



Practical Concurrent and Parallel Programming II

Shared Memory I

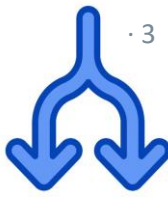
Raúl Pardo

Groups and Oral Feedback Sessions



- As of Sep 3rd at 10.00
 - 21 people do not have a group
 - Please contact us if you are having trouble finding a group
 - 65 people have not booked an oral feedback slot
 - This also means that 44 people have a group and did not book a slot
 - Please contact us if you cannot attend existing slots

Submission next week (a few remarks)



- Next week on Monday at 13.59 is the first submission deadline
- Your 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
- Each member of a group must submit a link to the same repository

- 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



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

Previously on PCPP...



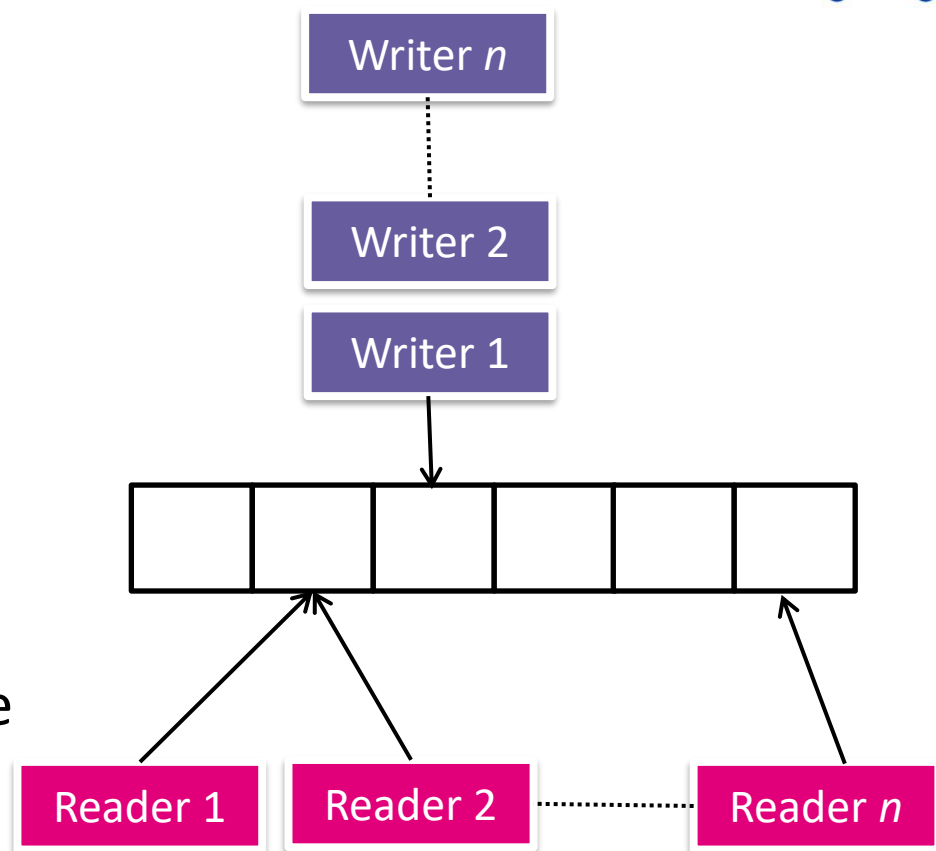
- Introduction to concurrency
- Java threads
- The Mutual Exclusion Problem
- Java Locks
- Happens-before (today)

Readers-Writers Problem

· 6



- 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



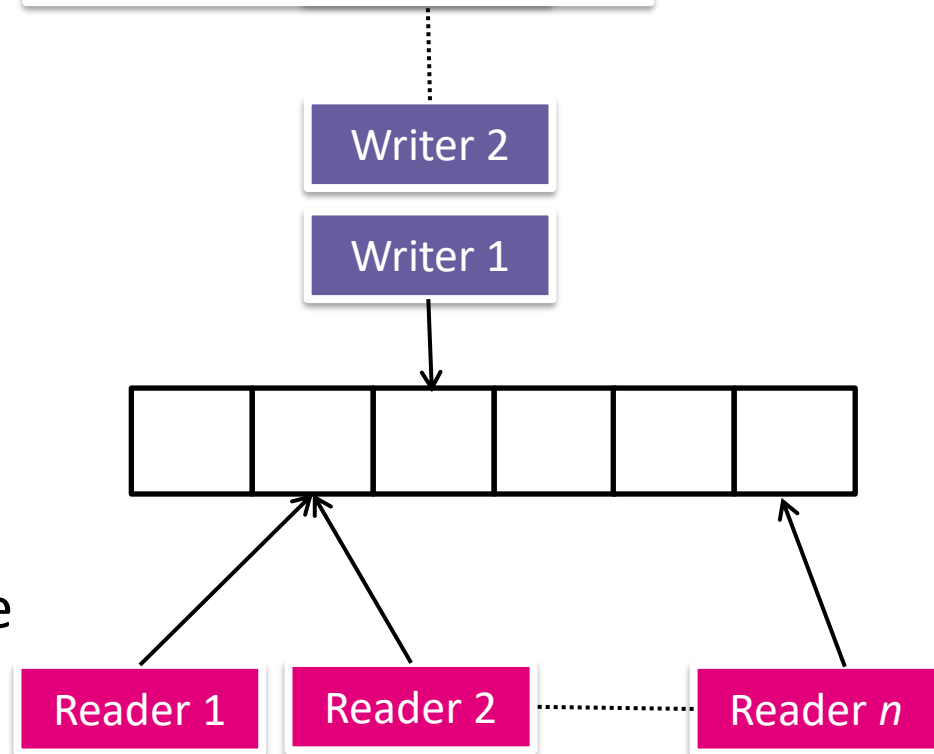
Readers-Writers Problem

· 6



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



- 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



- 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 (ens)
 - 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



Can race conditions appear in a monitor method?



- 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 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();}
}
```



- 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)

```
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 (InterruptedException e) {}  
    finally {}  
}
```

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

```
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();}
}
```



- 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();  
        // ...  
    }  
    finally {  
        lock.unlock();  
    }  
}
```

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

```
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();}
```




- 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();}
}
```

If so, we put the thread to wait

If not, the writer takes the lock

```
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();}
}
```

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

```
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();}
}
```

We release the writer lock unconditionally

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

```
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

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 {lock.unlock();}
}
```

We release the writer lock unconditionally

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

```
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

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();
    }
```

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

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 {lock.unlock();}
}
```

We release the writer lock unconditionally

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

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 {}  
}
```

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

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>

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

```
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
- Why does this happen?

- 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)



```
// 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

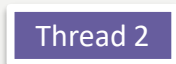
Lock queue (a.k.a. entry queue)



```
// 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)



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

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

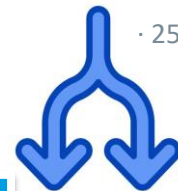
public void m_executes_signal() {
    ...
    c.signal();
    ...
}
```



Condition queue



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



- Thread 2 is selected (non-deterministic) 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();
    ...
}

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

Thread 2

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

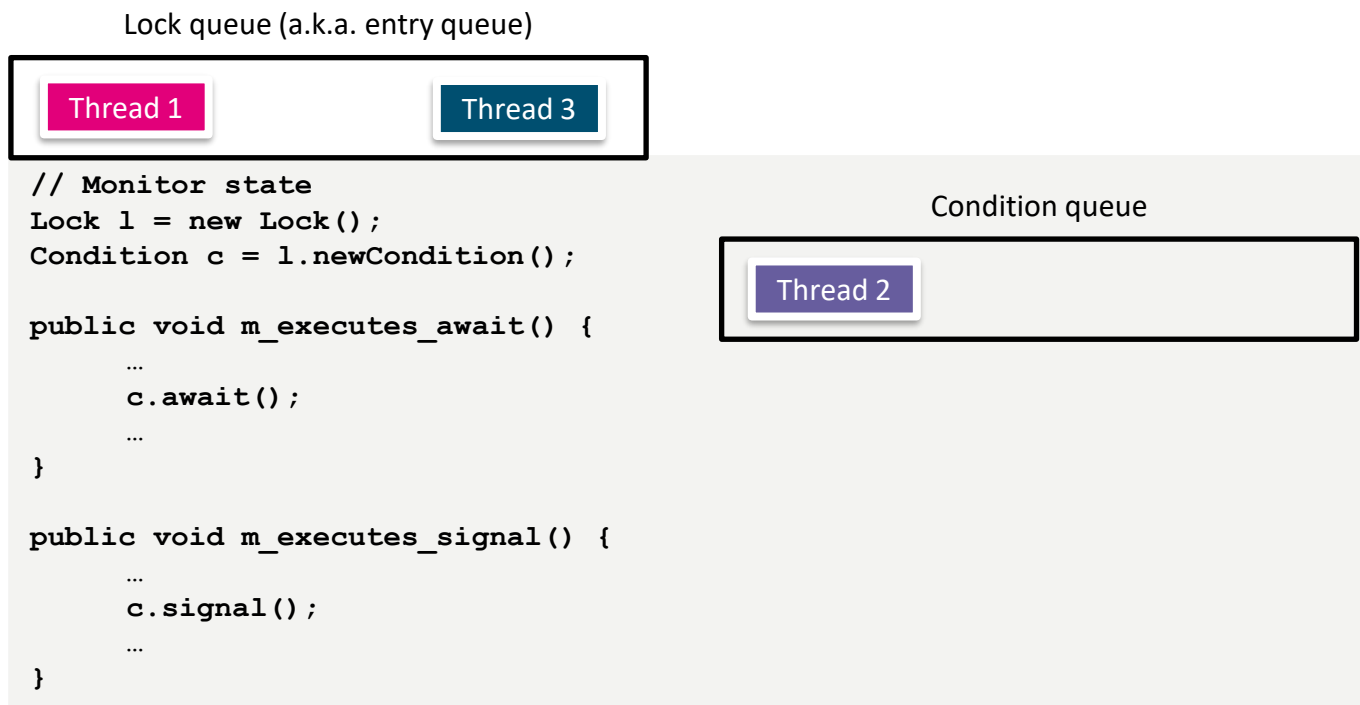
Condition queue



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



- Thread 2 executes await and goes to the condition queue





- 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)

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();
    ...
}
```

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

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();
    ...
}
```

Condition queue



Thread 3



- Thread 3 releases the lock and finishes execution

Lock queue (a.k.a entry queue)

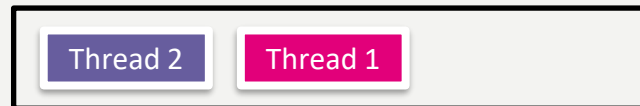


```
// Monitor state
Lock l = new Lock();
Condition c = l.newCondition();

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

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

Condition queue





- Thread 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)

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();
    ...
}
```

Note that in the condition queue nothing ensures fairness either

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
 - Mesa semantics: Threads going to the entry queue to compete for entering the monitor again
 - Java semantics is Mesa.
 - Hoare semantics: 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

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

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

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



- 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?

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

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

```
public 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();}
}
```

A note on busy-wait

· 37



- 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();}
}
```

- 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() {  
        ...  
    }  
}
```



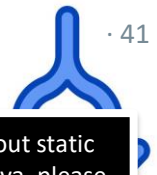
```
class C {  
    public T method() {  
        synchronized (this) {  
            ...  
        }  
    }  
}
```

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();
```

Java Intrinsic Locks | **synchronized**

· 41



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) {  
            ...  
        }  
    }  
}
```

Java Intrinsic Locks | **synchronized**

· 41



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();
```


Java Intrinsic Locks | **synchronized**

· 42



- 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();

    ...

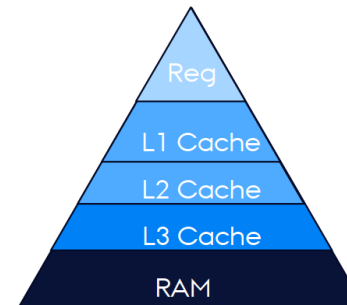
    o.notifyAll();
}
```

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)

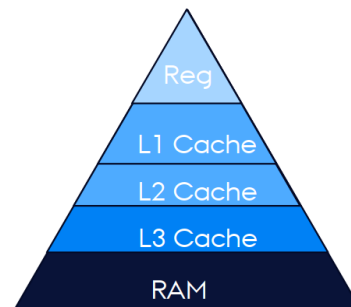


Wasted inst
0
12
36
108
678

- 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)

More precisely, in the absence of a happen-before relation between statements of different threads, it is not guaranteed that they will see/view the same shared memory state



Wasted inst
0
12
36
108
678



- 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");
```



- Complete program in `NoVisibility1.java`

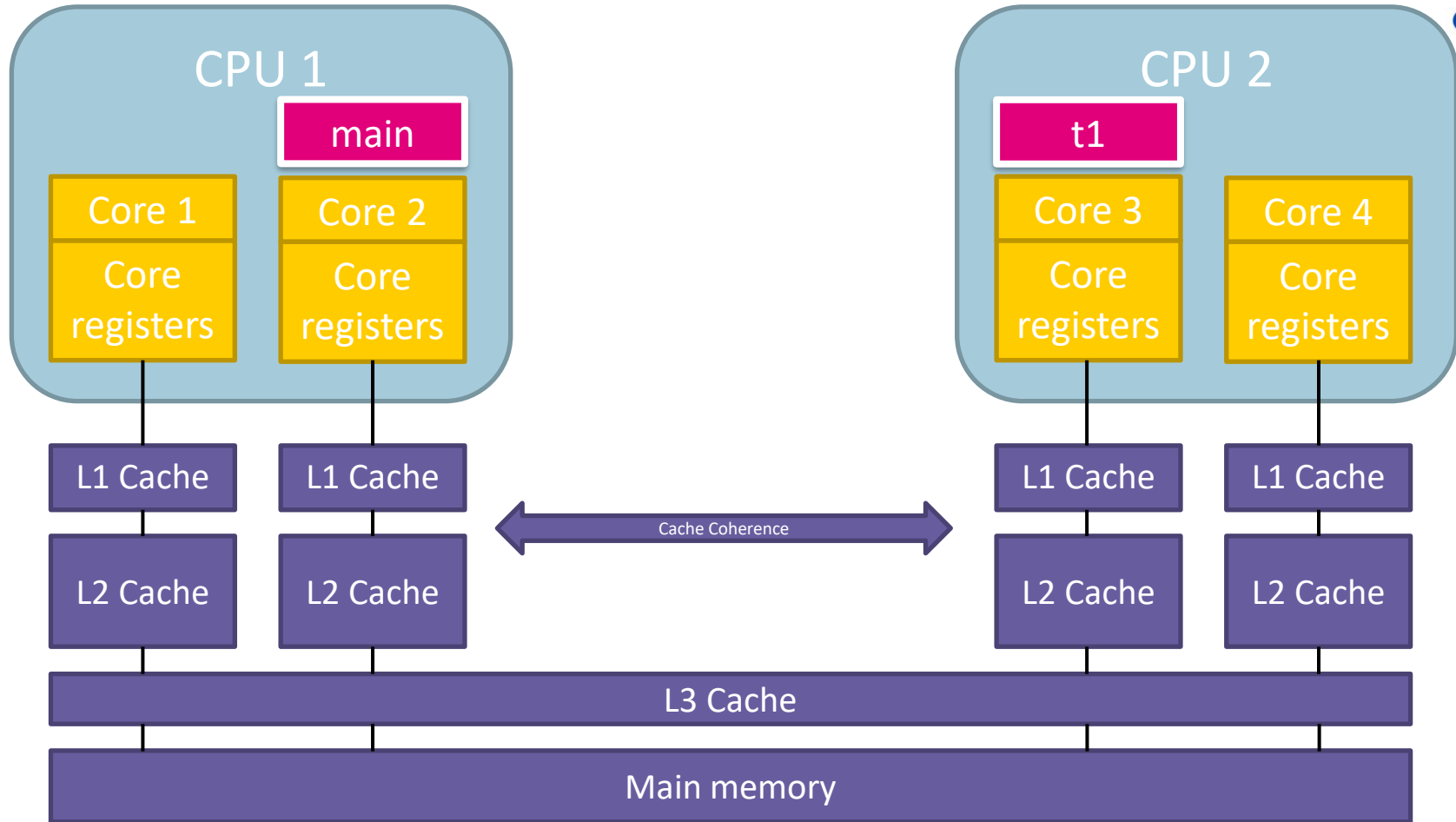
What are the possible outputs of this program?

```
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");
```

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

Visibility | Memory Hierarchy (simplified)

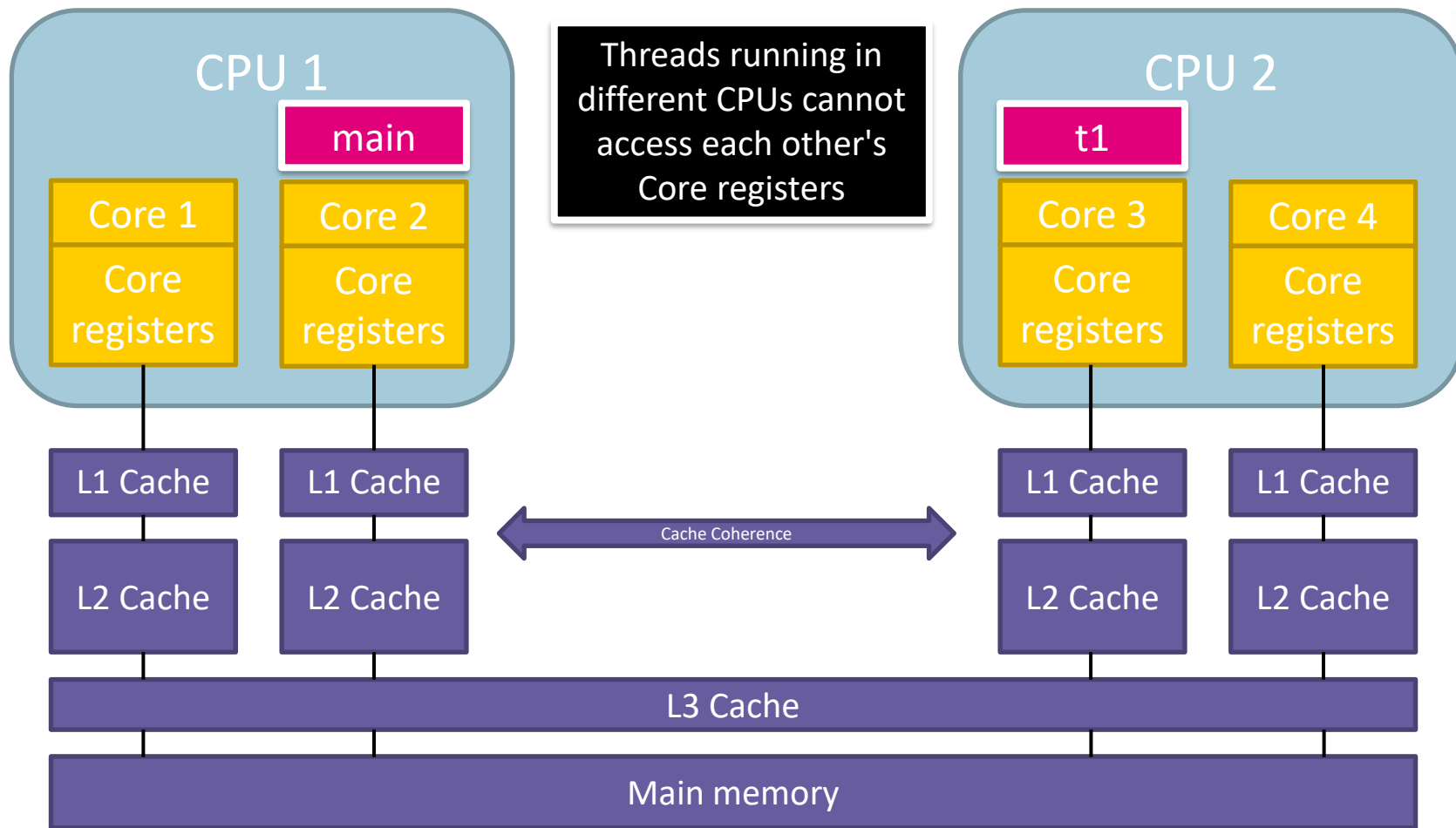
· 47



Drawing and explanation inspired from: <https://www.youtube.com/watch?v=nNXzDS6dQg>

Visibility | Memory Hierarchy (simplified)

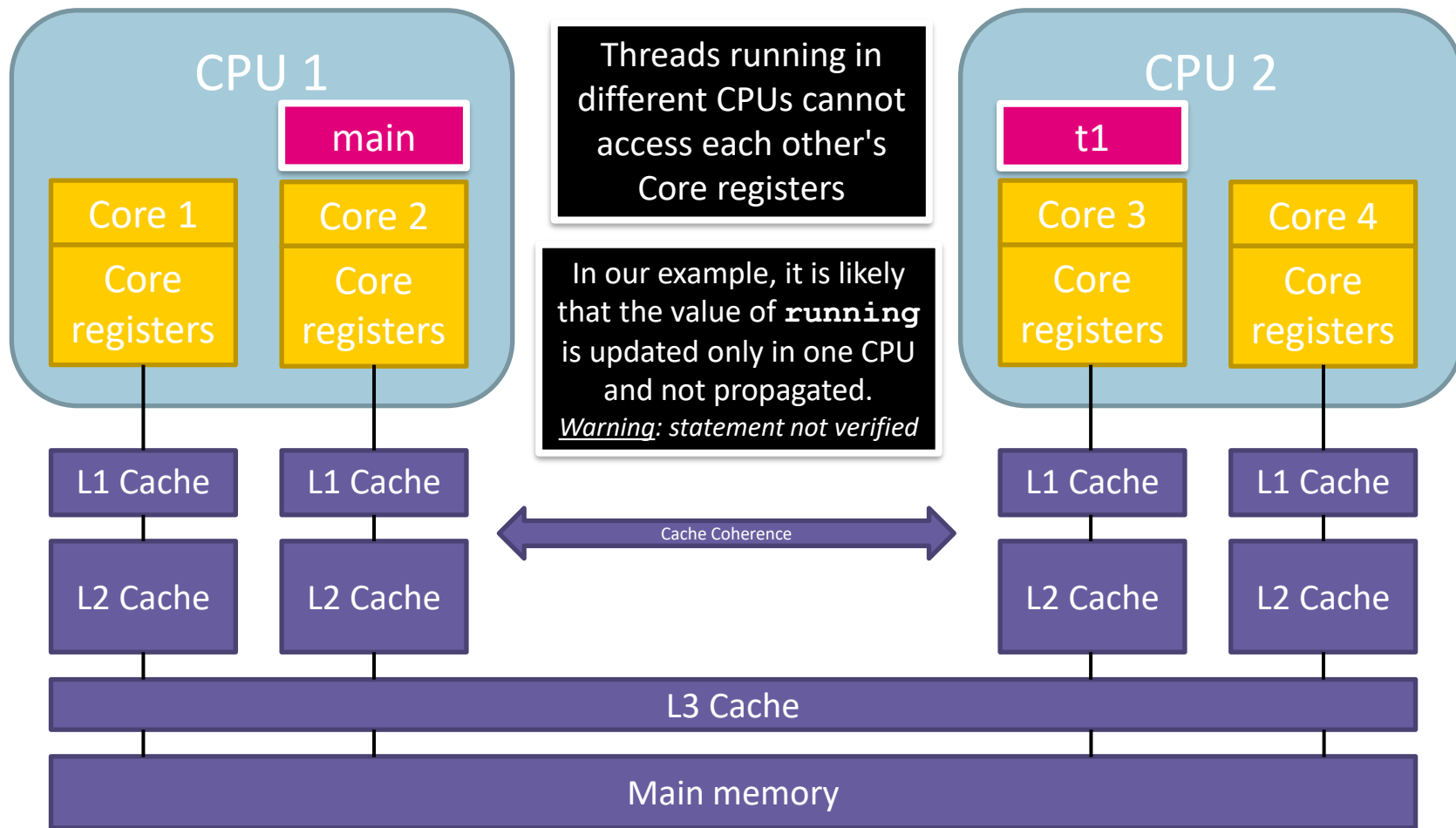
· 47



Drawing and explanation inspired from: <https://www.youtube.com/watch?v=nNXzDS6dQg>

Visibility | Memory Hierarchy (simplified)

· 47



Drawing and explanation inspired from: <https://www.youtube.com/watch?v=nNXzDS6dQg>



- Why do visibility problems occur?
 - Simple: lack of happens-before relation between operations, see [JLS](#)
 - In the program below, it holds
 - $t1(\text{while}(\text{running})) \nrightarrow \text{main}(\text{running} := \text{false})$ and $\text{main}(\text{running} := \text{false}) \nrightarrow t1(\text{while}(\text{running}))$
 - 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

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

```
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");
```



- Establishing a happen-before relation enforces visibility
 - In the program below, it holds
 - $while(running) \rightarrow running := false$ or $running := false \rightarrow while(running)$
 - Consequently, the CPU is not allowed to keep the value of `running` in the register of the CPU or cache and must 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`



- Establishing a happen-before relation enforces visibility
 - In the program below, it holds
 - $while(running) \rightarrow running := false$ or $running := false \rightarrow while(running)$
 - Consequently, the CPU is not allowed to keep the value of `running` in the register of the CPU or cache and must flush it to main memory

Why do we write
“or” here?

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`



What are the possible outputs of this program?

- Establishing a happen-before relation enforces visibility
 - In the program below, it holds
 - $while(running) \rightarrow running := false$ or $running := false \rightarrow while(running)$
 - Consequently, the CPU is not allowed to keep the value of running in the register of the CPU or cache and must flush it to main memory

Why do we write “or” here?

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`

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 operations between these instructions

The JIT compiler or CPU are allowed to perform this reordering



The lack of dependences in intra-thread operations and happens-before relation allows the reordering resulting in the output (0,0)

- Due to lack of synchronization operations, we cannot establish a happen-before relation with operations among threads either, thus
 - $one(a := 1) \nrightarrow other(b := 1)$ and $one(a := 1) \nrightarrow other(y := a)$
 - Analogous with $x := b, b := 1, y := a$
- The JIT compiler and CPU can reorder operations so that the 4 following interleavings are valid
 - $one(x := 0), other(y := 0), one(a := 1), other(b := 1)$
 - $one(x := 0), other(y := 0), other(b := 1), one(a := 1)$
 - $other(y := 0), one(x := 0), one(a := 1), other(b := 1)$
 - $other(y := 0), one(x := 0), other(b := 1), one(a := 1)$

these are the 4 possible interleavings that produce (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 + ")");
```



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

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



Establishing a happen-before relation prevents (some) reordering.

Now the output (0,0) is not possible.

- The intrinsic monitor introduces happens-before relations
(monitor rule) $one(x := b) \rightarrow other(b := 1)$ or $other(y := a) \rightarrow one(a := 1)$
(program order rule) $one(a := 1) \rightarrow one(x := b)$ and $other(b := 1) \rightarrow other(y := a)$
- Combining these relations, we have
(1) $one(a := 1) \rightarrow one(x := b) \rightarrow other(b := 1) \rightarrow other(y := a)$
(2) $other(b := 1) \rightarrow other(y := a) \rightarrow one(a := 1) \rightarrow one(x := b)$
- Consider one of the (0,0) interleavings (see prev. slides)
 $one(x := 0), other(y := 0), one(a := 1), other(b := 1)$
- The interleaving contradicts
 $one(a := 1) \rightarrow other(y := a)$ in (1) and
 $other(b := 1) \rightarrow one(x := b)$ in (2).
Thus, the interleaving is not valid.
- Similar reasoning can be used to find contradictions in the remaining 3 cases.

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

The lack of data dependences
still allows for re-orderings
within the critical section (JLS)

"It should be noted that the presence of a happens-before relationship between two actions does not necessarily imply that they have to take place in that order in an implementation. If the reordering produces results consistent with a legal execution, it is not illegal." (JLS)



Reordering can still happen within critical sections!

- Happen-before relations of the program

(1) $one(a := 1) \rightarrow one(x := b) \rightarrow other(b := 1) \rightarrow other(y := a)$
(2) $other(b := 1) \rightarrow other(y := a) \rightarrow one(a := 1) \rightarrow one(x := b)$

- This is a valid interleaving

$one(a := 1), one(x := b), other(b := 1), other(y := a)$

- The output of this interleaving is (0,1)
- The following reordering does not satisfy the happens before constraints, but can occur because the observable behaviour is the same as above, i.e., it outputs (0,1)

$one(x := b), one(a := 1), other(b := 1), other(y := a)$

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

"It should be noted that the presence of a happens-before relationship between two actions does not necessarily imply that they have to take place in that order in an implementation. If the reordering produces results consistent with a legal execution, it is not illegal." ([JLS](#))



Reordering can still happen within critical sections!

- Happen-before relations of the program

(1) $one(a := 1) \rightarrow one(x := b) \rightarrow other(b := 1) \rightarrow other(y := a)$
(2) $other(b := 1) \rightarrow other(y := a) \rightarrow one(a := 1) \rightarrow one(x := b)$

- This is a valid interleaving

$one(a := 1), one(x := b), other(b := 1), other(y := a)$

- The output of this interleaving is (0,1)
- The following reordering does not satisfy the happens before constraints, but can occur because the observable behaviour is the same as above, i.e., it outputs (0,1)

$one(x := b), one(a := 1), other(b := 1), other(y := a)$

That said, we do not need to worry about this: "Once the determination that the code is correctly synchronized is made, the programmer does not need to worry that reorderings will affect his or her code." ([JLS](#))

```
// 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();
// ... later ...
one.join();
other.join();
println(" (" + x + ", " + y + ") ");
```

- 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

- Volatile variables in terms of reads/writes and happens-before
 - A write to a volatile variable happens before any subsequent read to the volatile variable
- *Volatile variables cannot be used to ensure mutual exclusion!*
 - Note that neither reads or writes are blocking operations



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 + ")");
```



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

- Because of volatile and program order we have
(Program order)
 $one(a := 1) \rightarrow one(x := b)$ and
 $other(b := 1) \rightarrow other(y := a)$
(Volatile)
 $one(a := 1) \rightarrow other(y := a)$ or $other(y := a) \rightarrow one(a := 1)$ and
 $other(b := 1) \rightarrow one(x := b)$ or $one(x := b) \rightarrow other(b := 1)$
- Recall the interleavings for output (0,0)
 $one(x := 0), other(y := 0), one(a := 1), other(b := 1)$ (1)
 $one(x := 0), other(y := 0), other(b := 1), one(a := 1)$ (2)
 $other(y := 0), one(x := 0), one(a := 1), other(b := 1)$ (3)
 $other(y := 0), one(x := 0), other(b := 1), one(a := 1)$ (4)
- The only volatile premises that allowed these interleavings are
 $other(y := a) \rightarrow one(a := 1)$ and
 $one(x := b) \rightarrow other(b := 1)$
- Using the program order premises, we can derive
 $other(b := 1) \rightarrow other(y := a) \rightarrow one(a := 1) \rightarrow one(x := b)$
 $one(a := 1) \rightarrow one(x := b) \rightarrow other(b := 1) \rightarrow other(y := a)$
- None of the interleavings (1)-(4) satisfy these two properties. Thus, (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 + ") ");
```



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

- Because of volatile and program order we have

(Program order)

$one(a := 1) \rightarrow one(x := b)$ and
 $other(b := 1) \rightarrow other(y := a)$

(Volatile)

$one(a := 1) \rightarrow other(y := a)$ or $other(y := a) \rightarrow one(a := 1)$ and
 $other(b := 1) \rightarrow one(x := b)$ or $one(x := b) \rightarrow other(b := 1)$

- Recall the interleavings for output (0,0)

$one(x := 0), other(y := 0), one(a := 1), other(b := 1)$ (1)
 $one(x := 0), other(y := 0), other(b := 1), one(a := 1)$ (2)
 $other(y := 0), one(x := 0), one(a := 1), other(b := 1)$ (3)
 $other(y := 0), one(x := 0), other(b := 1), one(a := 1)$ (4)

- The only volatile premises that allowed these interleavings are
 $other(y := a) \rightarrow one(a := 1)$ and
 $one(x := b) \rightarrow other(b := 1)$
- Using the program order premises, we can derive

$other(b := 1) \rightarrow other(y := a) \rightarrow one(a := 1) \rightarrow one(x := b)$
 $one(a := 1) \rightarrow one(x := b) \rightarrow other(b := 1) \rightarrow other(y := a)$

- None of the interleavings (1)-(4) satisfy these two properties. Thus, (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

- 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.*

- 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
- The good news: 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 consult the language memory model (happens-before relation)
- Even better news: 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