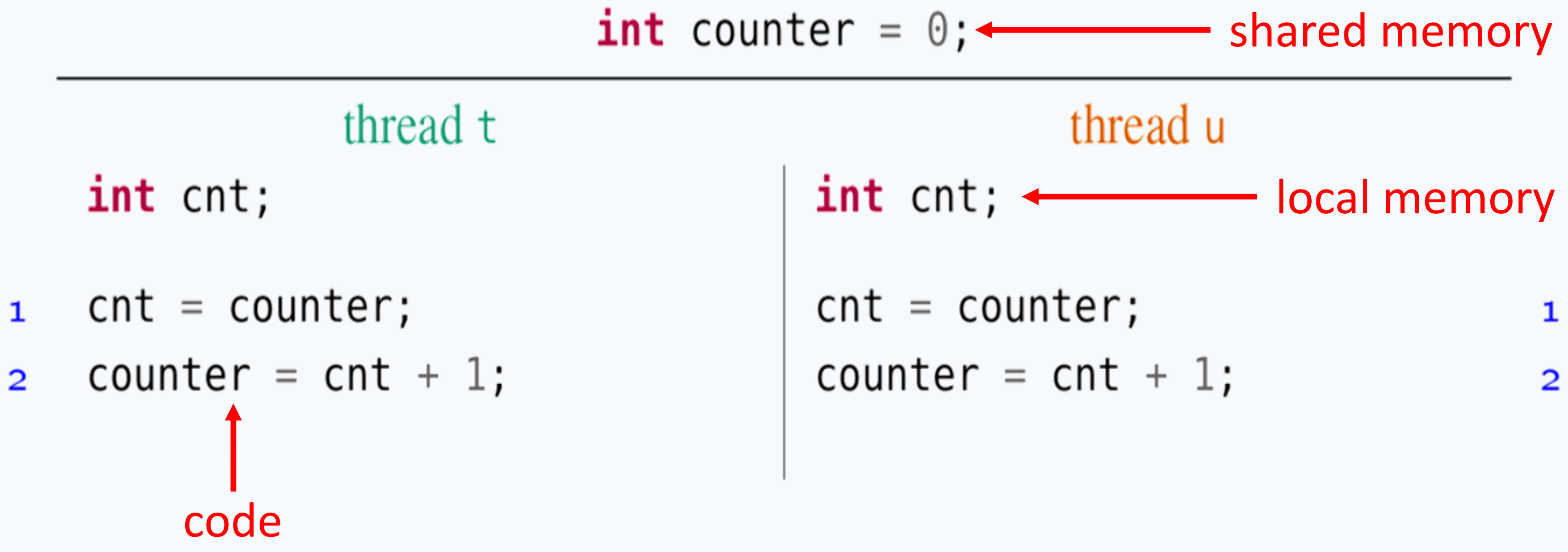
Trace abstractions



Atomic/linearized:	the effects of each thread appear as if they happened instantaneously, when the trace snapshot is taken, in the thread’s sequential order
Complete:	the trace includes all intermediate atomic states
Interleaved:	the trace is an interleaving of each thread’s linear trace (in particular, no simultaneity)

Abstraction of concurrent programs

When convenient, we will use an abstract notation for multi-threaded applications, which is similar to the pseudo-code used in Ben-Ari's book but uses Java syntax.



```
                 int counter = 0;
-----------------------------------------------------
        thread t           |        thread u
  int cnt;                 |  int cnt;
                           |
  cnt = counter;           |  cnt = counter;
  counter = cnt + 1;       |  counter = cnt + 1;
```

Each line of code includes exactly one instruction that can be executed atomically:

- atomic statement ≅ single read or write to global variable
- precise definition is tricky in Java, but we will learn to avoid pitfalls