Concurrent and Distributed Systems

Talk Two

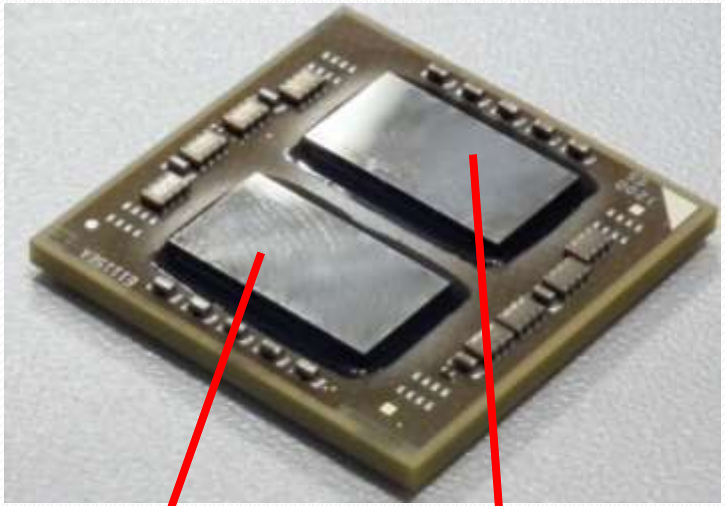
Basics of Concurrency

Maxim Mozgovoy

Parallelism: Real and Abstract

How several algorithms can work in parallel?

In case of multicore / multiprocessor systems and networks each CPU can work on its own task:

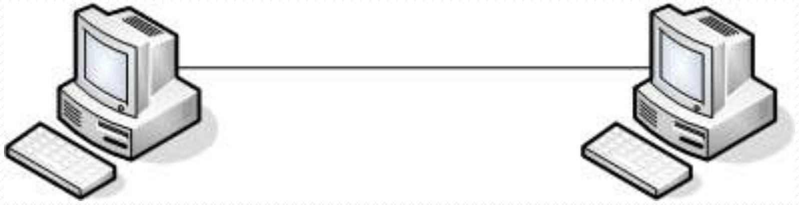


Runs task 1 Runs task 2

dual-core CPU

computational network

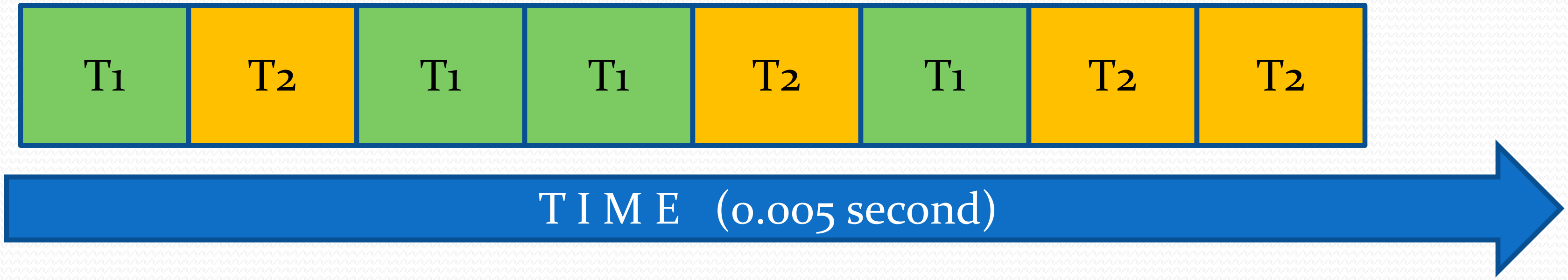
Runs task 1 Runs task 2



Parallelism: Real and Abstract

If tasks > processors,

the system implements abstract parallelism with time slicing:



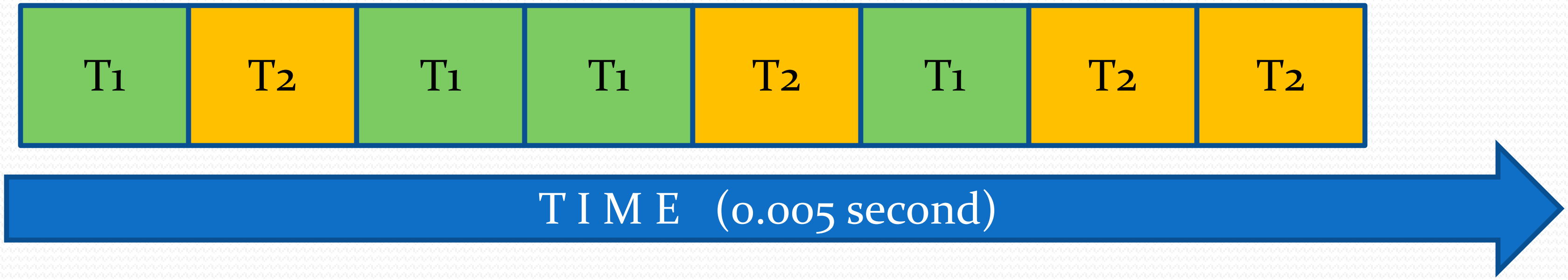
T1 T2 T1 T1 T2 T1 T2 T2

TIME (0.005 second)

For the user, it looks like both task are executed in parallel

(of course, no speed improvements in this case;
but the best speed is not always needed)

Parallelism: Real and Abstract



T1 T2 T1 T1 T2 T1 T2 T2

TIME (0.005 second)

Time slicing is possible because modern processors and operating systems support preemptive multitasking:

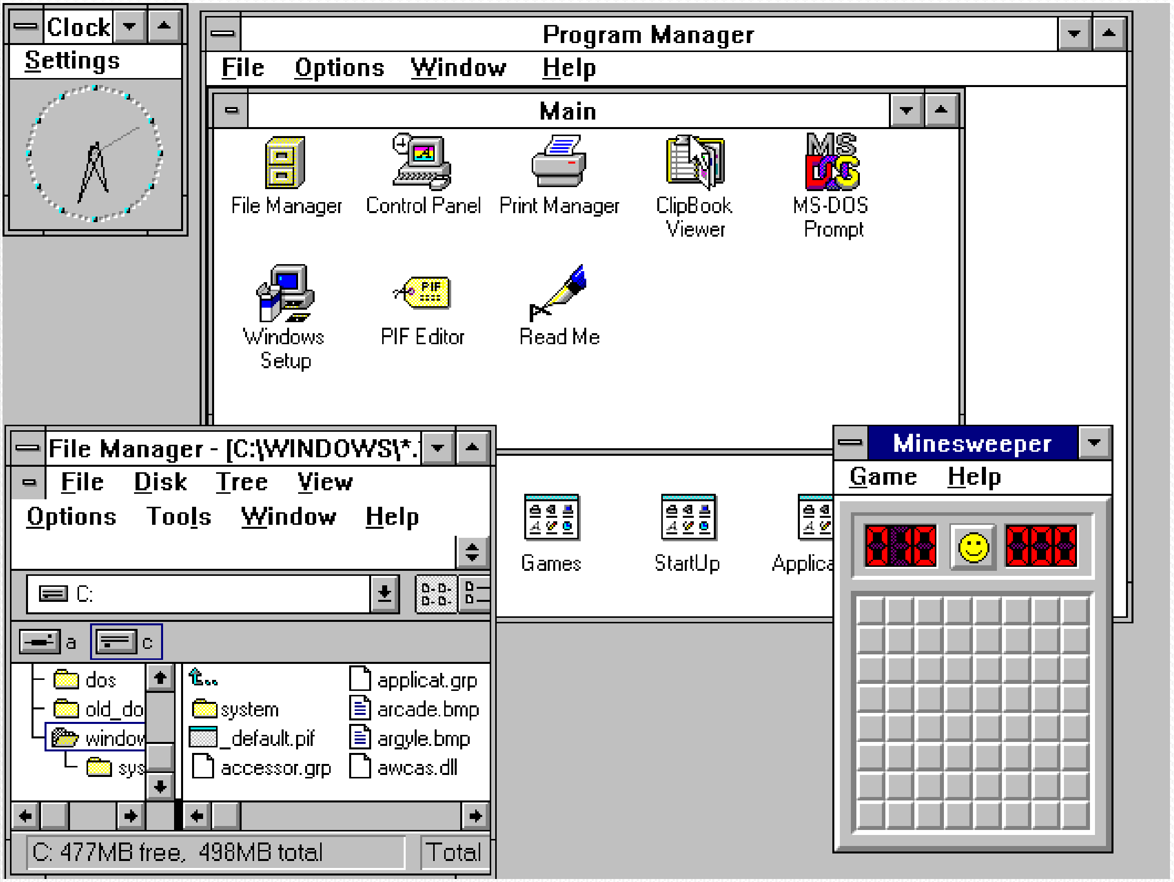
OS scheduler can interrupt any process and give some time to another process (such operations have to be supported by the CPU hardware).

It is scheduler’s job to prevent starvation, i.e., to give a time slot to each process. However, users can affect scheduling policy by changing process priorities.

Parallelism: Real and Abstract

Older operating systems used cooperative multitasking:

each parallel process had to do some work and then switch execution to another process (so it was a programmer’s job)

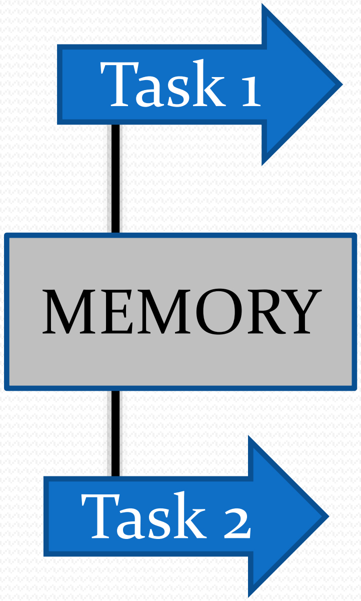


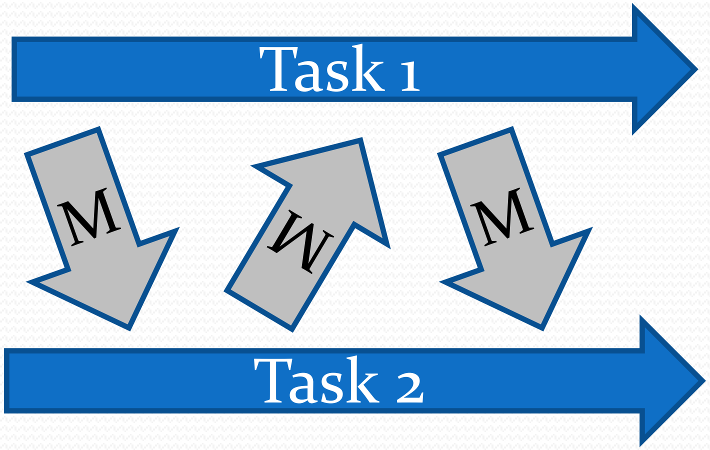
Example:
Windows 3.1 (1992)

If some process does not switch execution, the whole system freezes ☺

Where is the Data?

Parallel processes can store data in the shared memory, or in independent memory units and use message passing.





The choice often defined by equipment (multicore vs network)
But sometimes the developer can decide which model to use.

Parallelism in Our Course

We will develop concurrent / parallel systems using ordinary desktop computers.

For us, it does not matter how the OS runs several processes on one machine (is parallelism real or abstract).

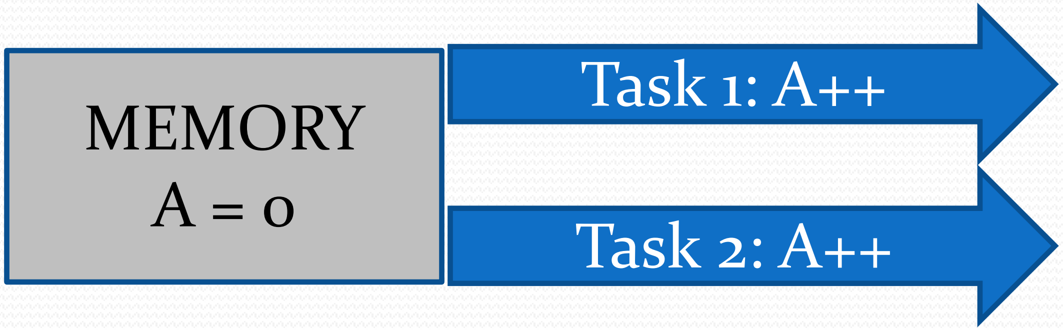
(The case is different for distributed systems,
but we will discuss networking later).

We will begin with concurrent systems working in shared memory model.

Processes that use the same memory are usually called threads.

Atomic Operations (1)

Difficulties arise when we need to coordinate processes.



MEMORY

A = 0

Task 1: A++

Task 2: A++

Expected result:

A = 2

However, it is possible
that machine code for A++ is:

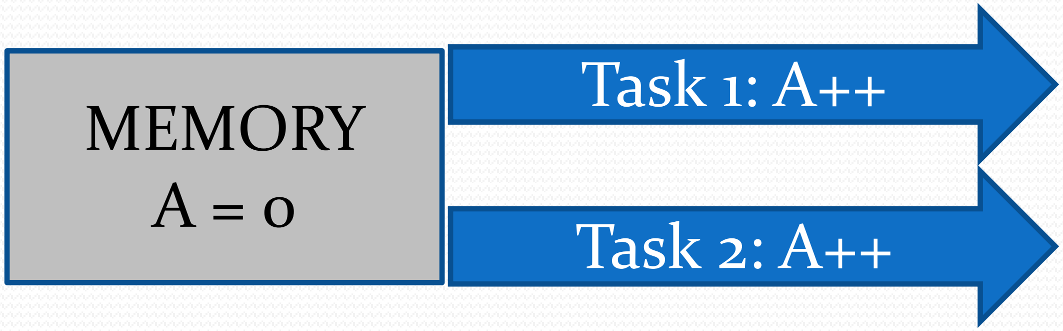
READ register X, (A)
INCREASE X
WRITE (A), register X

Then we might get:

T1	READ X, (A)	// $X_1 = 0$
T2	READ X, (A)	// $X_2 = 0$
T1	INCREASE X	// $X_1 = 1$
T2	INCREASE X	// $X_2 = 1$
T1	WRITE (A), X	// A = 1
T2	WRITE (A), X	// A = 1
		// A = 1

Atomic Operations (2)

Coordination is possible if we know that some operations are atomic and can only be executed in whole.



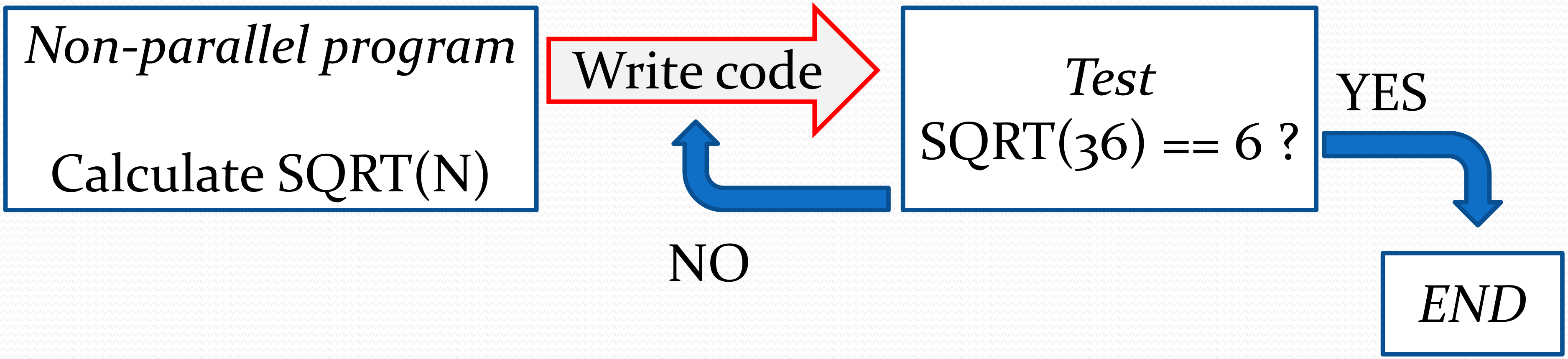
Always makes A = 2 if A++ is atomic.

Different CPUs or virtual machines have different atomic ops, but you can always convert any code fragment into an atomic operation (we will discuss later how to do it).

Scenarios (1)

Still, coordination is a challenge.

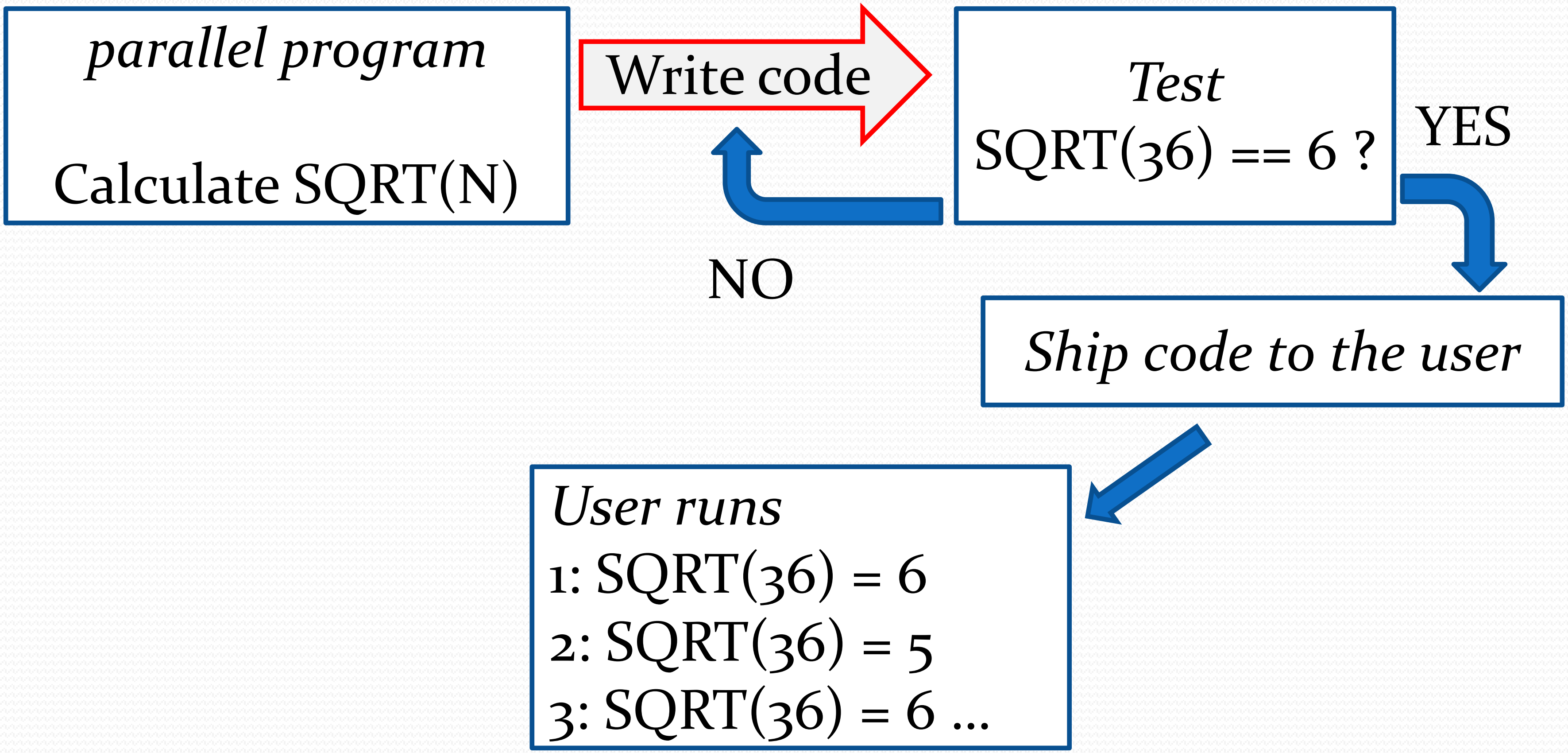
The biggest problem arise when the program sometimes prints different results. How to debug it?!



Scenarios (2)

Still, coordination is a challenge.

The biggest problem arise when the program sometimes prints different results. How to debug it?!



parallel program

Calculate SQRT(N)

Write code

Test

SQRT(36) == 6 ?

YES

NO

Ship code to the user

User runs

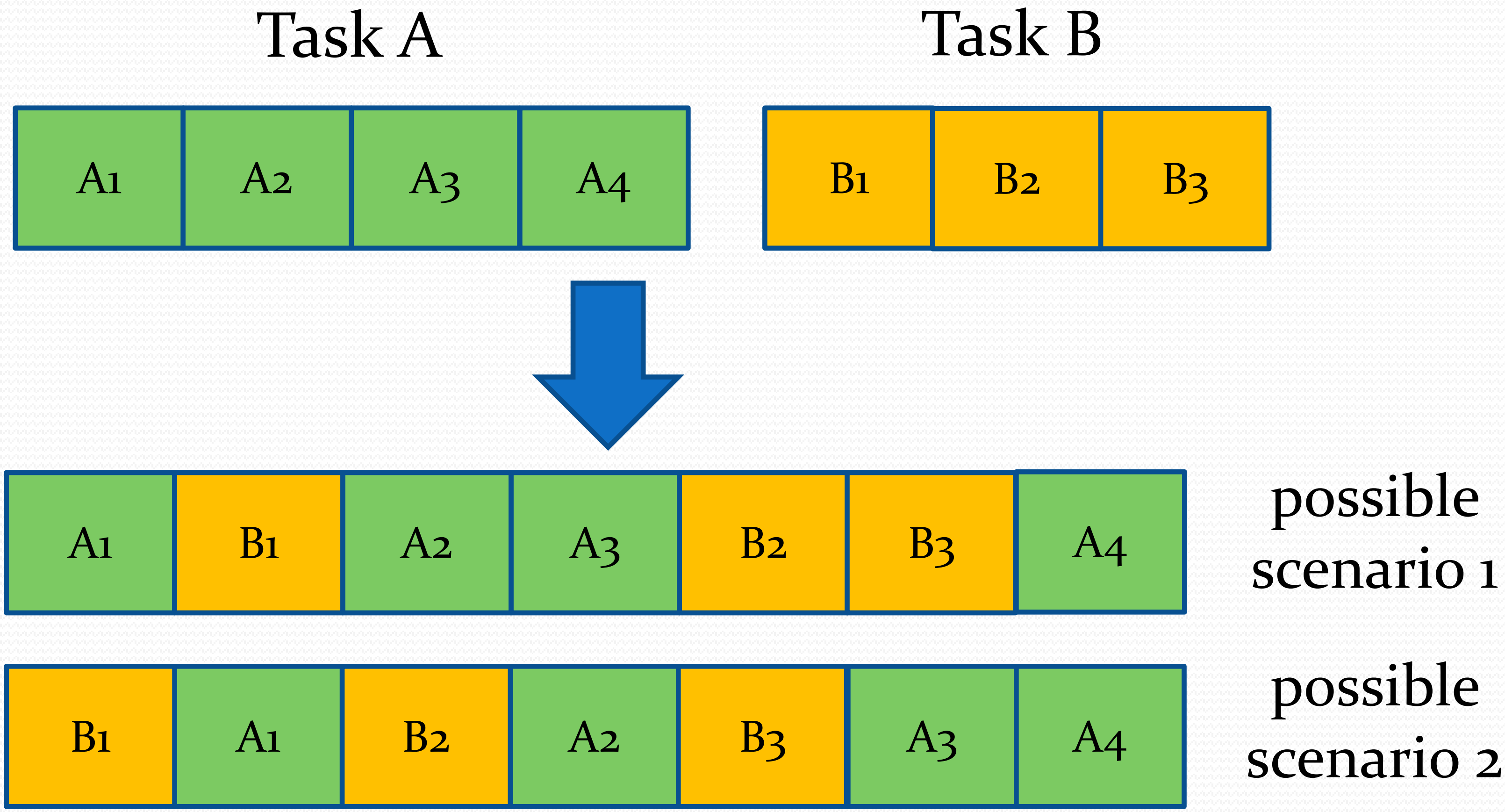
1: SQRT(36) = 6

2: SQRT(36) = 5

3: SQRT(36) = 6 …

Interleaving Model

When designing a concurrent program, it is convenient to treat it as a sequence of arbitrarily interleaved atomic statements:



Task A

A1 A2 A3 A4

Task B

B1 B2 B3

A1 B1 A2 A3 B2 B3 A4

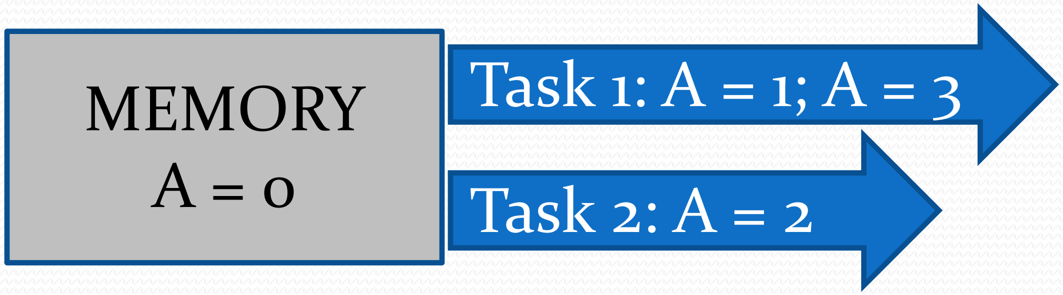
possible scenario 1

B1 A1 B2 A2 B3 A3 A4

possible scenario 2

Understanding State Space Diagrams

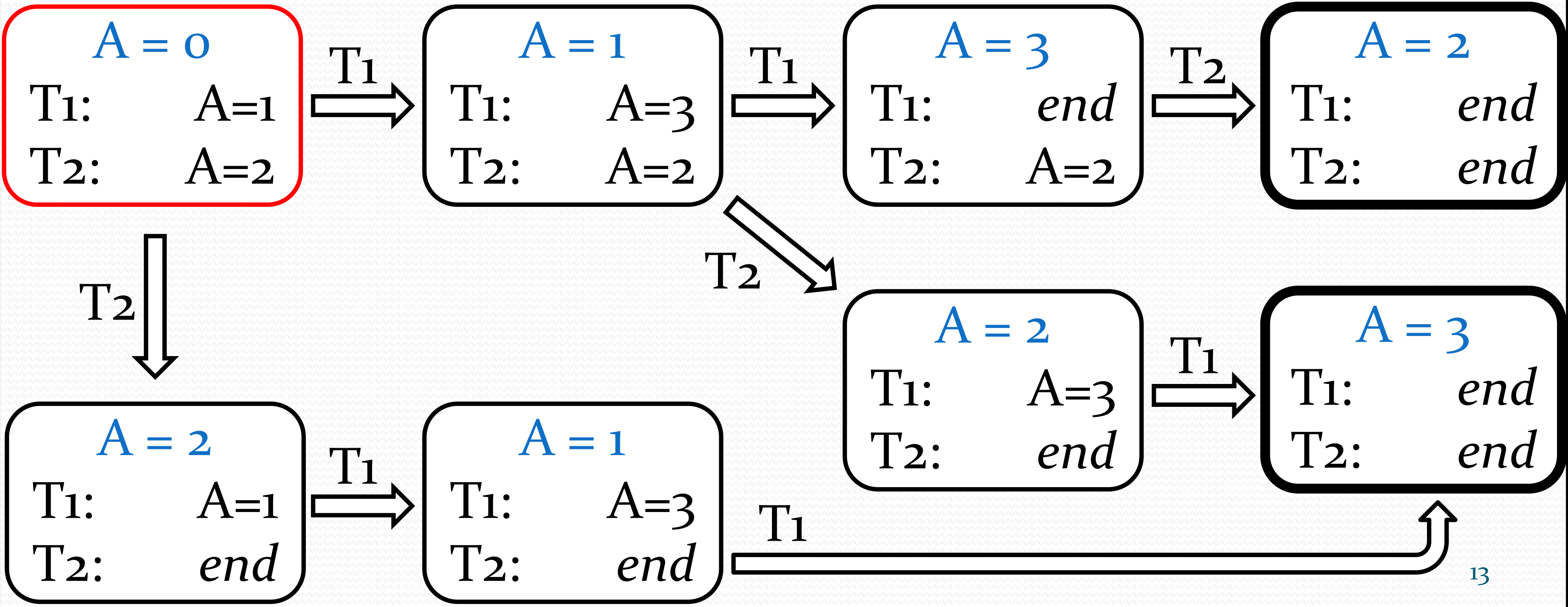
We can visualize different scenarios with state space diagrams:



MEMORY
A = 0

Task 1: A = 1; A = 3

Task 2: A = 2



A = 0
T1: A=1
T2: A=2

T1

A = 1
T1: A=3
T2: A=2

T1

A = 3
T1: *end*
T2: A=2

T2

A = 2
T1: *end*
T2: *end*

T2

A = 2
T1: A=3
T2: *end*

T1

A = 3
T1: *end*
T2: *end*

T2

A = 2
T1: A=1
T2: *end*

T1

A = 1
T1: A=3
T2: *end*

T1

The Goal

The programs should always provide the same results, and can be run on different computers.

We cannot assume atomicity of programming language instructions in general (a scheduler can swap threads anytime).

We can rely on special instructions, guaranteed to be atomic (to be discussed next week).

To check that the program provides the same results, we can analyze a (simplified) state space diagram.

Built-in Parallel Instructions

Prog. languages support concurrent programming differently. Some have special instructions (e.g, Chapel language):

```
var N: int = 500;
var A: A[1..N] int;

for i in 1..N do A[i] = i;       // ordinary loop
forall i in 1..N do A[i] = i;    // concurrent loop
```

The forall statement tells the compiler that the iterations are independent, and if multicore hardware is available, work should be done in parallel.

The coforall statement creates as many tasks as there are loop iterations regardless of available hardware.

Basic Concurrency in Java (1)

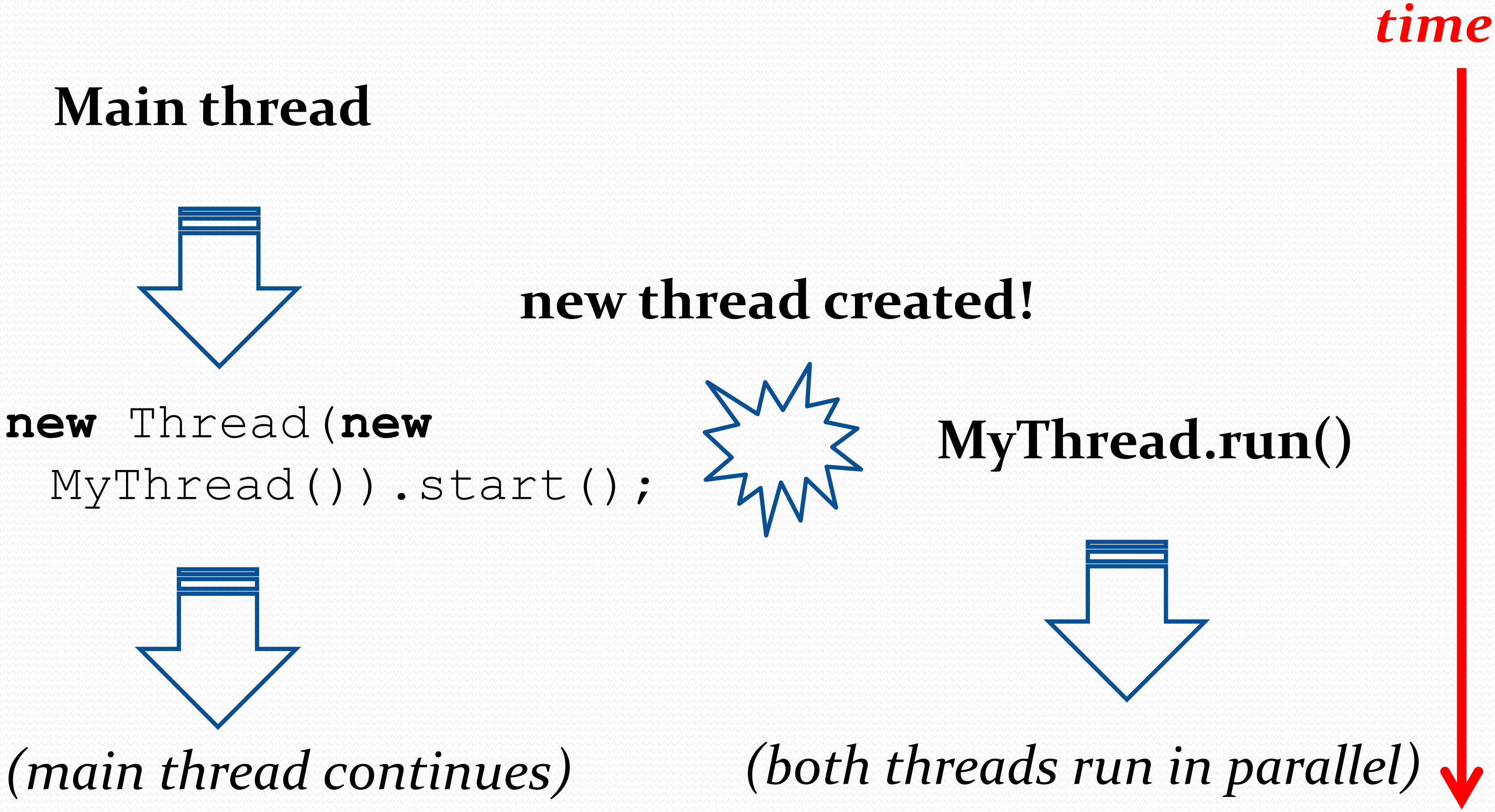
Languages such as Java and C++ use additional libraries.
So, the ordinary Java/C++ syntax does not change.

In Java, class Thread and interface Runnable (check Java API documentation!) can be used to make new processes:

```
class MyThread implements Runnable {
   public void run() { /* here is your code */ }
}
…
new Thread(new MyThread()).start(); // run it!
```

Basic Concurrency in Java (2)

The threads run as follows:



time

Main thread

new thread created!

```
new Thread(new
  MyThread()).start();
```

MyThread.run()

(main thread continues)

(both threads run in parallel)

Basic Concurrency in Java (3)

```
import java.util.Random;
class MyThread implements Runnable {
   private int ID;
   private Random rand;

   public MyThread(int id)    { ID = id;  rand = new Random(); }
   public void DoSleep()      { try { Thread.sleep(rand.nextInt(200)); }
                                catch(Exception e) {} }
   public void run() {
      for (int i = 0; i < 5; i++)  {
          System.out.println("ID: " + ID);
          DoSleep(); }
  }
}
```

Each thread executes:

```
DO 5 times:
   Print ID
   Sleep 0…200 msec
END
```

Basic Concurrency in Java (4)

```
// Create and run three threads:

class HelloThread {
    static public void main(String args[]) {
        for(int id = 1; id <= 3; ++id)
            new Thread(new MyThread(id)).start();
    }
}
```

A possible printout:

```
ID: 1        ID: 3
ID: 3        ID: 3
ID: 2        ID: 2
ID: 3        ID: 3
ID: 2        ID: 1
ID: 1        ID: 1
ID: 1        ID: 2
...          ID: 2
```

How Thread and Runnable Work?

- Java library uses the capabilities of the underlying operating system (Windows, Linux, etc.)
- When you create new Thread, Java asks the OS to start a new thread (process).
- If the computer has several processors, different threads can be run on different processors!
- If the number of physical processors is not enough, the system will use abstract parallelism (i.e., task switching).

- Basically, you don’t need to care how it works. It is operating system’s job to organize a multitasking environment.
- (By the way, making a good scheduler is not easy).

Communication with Variables (1)

- Some communication between threads can be achieved with global variables. This technique is safe if done right.
- In particular, reading and writing are atomic operations for **volatile** variables.
- Also, when the value of a global **volatile** variable changes, this change is immediately visible to all threads.
- (For non-volatile variables, a value might be cached: thread A sets a new value, but thread B still sees an old value).
- For example, suppose I want to run several threads until one of them generates a random value of 12345. It is safe to use a shared **volatile** flag to indicate whether a result is found (*see the next slide*)

Communication with Variables (2)

```
import java.util.Random;
class MyThread implements Runnable {
   private Random rand;
   private static volatile boolean found = false;
   public MyThread()    {rand = new Random(); }
   public void run() {
      while (!found)
         if(rand.nextInt(100000) == 12345)
            found = true;
   }
}
```

```
class HelloThread {
   static public void main(String args[]) {
      new Thread(new MyThread()).start();
      new Thread(new MyThread()).start();
   }
}
```

Conclusions

- For now, we will concentrate on concurrent programs that work under abstract parallelism (using one or more CPUs)
- Concurrent programs use shared memory, and their instructions are executed in unknown order by the scheduler.
- Normally, a programmer does not know which instructions are atomic (uninterruptable) on a given machine.
- In Java, you can make new processes (threads) by using class Thread and interface Runnable.
- We can visualize possible scenarios of a concurrent program's execution by using state space diagrams.