Practical Concurrent and Parallel Programming II

Shared Memory I

Raúl Pardo

Groups and Oral Feedback Sessions

- As of Sep 1st, at 09.00
  - 76 people do not have a group
    - *Please contact us if you are having trouble finding a group*
  - 130 people have not booked and oral feedback slot
    - This also means that 54 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 07.59 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

Previously on PCPP…

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

Agenda

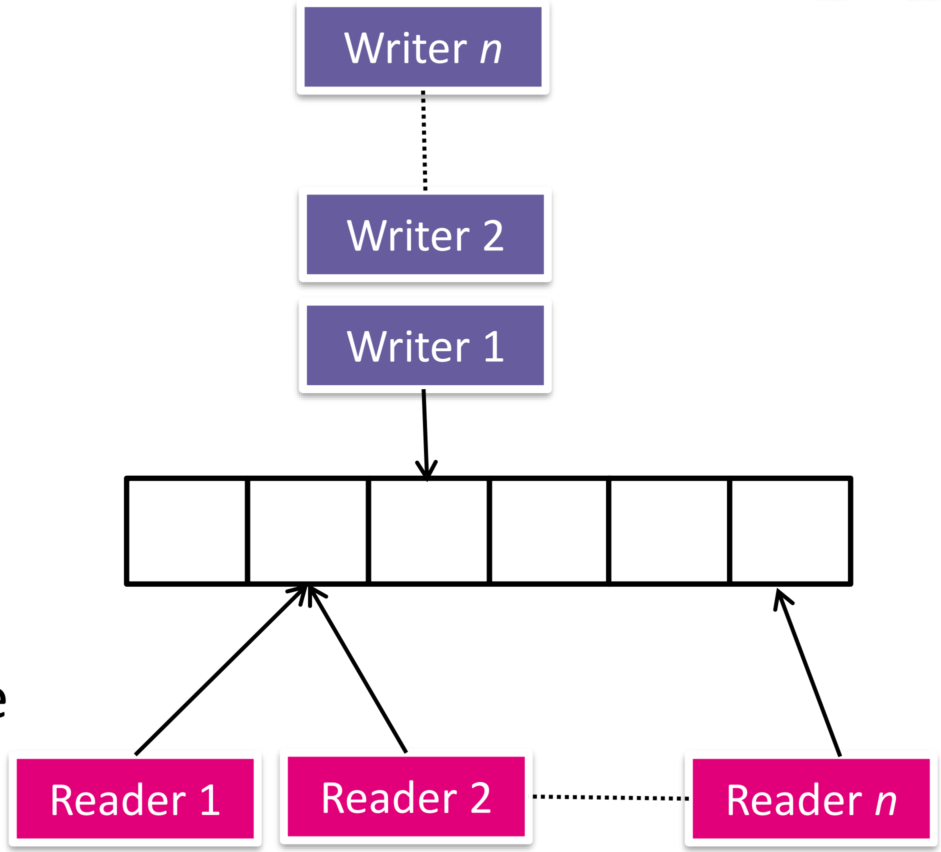
- Readers and Writers Problem
- Monitors
- Fairness
- Java Intrinsic Locks (**`synchronized`**)
- Hardware and Programming Language Concurrency Issues
- Visibility
- Reordering
- Happens-before
- Volatile variables (**`volatile`**)

Readers-Writers Problem

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

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

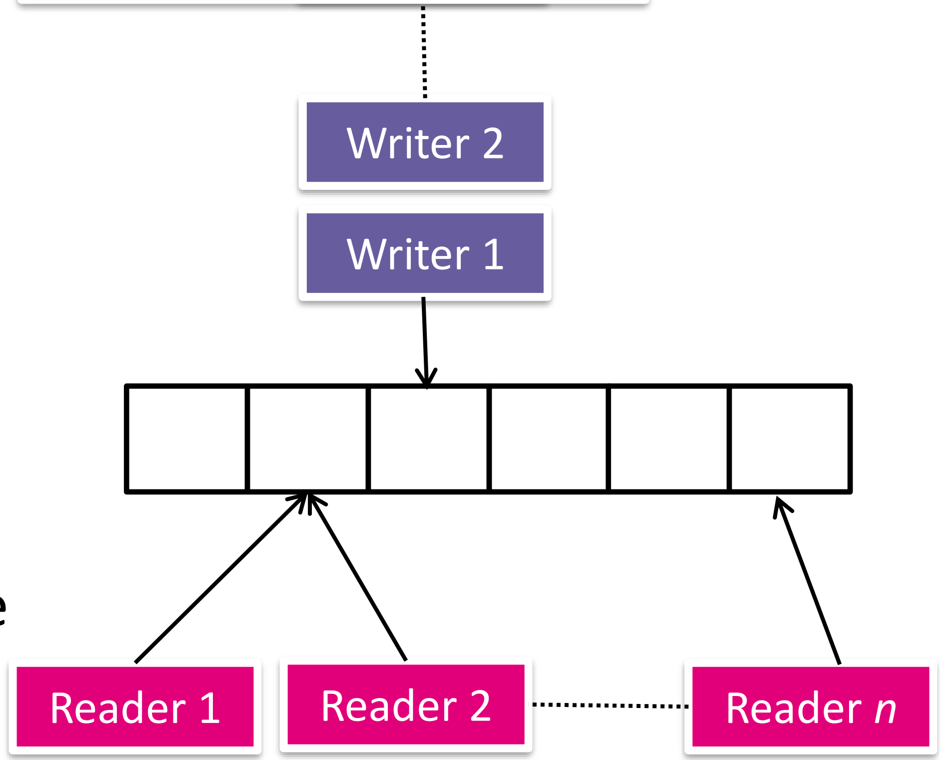
- At most one thread can write at the same time



Readers-Writers Problem

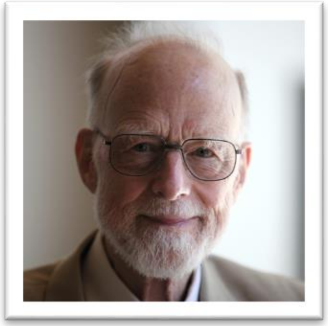
Can we solve this problem using locks as we saw last week?

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



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

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 thread blocked on the queue, if any
  - **`signalAll()`** – wakes up all threads blocked on the queue, if any
- When threads wake up they acquire the lock immediately (before the execute anything else)

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) {
               c.await()
          }
     }
     catch (InterruptedException e) {…}
     finally {l.unlock();}
}
```

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) {
               c.await()
          }
     }
     catch (InterruptedException e) {…}
     finally {l.unlock();}
}
```

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

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

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();}
}

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 (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
}
```

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

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

We release the writer lock unconditionally

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

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();
```

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

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

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

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

We release the writer lock unconditionally

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

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++;
                                          n e) {…}
}

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

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

```
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++;
      }
      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

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

- The monitor seems to be *unfair* towards writers

Fairness in Monitors

- 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.

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

Condition 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();
      …
}
```

Fairness in Monitors

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

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

Thread 1 Thread 3

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

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

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

Condition queue

Fairness in Monitors

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

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

Thread 1

Thread 3

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

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

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

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

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

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

Condition queue

Thread 2 | Thread 1

Fairness in Monitors

- Thread 1 is selected and executes await as well

Lock queue (a.k.a entry queue)

Thread 3

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

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

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

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

Condition queue

Thread 2 | Thread 1

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

Condition queue

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

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

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

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 priority over threads waiting on the entry queue
    - In our example, any thread in the entry queue could be selected; independently on whether it came from the condition queue

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

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

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

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

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 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();}
}
```

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.
- 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
}
```

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();
```

These two threads are using different locks, as `method()` uses the instance lock

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) {
            …
        }
    }
}
```

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`

- 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

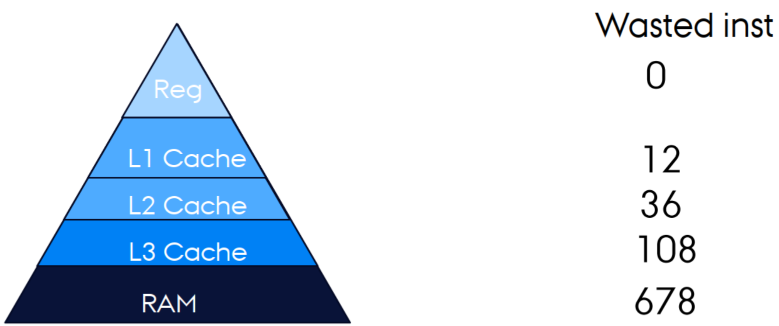
Visibility

- *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

Visibility

- *<u>In the absence of synchronization operations the CPU is allowed to keep data in the CPU's registers/cache</u>*
  - Thus, *<u>it might not be visible for threads running on a different CPU</u>*
  - 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

- 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

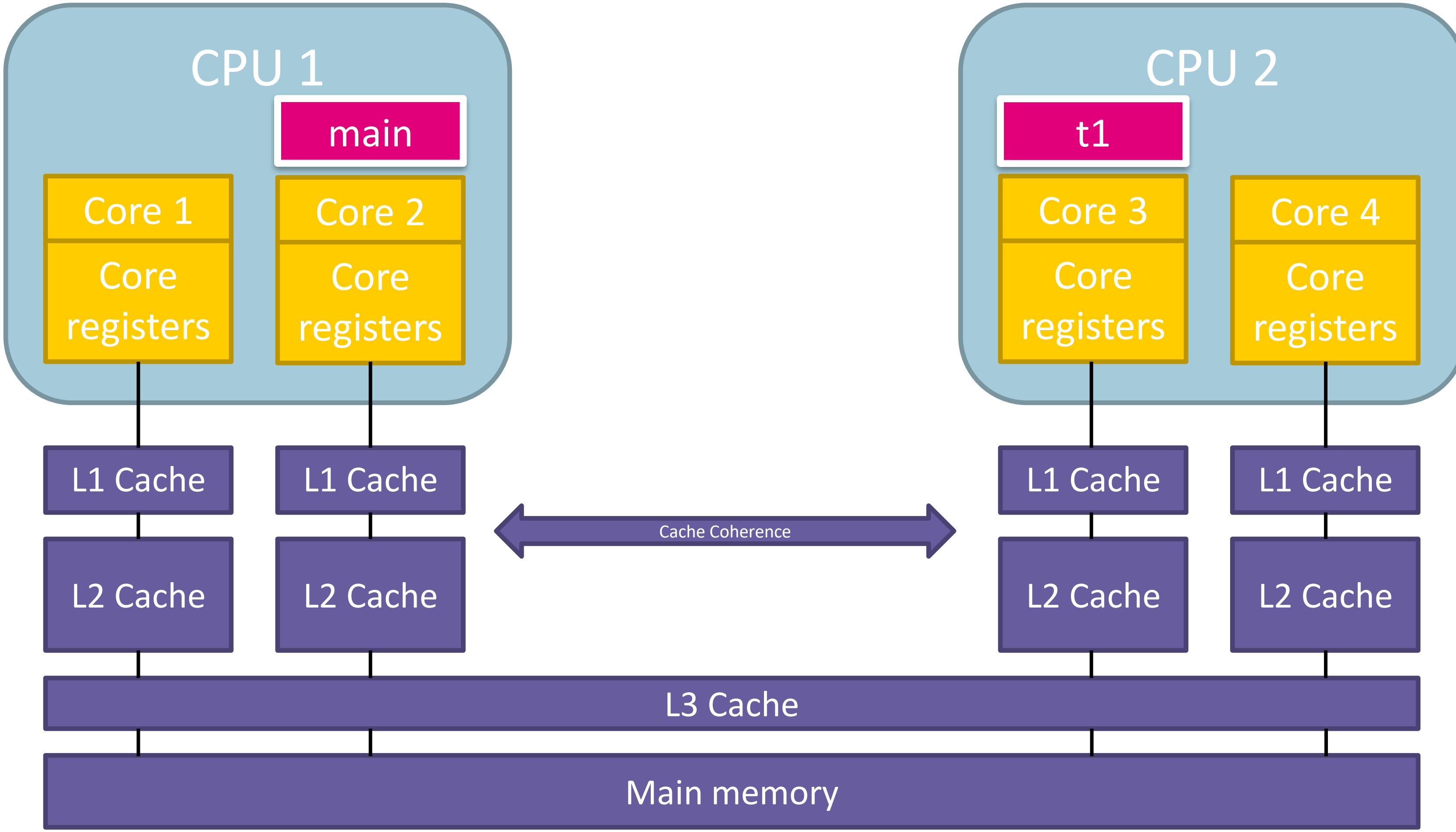
- Complete program in **`NoVisibility1.java`**

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");
```

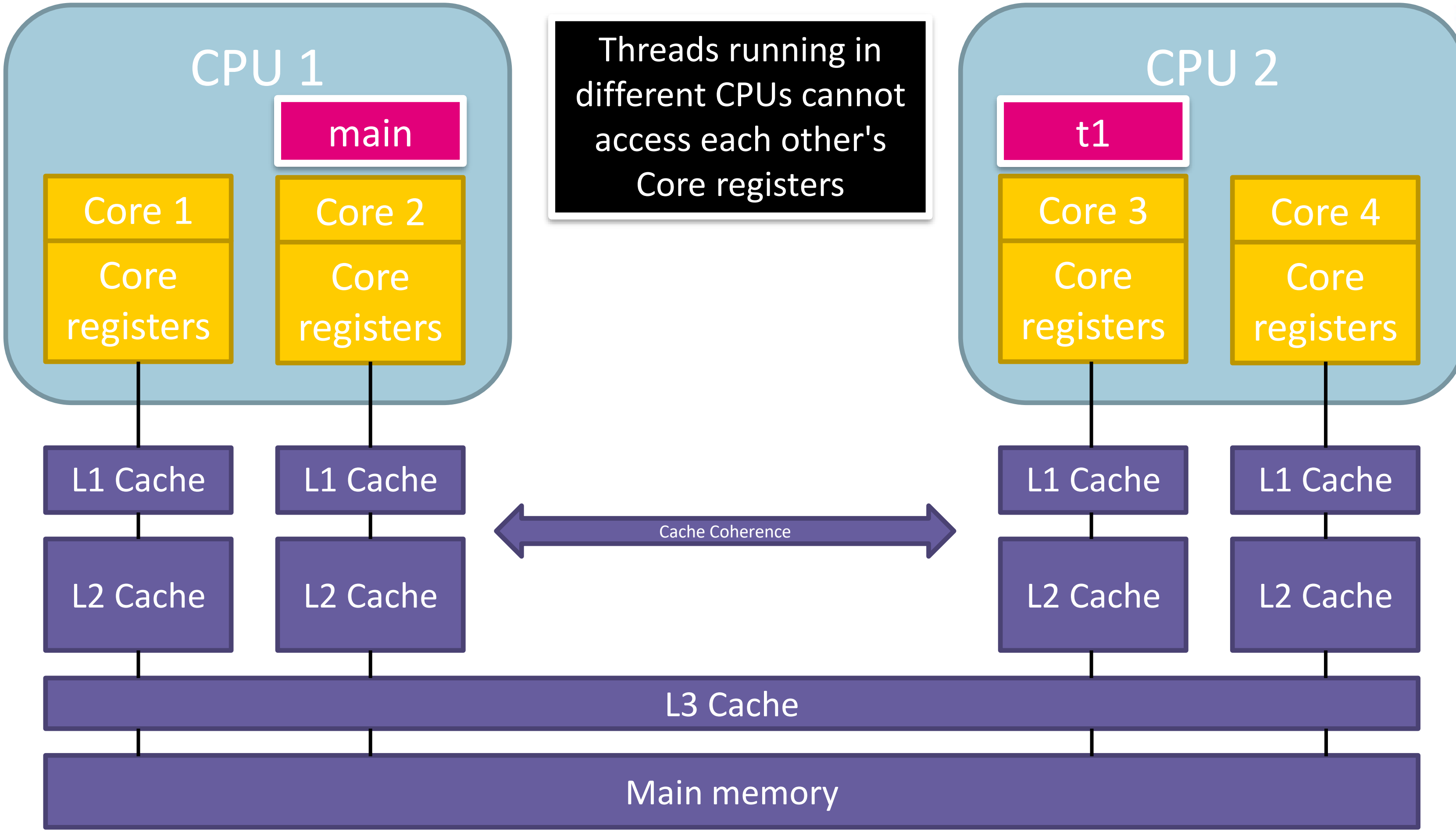
<u>DISCLAIMER</u>: This is not a “Java issue”. It can happen in any programming language with support for concurrency

Visibility | Memory Hierarchy (simplified)



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

Visibility | Memory Hierarchy (simplified)

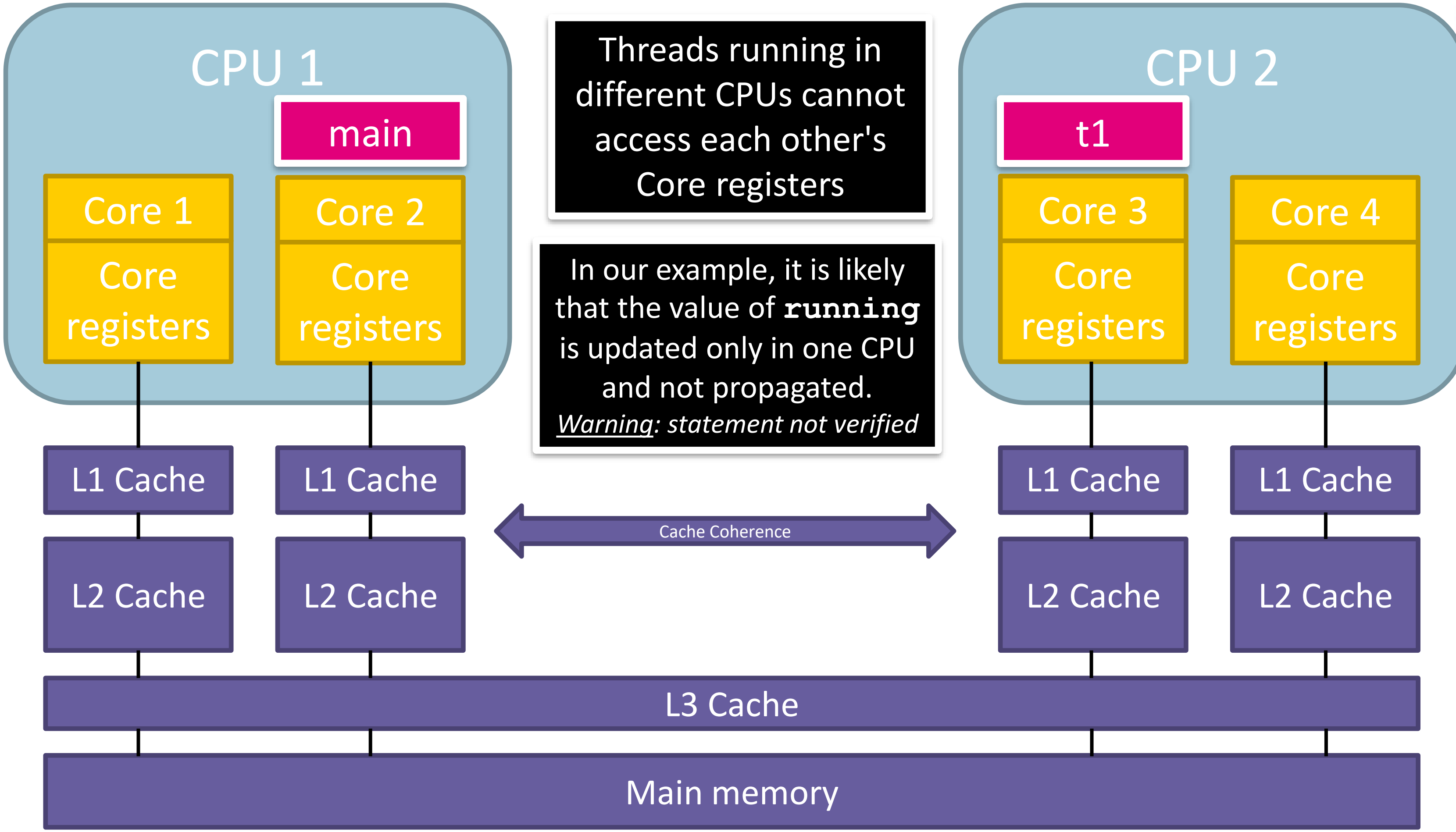


Drawing and explanation inspired from: https://www.youtube.com/watch?v=nNXkzDS6dOg

Visibility | Memory Hierarchy (simplified)

Threads running in different CPUs cannot access each other's 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

Happens-before

- A *memory model* characterizes the set of <u>*valid*</u> executions of a concurrent program (interleavings) in terms of a *happens-before relation* (may be called differently in other languages)

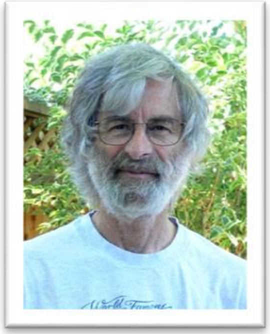
- We say that an operation a <u>*happens-before*</u> an operation b in an interleaving, denoted as $a \rightarrow b$, iff
  - a and b belong to the same thread and a appears before b in the thread definition
  - a is an **`unlock()`** and b is a **`lock()`** on the same lock
  - Given an operation c, we have that $a \rightarrow c$ and $c \rightarrow b$ (transitivity)

- In the absence of *happens-before* relation between operations, we say that operations are executed *concurrently*
  - Sometimes denoted as $a \mid\mid b$

- <u>The JVM ensures that if $a \rightarrow b$ then effect of operation *a* is *visible* by operation *b*</u>

Happens-before

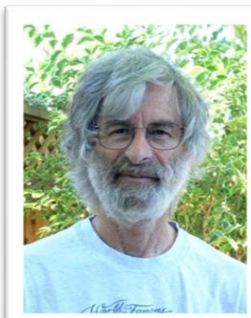
- A *memory model* characterizes the set of <u>*valid*</u> executions of a concurrent program (interleavings) in terms of a *happens-before relation* (may be called differently in other languages)
- We say that an operation a <u>*happens-before*</u> an operation b in an interleaving, denoted as $a \rightarrow b$, iff
  - a and b belong to the same thread and a appears before b in the thread definition
  - a is an **`unlock()`** and b is a **`lock()`** on the same lock
  - Given an operation c, we have that $a \rightarrow c$ and $c \rightarrow b$ (transitivity)
- In the absence of *happens-before* relation between operations, we say that operations are executed *concurrently*
  - Sometimes denoted as $a \mid\mid b$
- <u>The JVM ensures that if $a \rightarrow b$ then effect of operation *a* is *visible* by operation *b*</u>
- "Happened-before" was first introduced by Leslie Lamport for distributed systems
  - See optional readings



Happens-before

We will focus on the Java happens-before relation (page 341 Goetz and JLS documentation)

- A *memory model* characterizes the set of *valid* executions of a concurrent program (interleavings) in terms of a *happens-before relation* (may be called differently in other languages)
- We say that an operation a *happens-before* an operation b in an interleaving, denoted as $a \rightarrow b$, iff
  - a and b belong to the same thread and a appears before b in the thread definition
  - a is an **`unlock()`** and b is a **`lock()`** on the same lock
  - Given an operation c, we have that $a \rightarrow c$ and $c \rightarrow b$ (transitivity)
- In the absence of *happens-before* relation between operations, we say that operations are executed *concurrently*
  - Sometimes denoted as $a \parallel b$
- The JVM ensures that if $a \rightarrow b$ then effect of operation *a* is *visible* by operation *b*
- "Happened-before" was first introduced by Leslie Lamport for distributed systems
  - See optional readings

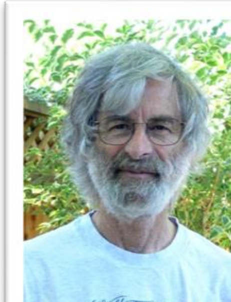


Happens-before

We will focus on the Java happens-before relation (page 341 Goetz and JLS documentation)

- A *memory model* characterizes the set of *valid* executions of a concurrent program (interleavings) in terms of a *happens-before relation* (may be called differently in other languages)
- We say that an operation a *happens-before* an operation b in an interleaving, denoted as $a \rightarrow b$, iff
  - a and b belong to the same thread and a appears before b in the thread definition
  - a is an **`unlock()`** and b is a **`lock()`** on the same lock
  - Given an operation c, we have that $a \rightarrow c$ and $c \rightarrow b$ (transitivity)
- In the absence of *happens-before* relation between operations, we say that operations are executed *concurrently*
  - Sometimes denoted as $a \,||\, b$
- The JVM ensures that if $a \rightarrow b$ then effect of operation *a* is *visible* by operation *b*

"Don't be brainwashed by programming languages. Free your mind with mathematics." Time for one question before we go straight to the next talk. #HLF18



Visibility

- Why do visibility problems occur?
  - Simple: lack of happens-before relation between operations, see JLS
  - In the program below, for all interleavings it holds that
  - $t1(while(running)) \nrightarrow main(running \coloneqq false)$ and $main(running \coloneqq false) \nrightarrow t1(while(running))$
  - Consequently, the CPU is allowed to keep the value of running in the register of the CPU or cache and not flush it to main memory

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");
```

Visibility

- Establishing a happen-before relation enforces visibility
  - In the program below, it holds for all interleavings that
  - $while(running) \rightarrow running \coloneqq false$ or
    $running \coloneqq false \rightarrow while(running)$
  - 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

This should be demonstrated rigorously using the rules of happens-before (see examples below and challenging exercises for this week)

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`

Reordering

- 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

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;
});
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`**

This program can output (0,0)

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

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

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 operations between these instructions

The JIT compiler or CPU are allowed to perform this reordering

Reordering | Happens-before

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 among operations among threads either, thus
  - $one(a \coloneqq 1) \nrightarrow other(b \coloneqq 1)$ and $one(a \coloneqq 1) \nrightarrow other(y \coloneqq a)$
  - Analogous with $x \coloneqq b, b \coloneqq 1, y \coloneqq a$

- The JIT compiler and CPU can reorder operations so that the 4 following interleavings are valid
  - $one(x \coloneqq 0), other(y \coloneqq 0), one(a \coloneqq 1), other(b \coloneqq 1)$
  - $one(x \coloneqq 0), other(y \coloneqq 0), other(b \coloneqq 1), one(a \coloneqq 1)$
  - $other(y \coloneqq 0), one(x \coloneqq 0), one(a \coloneqq 1), other(b \coloneqq 1)$
  - $other(y \coloneqq 0), one(x \coloneqq 0), other(b \coloneqq 1), one(a \coloneqq 1)$

  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+")");
```

Reordering | Happens-before

See **`PossibleReorderingSynchronized.java`**

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+")");
```

Reordering | Happens-before

Example of happens-before reasoning: If an exercise asks to use happens-before reasoning, you must: i) state the desired property (i.e., the pair of operations that must be related by happens-before), ii) list all happens-before pairs in the program, iii) apply transitivity to derive the pair of operations in i).

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

- We must show, for all interleavings, that
  $one(a \coloneqq 1) \rightarrow other(y \coloneqq a)$ or $one(b \coloneqq 1) \rightarrow other(x \coloneqq b)$
- The intrinsic monitor introduces happens-before pairs
  (monitor rule)
  $one(x \coloneqq b) \rightarrow other(b \coloneqq 1)$ or
  $other(y \coloneqq a) \rightarrow one(a \coloneqq 1)$
  (program order rule)
  $one(a \coloneqq 1) \rightarrow one(x \coloneqq b)$ and
  $other(b \coloneqq 1) \rightarrow other(y \coloneqq a)$
- Combining these pairs, we have two possible sets of happen-before pairs for all interleavings in this program
  (1) $one(a \coloneqq 1) \rightarrow one(x \coloneqq b) \rightarrow other(b \coloneqq 1) \rightarrow other(y \coloneqq a)$
  (2) $other(b \coloneqq 1) \rightarrow other(y \coloneqq a) \rightarrow one(a \coloneqq 1) \rightarrow one(x \coloneqq b)$
- By (1) and transitivity: $one(a \coloneqq 1) \rightarrow other(y \coloneqq a)$
- By (2) and transitivity: $one(b \coloneqq 1) \rightarrow other(x \coloneqq b)$

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

Reordering | Happens-before

<u>Example of happens-before reasoning</u>: If an exercise asks to use happens-before reasoning, you must: i) state the desired property (i.e., the pair of operations that must be related by happens-before), ii) list all happens-before pairs in the program, iii) apply transitivity to derive the pair of operations in i).

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

"or" here is necessary due to the non-deterministic execution order; this is known as synchronization order in the Java Memory Model

- We must show, for all interleavings:
  $one(a \coloneqq 1) \rightarrow other(y \coloneqq a)$ or $one(b \coloneqq 1) \rightarrow other(x \coloneqq b)$

- The intrinsic monitor introduces happens-before pairs

  (monitor rule)
  $one(x \coloneqq b) \rightarrow other(b \coloneqq 1)$ or
  $other(y \coloneqq a) \rightarrow one(a \coloneqq 1)$

  (program order rule)
  $one(a \coloneqq 1) \rightarrow one(x \coloneqq b)$ and
  $other(b \coloneqq 1) \rightarrow other(y \coloneqq a)$

- Combining these pairs, we have two possible sets of happen-before pairs for all interleavings in this program

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

- By (1) and transitivity: $one(a \coloneqq 1) \rightarrow other(y \coloneqq a)$
- By (2) and transitivity: $one(b \coloneqq 1) \rightarrow other(x \coloneqq b)$

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

Reordering | Happens-before

Example of happens-before reasoning: If an exercise asks to use happens-before reasoning, you must: i) state the desired property (i.e., the pair of operations that must be related by happens-before), ii) list all happens-before pairs in the program, iii) apply transitivity to derive the pair of operations in i).

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

- We must show, for all interleavings, that
  $one(a \coloneqq 1) \rightarrow other(y \coloneqq a)$ or $one(b \coloneqq 1) \rightarrow other(x \coloneqq b)$

Here I'm abusing notation and merging "one(x:=b) -> unlock() -> lock() -> other(b:=1)" as "one(x:=b) -> other(b:=1)" and similarly for the other disjunct

- The intrinsic monitor introduces

  (monitor rule)
  $one(x \coloneqq b) \rightarrow other(b \coloneqq 1)$ or
  $other(y \coloneqq a) \rightarrow one(a \coloneqq 1)$

  (program order rule)
  $one(a \coloneqq 1) \rightarrow one(x \coloneqq b)$ and
  $other(b \coloneqq 1) \rightarrow other(y \coloneqq a)$

- Combining these pairs, we have two possible sets of happen-before pairs for all interleavings in this program

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

- By (1) and transitivity: $one(a \coloneqq 1) \rightarrow other(y \coloneqq a)$
- By (2) and transitivity: $one(b \coloneqq 1) \rightarrow other(x \coloneqq b)$

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

ition
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+")");
```

Reordering | Happens-before

Example of happens-before reasoning: If an exercise asks to use happens-before reasoning, you must: i) state the desired property (i.e., the pair of operations that must be related by happens-before), ii) list all happens-before pairs in the program, iii) apply transitivity to derive the pair of operations in i).

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

- We must show, for all interleavings, that
  $one(a \coloneqq 1) \rightarrow other(y \coloneqq a)$ or $one(b \coloneqq 1) \rightarrow other(x \coloneqq b)$

- The intrinsic monitor introduces happens-before pairs
  (monitor rule)
  $one(x \coloneqq b) \rightarrow other(b \coloneqq 1)$ or
  $other(y \coloneqq a) \rightarrow one(a \coloneqq 1)$
  (program order rule)
  $one(a \coloneqq 1) \rightarrow one(x \coloneqq b)$ and
  $other(b \coloneqq 1) \rightarrow other(y \coloneqq a)$

- Combining these pairs, we have two possible sets of happen-before pairs for all interleavings in this program
  (1) $one(a \coloneqq 1) \rightarrow one(x \coloneqq b) \rightarrow other(b \coloneqq 1) \rightarrow other(y \coloneqq a)$
  (2) $other(b \coloneqq 1) \rightarrow other(y \coloneqq a) \rightarrow one(a \coloneqq 1) \rightarrow one(x \coloneqq b)$

- By (1) and transitivity: $one(a \coloneqq 1) \rightarrow other(y \coloneqq a)$
- By (2) and transitivity: $one(b \coloneqq 1) \rightarrow other(x \coloneqq b)$

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

Reordering

> *"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](JLS))*

Reordering can still happen within critical sections!

- Happens-before pairs for all interleavings of the program

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

- This is a valid interleaving

$$one(a \coloneqq 1), one(x \coloneqq b), other(b \coloneqq 1), other(y \coloneqq a)$$

- The output of this interleaving is (0,1)

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

$$one(x \coloneqq b), one(a \coloneqq 1), other(b \coloneqq 1), other(y \coloneqq a)$$

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

Reordering

> *"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)

<u>Reordering can still happen within critical sections!</u>

- Happens-before pairs for all interleavings of the program

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

- This is a valid interleaving

$$one(a \coloneqq 1), one(x \coloneqq b), other(b \coloneqq 1), other(y \coloneqq a)$$

- The output of this interleaving is (0,1)

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

$$one(x \coloneqq b), one(a \coloneqq 1), other(b \coloneqq 1), other(y \coloneqq a)$$

```
// 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();
ther.join();
rintln("("+x+","+y+")");
```

<u>If you show the existence of happens-before relation between the required operations, then you do not need to worry about this</u>: *"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)

`volatile`

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

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

- Volatile variables cannot be reordered

`volatile` | happens-before

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

volatile

See **PossibleReorderingVolatile.java**

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

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

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

volatile

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

- Because of volatile and program order we have (for all interleavings)

  (Program order)
  $one(a \coloneqq 1) \rightarrow one(x \coloneqq b)$ and
  $other(b \coloneqq 1) \rightarrow other(y \coloneqq a)$

  (Volatile)
  $one(a \coloneqq 1) \rightarrow other(y \coloneqq a)$ or
  $other(b \coloneqq 1) \rightarrow one(x \coloneqq b)$

  These are enforced because volatile variables cannot be reordered

- Using the volatile rule we obtained the desired property (recall the required property forbid the output (0,0))

  $one(a \coloneqq 1) \rightarrow other(y \coloneqq a)$ or
  $one(b \coloneqq 1) \rightarrow other(x \coloneqq b)$

```
// 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

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

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

- Because of volatile and program order we have (for all interleavings)

  (Program order)
  $one(a \coloneqq 1) \rightarrow one(x \coloneqq b)$ and
  $other(b \coloneqq 1) \rightarrow other(y \coloneqq a)$

  (Volatile)
  $one(a \coloneqq 1) \rightarrow other(y \coloneqq a)$ or
  $other(b \coloneqq 1) \rightarrow one(x \coloneqq b)$

  These are enforced because volatile variables cannot be reordered

- Using the volatile rule we obtained the desired property (recall the required property forbid the output (0,0))

  $one(a \coloneqq 1) \rightarrow other(y \coloneqq a)$ or
  $one(b \coloneqq 1) \rightarrow other(x \coloneqq b)$

```
// 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` 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 (and locks). I strongly recommend you follow their advice.

That said, *<u>all recommendations follow logically from the reasoning we have presented here.</u>*

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 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