



UNIVERSITY OF
GOTHENBURG

Introduction to concurrent programming

Lecture 1 of TDA384/DIT391

Principles of Concurrent Programming

Sandro Stucki

Chalmers University of Technology | University of Gothenburg
SP3 2018/2019

Based on course slides by Carlo A. Furia

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: \perp	counter: 0
2	pc: 7 cnt: 0	counter: 0
3	pc: 8 cnt: 0	counter: 1
4	pc: 6 cnt: \perp	counter: 1
5	pc: 7 cnt: 1	counter: 1
6	pc: 8 cnt: 1	counter: 2
7	done	counter: 2

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

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

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: \perp	counter: 0
2	pc: 7 cnt: 0	counter: 0
3	pc: 8 cnt: 0	counter: 1
4	pc: 6 cnt: \perp	counter: 1
5	pc: 7 cnt: 1	counter: 1
6	pc: 8 cnt: 1	counter: 2
7	done	counter: 2

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: \perp	counter: 0
2	pc: 7 cnt: 0	counter: 0
3	pc: 8 cnt: 0	counter: 1
4	pc: 6 cnt: \perp	counter: 1
5	pc: 7 cnt: 1	counter: 1
6	pc: 8 cnt: 1	counter: 2
7	done	counter: 2

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: \perp	counter: 0
2	pc: 7 cnt: 0	counter: 0
3	pc: 8 cnt: 0	counter: 1
4	pc: 6 cnt: \perp	counter: 1
5	pc: 7 cnt: 1	counter: 1
6	pc: 8 cnt: 1	counter: 2
7	done	counter: 2

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: \perp	counter: 0
2	pc: 7 cnt: 0	counter: 0
3	pc: 8 cnt: 0	counter: 1
4	pc: 6 cnt: \perp	counter: 1
5	pc: 7 cnt: 1	counter: 1
6	pc: 8 cnt: 1	counter: 2
7	done	counter: 2

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: \perp	counter: 0
2	pc: 7 cnt: 0	counter: 0
3	pc: 8 cnt: 0	counter: 1
4	pc: 6 cnt: \perp	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**.

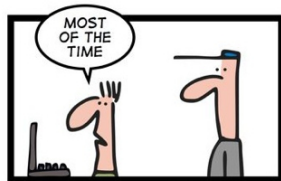
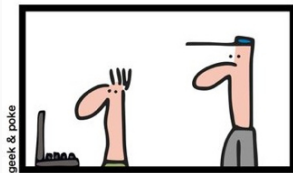
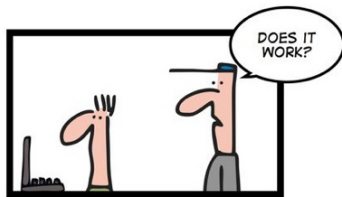
What?!

```
$ 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!

SIMPLY EXPLAINED



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)

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.

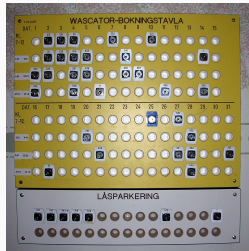


Photo: Wolfgang Mozart 2010.

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).

Concurrency vs. parallelism

Real world situation



Photo: Daniel Mott 2009.

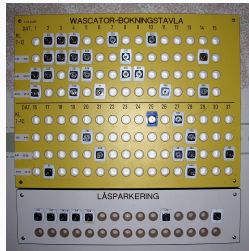


Photo: Wolfgang Mozart 2010.

Challenges:

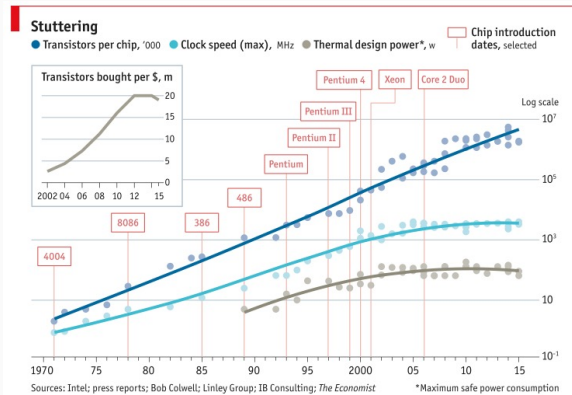
- concurrency: everyone gets to do their laundry (fairness), machines are operated by at most one user (mutual exclusion);
- parallelism: distribute load evenly over machines/rooms (load balancing).

Solutions: schedules, locks, signs/indicators. . .

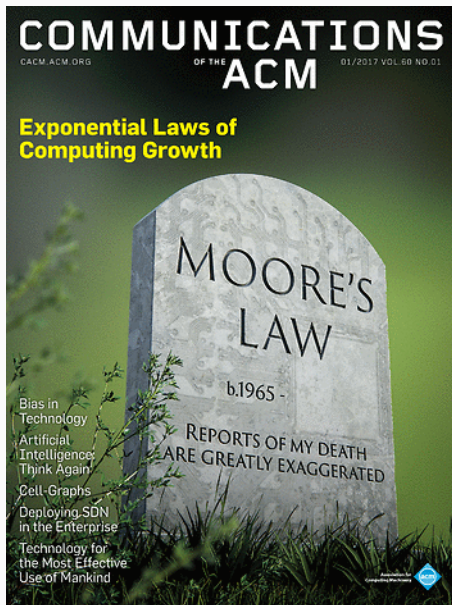
Moore's law and its end

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



Concurrency everywhere

The end of Moore's law is 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, ...

Amdahl's law: concurrency is no free lunch

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

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

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

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

Amdahl's law: examples

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

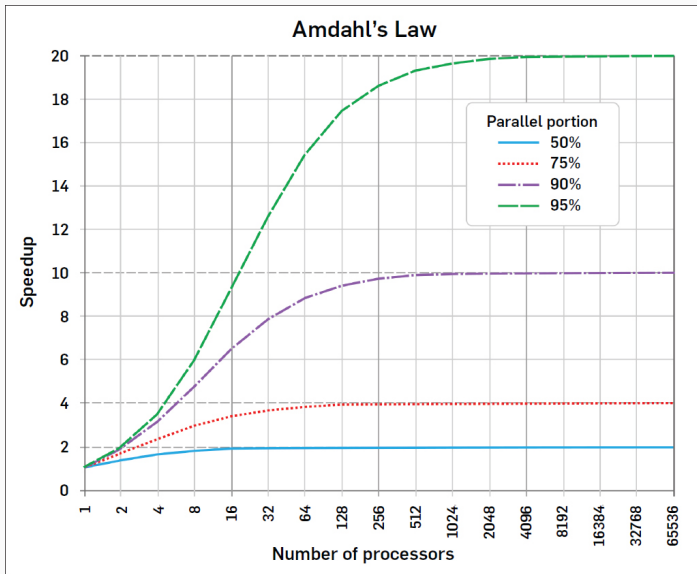
With $n = 10$ processors, how close can we get to a 10x speedup?

% 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



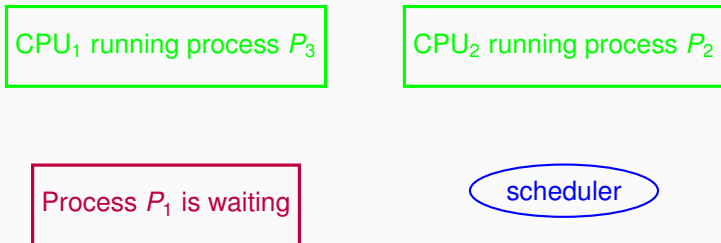
Basic terminology and abstractions

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:

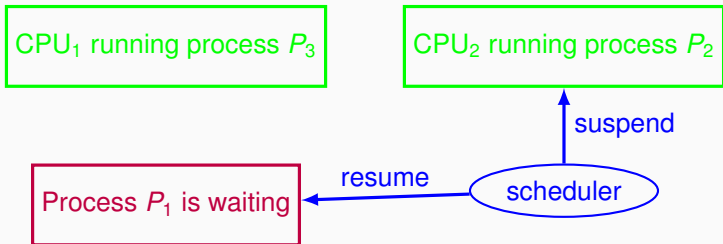


Processes

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:

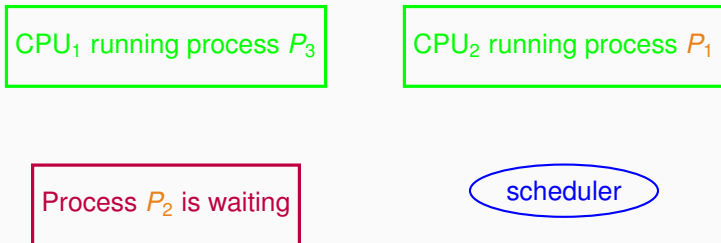


Processes

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:



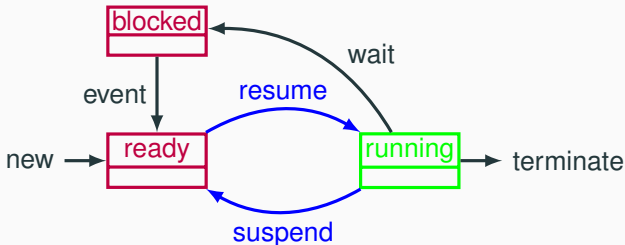
Process states

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

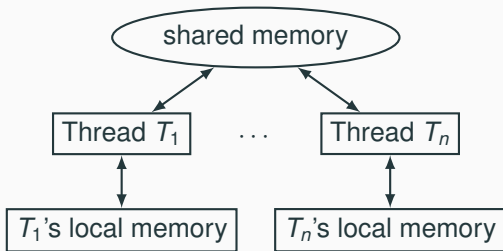
running: running on some CPU



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:

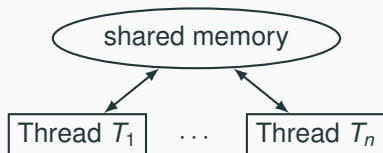
processes: executing units that do not share memory (in **Erlang**)

threads: executing units that share memory (in **Java**)

Shared memory vs. message passing

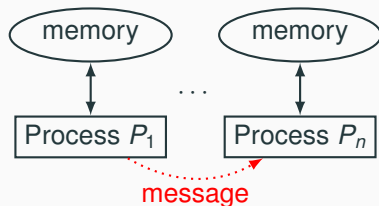
Shared memory models:

- 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();
```

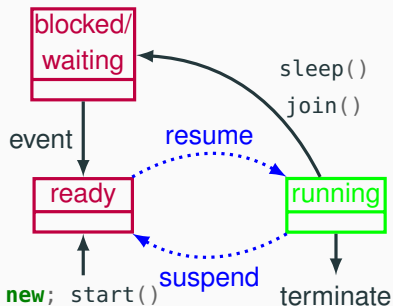
```
    Thread t = new Thread(c);
```

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

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

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

States of a Java thread

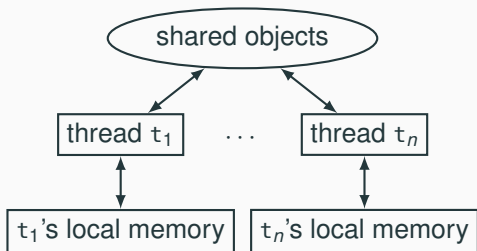


Resuming and suspending is done by the JVM scheduler, outside the program's control.

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.

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 : ⊥	pc _u : 6 cnt _u : ⊥	counter: 0
2	pc _t : 7 cnt _t : 0	pc _u : 6 cnt _u : ⊥	counter: 0
3	pc _t : 8 cnt _t : 0	pc _u : 6 cnt _u : ⊥	counter: 1
4	done	pc _u : 6 cnt _u : ⊥	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 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 : ⊥	pc _u : 6 cnt _u : ⊥	counter: 0
2	pc _t : 7 cnt _t : 0	pc _u : 6 cnt _u : ⊥	counter: 0
3	pc _t : 8 cnt _t : 0	pc _u : 6 cnt _u : ⊥	counter: 1
4	done	pc _u : 6 cnt _u : ⊥	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 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 : ⊥	pc _u : 6 cnt _u : ⊥	counter: 0
2	pc _t : 7 cnt _t : 0	pc _u : 6 cnt _u : ⊥	counter: 0
3	pc _t : 8 cnt _t : 0	pc _u : 6 cnt _u : ⊥	counter: 1
4	done	pc _u : 6 cnt _u : ⊥	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 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 : ⊥	pc _u : 6 cnt _u : ⊥	counter: 0
2	pc _t : 7 cnt _t : 0	pc _u : 6 cnt _u : ⊥	counter: 0
3	pc _t : 8 cnt _t : 0	pc _u : 6 cnt _u : ⊥	counter: 1
4	done	pc _u : 6 cnt _u : ⊥	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 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 : ⊥	pc _u : 6 cnt _u : ⊥	counter: 0
2	pc _t : 7 cnt _t : 0	pc _u : 6 cnt _u : ⊥	counter: 0
3	pc _t : 8 cnt _t : 0	pc _u : 6 cnt _u : ⊥	counter: 1
4	done	pc _u : 6 cnt _u : ⊥	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 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 : ⊥	pc _u : 6 cnt _u : ⊥	counter: 0
2	pc _t : 7 cnt _t : 0	pc _u : 6 cnt _u : ⊥	counter: 0
3	pc _t : 8 cnt _t : 0	pc _u : 6 cnt _u : ⊥	counter: 1
4	done	pc _u : 6 cnt _u : ⊥	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 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 : ⊥	pc _u : 6 cnt _u : ⊥	counter: 0
2	pc _t : 7 cnt _t : 0	pc _u : 6 cnt _u : ⊥	counter: 0
3	pc _t : 8 cnt _t : 0	pc _u : 6 cnt _u : ⊥	counter: 1
4	done	pc _u : 6 cnt _u : ⊥	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 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 : ⊥	pc _u : 6 cnt _u : ⊥	counter: 0
2	pc _t : 7 cnt _t : 0	pc _u : 6 cnt _u : ⊥	counter: 0
3	pc _t : 8 cnt _t : 0	pc _u : 6 cnt _u : ⊥	counter: 1
4	done	pc _u : 6 cnt _u : ⊥	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 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 : ⊥	pc _u : 6 cnt _u : ⊥	counter: 0
2	pc _t : 7 cnt _t : 0	pc _u : 6 cnt _u : ⊥	counter: 0
3	pc _t : 8 cnt _t : 0	pc _u : 6 cnt _u : ⊥	counter: 1
4	done	pc _u : 6 cnt _u : ⊥	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 : ⊥	pc _u : 6 cnt _u : ⊥	counter: 0
2	pc _t : 7 cnt _t : 0	pc _u : 6 cnt _u : ⊥	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

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 : ⊥	pc _u : 6 cnt _u : ⊥	counter: 0
2	pc _t : 7 cnt _t : 0	pc _u : 6 cnt _u : ⊥	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

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 : ⊥	pc _u : 6 cnt _u : ⊥	counter: 0
2	pc _t : 7 cnt _t : 0	pc _u : 6 cnt _u : ⊥	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

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 : ⊥	pc _u : 6 cnt _u : ⊥	counter: 0
2	pc _t : 7 cnt _t : 0	pc _u : 6 cnt _u : ⊥	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

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 : ⊥	pc _u : 6 cnt _u : ⊥	counter: 0
2	pc _t : 7 cnt _t : 0	pc _u : 6 cnt _u : ⊥	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

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 : ⊥	pc _u : 6 cnt _u : ⊥	counter: 0
2	pc _t : 7 cnt _t : 0	pc _u : 6 cnt _u : ⊥	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

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 : ⊥	pc _u : 6 cnt _u : ⊥	counter: 0
2	pc _t : 7 cnt _t : 0	pc _u : 6 cnt _u : ⊥	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

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 : ⊥	pc _u : 6 cnt _u : ⊥	counter: 0
2	pc _t : 7 cnt _t : 0	pc _u : 6 cnt _u : ⊥	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

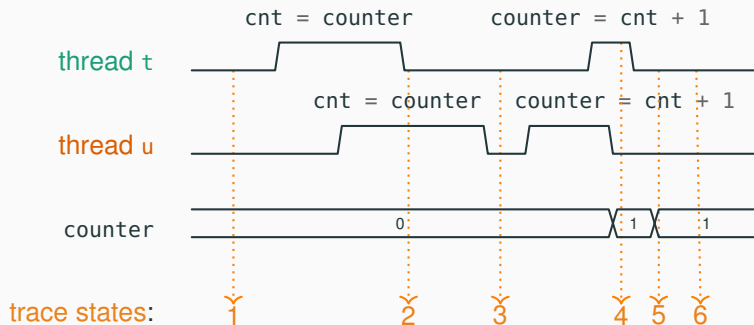
Traces

#	t'S LOCAL	u'S LOCAL	SHARED
1	pc _t : 6 cnt _t : ⊥	pc _u : 6 cnt _u : ⊥	counter: 0
2	pc _t : 7 cnt _t : 0	pc _u : 6 cnt _u : ⊥	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



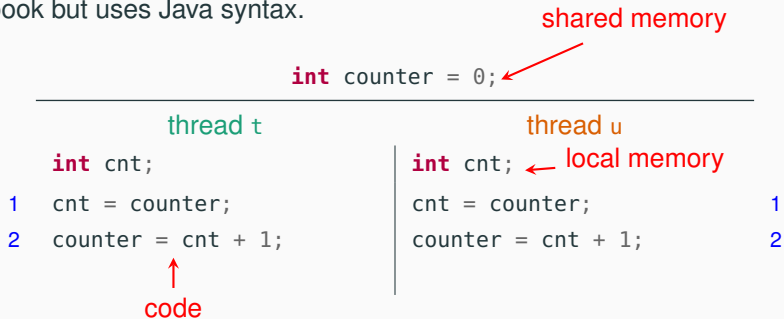
atomic/linearized: the effects of each thread appear as if they happened **instantaneously**, when the trace snapshot is taken, in the thread's **sequential order**

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.



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

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

These slides' license

© 2016–2019 Carlo A. Furia, Sandro Stucki



Except where otherwise noted, this work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

To view a copy of this license, visit

<http://creativecommons.org/licenses/by-sa/4.0/>.