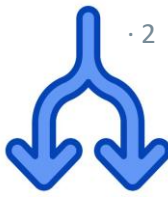# Practical Concurrent and Parallel Programming II

Shared Memory I

Raúl Pardo

- As of Aug 31st
  - 30 people do not have a group
  – *Please contact us if you are having trouble finding a group*

  - 65 people have not booked and oral feedback slot
  – *Please contact us if you cannot attend existing slots*

You must sign up for an oral feedback slot before next week

After you register in an oral feedback slot,
please do not change it unless it is strictly necessary

Submission next week (a few remarks)
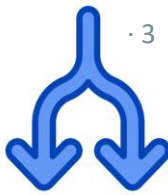
- Next week on Monday at 07.59 is the first submission deadline

- Your group submission must contain a link to a GitHub repository

- The repository must be readable (not private) by TAs and teachers
  - It must be possible to get a copy of your repository with a simple clone command

- You are not allowed to make changes to the repository after the submission deadline

- One of the members of the group submits a link to the same repository
  - Make sure that you submit as a group

- If you were eligible for examination last year, you do not need to resubmit the assignments (neither join a group nor booking a slot)
 - Notify us so that we can verify your assignments from last year

- Introduction to concurrency
- Java threads
- The Mutual Exclusion Problem (today)
- Java Locks (today)

- Readers and Writers Problem
- Monitors
- Fairness
- Java Intrinsic Locks (**`synchronized`**)
- Hardware and Programming Language Concurrency Issues
- Visibility
- Reordering
- Volatile variables (**`volatile`**)

- Consider a shared data structure (e.g., an array, list set, …) where threads may read and write

- Many threads can read from the structure as long as no thread is writing

- At most one thread can write at the same time

Writer *n*

Writer 2

Writer 1

Reader 1

Reader 2

Reader *n*

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2025

Readers-Writers Problem

- Consider a shared data structure (e.g., an array, list set, …) where threads may read and write
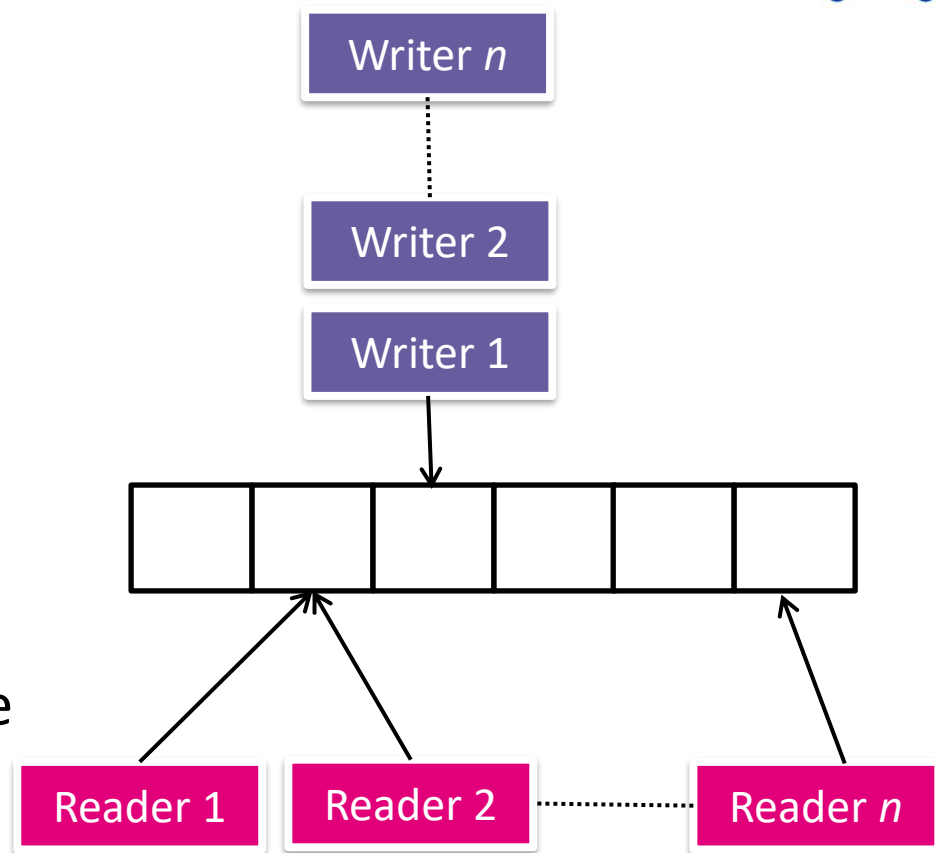
- Many threads can read from the structure as long as no thread is writing
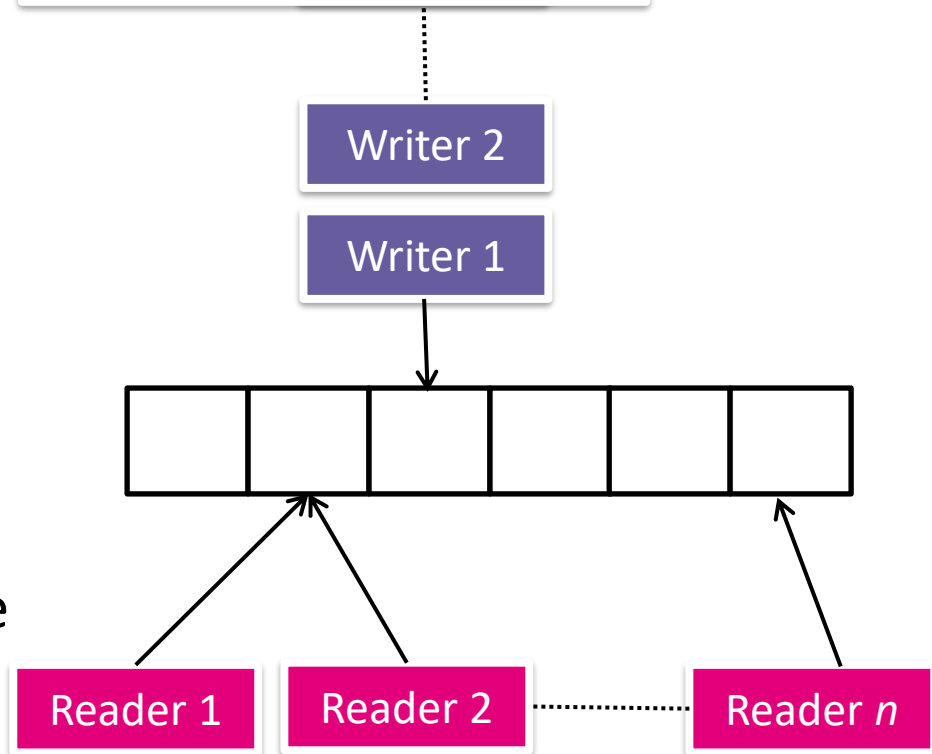
- At most one thread can write at the same time

Can we solve this problem using locks as we saw last week?

Writer 2

Writer 1

Reader 1   Reader 2 .... Reader *n*

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2025

- A *monitor* is a structured way of encapsulating data, methods and synchronization in a single modular package
    - First introduced by Tony Hoare (right photo, see optional readings) and the Danish computer scientist Per Brinch Hansen (left photo)

- *A monitor consists of:*

    - Internal state (data)

    - Methods (procedures)
        - All methods in a monitor are mutually exclusive (ensured via locks)
        - Methods can only access internal state

    - Condition variables (or simply conditions)
        - Queues where the monitor can put threads to wait

- In Java (and generally in OO), monitors are conveniently implemented as classes

- Conditions are used when a thread must wait for something to happen, e.g.,
  - A writer thread waiting for all readers and/or writer to finish
  - A reader waiting for the writer to finish

- Queues in condition variables provide the following interface:
  - `await()` – releases the lock, and blocks the thread (on the queue)
  - `signal()` – wakes up a thread blocked on the queue, if any
  - `signalAll()` – wakes up all threads blocked on the queue, if any

- When threads wake up they acquire the lock immediately (before the execute anything else)

- The snippet on the right shows a common structure for monitors in Java (pseudo-code)
  - See, e.g., `ReadWriteMonitor.java` for an actual implementation

- State variables are accessible to all methods in the monitor

- The method is mutually exclusive (using a `ReentrantLock`)

- Note also the use of the condition variable, and how it is associated to the lock
  - `await()` may throw `InterruptedExceptions`

```
…
// state variables
int i = 0;
Lock l = new ReentrantLock();
Condition c = l.newCondition();
…

// method example
public void method(…) {
    l.lock()
    try{
        while(i>0) {
            c.await()
        }
    }
    catch (InterruptedException e) {…}
    finally {l.unlock();}
}
```

- The snippet on the right shows a common structure for monitors in Java (pseudo-code)
  - See, e.g., `ReadWriteMonitor.java` for an actual implementation

- State variables are accessible to all methods in the monitor

- The method is mutually exclusive (using a `ReentrantLock`)

- Note also the use of the condition variable, and how it is associated to the lock
  - `await()` may throw `InterruptedExceptions`

```
…
// state variables
int i = 0;
Lock l = new ReentrantLock();
Condition c = l.newCondition();
…

// method example
public void method(…) {
    l.lock()
    try{
        while(i>0) {
            c.await()
        }
    }
    catch (InterruptedException e) {…}
    finally {l.unlock();}
}
```

This is the **condition variable** or **condition**

Do not confuse them with conditions in if-statements or while-statements

- First, we define the _state of the monitor_
- An integer counts the current number of reader threads
- A boolean marks whether a thread is writing
- We use **ReentrantLock** to ensure mutual exclusion
- We use a **Condition** variable to selectively decide whether a thread must wait to read/write (see next slide)

```java
public class ReadWriteMonitor {
    private int readers        = 0;
    private boolean writer     = false;
    private Lock lock          = new ReentrantLock();
    private Condition condition = lock.newCondition();
…
}
```

- We define four methods to lock and unlock read and write access to the shared resource

```
public void readLock() {
    lock.lock();
    try {
        while(writer)
          condition.await();
        readers++;
    }
    catch (InterruptedException e) {…}
    finally {lock.unlock();}
}

public void readUnlock() {
    lock.lock();
    try {
        readers--;
        if(readers==0)
          condition.signalAll();
    }
    finally {lock.unlock();}
}
```

```
public void writeLock() {
    lock.lock();
    try {
        while(readers > 0 || writer)
          condition.await();
        writer=true;
    }
    catch (InterruptedException e) {…}
    finally {lock.unlock();}
}

public void writeUnlock() {
    lock.lock();
    try {
        writer=false;
        condition.signalAll();
    }
    finally {lock.unlock();}
}
```

- We define four methods to lock and unlock read and write access to the shared resource

```
public void readLock() {
    lock.lock();
    try {
        while(writer)
          condition.await();
        readers++;
    }
    catch (Int
    finally {l
}

public void readUnlock() {
    lock.lock();
    try {
        readers--;
        if(readers==0)
          condition.signalAll();
    }
    finally {lock.unlock();}
}
```

We check whether a writer is accessing the resource

If there is a writer, then we put the thread to wait

Otherwise, we increase the number of readers and let the thread proceed

```
public void writeLock() {
    lock.lock();
    try {
        while(readers > 0 || writer)
          condition.await();
        writer=true;
    }
    catch (InterruptedException e) {…}
    finally {lock.unlock();}
}

public void writeUnlock() {
    lock.lock();
    try {
        writer=false;
        condition.signalAll();
    }
    finally {lock.unlock();}
}
```

- We define four methods to lock and unlock read and write access to the shared resource

```java
public void readLock() {
    lock.lock();
    try {
        while(writer)
          condition.await();
        readers++;
    }
    catch (InterruptedException e) {…}
    finally {lock.unlock();}
}

public void readUnlock() {
    lock.lock();
    try {
        readers--;
        if(readers==0)
          condition.signalAll();
    }
    finally {lock.unlock();}
}
```

```java
public void writeLock() {
    lock.lock();
    try {
        while(readers > 0 || writer)
          condition.await();
        writer=true;
    }
    catch (InterruptedException e) {…}
    finally {lock.unlock();}
}

public void writeUnlock() {
    lock.lock();
    try {
        writer=false;
        condition.signalAll();
    }
    finally {lock.unlock();}
}
```

We decrease the number of readers unconditionally

If there are no more readers, we signal condition

- We define four methods to lock and unlock read and write access to the shared resource

We check whether there are readers or a writer accessing the resource

```
public void readLock() {
    lock.lock();
    try {
        while(writer)
          condition.await()
        readers++;
    }
    catch (InterruptedExcep
    finally {lock.unlock();
}

public void readUnlock() {
    lock.lock();
    try {
        readers--;
        if(readers==0)
          condition.signalAll();
    }
    finally {lock.unlock();}
}
```

If so, we put the thread to wait

If not, the writer takes the lock

```
public void writeLock(
    lock.lock();
    try {
        while(readers > 0 || writer)
          condition.await();
        writer=true;
    }
    .tch (InterruptedException e) {…}
    finally {lock.unlock();}
}

public void writeUnlock() {
    lock.lock();
    try {
        writer=false;
        condition.signalAll();
    }
    finally {lock.unlock();}
}
```

Monitors | Readers-Writers Problem

- We define four methods to lock and unlock read and write access to the shared resource

```java
public void readLock() {
    lock.lock();
    try {
        while(writer)
          condition.await();
        readers++;
    }
    catch (InterruptedException e) {…}
    finally {lock.unlock();}
}

public void readUnlock() {
    lock.lock();
    try {
        readers--;
        if(readers==0)
          condition.signalAll();
    }
    finally {lo
}
```

```java
public void writeLock() {
    lock.lock();
    try {
        while(readers > 0 || writer)
          condition.await();
        writer=true;
    }
    catch (InterruptedException e) {…}
    finally {lock.unlock();}
}

public void writeUnlock() {
    lock.lock();
    try {
        writer=false;
        condition.signalAll();
    }
    finally {lock.unlock();}
}
```

We release the writer lock unconditionally

We signal the condition for other threads to access the resource, if any

- We define four methods to lock and unlock read and write access to the

> Do we need the **while** in the locking methods, wouldn´t it suffice with an **if**?

```
public void readLock()
    lock.lock();
    try {
        while(writer)
          condition.await();
        readers++;
    }
    catch (InterruptedException e) {…}
    finally {lock.unlock();}
}

public void readUnlock() {
    lock.lock();
    try {
        readers--;
        if(readers==0)
          condition.signalAll();
    }
    finally {l
}
```

```
     id writeLock() {
    lock.lock();
    try {
        while(readers > 0 || writer)
          condition.await();
        writer=true;
    }
    catch (InterruptedException e) {…}
    finally {lock.unlock();}
}

public void writeUnlock() {
    lock.lock();
    try {
        writer=false;
        condition.signalAll();
    
    finally {lock.unlock();}
}
```

> We release the writer lock unconditionally

> We signal the condition for other threads to access the resource, if any

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2025

- We define four methods to lock and unlock read and write access to the

Do we need the **while** in the locking methods, wouldn´t it suffice with an **if**?

```
public void readLock()
    lock.lock();
    try {
        while(writer)
          condition.await();
```

Note: Threads waiting on a condition variable may spuriously wake up
(https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Condition.html)

```
public void readUnlock() {
    lock.lock();
    try {
        readers--;
        if(readers==0)
          condition.signalAll();
    }
    finally {lo
}
```

```
    oid writeLock() {
    lock.lock();
    try {
        while(readers > 0 || writer)
          condition.await();
        writer=true;
    }
    catch (InterruptedException e) {…}
    finally {lock.unlock();}
}

public void writeUnlock() {
    lock.lock();
    try {
        writer=false;
        condition.signalAll();
    }
    finally {lock.unlock();}
}
```

We release the writer lock unconditionally

We signal the condition for other threads to access the resource, if any

- We define four methods to lock and unlock read and write access to the shared resource

```
public void readLock() {
    lock.lock();
    try {
        while(writer)
          condition.await();
        readers++;
    }
    catch (InterruptedException e) {…}
    finally {lock.unlock();}
}

public void readUnlock() {
    lock.lock();
    try {
        readers--;
        if(readers==0)
          condition.signalAll();
    }
    finally {lock.unlock();}
}
```

```
public void writeLock() {
    lock.lock();
    try {
        while(readers > 0 || writer)
          condition.await();
        writer=true;
    }
    catch (InterruptedException e) {…}
    finally {lock.unlock();}
}

public void writeUnlock() {
    lock.lock();
    try {
        writer=false;
        condition.signalAll();
    }
    finally {lock.unlock();}
}
```

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2025

- We define four methods to lock and unlock read and write access to the shared resource

```java
public void readLock() {
    lock.lock();
    try {
        while(writer)
          condition.await();
        readers++;
        ...
    } catch ( ... n e) {…}
    ...
}

public void readUnlock() {
    lock.lock();
    try {
        readers--;
        if(readers==0)
          condition.signalAll();
    }
    finally {lock.unlock();}
}
```

> Is it necessary to check whether **readers==0**?

```java
public void writeLock() {
    lock.lock();
    try {
        while(readers > 0 || writer)
          condition.await();
        writer=true;
    }
    catch (InterruptedException e) {…}
    finally {lock.unlock();}
}

public void writeUnlock() {
    lock.lock();
    try {
        writer=false;
        condition.signalAll();
    }
    finally {lock.unlock();}
}
```

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2025

- We define four methods to lock and unlock read and write access to the shared resource

```java
public void readLock() {
    lock.lock();
    try {
        while(writer)
         condition.await();
        readers++;
    }
    ca
    fi
}
```

Read-write locks are part of the `java.util.concurrent.locks`
package: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReadWriteLock.html

```java
public void writeLock() {
    lock.lock();
    try {
        while(readers > 0 || writer)
         condition.await();
        writer=true;
```

```java
public void readUnlock() {
    lock.lock();
    try {
        readers--;
        if(readers==0)
         condition.signalAll();
    }
    finally {lock.unlock();}
}
```

```java
public void writeUnlock() {
    lock.lock();
    try {
        writer=false;
        condition.signalAll();
    }
    finally {lock.unlock();}
}
```

- Now we start several reader and writer threads

```
ReadWriteMonitor m = new ReadWriteMonitor();
for (int i = 0; i < 10; i++) {

    // start a reader
    new Thread(() -> {
        m.readLock();
        System.out.println(" Reader " + Thread.currentThread().getId() + " started reading");
        // read
        System.out.println(" Reader " + Thread.currentThread().getId() + " stopped reading");
        m.readUnlock();
    }).start();

    // start a writer
    new Thread(() -> {
        m.writeLock();
        System.out.println(" Writer " + Thread.currentThread().getId() + " started writing");
        // write
        System.out.println(" Writer " + Thread.currentThread().getId() + " stopped writing");
        m.writeUnlock();
    }).start();
}
```

- Let's run the **`ReadersWriters.java`** file

- Let's run the **ReadersWriters.java** file

- Most of the time (or always):
  - First all readers are executed
  - Then all writers are executed

- The monitor seems to be *unfair* towards writers

- In this course, *fairness* refers to *absence of starvation*

- To reason about fairness in a monitor, we should understand how the monitor puts threads to wait and wakes them up

- Monitors have two queues where threads may wait
  - Lock queue (a.k.a. entry queue)
  - Condition variable queue
    - Note: We call it here "queues" for historic reasons, but they do not behave like queues in Java. They are more like sets.

- Consider a monitor with three threads waiting on the lock queue

Lock queue (a.k.a entry queue)

| Thread 1 | Thread 2 | Thread 3 |

```
// Monitor state
Lock l = new Lock();
Condition c = l.newCondition();

public void m_executes_await() {
    …
    c.await();
    …
}

public void m_executes_signal() {
    …
    c.signal();
    …
}
```

Condition queue

Fairness in Monitors

- Thread 2 is selected (non-deterministically), acquires the lock and proceeds to execute a method

Lock queue (a.k.a. entry queue)

Thread 1   Thread 3

```
// Monitor state
Lock l = new Lock();
Condition c = l.newCondition();

public void m_executes_await() {
    …
    c.await();    Thread 2
    …
}

public void m_executes_signal() {
    …
    c.signal();
    …
}
```

Condition queue

Fairness in Monitors
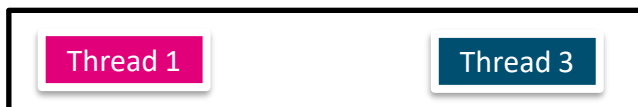
- Thread 2 is selected (non-deterministically), acquires the lock and proceeds to execute a method

Lock queue (a.k.a. entry queue)

Thread 1   Thread 3

```
// Monitor state
Lock l = new Lock();
Condition c = l.newCondition();

public void m_executes_await() {
    …
    c.await();      Thread 2
    …
}

public void m_executes_signal() {
    …
    c.signal();
    …
}
```
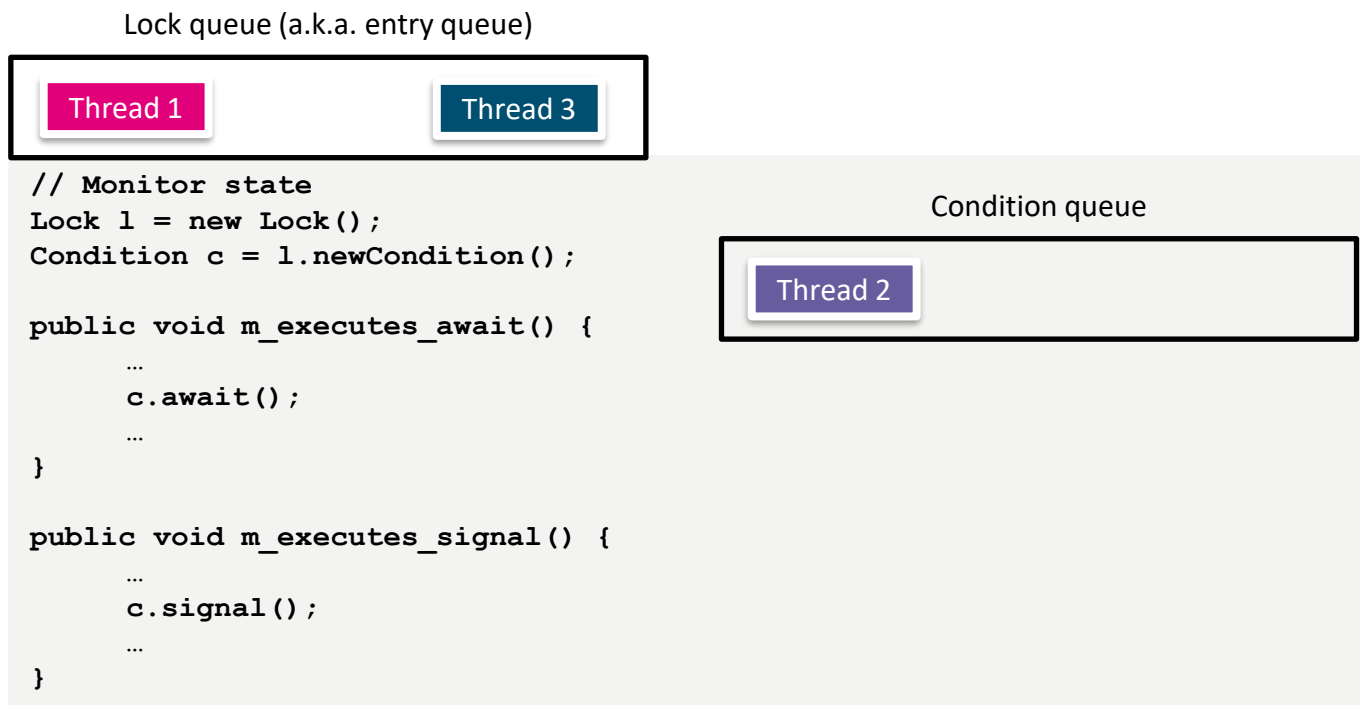
Condition queue

ReentrantLock has a **fair** flag that, when set to **true**, ensures that always the thread waiting longest in the entry queue is selected

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2025

- Thread 2 executes await and goes to the condition queue

Lock queue (a.k.a. entry queue)

Thread 1	Thread 3

```
// Monitor state
Lock l = new Lock();
Condition c = l.newCondition();

public void m_executes_await() {
    …
    c.await();
    …
}

public void m_executes_signal() {
    …
    c.signal();
    …
}
```

Condition queue

Thread 2

- Thread 1 is selected and executes await as well

Lock queue (a.k.a entry queue)

Thread 3

```
// Monitor state
Lock l = new Lock();
Condition c = l.newCondition();

public void m_executes_await() {
    …
    c.await();
    …
}

public void m_executes_signal() {
    …
    c.signal();
    …
}
```

Condition queue

Thread 2   Thread 1

- Thread 1 is selected and executes await as well

Lock queue (a.k.a entry queue)

Reminder: At this point a spurious wake-up can occur and Thread 1 or 2 could go back to the entry queue unexpectedly
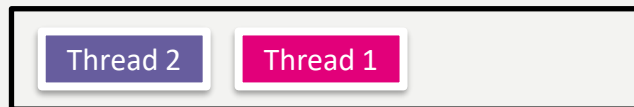
| | Thread 3 |

```
// Monitor state
Lock l = new Lock();
Condition c = l.newCondition();

public void m_executes_await() {
    …
    c.await();
    …
}

public void m_executes_signal() {
    …
    c.signal();
    …
}
```

Condition queue

| Thread 2 | Thread 1 |

- Thread 3 is selected and executes signal

Lock queue (a.k.a entry queue)

```
// Monitor state
Lock l = new Lock();
Condition c = l.newCondition();

public void m_executes_await() {
    …
    c.await();
    …
}

public void m_executes_signal() {
    …
    c.signal();     Thread 3
    …
}
```

Condition queue

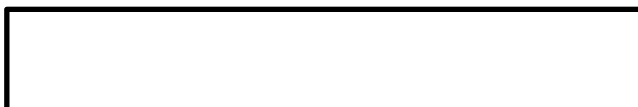Thread 2   Thread 1

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2025

- Thread 3 releases the lock and finishes execution

Lock queue (a.k.a entry queue)

```
// Monitor state
Lock l = new Lock();
Condition c = l.newCondition();

public void m_executes_await() {
    …
    c.await();
    …
}

public void m_executes_signal() {
    …
    c.signal();
    …
}
```

Condition queue

| Thread 2 | Thread 1 |

Fairness in Monitors

- Thread 1 is selected (non-deterministically) to go back to the entry queue (as a consequence of executing signal by Thread 3)

Lock queue (a.k.a. entry queue)

Thread 1

```
// Monitor state
Lock l = new Lock();
Condition c = l.newCondition();

public void m_executes_await() {
    …
    c.await();
    …
}

public void m_executes_signal() {
    …
    c.signal();
    …
}
```

Condition queue

Thread 2

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2025

Fairness in Monitors

- Thread 1 is selected (non-deterministically) to go back to the entry queue (as a consequence of executing signal by Thread 2)

Lock queue (a.k.a. entry queue)

Thread 1

Note that in the condition queue nothing ensures fairness either

```
// Monitor state
Lock l = new Lock();
Condition c = l.newCondition();

public void m_executes_await() {
    …
    c.await();
    …
}

public void m_executes_signal() {
    …
    c.signal();
    …
}
```

Condition queue

Thread 2

- If instead we use **signalAll()**, then both threads go to the entry queue

Lock queue (a.k.a. entry queue)

| Thread 1 | Thread 2 |

```
// Monitor state
Lock l = new Lock();
Condition c = l.newCondition();

public void m_executes_await() {
    …
    c.await();
    …
}

public void m_executes_signal() {
    …
    c.signalAll();
    …
}
```

Condition queue

- Be aware that different languages have different signalling semantics for monitors

  - <u>Mesa semantics</u>: Threads going to the entry queue to compete for entering the monitor again
    - Java semantics is Mesa.

  - <u>Hoare semantics</u>: Threads waiting on a condition variable have priority over threads waiting on the entry queue
    - In our example, any thread in the entry queue could be selected; independently on whether it came from the condition queue

- *Absence of starvation*: if a thread is ready to enter the critical section, it must eventually do so

- In our writers and readers example, writes may *starve* if readers keep coming

Readers can come as long as there are no writers, but writers need to wait until there are 0 readers

```
public void readLock() {
    lock.lock();
    try {
        while(writer)
         condition.await();
        readers++;
    }
    catch (InterruptedException e) {…}
    finally {lock.unlock();}
}
```

```
lic void writeLock() {
    lock.lock();
    try {
        while(readers > 0 || writer)
          condition.await();
        writer=true;
    }
    catch (InterruptedException e) {…}
    finally {lock.unlock();}
}
```

- Writers may set the writer flag to true to indicate that they are waiting to enter
- See **FairReadWriteMonitor.java**

```
public void readLock() {
    lock.lock();
    try {
        while(writer)
         condition.await();
        readers++;
    }
    catch (InterruptedException e) {…}
    finally {lock.unlock();}
}
```

```
public void writeLock() {
    lock.lock();
    try {
        while(writer)
         condition.await();
        writer=true;
        while(readers > 0)
         condition.await();

    }
    catch (InterruptedException e) {…}
    finally {lock.unlock();}
}
```

- Writers may set the writer flag to true to indicate that they are waiting to enter
- See `FairReadWriteMonitor.java`

Does this solution ensure that if a writer is ready to write it will eventually do it?

```java
public void readLock() {
    lock.lock();
    try {
        while(writer)
         condition.await();
        readers++;
    }
    catch (InterruptedException e) {…}
    finally {lock.unlock();}
}
```

```java
public void writeLock() {
    lock.lock();
    try {
        while(writer)
         condition.await();
        writer=true;
        while(readers > 0)
         condition.await();

    }
    catch (InterruptedException e) {…}
    finally {lock.unlock();}
}
```

- *Busy-wait* is an *alternative to blocking* a thread to wait until some condition holds or to enter the critical section

- The main difference with **lock()** or **await()** is that the thread does not transition to the "blocked" state

- Generally, busy-wait is a bad idea,
  - Threads may consume computing resources to check a condition that has not been updated
  - In this course, we will never ask you to use busy-wait
  - *Exercise solutions using busy-wait will not be approved*

- Very rarely busy-wait may be preferred over blocking the thread

```
…
// state variables
int i = 0;
Lock l = new ReentrantLock();
…


// method example
public void method(…) {
    l.lock()
    try{
        // busy-wait
        while(i>0) {
            // do nothing
        }
    }
    catch (InterruptedException e) {…}
    finally {l.unlock();}
}
```

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2025

- In Java, all objects have an intrinsic lock associated to it with a condition variable
  - I find it more correct to call them *intrinsic monitors* since they contain a condition variable. In fact, in the Java Language Specification they are called monitors.
- Intrinsic locks are accessed via the **`synchronized`** keyword
- These two code snippets are equivalent (for practical purposes)

```
Lock l = new Lock();

l.lock()
try {
    // critical section code
} finally {
    l.unlock()
}
```

```
Object o = new Object();

synchronized (o) {
    // critical section code
}
```

- **synchronized** can also be used on methods
  - The intrinsic lock associated to an instance of the object is used

```
class C {
    public synchronized T method() {
        …
    }
}
```

⬅➡

```
class C {
    public T method() {
        synchronized (this) {
            …
        }
    }
}
```

- **`synchronized`** can also be used on methods
  - The intrinsic lock associated to an instance of the object is used

```
class C {
    public synchronized T method() {
        …
    }
}
```

⟷

```
class C {
    public T method() {
        synchronized (this) {
            …
        }
    }
}
```
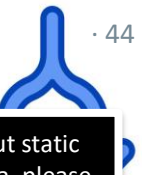
These two threads are using different locks, as `method()` uses the instance lock

```
new Thread(() -> {
    C c1 = new C();
    c1.method()
}).start();

new Thread(() -> {
    C c1 = new C();
    c1.method()
}).start();
```

Note: If you don´t know about static variables, methods, etc. in java, please let us know and we will point you to relevant literature.
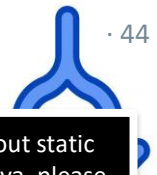
- **`synchronized`** can also be used on _static_ methods
  - The intrinsic lock associated the class runtime object is used

```
class C {
    public synchronized static T method() {
        …
    }
}
```



```
class C {
    public static T method() {
        synchronized (C.class) {
            …
        }
    }
}
```

> Note: If you don´t know about static variables, methods, etc. in java, please let us know and we will point you to relevant literature.

- **`synchronized`** can also be used on _static_ methods
  - The intrinsic lock associated the class runtime object is used

```
class C {
    public synchronized static T method() {
        …
    }
}
```

```
class C {
    public static T method() {
        synchronized (C.class) {
            …
        }
    }
}
```

These two objects use the same lock because they use the class lock, which is the same for all object instances

```
new Thread(() -> {
    C c1 = new C();
    c1.method()
}).start();

new Thread(() -> {
    C c1 = new C();
    c1.method()
}).start();
```

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2025

- The condition variable in intrinsic locks is accessed via the methods **`wait()`,`notify()`, `notifyAll()`**
- These are equivalent to **`await()`,`signal()`,`signalAll()`** in **`ReentrantLock`**.
- When using **`synchronized`** in methods use **`this.wait()`, `this.notify()`**, etc…
- These two code snippets are equivalent (for practical purposes)

```
Lock l = new Lock();
Condition c = l.addCondition()

l.lock()
try {
    // critical section code
    while(property)
     c.await();

    …

    c.signalAll();

} finally {
    l.unlock()
}
```

```
Object o = new Object();

synchronized (o) {
    // critical section code
    while(property)
     o.wait();

    …

    o.notifyAll();
}
```

Hardware and Programming Language Concurrency issues

- *In the absence of synchronization operations the CPU is allowed to keep data in the CPU's registers/cache*
  - Thus, *it might not be visible for threads running on a different CPU*
  - These are hardware optimizations to increase performance

- *In the absence of synchronization operations the CPU is allowed to keep data in the CPU's registers/cache*
  - Thus, *it might not be visible for threads running on a different CPU*
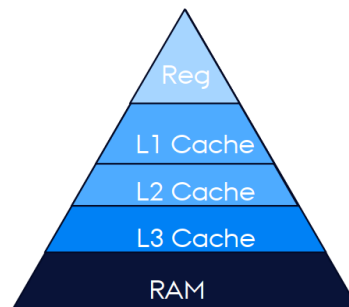 - These are hardware optimizations to increase performance

Processor @3.3Ghz app 0.1 ns pr instruction
L1 Data Cache Latency = 4 cycles
L2 Cache Latency = 12 cycles
L3 Cache Latency = 36 cycles (3.4 GHz i7-4770)
RAM Latency = 36 cycles + 57 ns (3.4 GHz i7-4770)

Wasted inst

Reg	0
L1 Cache	12
L2 Cache	36
L3 Cache	108
RAM	678

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2025

- Complete program in `NoVisibility1.java`

What are the possible outputs of this program?

```
boolean running = true;
Thread t1 = new Thread(() -> {
   while (running) {
      /* do nothing */
   }
   System.out.println("t1 finishing execution");
})
t1.start();
try{Thread.sleep(500);}catch(…){…}
running = false;
System.out.println("Main finishing execution");
```

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2025

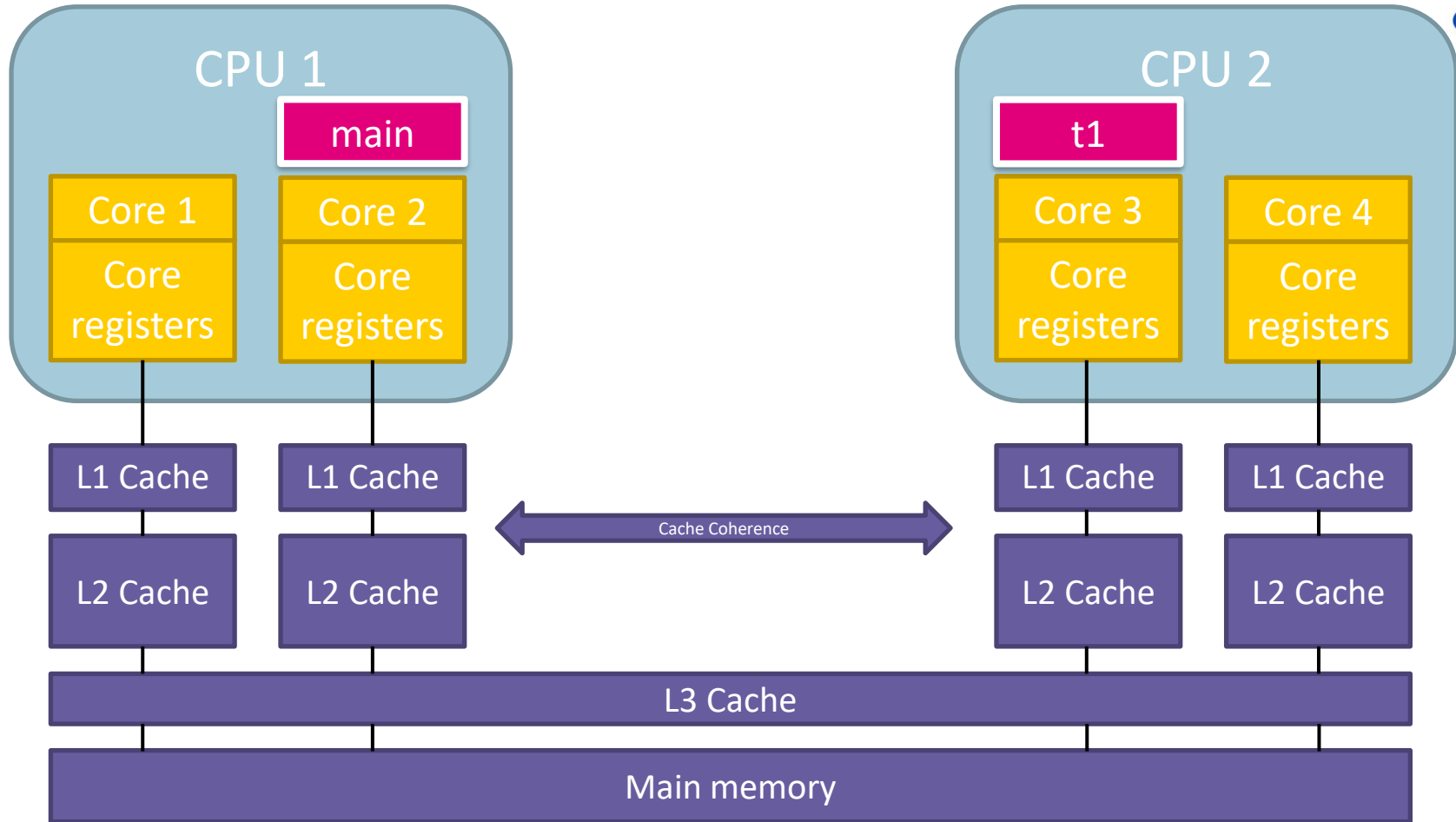- Complete program in `NoVisibility1.java`

What are the possible outputs of this program?

```java
boolean running = true;
Thread t1 = new Thread(() -> {
    while (running) {
        /* do nothing */
    }
    System.out.println("t1 finishing execution");
})
t1.start();
try{Thread.sleep(500);}catch(…){…}
running = false;
System.out.println("Main finishing execution");
```

DISCLAIMER: This is not a "Java issue". It can happen in any programming language with support for concurrency

CPU 1

main

Threads running in different CPUs cannot access each other's Core registers

CPU 2

t1

Core 1 | Core registers

Core 2 | Core registers

In our example, it is likely that the value of `running` is updated only in one CPU and not propagated.
Warning: statement not verified

Core 3 | Core registers

Core 4 | Core registers

L1 Cache | L1 Cache | Cache Coherence | L1 Cache | L1 Cache

L2 Cache | L2 Cache | L2 Cache | L2 Cache

L3 Cache

Main memory

Drawing and explanation inspired from: https://www.youtube.com/watch?v=nNXkzDS6dOg

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2025

- Why do visibility problems occur?
    - Lack of synchronization operations between threads
    - The main thread and thread t1 do not use any synchronization operations
    - Consequently, the CPU is allowed to keep the value of running in the register of the CPU or cache and not flush it to main memory

```java
boolean running = true;
Thread t1 = new Thread(() -> {
    while (running) {
        /* do nothing */
    }
    System.out.println("t1 finishing execution");
})
t1.start();
try{Thread.sleep(500);}catch(…){…}
running = false;
System.out.println("Main finishing execution");
```

- Adding synchronization between threads ensures visibility
  - We must establish an execution order between the read and the write on **running** performed by the two threads

    > The Java memory model precisely defines how synchronization must be used. More about this next week.

  - In this program, the CPU is <u>not</u> allowed to keep the value of running in the register of the CPU or cache and <u>must</u> flush it to main memory

> Concretely, when **unlock()** is executed, CPU registers and low-level cache are flushed (entirely) to memory levels shared by all CPUs

```
boolean running = true;
Object o = new Object();
Thread t1 = new Thread(() -> {
    while (running) {
        synchronized(o){/* do nothing */}
    }
    System.out.println("t1 finishing execution");
})
t1.start();
try{Thread.sleep(500);}catch(…){…}
synchronized(o){running = false;}
System.out.println("Main finishing execution");
```

> See the complete program:
> **NoVisibility1Synchronized.java**

- In the absence of _data dependences_ or _synchronization operations_,
  - the processor (CPU) or
  - Just-In-Time compiler (JIT) in the Java Virtual Machine (JVM)

  can reorder Java bytecode operations

- Thus, write accesses may be perceived as reordered as compared to the order in the definition of the thread

- Reordering is intended to increase performance
  - For instance, parallelizing tasks
  - Most programming languages incorporate these optimizations

- Complete program in **PossibleReordering.java**

```java
// shared variables
x=0;y=0;
a=0;b=0;

// Threads definition
Thread one = new Thread(() -> {
    a=1;
    x=b;
});
Thread other = new Thread(() -> {
    b=1;
    y=a;
});
one.start();other.start();
one.join();other.join();
System.out.println("("+x+","+y+")");
```

- Complete program in **PossibleReordering.java**

This program can output (0,0)

```
// shared variables
x=0;y=0;
a=0;b=0;

// Threads definition
Thread one = new Thread(() -> {
    a=1;
    x=b;
});
Thread other = new Thread(() -> {
    b=1;
    y=a;
});
one.start();other.start();
one.join();other.join();
System.out.println("("+x+","+y+")");
```

- Complete program in **PossibleReordering.java**

```
// shared variables
x=0;y=0;
a=0;b=0;

// Threads definition
Thread one = new Thread(() -> {
    a=1; // x=b
    x=b; // a=1
});
Thread other = new Thread(() -> {
    b=1; // y=a
    y=a; // b=1
});
one.start();other.start();
one.join();other.join();
System.out.println("("+x+","+y+")");
```

No data dependencies or synchronization operations between these instructions

The JIT compiler or CPU are allowed to perform this reordering

See `PossibleReorderingSynchronized.java`

Using synchronization can avoid the (0,0) output, but not reordering

- This program can output neither (0,0) nor (1,1)

- The reason is that mutual exclusion is enforced in the execution of each thread
  - Only the outputs (0,1) and (1,0) are possible in this case

- Note that reordering does not affect the intra-thread behaviour (i.e., the behaviour of the thread when executed in isolation)
  - This is one of the constraints for compiler or CPU optimizations when reordering operations

```java
// shared variables
x=0;y=0;
a=0;b=0;
Object o = new Object();

// Threads definition
Thread one = new Thread(() -> {
    synchronized (o) {
      a=1;
      x=b;
    }
});
Thread other = new Thread(() -> {
    synchronized (o) {
      b=1;
      y=a;
    }
});
one.start();other.start();
one.join();other.join();
System.out.println("("+x+","+y+")");
```

`volatile`

- Java provides a weak form of synchronization via the variable/field modifier `volatile`

- Volatile variables are not stored in CPU registers or low levels of cache hidden from other CPUs
  - Writes to volatile variables flush registers and low level cache to shared memory levels

- Volatile variables cannot be reordered

- *<u>Volatile variables cannot be used to ensure mutual exclusion!</u>*
  - Note that neither reads or writes are blocking operations

volatile

See **PossibleReorderingVolatile.java**

- In this program the output (0,0) is not possible

```
// shared variables
x=0;
y=0;
volatile a=0;
volatile b=0;

// Threads definition
Thread one = new Thread(() -> {
    a=1;
    x=b;
});
Thread other = new Thread(() -> {
    b=1;
    y=a;
});
one.start();other.start();
one.join();other.join();
System.out.println("("+x+","+y+")");
```

- Volatile variables can

  - Ensure visibility

  - Prevent reordering

- Locking can

  - Ensure visibility

  - Prevent reordering

  - Ensure mutual exclusion

- The visibility and reordering issues exist in <u>all programming languages</u>

- Different programming languages address them in different ways

  - Different ways to enforce volatile variable behaviour

  - Different ways to synchronize threads

  - Different compilation strategies

- <u>Memory models</u> formally describe the valid executions behaviour of concurrent programs (including visibility and reordering)

- Next week we will see the Java memory model

- Learning how to use the memory model of a programming language will give you the ability of precisely predicting the behaviour of your concurrent programs

- All programming languages specialized in concurrency have a memory model