Introduction to concurrent programming

Lecture 1 of TDA384/DIT391
(Principles of Concurrent Programming)

Carlo A. Furia

Chalmers University of Technology – University of Gothenburg

SP3 2017/2018

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

```java
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;
  }
}
```

```java
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: ⊥	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: ⊥	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: ⊥	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

```java
                          public class ConcurrentCount {
                            public static void main(String[] args) {
                              CCounter counter = new CCounter();
                              // threads t and u, sharing counter
public class CCounter         Thread t = new Thread(counter);
    extends Counter           Thread u = new Thread(counter);
    implements Runnable       t.start(); // increment once
{                             u.start(); // increment twice
    // threads                try { // wait for t and u to terminate
    // will execute             t.join(); u.join();
    // run()                  } 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!
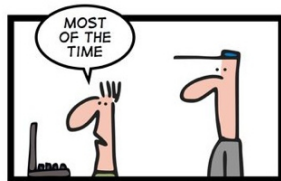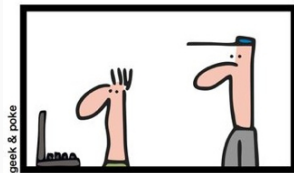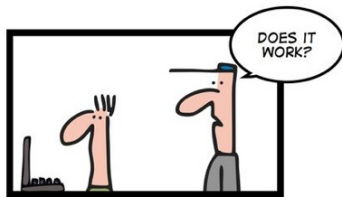
Why concurrency?

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

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)
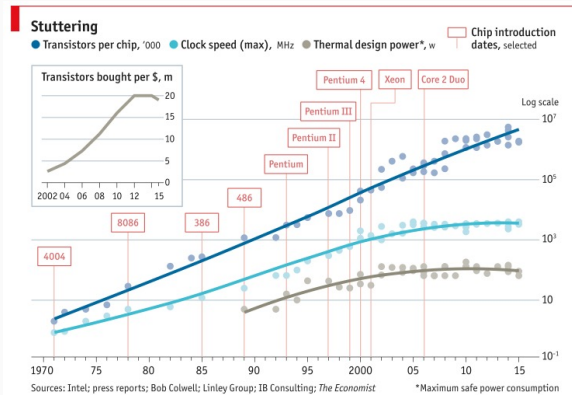
parallelism: efficient execution of fractions of a complex task on multiple processing units (physical parallelism)

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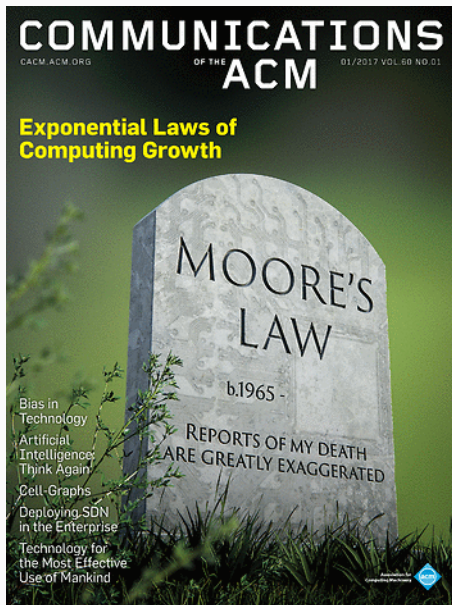- Parallelism is mainly about speeding up computations by taking advantage of redundant hardware

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

Stuttering

● Transistors per chip, '000 ● Clock speed (max), MHz ● Thermal design power*, w   ☐ Chip introduction dates, selected

Transistors bought per $, m

Pentium 4   Xeon   Core 2 Duo

Pentium III

Pentium II

Pentium

486

8086

386

4004

Log scale

1970   75   80   85   90   95   2000   05   10   15

Sources: Intel; press reports; Bob Colwell; Linley Group; IB Consulting; The Economist   *Maximum safe power consumption

Concurrency everywhere

The end of Moore's law is having a major impact on the practice of programming:

- before: CPUs get faster 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: CPUs do not get faster but add more and more parallel cores
    - 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

$$\text{maximum speedup} = \frac{1}{\underbrace{(1-p)}_{\text{sequential part}} + \underbrace{p/n}_{\text{parallel part}}}$$
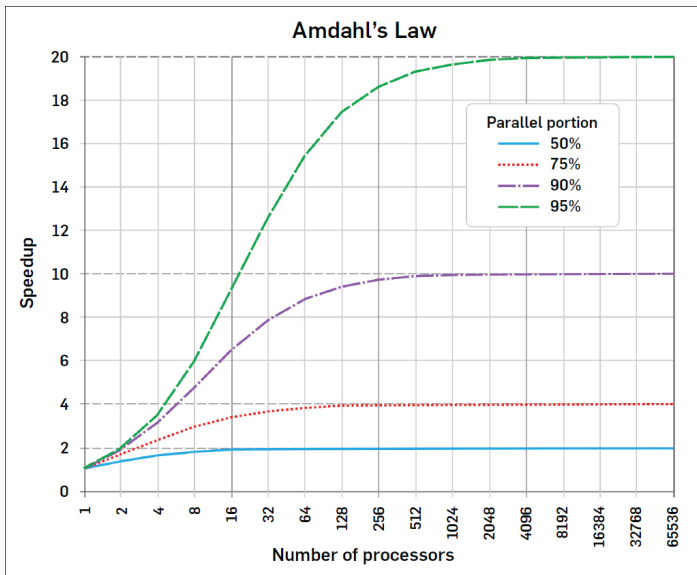
With $n = 10$ processors, how close can we get to a 10x speedup?

% 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

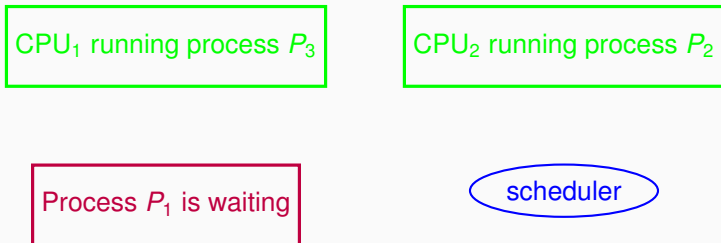# Basic terminology and abstractions

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:

CPU$_1$ running process P_3

CPU$_2$ running process P_2

Process P_1 is waiting

scheduler

Processes

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

- identifier
- program counter
- memory space

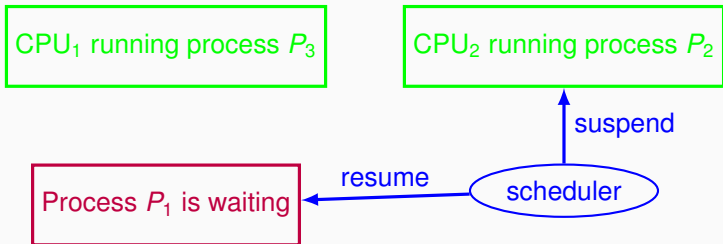The runtime/operating system schedules processes for execution on the available processors:

Processes

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

- identifier
- program counter
- memory space

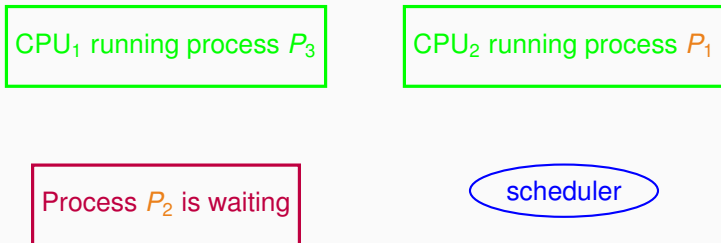The runtime/operating system schedules processes for execution on the available processors:

CPU$_1$ running process P_3

CPU$_2$ running process P_1

Process P_2 is waiting

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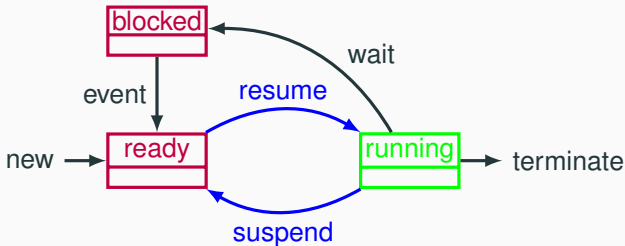
The scheduler is the system unit in charge of setting process states:

ready: ready to be executed, but not allocated to any CPU
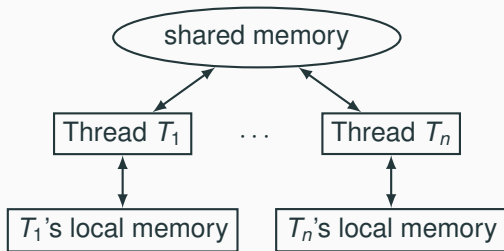**blocked:** waiting for an event to happen
running: running on some CPU

Threads

A thread is a lightweight process – an independent unit of execution on 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:
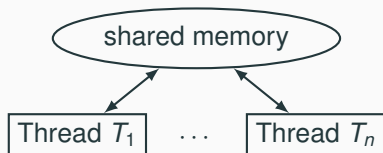
processes: executing units that do not share memory (in Erlang)

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
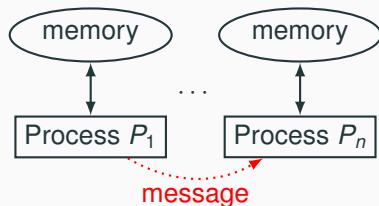
Java threads

Java threads

Two ways to build multi-threaded programs in Java:

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

```java
public class CCounter
  implements Runnable              CCounter c = new CCounter();
{
  // thread's computation:         Thread t = new Thread(c);
  public void run() {              Thread u = new Thread(c);
    int cnt = counter;
    counter = cnt + 1;             t.start();
  }                                u.start();
}
```

States of a Java thread
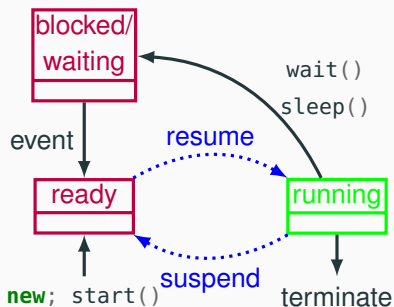
Resuming and suspending is done by the JVM scheduler, outside the program's control

For a thread object `t`:

- `t.start()`: the thread is ready for execution
- `t.sleep(n)`: block the thread for `n` milliseconds (correct timing depends on JVM implementation)
- `t.wait()`: block the thread until an event occurs
- `t.join()`: block the current thread until `t` terminates
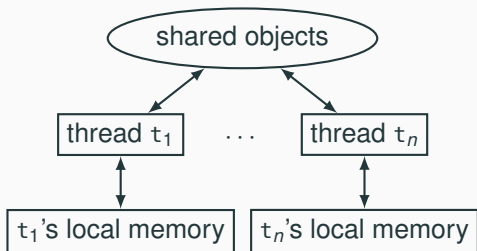
Shared vs. thread-local memory:

- shared objects: the object on which the thread operate, 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.

```
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 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: $\bot$	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: 8 cnt_t: 0	pc_u: 6 cnt_u: $\bot$	counter: 1
4	done	pc_u: 6 cnt_u: $\bot$	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

```
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: $\bot$	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: 8 cnt_t: 0	pc_u: 6 cnt_u: $\bot$	counter: 1
4	done	pc_u: 6 cnt_u: $\bot$	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: $\bot$	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: 8 cnt_t: 0	pc_u: 6 cnt_u: $\bot$	counter: 1
4	done	pc_u: 6 cnt_u: $\bot$	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: $\bot$	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: 8 cnt_t: 0	pc_u: 6 cnt_u: $\bot$	counter: 1
4	done	pc_u: 6 cnt_u: $\bot$	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

```java
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 possible execution of the concurrent counter

```java
public class CCounter implements Runnable {
    int counter = 0;      // shared object state

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

#	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

```java
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 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: $\bot$	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: 8 cnt_t: 0	pc_u: 6 cnt_u: $\bot$	counter: 1
4	done	pc_u: 6 cnt_u: $\bot$	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 : $\bot$	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

```
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: $\bot$	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

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: $\bot$	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

```
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: $\bot$	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

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

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: $\bot$	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

One alternative execution of the concurrent counter

```java
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: $\bot$	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

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: $\bot$	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

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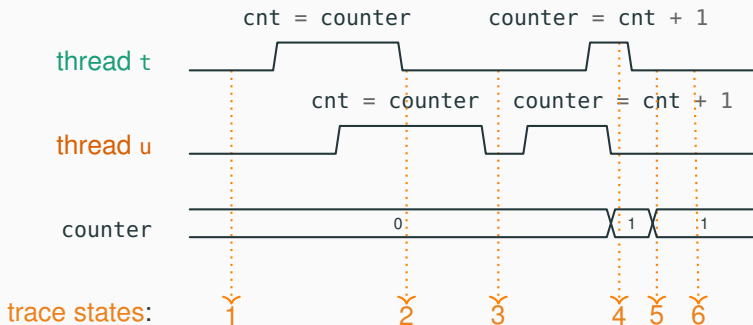
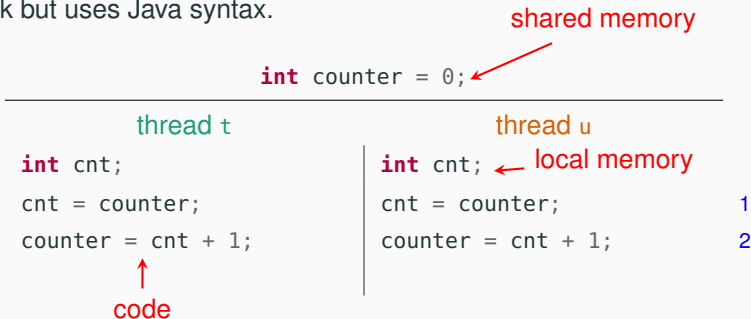**atomic/linearized:** the effects of each thread appear as if they
happened instantaneously, when the trace snapshot is
taken, in the thread's sequential order

complete: the trace include 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.

shared memory

```
int counter = 0;
```

thread t	thread u
`int cnt;`	`int cnt;` local memory
1 `cnt = counter;`	`cnt = counter;` 1
2 `counter = cnt + 1;`	`counter = cnt + 1;` 2

code

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

© 2016–2018 Carlo A. Furia