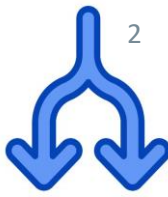# Practical Concurrent and Parallel Programming II

Shared Memory I

Raúl Pardo

Assessing the usefulness of exercises guide

We would like to have your feedback on the exercises guide

- https://www.menti.com/zskzs5ibm2

- Please feel free to share comments/suggestions on how to improve the exercises guide in the forum
  - For instance, problems you encounter when following the guide
  - Please feel free to share your comments anonymously

- Feel free to also ask for help on running exercises in the forum
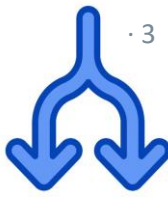
- As of Sep 6th at 8:00

  - 21 people do not have a group
  - _Please contact us if you are having trouble finding a group_

  - 40 people have not booked and oral feedback slot
  - This also means that 19 people have a group and did not book a slot
  - _Please contact us if you cannot attend existing slots_

- Schedule for oral feedback available in the course GitHub repo
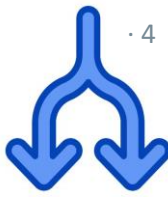
- TAs will be in room(s) 2A12-14

- Suggestion for finding a seat in exercise sessions:
  - First, try to find a spot in 2A12-14
  - If no spots are available, use any of the other two available rooms (2A54 & 4A58) and if you have questions go to 2A12-14

Believe it or not, last week we spent almost two hours talking about a program that increments a long integer...

- Introduction to concurrency
- Java threads
- The Mutual Exclusion Problem
- Java Locks

- Readers and Writers Problem
- Monitors
- Java Intrinsic Locks (**`synchronized`**)
- Hardware and Programming Language Concurrency Issues

  - Visibility

  - Reordering
- Volatile variables (**`volatile`**)

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

Writer *n*

Writer 2

Writer 1

Reader 1
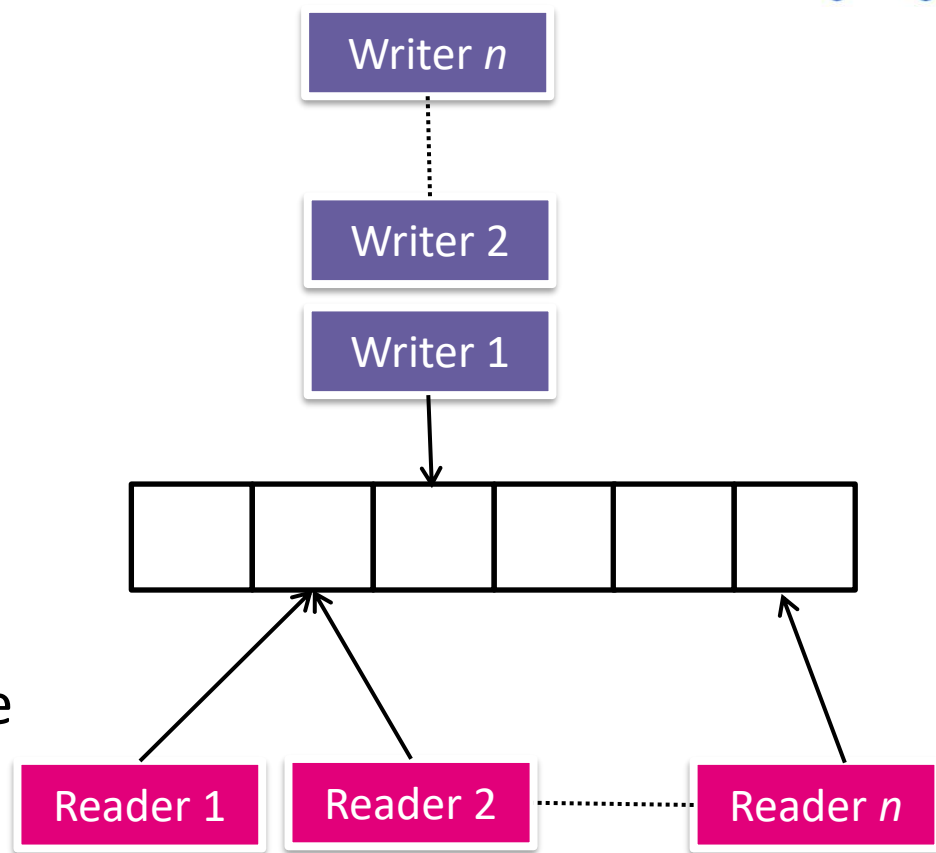
Reader 2

Reader *n*

Readers-Writers Problem

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

- 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

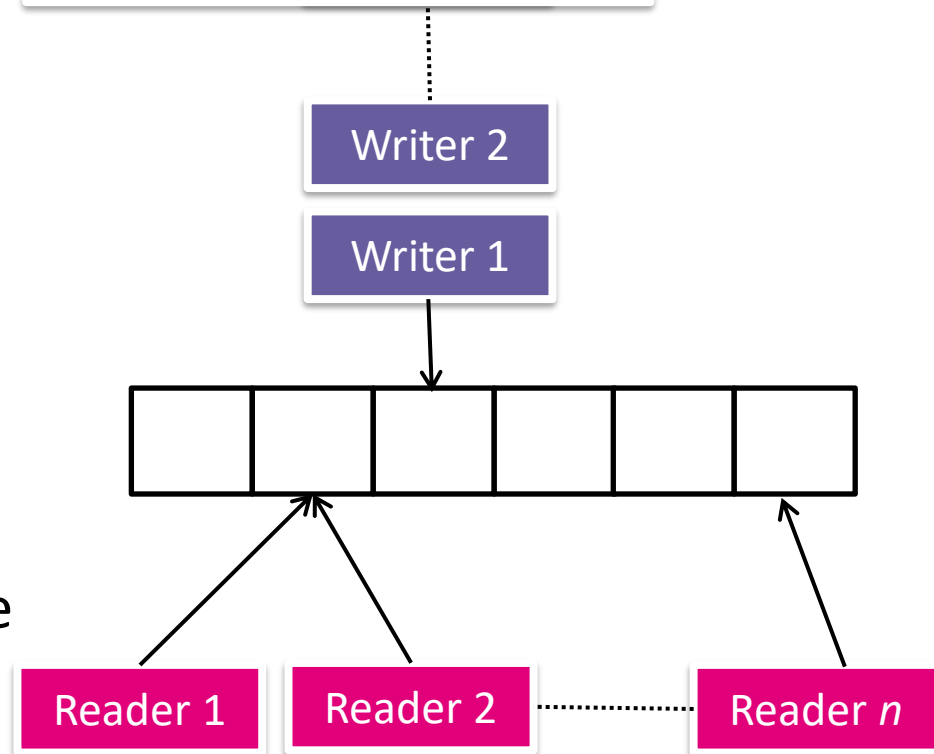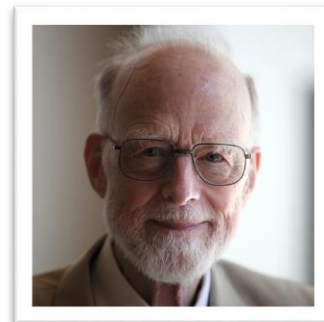- At most one thread can write at the same time

Writer 2

Writer 1

Reader 1   Reader 2   ⋯⋯⋯   Reader *n*

- A *monitor* is a structured way of encapsulating data, methods and synchronization in a single modular package
  - First introduced by Tony Hoare (see optional readings)

- *A monitor consists of:*
  - <u>Internal state (data)</u>
  - <u>Methods (procedures)</u>
    – All methods in a monitor are mutually exclusive (ensured via locks)
    – Methods can only access internal state
  - <u>Condition variables (or simply conditions)</u>
    – Queues where the monitor can put threads to wait

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

Monitors

- A *monitor* is a structured way of encapsulating data, methods and synchronization in a single modular package
  - First introduced by Tony Hoare (see optional readings)

- *A monitor consists of:*
  - <u>Internal state (data)</u>
  - <u>Methods (procedures)</u>
    - All methods in a monitor are mutually exclusive (ensured via locks)
    - Methods can only access internal state
  - <u>Condition variables (or simply conditions)</u>
    - Queues where the monitor can put threads to wait

> Can race conditions appear in a monitor method?

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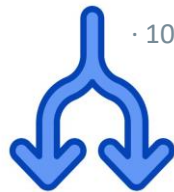
Conditions

- 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 threads blocked on the queue, if any
  - `signalAll()` – wakes up all threads blocked on the queue, if any

- When threads wake up the 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) {
            condition.await()
        }
    }
    catch (InterruptedException e) {…}
    finally {l.unlock();}
}
```

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2021

- 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

```
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 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

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

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

```
                                   e) {…}
}

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

We decrease the number of readers unconditionally

If there are no more readers, we signal condition

```
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 – F2021

- 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 (InterruptedExcep
    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 check whether there are readers or a writer accessing the resource

If so, we put the thread to wait

If not, the writer takes the lock

- 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 release the writer lock unconditionally

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

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2021

- 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`?

```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 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`?

```java
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)

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

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

```java
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

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2021

- 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
}
```

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

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 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();}
}
```

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2021

- 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

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

- Why does this happen?

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

- Examples of fairness:
  - Threads should not be scheduled based on the tasks they perform or their computational costs (e.g., writers threads in our example)
  - If two threads compete to enter the monitor (i.e., execute a method), the thread waiting longest should have priority

- 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

- 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

Condition queue

```
// 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();
    …
}
```

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2021

- Thread 2 is selected (non-deterministically), acquires the lock and proceeds to execute a method
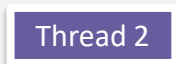
Lock queue (a.k.a. entry queue)

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

Fairness in Monitors

- Thread 2 is selected (non-determinist proceeds to execute a method

What thread would have been selected in this example if we had enabled the `fair` flag?

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();
    …
}
```
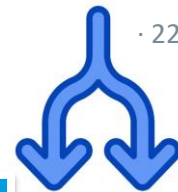
Condition queue

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

Fairness in Monitors

- 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

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2021
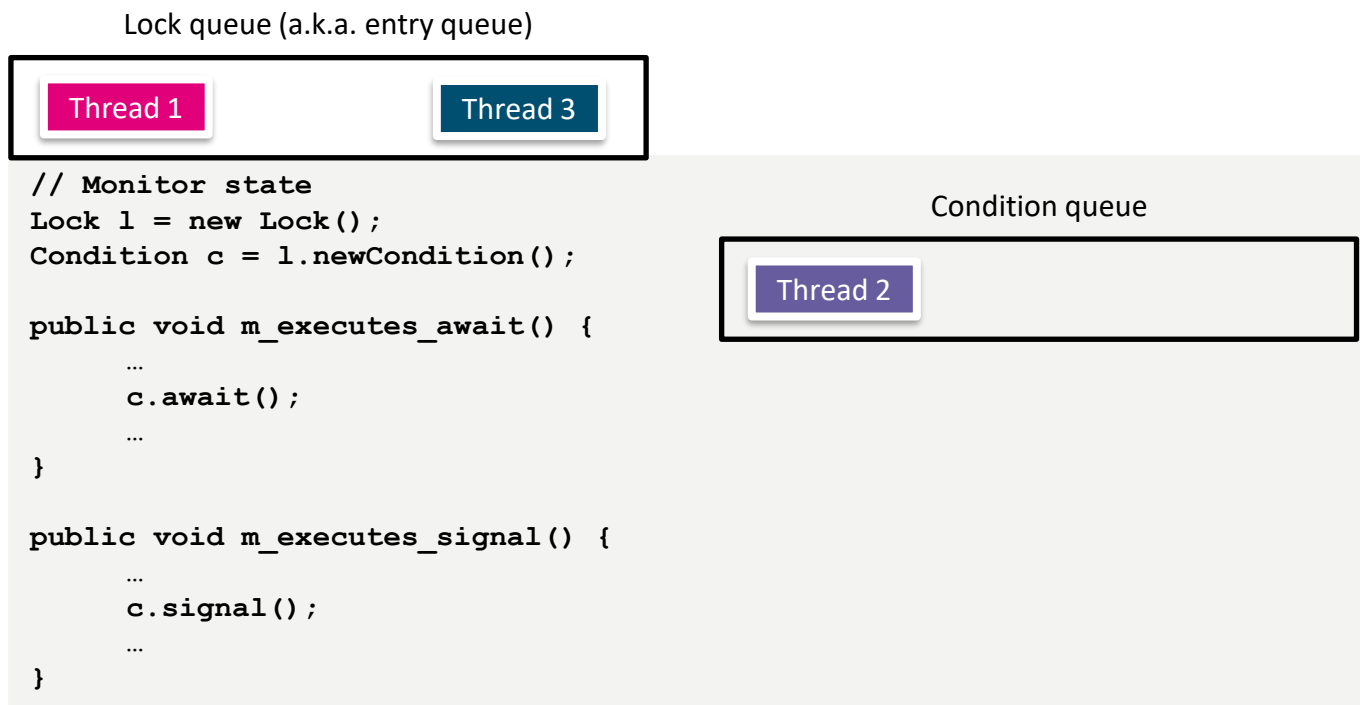
- Thread 1 is selected and executes await as well

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();
    …
}
```
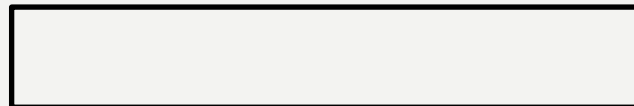
Condition queue

Thread 2   Thread 1

Fairness in Monitors

- Thread 1 is selected and executes await as well

Lock queue (a.k.a entry queue)

Thread 3

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

```
// 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 – F2021

- 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 |

- 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

- 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)

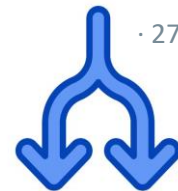Note that in the condition queue nothing ensures fairness either

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

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

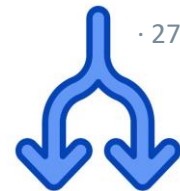Lock queue (a.k.a. entry queue)

| 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

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2021

Signalling semantics

- 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 preference 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();}
}
```

Starvation (revisited)

- _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

Can this starvation problem be fixed?

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

```
public void writeLock() {
    lock.lock();
    try {
        while(writer)
         condition.await();
        writer=true;
        while(readsAcquires != readsReleases)
         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 will eventually do it?

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

```
public void writeLock() {
    lock.lock();
    try {
        while(writer)
         condition.await();
        writer=true;
        while(readsAcquires != readsReleases)
         condition.await();

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

- There exist two notions of fairness
  - <u>Weak fairness</u>: A thread that is continuously active will eventually make progress
  - <u>Strong fairness</u>: A thread that is infinitely often active will eventually make progress

- There exist two notions of fairness

  - <u>Weak fairness</u>: A thread that is continuously active will eventually make progress

  - <u>Strong fairness</u>: A thread that is infinitely often active will eventually make progress

What type of fairness does **ReentrantLock** with the **fair** flag ensures?

- There exist two notions of fairness

    - <u>Weak fairness</u>: A thread that is continuously active will eventually make progress

    - <u>Strong fairness</u>: A thread that is infinitely often active will eventually make progress

Does **ReentrantLock** with the **fair** flag solve the writer starvation problem?

What type of fairness does **ReentrantLock** with the **fair** flag ensures?

```
…
// 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();}
}
```

Would it be ok to do this?

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2021

- *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 be rejected*

- Very rarely busy-wait may be preferred over blocking the thread
  - When the thread waits for a very short time it might be more efficient to use busy-wait
  - However, as we have discussed, reasoning about how it takes for a thread to do anything is pointless in concurrency

```
…
// 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();}
}
```

- 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.
- Locks are accessed via the **`synchronized`** keyword
- These two code snippets are equivalent

```
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) {
            …
        }
    }
}
```

Are these two threads using the same intrinsic lock?

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

new Thread(() -> {
    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 <u>*static*</u> 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) {
            …
        }
    }
}
```

Are these two threads using the same intrinsic lock?

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

new Thread(() -> {
    C c2 = new C();
    c2.method()
}).start();
```

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2021

- 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

```
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 (Java) Concurrency issues

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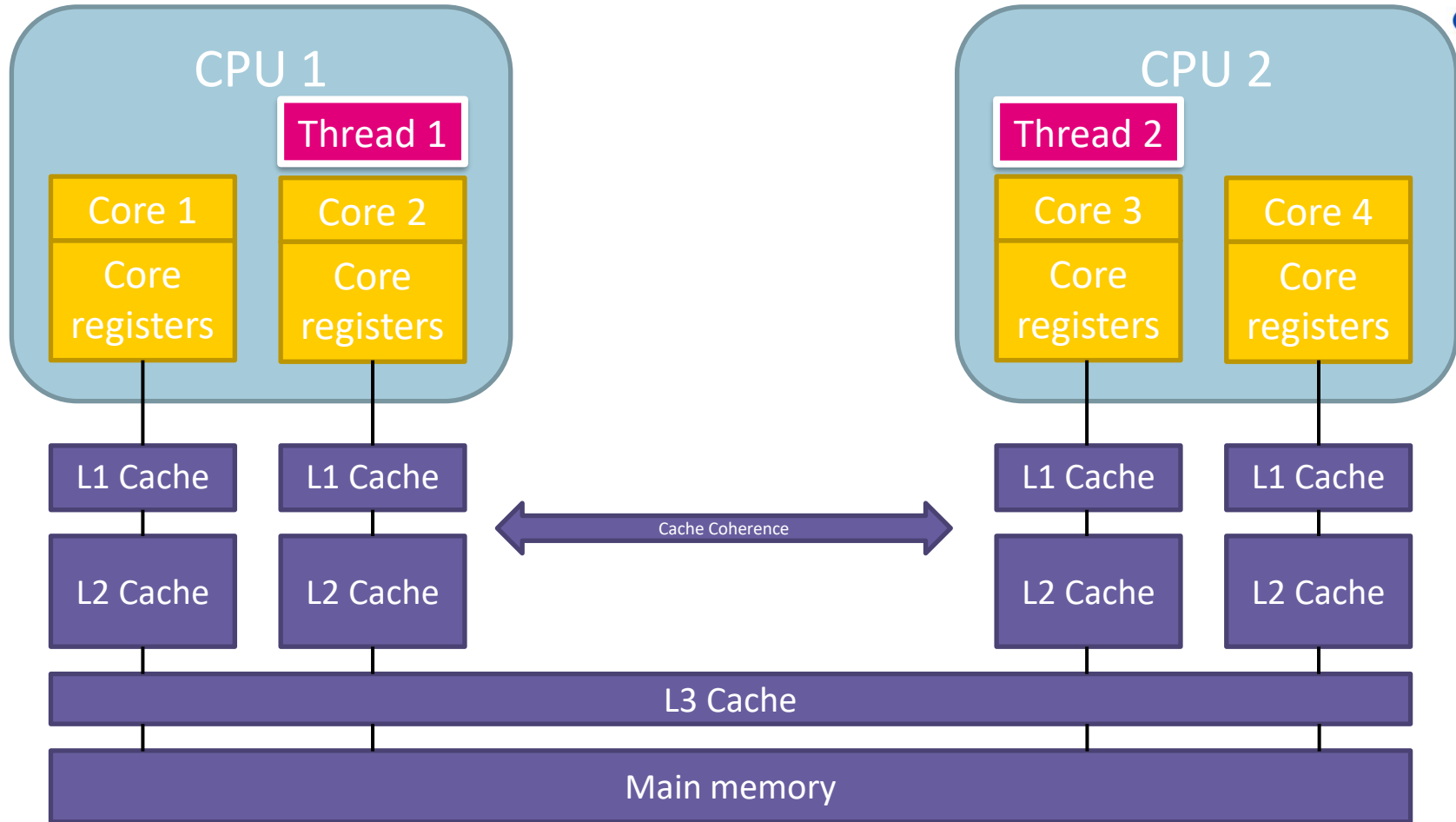 - These are hardware optimizations to increase performance

- Complete program in **NoVisibility1.java**

What are the possible outputs of this program?

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

Visibility | Memory Hierarchy (simplified)

CPU 1

Thread 1

Threads running in different CPUs cannot access each others Core registers

CPU 2

Thread 2

Core 1	Core 2
Core registers	Core registers

Core 3	Core 4
Core registers	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 – F2021

Visibility | Memory Hierarchy (simplified)

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

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2021
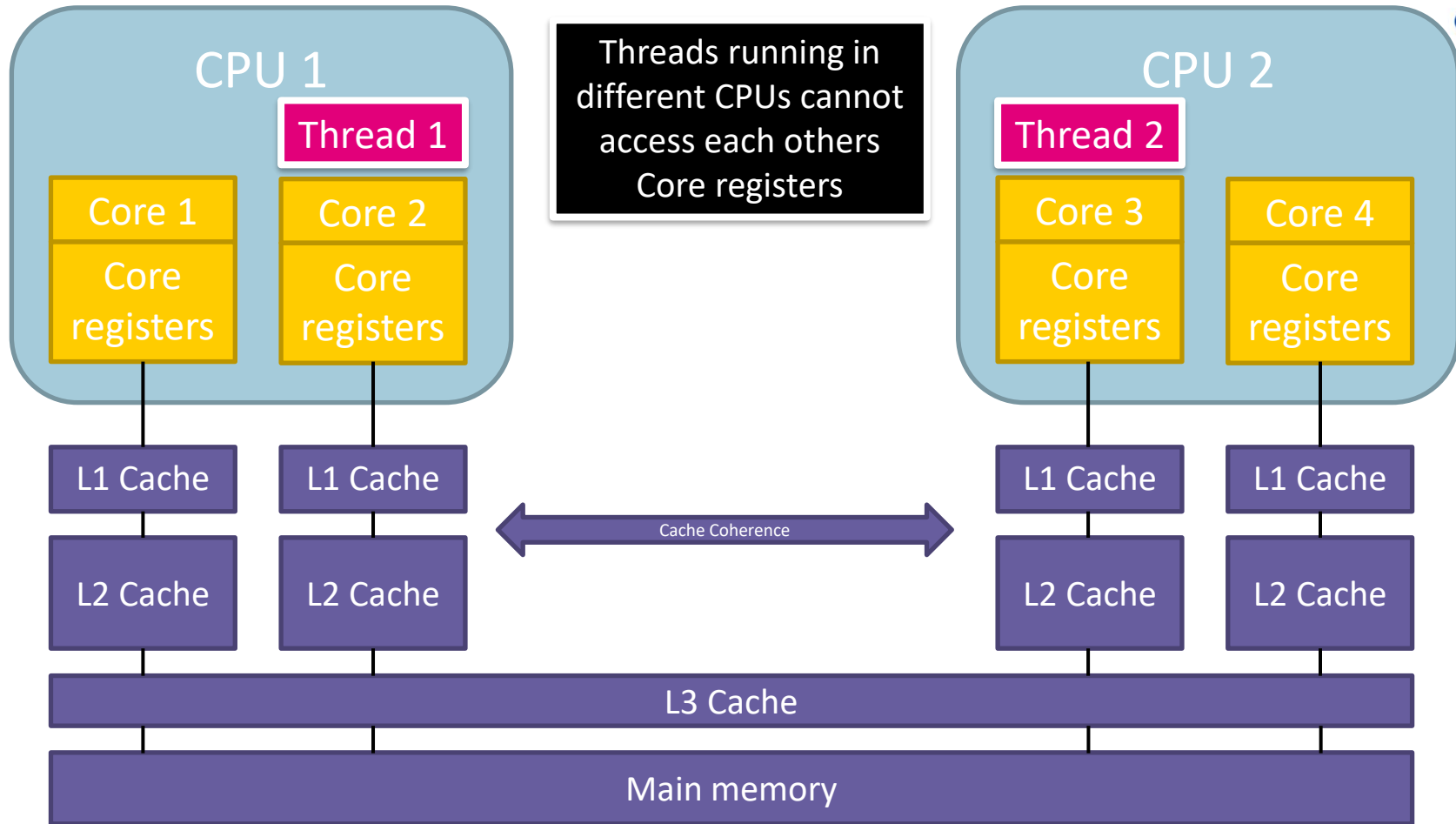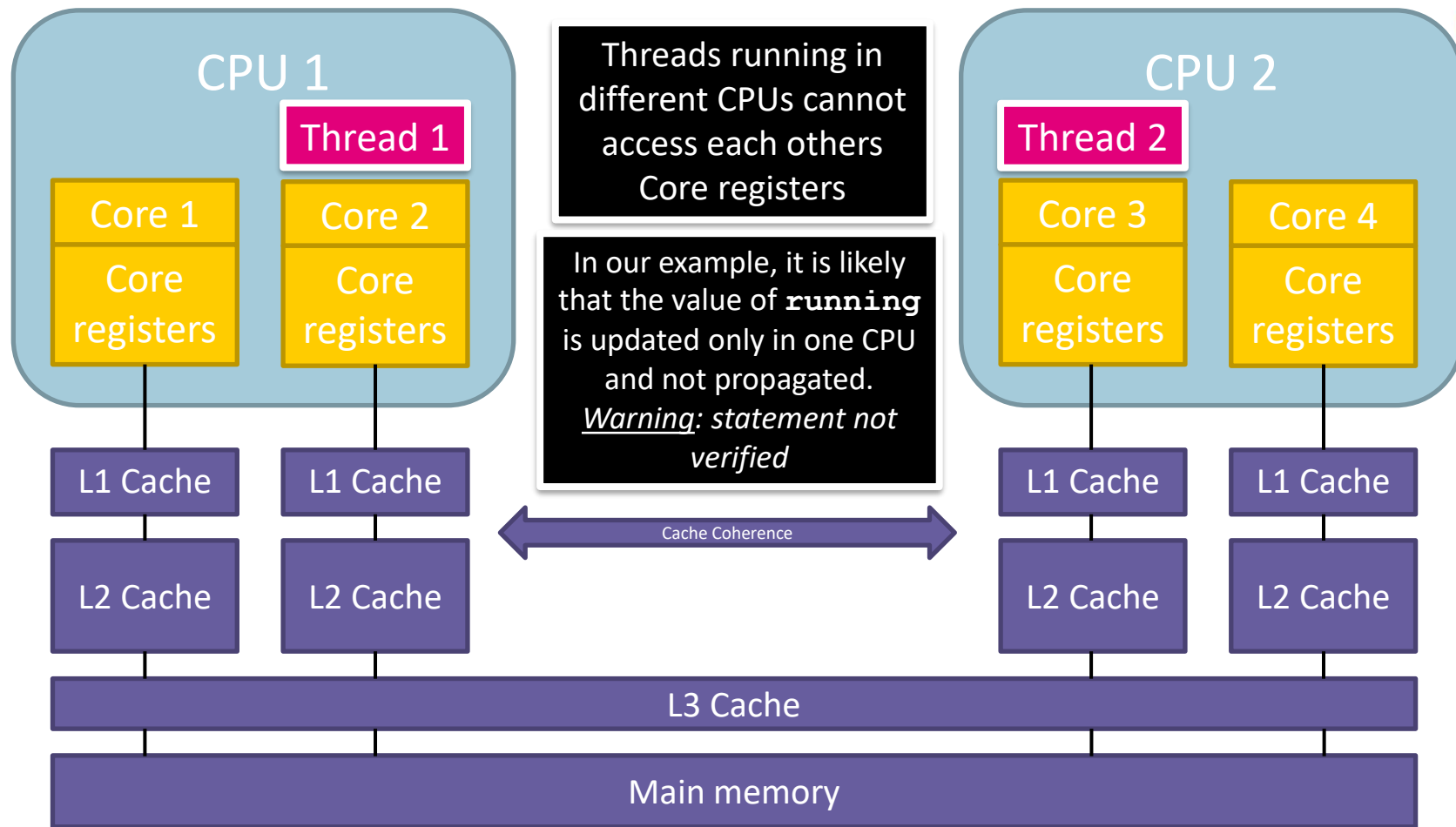
- Why do visibility problems occur?
  - <u>Simple</u>: lack of happens-before relation between operations
  - In the program below, it holds
  - $while(running) \not\twoheadrightarrow running := false$ and
    $$running := false \not\twoheadrightarrow while(running)$$
  - Consequently, the CPU is allowed to keep the value of running in the register of the CPU or cache and not flush it to main memory

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

Here we abuse notation and use **`while(running)`** to denote the operation of checking the condition **`running==true`**

~~ems occur?~~

~~ppens-before relation between operations~~

~~the program below, it holds~~

- $while(running) \nrightarrow running := false$ and $running := false \nrightarrow while(running)$

- 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

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

- Establishing a happen-before relation enforces visibility
  - We can use locks or synchronized (as they are equivalent)
  - In the program below, it holds
  - $while(running) \rightarrow running := false$ <u>or</u>
    $running := false \rightarrow while(running)$
  - Consequently, 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

Precisely, 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();
new Thread(() -> {
    while (running) {
        synchronized(o){/* do nothing */}
    }
    System.out.println("Hasta la vista, baby");
}).start();
try{Thread.sleep(500);}catch(…){…}
Synchronized (o) {running = false;}
System.out.println("Main thread finishing execution");
```

See the complete program:
`NoVisibility1Synchronized.java`

- Establishing a happen-before relation enforces visibility
  - We can use locks or synchronized (as they are equivalent)
  - In the program below, it holds
  - $while(running) \rightarrow running := false$ or
    $$running := false \rightarrow while(running)$$

  > Why did I write "or" here

  - Consequently, 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

> Precisely, 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();
new Thread(() -> {
    while (running) {
        synchronized(o){/* do nothing */}
    }
    System.out.println("Hasta la vista, baby");
}).start();
try{Thread.sleep(500);}catch(…){…}
Synchronized (o) {running = false;}
System.out.println("Main thread finishing execution");
```

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

What are the possible outputs of this program?

- Establishing a happen-before relation enforces visibility
  - We can use locks or synchronized (as they are equivalent)
  - In the program below, it holds
  - $while(running) \rightarrow running := false$ <u>or</u>
    $running := false \rightarrow while(running)$

    Why did I write "or" here

  - Consequently, 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

Precisely, 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();
new Thread(() -> {
    while (running) {
        synchronized(o){/* do nothing */}
    }
    System.out.println("Hasta la vista, baby");
}).start();
try{Thread.sleep(500);}catch(…){…}
Synchronized (o) {running = false;}
System.out.println("Main thread finishing execution");
```

See the complete program:
`NoVisibility1Synchronized.java`

- In the absence of data dependences or "synchronization", the Just-In-Time (JIT) compiler is allowed to reorder java bytecode operations
  - Thus, writing instructions may be perceived as reordered as compared to the order in the definition of the thread
  - Reordering is intended to increase performance (e.g., parallelizing tasks)

- Complete program in **PossibleReordering.java**

Can this program 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 between these instructions

The JIT compiler is allowed to perform this reordering

The lack of happens-before relation allows the reordering resulting in the output (0,0)

- Due to lack data dependences we have
  - $a := 1 \nrightarrow x := b$ and
    $$b := 1 \nrightarrow y := a$$

- Of course, due to lack of synchronization we cannot establish a happen-before relation with operations among threads either, thus
  - $a := 1 \nrightarrow b := 1$ and $a := 1 \nrightarrow y := a$
  - Analogous with $x := b, b := 1, y := a$

- Consequently, the JIT compiler is allowed to choose any ordering, e.g.,
  - $x := 0, y := 0, a := 1, b := 1$

```
// 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+")");
```

See **PossibleReorderingSynchronized.java**

Establishing a happen-before relation prevents (some) reordering. Now the output (0,0) is not possible.

```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+")");
```

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2021

Reordering

See **PossibleReorderingSynchronized.java**

Establishing a happen-before relation prevents (some) reordering. Now the output (0,0) is not possible.

WARNING: a difficult to digest slide is coming, but don't despair, a summary slide will follow.


```
// 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+")");
```

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2021

Establishing a happen-before relation prevents (some) reordering. Now the output (0,0) is not possible.

- Assume that thread one takes the lock first, then we have
- $a := 1 \rightarrow b := 1$ and
  $a := 1 \rightarrow y := a$ and
  $x := b \rightarrow b := 1$ and
  $$x := b \rightarrow y := a$$

The lack of data dependences still allows for reorderings within the critical section

- Assume, by contradiction, that the program outputs (0,0), then it must hold that
- $x := b \rightarrow y := a$ and
  $y := a \rightarrow a := 1$
  However, note that $y := a \rightarrow a := 1$ contradicts our assumption $a := 1 \rightarrow y := a$. So interleavings satisfying this ordering cannot occur.

- The same holds for the case when other takes the lock first. You can try to write it down at home.

```
// 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+")");
```

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2021

Now an informal yet rigorous (hopefully clearer) explanation to whether this program can output (0,0)

- In order to output (0,0), x=b and y=a must be executed before a=1 and b=1, respectively.

- Because {a=1,x=b} and {b=1,y=a} are protected by the same lock the only possible executions are:
  - {a=1,x=b} and then {b=1,y=a}, or
  - {b=1,y=a} and then {a=1,x=b}

- Thus,
  - In the first case a is set to 1 before y=a is executed
  - In the second case, b is set to 1 before x=b is executed

- Consequently, either x or y are set to 1 before printing and x and the output (0,0) cannot occur.

```
// 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+")");
```

Can this program output (1,1)?

Now an informal yet rigorous (hopefully clearer) explanation to whether this program can output (0,0)

- In order to output (0,0), x=b and y=a must be executed before a=1 and b=1, respectively.

- Because {a=1,x=b} and {b=1,y=a} are protected by the same lock the only possible executions are:
  - {a=1,x=b} and then {b=1,y=a}, or
  - {b=1,y=a} and then {a=1,x=b}

- Thus,
  - In the first case a is set to 1 before y=a is executed
  - In the second case, b is set to 1 before x=b is executed

- Consequently, either x or y are set to 1 before printing and x and the output (0,0) cannot occur.

```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 low level cache to shared memory levels

- Volatile variables cannot be reordered with each other

- The least confusing way of thinking about volatile variables is in terms of reads/writes and happens-before
  - A <u>write</u> to a volatile variable happens before any <u>*subsequent read*</u> to the volatile variable
  - Another option is to see reads/writes to volatile variables as atomic operations
    – Remember that one-liner program statements are not always atomic, this also applies for one-liner program statements on volatile variables

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

- In this program the output (0,0) is not possible (explanation based on happens-before)

```java
// 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

See `PossibleReorderingVolatile.java`

- In this program the output (0,0) is not possible (explanation based on happens-before)

> WARNING: a difficult to digest slide is coming, but don't despair, a summary slide will follow.


```
// 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+")");
```

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2021

volatile

In this program the output (0,0) is not possible (explanation based on happens-before)

- Because of volatile we have (the following premises)
  $a := 1 \rightarrow x := b$ and
  $b := 1 \rightarrow y := a$

- Assume, by contradiction, that the output of the program is (0,0), then any of the following must hold
  $x := b \rightarrow y := a \rightarrow a := 1 \rightarrow b := 1$ (1) or
  $x := b \rightarrow y := a \rightarrow b := 1 \rightarrow a := 1$ (2) or
  $y := a \rightarrow x := b \rightarrow a := 1 \rightarrow b := 1$ (3) or
  $y := a \rightarrow x := b \rightarrow b := 1 \rightarrow a := 1$ (4)

- From (1) and (2) we can derive $x := b \rightarrow a := 1$ which contradicts the premise $a := 1 \rightarrow x := b$

- From (3) and (4) we can derive $y := a \rightarrow b := 1$ which contradicts the premise $b := 1 \rightarrow y := a$

- Therefore, 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

WARNING: Only in the new Java Memory Model (JMM). Previous versions of the JMM allowed for reordering of volatile variables
http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html#volatile
https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.3.1.4

In this program the output (0 ... on happens-before)

- Because of volatile we have (the following premises)
  $a := 1 \rightarrow x := b$ and
  $b := 1 \rightarrow y := a$

- Assume, by contradiction, that the output of the program is (0,0), then any of the following must hold
  $x := b \rightarrow y := a \rightarrow a := 1 \rightarrow b := 1$ (1) or
  $x := b \rightarrow y := a \rightarrow b := 1 \rightarrow a := 1$ (2) or
  $y := a \rightarrow x := b \rightarrow a := 1 \rightarrow b := 1$ (3) or
  $y := a \rightarrow x := b \rightarrow b := 1 \rightarrow a := 1$ (4)

- From (1) and (2) we can derive $x := b \rightarrow a := 1$ which contradicts the premise $a := 1 \rightarrow x := b$

- From (3) and (4) we can derive $y := a \rightarrow b := 1$ which contradicts the premise $b := 1 \rightarrow y := a$

- Therefore, 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`

WARNING: Only in the new Java Memory Model (JMM). Previous versions of the JMM allowed for reordering of volatile variables

http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html#volatile
https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.3.1.4

In this program the output (0 [...]
on happens-before)

- Because of volatile we have (the following premises)
  $a := 1 \rightarrow x := b$ and
  $b := 1 \rightarrow y := a$

- Assume, by contradiction, that the output of the program is (0,0), then any of the following must hold
  $x := b \rightarrow y := a \rightarrow a := 1 \rightarrow b := 1$ (1) or
  $x := b \rightarrow y := a \rightarrow b := 1 \rightarrow a := 1$ (2) or
  $y := a \rightarrow x := b \rightarrow a := 1 \rightarrow b := 1$ (3) or
  $y := a \rightarrow x := b \rightarrow b := 1 \rightarrow a := 1$ (4)

- From (1) and (2) we can derive $x := b \rightarrow a := 1$ which contradicts the premise $a := 1 \rightarrow x := b$

- From (3) and (4) we can derive $y := a \rightarrow b := 1$ which contradicts the premise $b := 1 \rightarrow y := a$

- Therefore, 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();
```

Why isn´t it necessary to declare **x** and **y** volatile as well?

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2021

`volatile`

WARNING: Only in the new Java Memory Model (JMM). Previous versions of the JMM allowed for reordering of volatile variables
http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html#volatile
https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.3.1.4

In this program the output (0 on happens-before)

- Because of volatile we have (the following premises)
$a := 1 \rightarrow x := b$ and
$b := 1 \rightarrow y := a$

We also have other premises coming from volatile, depending on the scheduler:
$a := 1 \rightarrow y := a$ or $y := a \rightarrow a := 1$ and
$b := 1 \rightarrow x := b$ or $x := b \rightarrow b := 1$

These premises come from our earlier definition of happens-before for volatile variables:
A write to a volatile variable happens-before any subsequent read to the volatile variable

Note that
$(a := 1 \rightarrow y := a$ or $y := a \rightarrow a := 1) \quad \neq \quad (a := 1 \nrightarrow y := a$ and $y := a \nrightarrow a := 1)$

- Therefore, the output (0,0) is not possible

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

inition
ew Thread(() -> {

new Thread(() -> {

her.start();
```

Why isn´t it necessary to declare **x** and **y** volatile as well?

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2021

`volatile`

Now an informal yet rigorous (hopefully clearer) explanation to whether this program can output (0,0)

- In order to output (0,0), x=b and y=a must be executed before a=1 and b=1, respectively.

- Because a and b are declared volatile,
  - if is y=a is executed then b must equal 1, and
  - if x=b is executed, then a must be equal 1

- Therefore,
  - if x=b is executed before y=a then, a must equal 1, and
  - if y=a is executed before x=b, then b must equal 1

- Since we know that either x=b is executed before y=a or vice-versa, then we can conclude that (0,0) is not a possible output

```
// 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

Can this program output (1,1)?

```
// 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

Pay attention to the difference between this programs:

- *Intrinsic locks (`synchronized`) ensure mutual exclusion in the execution of the body of the threads* (and preserves visibility and ordering between unlock() and lock())

- *Volatile variables do not ensure mutual exclusion; the body of the threads can be interleaved as the scheduler wishes*
  - However, *for each of those interleavings, visibility and execution ordering is preserved*

```
// 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+")");
```

```
// 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+")");
```

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2021

- Writing on a volatile variable <u>flashes memory for all variables</u> in CPU registers or cache
    - Thus it ensures visibility to writes on non-volatile variables prior that of the volatile variable
    - Volatile writes have the same *effect on memory* than exiting a monitor (`unlock()`)
        - Again, <u>*effect on memory, not on blocking*</u>

- This (very intricate) property of volatile variables can be sued to ensure visibility of many variables without using locks
    - I suggest you use it with a lot of care; if at all…

See `VolatileExample.java`

```
…
VolatileReaderWriter vrw = new VolatileReaderWriter();

// Threads definition
new Thread(() -> {
    vrw.reader();
}).start();

new Thread(() -> {
    vrw.writer();
}).start();

…

class VolatileReadWrite {
  int x = 0;
  volatile boolean v = false;

  public void writer() {
      x = 42;
      v = true;
  }

  public void reader() {
      if (v == true)
        System.out.println(x); guaranteed to see 42
      else
        System.out.println(x);
  }
}
```

- Writing on a volatile variable flashes

Can this program output 0?

- Thus it ensures visibility to writes on non-volatile variables prior that of the volatile variable
- Volatile writes have the same *effect on memory* than exiting a monitor (`unlock()`)
  - Again, *effect on memory, not on blocking*

- This (very intricate) property of volatile variables can be sued to ensure visibility of many variables without using locks
  - I suggest you use it with a lot of care; if at all…

See `VolatileExample.java`

```
…
VolatileReaderWriter vrw = new VolatileReaderWriter();

// Threads definition
new Thread(() -> {
    vrw.reader();
}).start();

new Thread(() -> {
    vrw.writer();
}).start();

…

class VolatileReadWrite {
  int x = 0;
  volatile boolean v = false;

  public void writer() {
      x = 42;
      v = true;
  }

  public void reader() {
      if (v == true)
        System.out.println(x); guaranteed to see 42
      else
        System.out.println(x);
  }
}
```

- Volatile variables can
  - Ensure visibility
  - Prevent reordering

- Locking can
  - Ensure visibility
  - Prevent reordering
  - Ensure mutual exclusion

volatile vs **Locks** Summary

- Volatile variables can
  - Ensure visibility
  - Prevent reordering

- Locking can
  - Ensure visibility
  - Prevent reordering
  - Ensure mutual exclusion

Goetz et. al. provide useful advice in using volatile variables. I strongly recommend you follow their advice.

In general, reasoning about volatile variables is hard, and locking should be preferred as it has much clearer and consistent semantics. However, volatile variables may have a lower impact in performance.

That said, _all recommendations follow logically from the reasoning we have presented here._

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2021

- What we have seen here applies only to (modern) Java

- Keep in mind that:
  - Not all programming languages have the same semantics for volatile
  - Not all hardware platforms treat visibility in the same way
  - Not all runtime environments reorder instructions in the same way

- <u>The good news</u>: the reasoning we have followed can be applied independently of the semantics of the hardware or runtime environment
  - When writing concurrent code in a different language, first look up the semantics for these notions (if you use them)

- <u>Even better news</u>: locks and monitors have very similar (or the same) semantics in all languages (as they are an abstract concept). So, in case of doubt, use locking.

> Perhaps less practical advice …, but more general