Practical Concurrent and Parallel Programming II

Shared Memory I

Raúl Pardo

Groups and Oral Feedback Sessions

- As of Sep 8th at 16.30
  - 26 people do not have a group
    - *Please contact us if you are having trouble finding a group*
  - 59 people have not booked and oral feedback slot
    - This also means that 33 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]

Submission next week (a few remarks)

- Next week on Friday at 7.59 we have the first submission
- Your submission must contain a link to a GitHub repository
- The repository must be readable (not private) by the TA's 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 solution (even if they all contain 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

Agenda

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

Previously on PCPP…

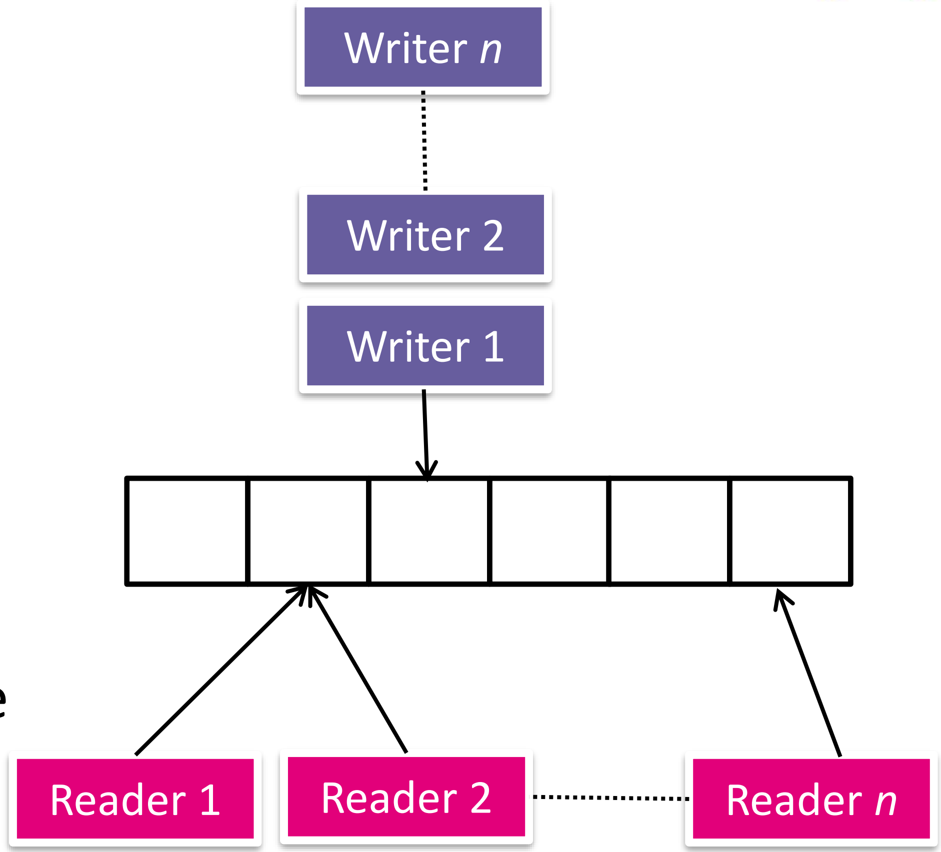
- Introduction to concurrency
- Java threads
- The Mutual Exclusion Problem
- Java Locks

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

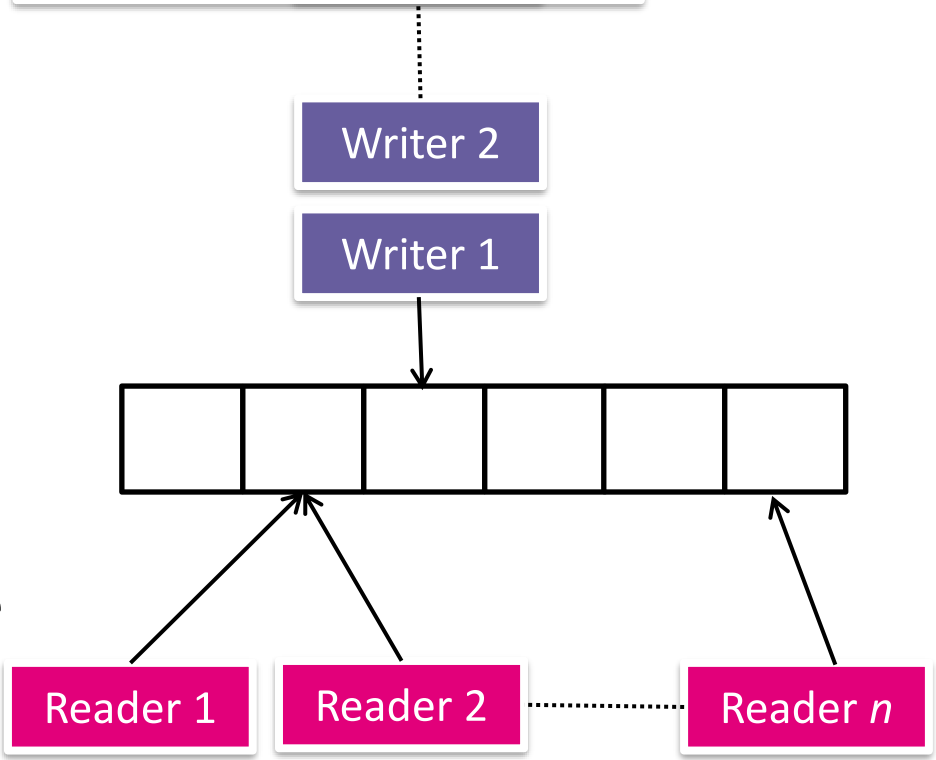
- At most one thread can write at the same time



Readers-Writers Problem

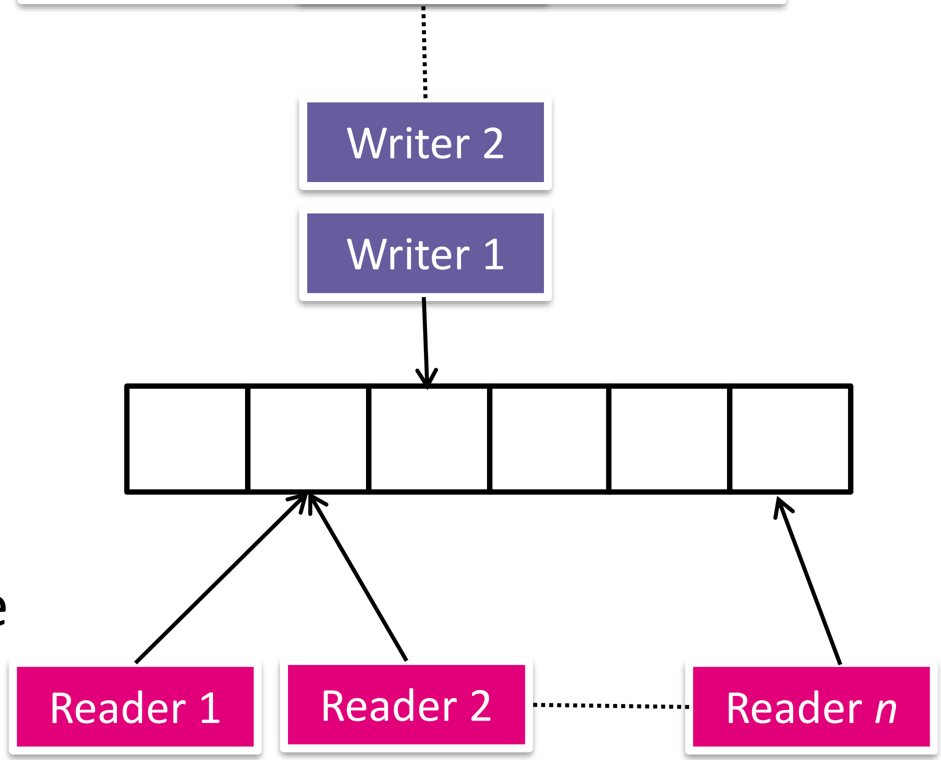
- Consider a shared data structure (e.g., an array, list set, …) where threads may read and write
- Many threads can read from the structure as long as no thread is writing
- At most one thread can write at the same time

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



Readers-Writers Problem

No because we need to put threads to wait depending on a condition



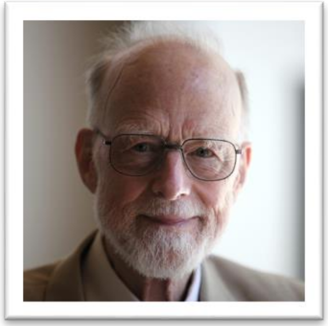
- Consider a shared data structure (e.g., an array, list set, …) where threads may read and write

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

- At most one thread can write at the same time

Monitors

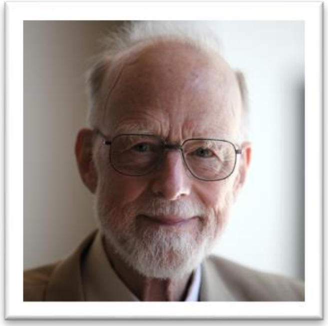
- A *monitor* is a structured way of encapsulating data, methods and synchronization in a single modular package
  - First introduced by Tony Hoare (right photo, see optional readings) and the Danish computer scientist Per Brinch Hansen (left photo)



- *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

Monitors

- 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)

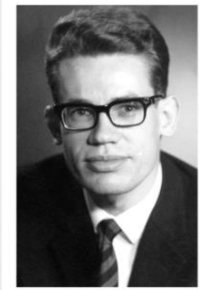


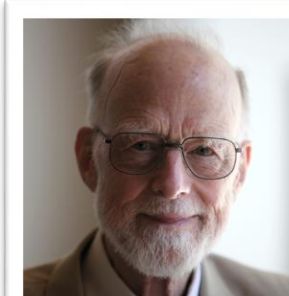
- *A monitor consists of:*
  - Internal state (data)
  - Methods (procedures)
    - All methods in a monitor are mutually exclusive (ens
    - Methods can only access internal state
  - Condition variables (or simply conditions)
    - Queues where the monitor can put threads to wait
- In Java (and generally in OO), monitors are conveniently implemented as classes

Can race conditions appear in a monitor method?

Monitors

- 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 (en
    - Methods can only access internal state
  - Condition variables (or simply conditions)
    - Queues where the monitor can put threads to wait

No because they only access internal state and all methods are mutually exclusive

- 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)

Monitor | Example I

- 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();}
}
```

Monitors | Readers-Writers Problem

- 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();
…
}
```

Monitors | Readers-Writers Problem

- 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();}
}
```

Monitors | Readers-Writers Problem

- 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();}
}
```

Monitors | Readers-Writers Problem

- 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();}
}

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

Monitors | Readers-Writers Problem

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

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

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

```
                                   n 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();}
}
```

Monitors | Readers-Writers Problem

- 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();
```

No, because the code for writers checks readers > 0 after waking up

```
                                  n 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();}
}
```

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

Monitors | Readers-Writers Problem

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

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

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

If so, we put the thread to wait

If not, the writer takes the lock

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

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

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

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

Monitors | Readers-Writers Problem

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

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

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

```
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

Monitors | Readers-Writers Problem

- 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 {lo
}
```

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

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

We release the writer lock unconditionally

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

Monitors | Readers-Writers Problem

- 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();
```

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

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

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

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

We release the writer lock unconditionally

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

Monitors | Readers-Writers Problem

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

Yes, firstly, due to spurious wake-ups. But, furthermore, if, for any reason (e.g., bugs in the implementation), the condition is signalled before the while-condition holds then the algorithm will work incorrectly

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

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

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

```
          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

Monitors | Readers-Writers Problem

- 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++;
      }
      ca
      fi
}
```

```
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

```
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();}
}
```

Monitors | Readers-Writers Problem

- 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();
}
```

Fairness in Monitors

- 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?

Fairness in Monitors

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

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

- Because, if readers are more frequent than writers (as it is the case when running this program), then writers will be locked out indefinitely. Note that readers can take the lock as long as there no writer, but writers require that no writer holds the lock and also that there are no readers

Fairness in Monitors

- 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

Fairness in Monitors

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

Lock queue (a.k.a entry queue)

Thread 1 | Thread 2 | Thread 3

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

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

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

Condition queue

Fairness in Monitors

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

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

Thread 1 Thread 3

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

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

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

Thread 2

Condition queue

Fairness in Monitors

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

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

Thread 1 Thread 3

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

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

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

Condition queue

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

Fairness in Monitors

- Thread 2 is selected (non-deterministi... 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

Condition queue

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

`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 1

- Thread 2 is selected (non-determinist
  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

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

Fairness in Monitors

- Thread 1 is selected and executes await as well

Lock queue (a.k.a entry queue)

Thread 3

Condition queue

Thread 2 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();
      …
}
```

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

Fairness in Monitors

- 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

Fairness in Monitors

- Thread 3 releases the lock and finishes execution

Lock queue (a.k.a entry queue)

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

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

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

Condition queue

Thread 2 Thread 1

Fairness in Monitors

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

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

Thread 1

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

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

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

Condition queue

Thread 2

Fairness in Monitors

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

Note that in the condition queue nothing ensures fairness either

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

Fairness in Monitors

- 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

Signalling semantics

- 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

Starvation (revisited)

- *<u>Absence of starvation</u>*: 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

```
lic void writeLock() {
      lock.lock();
      ry {
          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?

```
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

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

Starvation sol. | Readers-Writers Problem

- 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();}
}
```

Starvation sol. | Readers-Writers Problem

- 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();}
}
```

Starvation sol. | Readers-Writers Problem

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

Assuming that all threads run to competition, yes. But remember that the scheduler can still choose to never allow the writer thread to make progress.

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

```
public v
      loc
      try
             condition.await();
           writer=true;
           while(readsAcquires != readsReleases)
             condition.await();

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

Fairness

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

Fairness

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

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

Fairness

- There exist two notions of fairness
  - Weak fairness: A thread that is continuously active will eventually make progress
  - Strong fairness: 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?

Fairness

- 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

No, it only ensures that the writer thread will be scheduled fairly, but if there are many readers, it will simply wait again in the condition

Assuming that the algorithm is fair (e.g., the solution to reader-writer problem we presented), then it ensures weak fairness. This is because the writer will never starve in the lock entry queue

A note on busy-wait

- *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();}
}
```

Java Intrinsic Locks | `synchronized`

- 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
}
```

Java Intrinsic Locks | `synchronized`

- **`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) {
            …
        }
    }
}
```

Java Intrinsic Locks | `synchronized`

- **`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) {
            …
        }
    }
}
```

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

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

Are these two threads using the same intrinsic lock?

Java Intrinsic Locks | `synchronized`

- **`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) {
            …
        }
    }
}
```

No, synchronized on a regular method uses the instance lock. The two threads use different instances of the object

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

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

Java Intrinsic Locks | `synchronized`

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`

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) {
            …
        }
    }
}
```

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();
```

Java Intrinsic Locks | `synchronized`

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) {
            …
        }
    }
}
```

Yes, synchronized on a static method uses the class lock. Consequently, all instances of this class will use the same lock

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

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

Java Intrinsic Locks | **`synchronized`**

- 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

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

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

    …

    c.signalAll();

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

⟷

```java
Object o = new Object();

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

    …

    o.notifyAll();
}
```

Hardware and Programming Language (Java) Concurrency issues

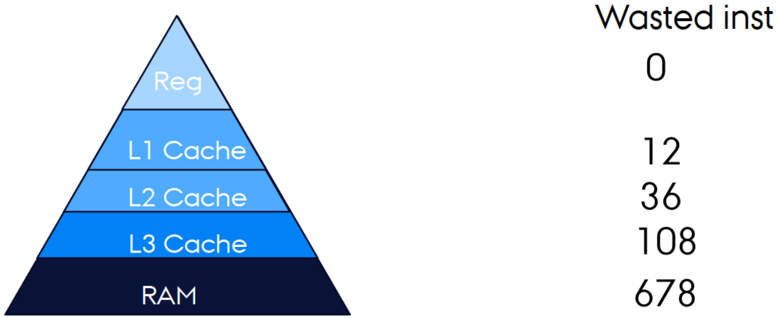
Visibility

- **In the absence of "synchronization" the CPU is allowed 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

Visibility

- **In the absence of "synchronization" the CPU is allowed 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

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)

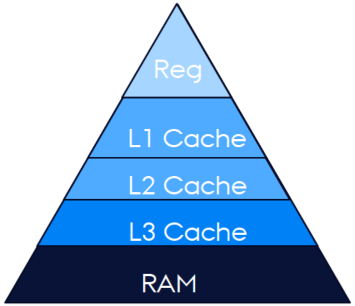


Visibility

- **In the absence of “synchronization” the CPU is allowed 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

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

Visibility

- Complete program in **`NoVisibility1.java`**

What are the possible outputs of this program?

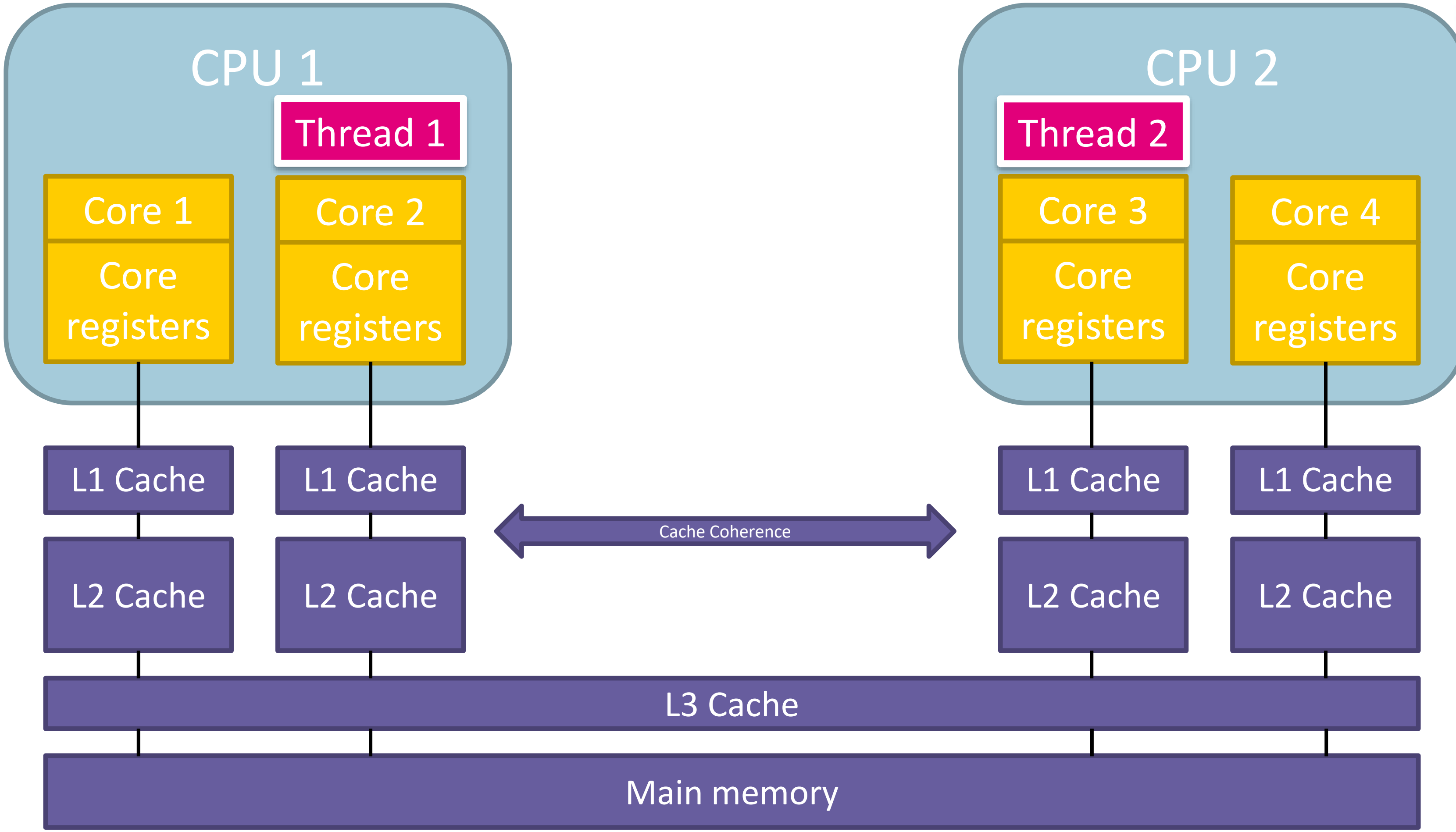
```
boolean running = true;
Thread t1 = new Thread(() -> {
   while (running) {
      /* do nothing */
   }
   System.out.println(“t1 finishing execution");
})
t1.start();
try{Thread.sleep(500);}catch(…){…}
running = false;
System.out.println("Main finishing execution");
```

Visibility

1. t1 finishing execution| Main finishing execution
2. Main finishing execution| t1 finishing execution
3. Main finishing execution (running==false is never visible to the inner thread)

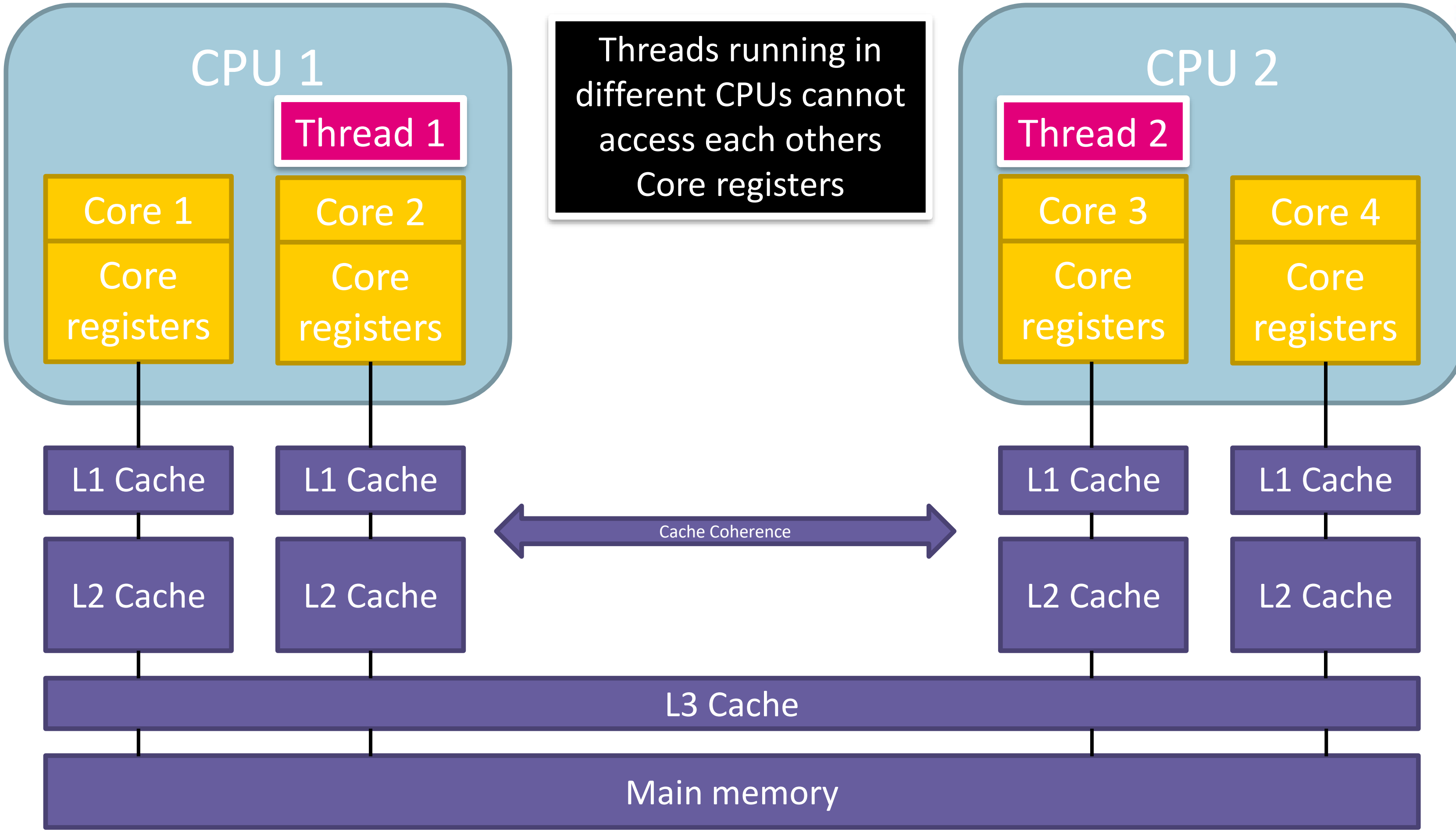
```
Thread t1 = new Thread(() -> {
    while (running) {
        /* do nothing */
    }
    System.out.println(“t1 finishing execution");
})
t1.start();
try{Thread.sleep(500);}catch(…){…}
running = false;
System.out.println("Main finishing execution");
```

Visibility | Memory Hierarchy (simplified)



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

Visibility | Memory Hierarchy (simplified)

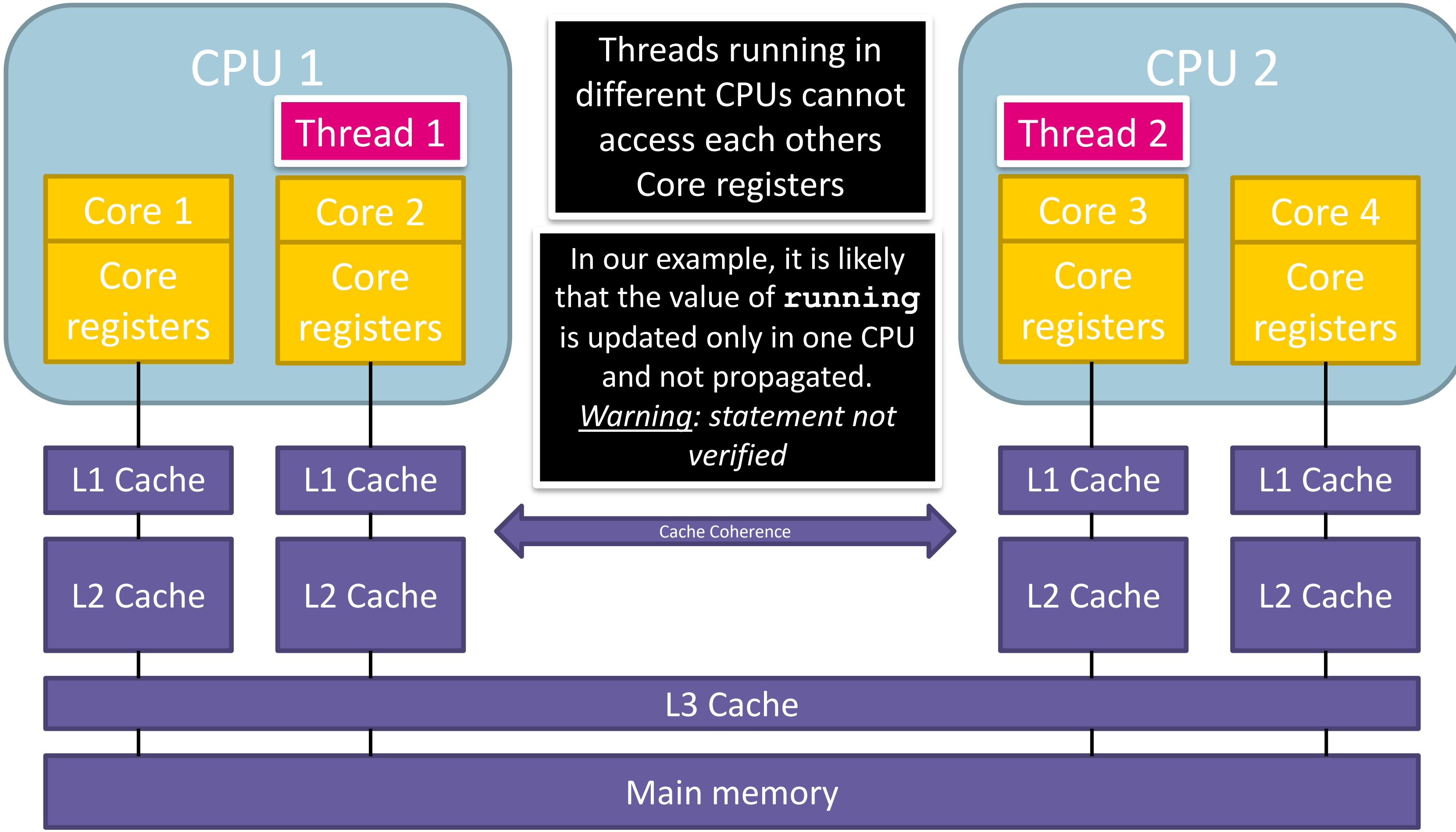


Drawing and explanation inspired from: https://www.youtube.com/watch?v=nNXkzDS6dOg

Visibility | Memory Hierarchy (simplified)

Threads running in different CPUs cannot access each others Core registers

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



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

Visibility

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

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

Visibility

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

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();
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`**

Visibility

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

Why did I write "or" here

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();
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
  - We can use locks or synchronized (as they are equivalent)
  - In the program below, it holds
  - $while(running) \rightarrow running \coloneqq false$ <u>or</u>
    $running \coloneqq false \rightarrow while(running)$
  - 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

Why did I write "or" here

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();
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`

Visibility

1. t1 finishing execution | Main finishing execution
2. Main finishing execution | t1 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
  - $while(running) \rightarrow running \coloneqq false$ <u>or</u> $running \coloneqq false \rightarrow while(running)$
  - Consequently, the CPU is <u>not</u> allowed to keep the va the register of the CPU or cache and <u>must</u> flush it to

Because both interleavings may occur, depending on what threads the scheduler decides to execute first

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();
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`**

Reordering

- 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)

Reordering

- 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+")");
```

Reordering

- Complete program in **`PossibleReordering.java`**

Yes, due to lack of happens-before relation between operations the JVM is allowed to reorder them

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

Reordering

- 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

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 data dependences we have
  - $a \coloneqq 1 \rightarrow x \coloneqq b$ or $x \coloneqq b \rightarrow a \coloneqq 1$ and $b \coloneqq 1 \rightarrow y \coloneqq a$ or $y \coloneqq a \rightarrow b \coloneqq 1$
- Of course, due to lack of synchronization we cannot establish a happen-before relation with operations among threads either, thus
  - $a \coloneqq 1 \nrightarrow b \coloneqq 1$ and $a \coloneqq 1 \nrightarrow y \coloneqq a$
  - Analogous with $x \coloneqq b, b \coloneqq 1, y \coloneqq a$
- Consequently, the JIT compiler can reorder operations so that the following interleavings is valid
  - $one(x \coloneqq 0), other(y \coloneqq 0), one(a \coloneqq 1), other(b \coloneqq 1)$

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

Reordering

This is stretching a bit too much the use of happens-before, as this step is a program transformation (not a product of possible interleavings). After the transformation one of them hold.

The lack of dependences in intra-threa [illegible] and happens-before relation allows the reo [illegible] resulting in the output (0,0)

- Due to lack of data dependences we have
  - $a \coloneqq 1 \rightarrow x \coloneqq b$ or $x \coloneqq b \rightarrow a \coloneqq 1$ and $b \coloneqq 1 \rightarrow y \coloneqq a$ or $y \coloneqq a \rightarrow b \coloneqq 1$
- Of course, due to lack of synchronization we cannot establish a happen-before relation with operations among threads either, thus
  - $a \coloneqq 1 \nrightarrow b \coloneqq 1$ and $a \coloneqq 1 \nrightarrow y \coloneqq a$
  - Analogous with $x \coloneqq b, b \coloneqq 1, y \coloneqq a$
- Consequently, the JIT compiler can reorder operations so that the following interleavings is valid
  - $one(x \coloneqq 0), other(y \coloneqq 0), one(a \coloneqq 1), other(b \coloneqq 1)$

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

Reordering

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

- The intrinsic monitor introduces the following happens-before relations (assuming `one` takes the lock first):
- $a \coloneqq 1 \rightarrow b \coloneqq 1$ and
  $a \coloneqq 1 \rightarrow y \coloneqq a$ and
  $x \coloneqq b \rightarrow b \coloneqq 1$ and
  $x \coloneqq b \rightarrow y \coloneqq a$

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

- Assume, by contradiction, that the program outputs (0,0). This can only be realised by the following interleaving
  $one(x \coloneqq 0), other(y \coloneqq 0), one(a \coloneqq 1), other(b \coloneqq 1)$
- However, note that $y \coloneqq a \rightarrow a \coloneqq 1$ holds in the interleaving, which contradicts our premise $a \coloneqq 1 \rightarrow y \coloneqq a$. So the interleaving 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+")");
```

`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

`volatile` | happens-before

- 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*</u> <u>read</u> to the volatile variable

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

volatile

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

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

- Assume, by contradiction, that the output of the program is (0,0). This can only happen as a result of any of the following interleavings
  $one(x \coloneqq b), other(y \coloneqq a), one(a \coloneqq 1), other(b \coloneqq 1)$ (1) or
  $one(x \coloneqq b), other(y \coloneqq a), other(b \coloneqq 1), one(a \coloneqq 1)$ (2) or
  $other(y \coloneqq a), one(x \coloneqq b), one(a \coloneqq 1), other(b \coloneqq 1)$ (3) or
  $other(y \coloneqq a), one(x \coloneqq b), other(b \coloneqq 1), one(a \coloneqq 1)$ (4)

- In (1) and (2), it holds $x \coloneqq b \rightarrow a \coloneqq 1$ which contradicts the premise $a \coloneqq 1 \rightarrow x \coloneqq b$

- In (3) and (4), it holds $y \coloneqq a \rightarrow b \coloneqq 1$ which contradicts the premise $b \coloneqq 1 \rightarrow y \coloneqq a$

- 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)
  $a \coloneqq 1 \rightarrow x \coloneqq b$ and
  $b \coloneqq 1 \rightarrow y \coloneqq a$
- Assume, by contradiction, that the output of the program is (0,0). This can only happen as a result of any of the following interleavings
  $one(x \coloneqq b), other(y \coloneqq a), one(a \coloneqq 1), other(b \coloneqq 1)$ (1) or
  $one(x \coloneqq b), other(y \coloneqq a), other(b \coloneqq 1), one(a \coloneqq 1)$ (2) or
  $other(y \coloneqq a), one(x \coloneqq b), one(a \coloneqq 1), other(b \coloneqq 1)$ (3) or
  $other(y \coloneqq a), one(x \coloneqq b), other(b \coloneqq 1), one(a \coloneqq 1)$ (4)
- In (1) and (2), it holds $x \coloneqq b \rightarrow a \coloneqq 1$ which contradicts the premise $a \coloneqq 1 \rightarrow x \coloneqq b$
- In (3) and (4), it holds $y \coloneqq a \rightarrow b \coloneqq 1$ which contradicts the premise $b \coloneqq 1 \rightarrow y \coloneqq a$
- 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)
  $a \coloneqq 1 \rightarrow x \coloneqq b$ and
  $b \coloneqq 1 \rightarrow y \coloneqq a$

- Assume, by contradiction, that the output of the program is (0,0). This can only happen as a result of any of the following interleavings
  $one(x \coloneqq b), other(y \coloneqq a), one(a \coloneqq 1), other(b \coloneqq 1)$ (1) or
  $one(x \coloneqq b), other(y \coloneqq a), other(b \coloneqq 1), one(a \coloneqq 1)$ (2) or
  $other(y \coloneqq a), one(x \coloneqq b), one(a \coloneqq 1), other(b \coloneqq 1)$ (3) or
  $other(y \coloneqq a), one(x \coloneqq b), other(b \coloneqq 1), one(a \coloneqq 1)$ (4)

- In (1) and (2), it holds $x \coloneqq b \rightarrow a \coloneqq 1$ which contradicts the premise $a \coloneqq 1 \rightarrow x \coloneqq b$

- In (3) and (4), it holds $y \coloneqq a \rightarrow b \coloneqq 1$ which contradicts the premise $b \coloneqq 1 \rightarrow y \coloneqq a$

- 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();
```

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

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)
  $a \coloneqq 1 \rightarrow x \coloneqq b$ and
  $b \coloneqq 1 \rightarrow y \coloneqq a$

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

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
$(a \coloneqq 1 \rightarrow y \coloneqq a \text{ or } y \coloneqq a \rightarrow a \coloneqq 1) \quad \neq \quad (a \coloneqq 1 \nrightarrow y \coloneqq a \text{ and } y \coloneqq a \nrightarrow a \coloneqq 1)$

- 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?

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 out

```
// shared variables
x=0;
y=0;
volatile a=0;
inition
ew Thread(() -> {
 new Thread(() -> {
her.start();
```

Because x and y are only read after the join, so there is already a happens-before relation between the write and the read

A note on `volatile` visibility

- 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 used 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);
  }
}
```

A note on **`volatile`** visibility

- 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 used 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);
  }
}
```

A note on **`volatile`** visibility

- Writing on a volatile variable flashes

Yes, but only if reader executes before writer executes x=42. The reason why x!=0 when v==true is that volatile variables force a flush of the complete cache, so x is flushed to main memory as well

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` vs `Locks` Summary

- 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.*

Visibility, Reordering & `volatile`

- 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 look up the semantics for these notions (if you use them)

- 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