Practical Concurrent and Parallel Programming III

Shared Memory II

Raúl Pardo

Assignment workload

- We would like to get an estimation on the amount of hours you spend on assignments

- Please go to the following mentimeter poll

  https://www.menti.com/alpseeeqzthb

  You should indicate the *amount of hours* that you spent *to complete Assignment 1*

  That is, the *amount of hours* that you spent on PCPP exercises in *the last two weeks combined*

Previously on PCPP…

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

Agenda

- Definitions of thread-safety
  - Classes
  - Programs
- Safe publication
- Immutability
- Instance confinement
- Synchronization primitives (synchronizers)
  - Semaphores
  - Barriers
- Producer-consumer problem

Program correctness

A (concurrent) program is ***correct*** *if and only if it satisfies its* ***specification***

Specification (informal)

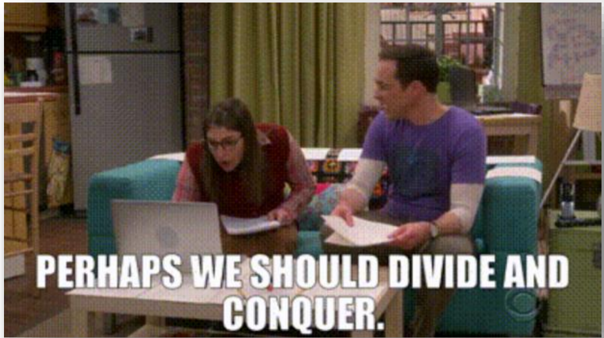
- A *specification* (or *spec*) is a rigorous statement that describes the expected/desired behaviour of a program
- Examples
  - Many readers can access the shared resource at the same time, and only one can write—if no readers are reading
  - The output of the program must be `counter*num_threads`
- Specifications can be as precise as formulae in some logic (propositional, temporal, first-order, etc.)
  - We will not cover these details in the course

Reasoning about concurrent programs

- We have already covered the basic concepts to reason about the *correctness* of concurrent programs
- Reasoning about correctness of concurrent programs is tricky
  - You have experienced this already in the assignments where you work with programs consisting in a few lines of code
- Imagine having to reason about applications with hundreds of lines of code and many classes
  - Server applications
  - Operating Systems
  - GUIs
  - …

Reasoning about concurrent programs

- We have already covered the basic concepts to reason about the *correctness* of concurrent programs
- Reasoning about correctness of concurrent programs is tricky
  - You have experienced this already in the assignments where you work with programs consisting in a few lines of code
- Imagine having to reason about applications with hundreds of lines of code and many classes
  - Server applications
  - Operating Systems
  - GUIs
  - …



Modular class-based reasoning

- It is more manageable to separately analyse parts of the code and then combine them in safe ways

- In Object Oriented languages (such as Java) we can focus on analysing thread-safety for classes

- This reduces the analysis to concurrent method calls and field accesses

Data Races (refined)

- *A **data race** occurs when two concurrent threads:*
  - *Access a shared memory location*
  - *At least one access is a write*
  - *<u>There is no happens-before relation between the accesses</u>*

New!

Data Races (refined)

- *A **data race** occurs when two concurrent threads:*
    - *Access a shared memory location*
    - *At least one access is a write*
    - *<u>There is no happens-before relation between the accesses</u>*

New!

Inspired by the Java memory model (JLS): *"A program is correctly synchronized if and only if all sequentially consistent executions are free of data races."*

Thread-safe class

A ***class*** *is said to be* ***thread-safe*** *if and only if no concurrent execution of method calls or field accesses (read/write) result in data races on the fields of the class*

PCPP teaching team

Thread-safe class

Inspired by the Java memory model (JLS): *"A program is correctly synchronized if and only if all sequentially consistent executions are free of data races."*

*A **class** is said to be **thread-safe** if and only if no concurrent execution of method calls or field accesses (read/write) result in data races on the fields of the class*

PCPP teaching team

Note that this definition is independent of class invariants as opposed to Goetz Chapter 4. This definition is more similar to Goetz Chapter 2, page 18.

Thread-safe class

IMPORTANT: In this course, *thread-safety* is not an umbrella term for code that seem to behave correctly in concurrent environments.

Inspired by the Java memory model (JLS): *"A program is correctly synchronized if and only if all sequentially consistent executions are free of data races."*

*A **class** is said to be **thread-safe** if and only if no concurrent execution of method calls or field accesses (read/write) result in data races on the fields of the class*

PCPP teaching team

Note that this definition is independent of class invariants as opposed to Goetz Chapter 4. This definition is more similar to Goetz Chapter 2, page 18.

Thread-safe class

IMPORTANT: In this course, *thread-safety* is not an umbrella term for code that seem to behave correctly in concurrent environments.

Inspired by the Java memory model (JLS): *"A program is correctly synchronized if and only if all sequentially consistent executions are free of data races."*

What is the specification in this definition?

*A **class** is said to be **thread-safe** if and only if no concurrent execution of method calls or field accesses (read/write) result in data races on the fields of the class*

Note that this definition is independent of class invariants as opposed to Goetz Chapter 4. This definition is more similar to Goetz Chapter 2, page 18.

PCPP teaching team

Thread-safe program

Do not confuse thread-safe classes with thread-safe programs.
Thread-safe programs are not defined in Goetz.

A concurrent ***program*** *is said to be* ***thread-safe***
if and only if it is race condition free

Inpired by the Java memory model *correctly synchronized program* (see previous slide), but we impose a different condition by requiring freedom of race conditions

PCPP teaching team

Thread-safety

It is very important to note that:

For any program p,

p only accesses thread-safe <u>classes</u>

$\not\Rightarrow$

p is a thread-safe <u>program</u>

Thread-safety

It is very important to note that:

For any program p,

p only accesses thread-safe <u>classes</u>

$\not\Rightarrow$

p is a thread-safe <u>program</u>

Programs using thread-safe classes may contain race conditions.

Thread-safe classes

- To analyse whether a class is thread-safe, we must simply ensure that for any concurrent execution of field access and methods calls—where at least one write access is executed—the operations are related by the happens-before relation

- In what follows, we list the elements to identify/consider:
  - Class state
  - Escaping
  - (Safe) publication
  - Immutability
  - Mutual exclusion

Thread-safe classes

- To analyse whether a class is thread-safe, we must simply ensure that for any concurrent execution of field access and methods calls—where at least one write access is executed—the operations are related by the happens-before relation

- In what follows, we list the elements to identify/consider:
  - Class state
  - Escaping
  - (Safe) publication
  - Immutability
  - Mutual exclusion

When asked to reason about the thread-safety of a class, you must always cover these elements

Class state

- By definition, (uncontrolled) concurrent access to the shared state (variables) leads to data races
- So, the first thing we need to do is to identify the fields that may be shared by several threads
- The state of a class involves the fields defined in the class
    - In a nutshell, our goal is to ensure that concurrent access to class state is free from data races

```
class C {
     // class state (variables)
     T s1;
     T s2;
     T s3;
     T s4;
     …

     // class methods
     T m1(…) {…}
     T m2(…) {…}
     T m3(…) {…}
     …
}
```

Only class state (only recommended)

- Methods should only manipulate class state or parameters
  - For instance, avoid the use of variables from parent classes

```
class C {
      // class state (variables)
      private int i = 0;

      // class methods
      public void synchronized n(List<Ingeter> l){
            l.add(42);
      }
}
```

Only class state (only recommended)

- Methods should only manipulate class state or parameters
  - For instance, avoid the use of variables from parent classes
- Methods should avoid using object references as parameters
  - We cannot guarantee happens-before relations with the referenced object

```
class C {
      // class state (variables)
      private int i = 0;

      // class methods
      public void synchronized n(List<Ingeter> l){
            l.add(42);
      }
}
```

```
// program using C

List<Integer> l = new ArrayList<Integer>();
C c = new C();
new Thread(() -> {
      l.add(1);
}).start();

new Thread(() -> {
      b.m(l); // the operations between the two
              // threads are not related by
              // happens-before
}).start();
```

Only class state (only recommended)

- Methods should only manipulate class state or parameters
  - For instance, avoid the use of variables from parent classes
- Methods should avoid using object references as parameters
  - We cannot guarantee happens-before relations with the referenced object
- That said, our *<u>definition of thread-safe class focuses on data races on the fields of the class</u>*
  - Therefore, these problems do not violate the definition

```
class C {
      // class state (variables)
      private int i = 0;

      // class methods
      public void synchronized n(List<Ingeter> l){
            l.add(42);
      }
}
```

```
// program using C

List<Integer> l = new ArrayList<Integer>();
C c = new C();
new Thread(() -> {
      l.add(1);
}).start();

new Thread(() -> {
      b.m(l); // the operations between the two
              // threads are not related by
              // happens-before
}).start();
```

Escaping


```
class Counter {
      // class state (variables)
      int i=0;

      // class methods
      public synchronized void inc(){i++;}
}
```

Escaping

- It is important to not expose shared state variables
- Otherwise, threads may use them without ensuring mutual exclusion
  - Thus, we cannot enforce a happens-before relation

```
class Counter {
      // class state (variables)
      int i=0;

      // class methods
      public synchronized void inc(){i++;}
}
```

```
// program using Counter

Counter c = new Counter();
new Thread(() -> {
      c.inc();
}).start();

new Thread(() -> {
      c.i++; // escaped the lock in inc()
}).start();
```

Escaping

- It is important to not expose shared state variables
- Otherwise, threads may use them without ensuring mutual exclusion
  - Thus, we cannot enforce a happens-before relation
- Defining all (shared) class state (primitive) variables as private ensures that these variables will only be accessed through public methods.
  - Thus, it is easier to control and reason about concurrent access

```
class Counter {
      // class state (variables)
      int i=0;

      // class methods
      public synchronized void inc(){i++;}
}
```

```
// program using Counter

Counter c = new Counter();
new Thread(() -> {
      c.inc();
}).start();

new Thread(() -> {
      c.i++; // escaped the lock in inc()
}).start();
```

Escaping


```
class IntArrayList {
    // class state
    private List<Integer> a = new ArrayList<Integer>();

    public synchronized void set(Integer index, Integer elem)
        {  a.set(index,elem); }

    public synchronized List<Integer> get() { return a; }
}
```

Can list `a` escape in `IntArrayList`?

```
class IntArrayList {
    // class state
    private List<Integer> a = new ArrayList<Integer>();

    public synchronized void set(Integer index, Integer elem)
        {  a.set(index,elem); }

    public synchronized List<Integer> get() { return a; }
}
```

Can list `a` escape in `IntArrayList`?


```
class IntArrayList {
    // class state
    private List<Integer> a = new ArrayList<Integer>();

    public synchronized void set(Integer index, Integer elem)
        {  a.set(index,elem); }

    public synchronized List<Integer> get() { return a; }
}
```

```
IntArrayList array = new IntArrayList();
new Thread(() -> {
    array.set(0,1); // access state with lock
}).start();
new Thread(() -> {
    array.get().set(0,42); // access state without locks
}).start();
```

Escaping

- Remember that when a method returns an object, we get a *reference* to that object

- Therefore, even if obtain the reference using locks, later we can modify the content of the object without locks

```
class IntArrayList {
    // class state
    private List<Integer> a = new ArrayList<Integer>();

    public synchronized void set(Integer index, Integer elem)
        {  a.set(index,elem); }

    public synchronized List<Integer> get() { return a; }
}
```

```
IntArrayList array = new IntArrayList();
new Thread(() -> {
    array.set(0,1); // access state with lock
}).start();
new Thread(() -> {
    array.get().set(0,42); // access state without locks
}).start();
```

Safe Publication

- Safe publication requires that initialization *happens-before* publication

Safe Publication

- Safe publication requires that <u>initialization *happens-before* publication</u>

- <u>Visibility issues</u> may appear <u>during initialization</u> of objects

```
public class UnsafeInitialization {
    private int x;
    private Object o;
    public UnsafeInitialization() {
        x = 42;
        o = new Object();
    }
}
```

Safe Publication

- Safe publication requires that <u>initialization *happens-before* publication</u>

- <u>Visibility issues</u> may appear <u>during initialization</u> of objects

```
public class UnsafeInitialization {
    private int x;
    private Object o;
    public UnsafeInitialization() {
        x = 42;
        o = new Object();
    }
}
```

- For the thread executing the constructor, there are no visibility issues, but if a reference to an instance of UnsafeInitialization object is accessible to another thread, it might not see **`x==42`** or **`o`** completely initialized

Safe Publication

- We can address visibility issues during initialization as follows

```
public class UnsafeInitialization {
    private volatile int x;
    private final Object o;
    public UnsafeInitialization() {
        x = 42;
        o = new Object();
    }
}
```

Safe Publication

- We can address visibility issues during initialization as follows

For primitive types, we can:

- Declare them as **`volatile`**
- Declare them as **`final`** (only works if the content is never modified)
- Initialize them as the default value: 0. (only works if the default value is acceptable)
- Declare them as **`static`** (only works if the field must be **`static`** in the class)
- Use corresponding atomic class from Java standard library: **`AtomicInteger`**

```
public class UnsafeInitialization {
    private volatile int x;
    private final Object o;
    public UnsafeInitialization() {
        x = 42;
        o = new Object();
    }
}
```

Safe Publication

- We can address visibility issues during initialization as follows

For primitive types, we can:

- Declare them as **`volatile`**
- Declare them as **`final`** (only works if the content is never modified)
- Initialize them as the default value: 0. (only works if the default value is acceptable)
- Declare them as **`static`** (only works if the field must be **`static`** in the class)
- Use corresponding atomic class from Java standard library: **`AtomicInteger`**

```
public class UnsafeInitialization {
    private volatile int x;
    private final Object o;
    public UnsafeInitializatio
        x = 42;
        o = new Object();
    }
}
```

For complex objects, we can:

- Declare them as **`final`**
- Initialize them as the default value: null. (only works if the default value is acceptable)
- Declare them as **`static`** (only works if the field must be **`static`** in the class)
- Use the **`AtomicReference`** class

Safe Publication

- The previous suggestions ensure safe publication because:
  - They established a *happens-before* relation between initialization and access the object's reference (publication)
    - *A write to a volatile field happens-before every subsequent read of that field.*
    - *The default initialization (zero, false, or null) of any object happens-before any other actions of a program.*
    - *The initialization of final and static fields happens-before any other actions of a program (after the constructor has finished its execution)*

  - At the JVM level, the reason is that
    - **`final`** and **`static`** fields cannot remain in cache after the constructor finishes
    - All fields are initialized with default values during class loading
    - writes on **`volatile`** are flushed to main memory (during initialization)

Safe Publication

- The previous suggestions ensure safe publication because:
  - They established a *happens-before* relation between initialization and access the object's reference (publication)

Defined by us from the JLS explanation and Goetz. You can use it for exercises in this course.

– *A write to a volatile field happens-before every subsequent read of that field.*

– *The default initialization (zero, false, or null) of any object happens-before any other actions of a program.*

– *The initialization of final and static fields happens-before any other actions of a program (after the constructor has finished its execution)*

  - At the JVM level, the reason is that

– **`final`** and **`static`** fields cannot remain in cache after the constructor finishes

– All fields are initialized with default values during class loading

– writes on **`volatile`** are flushed to main memory (during initialization)

Safe Publication

- The previous suggestions ensure safe publication because:
  - They established a *happens-before* relation between initialization and access the object's reference (publication)
    - *A write to a volatile field happens-before every subsequent read of that field.*
    - *The default initialization (zero, false, or null) of any object happens-before any other actions of a program.*
    - *The initialization of final and static fields happens-before any other actions of a program (after the constructor has finished its execution)*

Defined by us from the JLS explanation and Goetz. You can use it for exercises in this course.

If the constructor of the class leaks a reference of the object being constructed before it has completed its execution, then there is no happen-before relation with the accesses to the field

  - At the JVM level, the reason is that
    - **`final`** and **`static`** fields cannot remain in cache after the constructor finishes
    - All fields are initialized with default values during class loading
    - writes on **`volatile`** are flushed to main memory (during initialization)

Safe Publication

NOTE: For clarity and simplicity, up to now, we did not take initialization concerns into account. But from now on we will.

- The previous suggestions ensure safe publication because:
  - They established a *happens-before* relation between initialization and access the object's reference (publication)

Defined by us from the JLS explanation and Goetz. You can use it for exercises in this course.

  – *A write to a volatile field happens-before every subsequent read of that field.*
  – *The default initialization (zero, false, or null) of any object happens-before any other actions of a program.*
  – *The initialization of final and static fields happens-before any other actions of a program (after the constructor has finished its execution)*

If the constructor of the class leaks a reference of the object being constructed before it has completed its execution, then there is no happen-before relation with the accesses to the field

  - At the JVM level, the reason is that
  – `final` and `static` fields cannot remain in cache after the constructor finishes
  – All fields are initialized with default values during class loading
  – writes on `volatile` are flushed to main memory (during initialization)

Immutability

- An immutable object is one whose state cannot be changed after initialization
  - You can think of it as a constant
  - The **`final`** keyword in Java prevents modification of fields
    - Remember that variables assigned to an object only hold a reference to the object

- Since immutable objects do not change the state after initialization, data races can only occur during initialization

- An immutable class is one whose instances are immutable objects

Immutability

- To ensure thread-safety of immutable classes, we must ensure that:
  - No fields can be modified after publication
  - Objects are safely published
  - Access to the object's state does not escape

Immutability

- To ensure thread-safety of immutable classes, we must ensure that:
  - No fields can be modified after publication
  - Objects are safely published
  - Access to the object’s state does not escape

```
public final class ThreeStooges {
  private final Set<String> stooges = new HashSet<String>();

  public ThreeStooges () {
    stooges.add("Moe");
    stooges.add("Larry");
    stooges.add("Curly");
  }

  public Boolean isStooge(String name) {
    return stooges.contains(name)
  }
}
```

Goetz p. 47

Immutability

- To ensure thread-safety of immutable classes, we must ensure that:
  - No fields can be modified after publication
  - Objects are safely published
  - Access to the object's state does not escape

```
public final class ThreeStooges {
  private final Set<String> stooges = new HashSet<String>();

  public ThreeStooges () {
    stooges.add("Moe");
    stooges.add("Larry");
    stooges.add("Curly");
  }

  public Boolean isStooge(String name) {
    return stooges.contains(name)
  }
}
```

Why is this class thread-safe?
(tip: there are 3 main reasons)

Goetz p. 47

Mutual exclusion

- Whenever shared mutable state is accessed by several threads, we must ensure mutual exclusion

Thread-safe classes

- To analyse thread-safe in a class, we must identify/consider:
  - Identify the class state
  - Make sure that mutable class state does not escape
  - Ensure safe publication
  - Whenever possible define class state as immutable
  - If class state must be mutable, ensure mutual exclusion

Interesting section (4.5) on documenting synchronization in Goetz. Unfortunately, not widespread.

Thread-safe classes

Theorem: *These properties are sufficient to ensure the thread-safety of a class*—defined as data-race freedom for any pair of field accesses in any concurrent execution of method calls and field accesses

- To analyse thread-safe in a class, we must identify/consider:
  - Identify the class state
  - Make sure that mutable class state does not escape
  - Ensure safe publication
  - Whenever possible define class state as immutable
  - If class state must be mutable, ensure mutual exclusion

Interesting section (4.5) on documenting synchronization in Goetz. Unfortunately, not widespread.

Instance confinement

- *Instance confinement* refers to encapsulating access to a thread-unsafe object into a thread-safe class

Instance confinement

- *Instance confinement* refers to encapsulating access to a thread-unsafe object into a thread-safe class

```
public class PersonSet {
  private final Set<Person> mySet = new HashSet<Person>();

  public synchronized void addPerson (Person p) {
    mySet.add(p);
  }

  public synchronized boolean contains(Person p) {
    return mySet.contains(p);
  }
}
```

Goetz p. 59

Synchronized collections

- Java’s standard library provides a method to convert ordinary collections in to “synchronized” collections
  - `synchronizedCollection(Collection<T> c)`, `synchronizedList(List<T> l)`, `synchronizedSet(Set<T> s)`, …, `synchronizedXXX(XXX<T> x)` with `XXX` a Java collection.
  - Internally, these methods turn all the methods in the collection into synchronized
    – That is, they use the instance lock

Synchronized collections

- Java's standard library provides a method to convert ordinary collections in to "synchronized" collections
  - **`synchronizedCollection(Collection<T> c)`**, **`synchronizedList(List<T> l)`**, **`synchronizedSet(Set<T> s)`**, …, **`synchronizedXXX(XXX<T> x)`** with **`XXX`** a Java collection.
  - Internally, these methods turn all the methods in the collection into synchronized
    - That is, they use the instance lock

Let's look at the Javadoc
(https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#synchronizedList-java.util.List-)

p only accesses thread-safe <u>classes</u> ⇏ p is a thread-safe <u>program</u>

p only accesses thread-safe <u>classes</u> ⇏ p is a thread-safe <u>program</u>


```
List<Integer> l = new ArrayList<Integer>();
List<Integer> lSync = Collections.synchronizedList(l);

…

new Thread(() -> { addIfAbsent(lSync,1); }).start();
new Thread(() -> { addIfAbsent(lSync,1); }).start();

…

public void addIfAbsent(List l, Integer e) {
  if (!l.contains(e))
    l.add(e);
}
```

p only accesses thread-safe <u>classes</u> ⇏ p is a thread-safe <u>program</u>

Is this <u>program</u> thread-safe?

```
List<Integer> l = new ArrayList<Integer>();
List<Integer> lSync = Collections.synchronizedList(l);

…

new Thread(() -> { addIfAbsent(lSync,1); }).start();
new Thread(() -> { addIfAbsent(lSync,1); }).start();

…

public void addIfAbsent(List l, Integer e) {
  if (!l.contains(e))
    l.add(e);
}
```

Extending thread-safe classes

- Thread-safe classes may be extended to include compound actions
  - Intuitively, compound actions can be seen multiple method calls or field accesses within a critical section
  - A common examples are: *check-and-set,* iteration, navigation (*contains*)

```
public void addIfAbsent(List l, Integer e) {
  synchronized (l) {
    if (!l.contains(e))
      l.add(e);
  }
}
```

Thread uses the intrinsic lock of a synchronized collection

```
class ThreadSafeList {
  …
  public void synchronized addIfAbsent(T e) {
    if (!l.contains(e))
        l.add(e);
  }
  …
}
```

Thread-safe class is extended with a custom method to perform the action

Other synchronization primitives (synchronizers)

Semaphores

- Semaphores are synchronization primitives that allow at most *c* number of threads in the critical section where *c* is called the *capacity*
  - First introduced by Dijkstra

- A semaphore consists of:
  - An integer capacity (*c*), permits in Java
    - Initial number of threads allowed in the critical section
  - A method `acquire()`
    - Checks if *c* > 0, if so, it decrements capacity by one (c--) and allows the calling thread to make progress, otherwise it blocks the thread
    - It is a blocking call
  - A method `release()`
    - It checks whether there are waiting threads, if so, it wakes up one of them, otherwise it increases the capacity by one (c++)
    - It is non-blocking

Semaphores

- Semaphores are synchronization primitives that allow at most *c* number of threads in the critical section where *c* is called the *capacity*
  - First introduced by Dijkstra
- A semaphore consists of:
  - An integer capacity (*c*), [permits in Java]
    - Initial number of threads allowed in the critical section
  - A method **`acquire()`**
    - Checks if $c > 0$, if so, it decrements capacity by one (c--) and allows the calling thread to make progress, otherwise it blocks the thread
    - It is a blocking call
  - A method **`release()`**
    - It checks whether there are waiting threads, if so, it wakes up one of them, otherwise it increases the capacity by one (c++)
    - It is non-blocking

Semaphores (1968) appear before Monitors (1972)

Semaphores

- Semaphores are synchronization primitives that allow at most *c* number of threads in the critical section where *c* is called the

Synchronization primitives that only allow one thread in the critical section are called ***mutex*** (which is short for mutual exclusion)

Semaphores (1968) appear before Monitors (1972)

- An integer capacity (*c*), [permits in Java]
  – Initial number of threads allowed in the critical section
- A method **`acquire()`**
  – Checks if $c > 0$, if so, it decrements capacity by one (c--) and allows the calling thread to make progress, otherwise it blocks the thread
  – It is a blocking call
- A method **`release()`**
  – It checks whether there are waiting threads, if so, it wakes up one of them, otherwise it increases the capacity by one (c++)
  – It is non-blocking

Semaphores | Intuition

- You can think of a semaphore as a “bouncer” to enter a critical section or to be allowed to used a shared resource

Semaphores | Intuition

- You can think of a semaphore as a “bouncer” to enter a critical section or to be allowed to used a shared resource

Semaphores | Intuition

- You can think of a semaphore as a “bouncer” to enter a critical section or to be allowed to used a shared resource

Semaphores | Intuition

- You can think of a semaphore as a “bouncer” to enter a critical section or to be allowed to used a shared resource

Semaphores | Intuition

- You can think of a semaphore as a “bouncer” to enter a critical section or to be allowed to used a shared resource

Semaphores | Example

- Semaphores are typically used to control the number of threads accessing a resource (here we fix a maximum 5 readers and writers)

```
ReadWriteMonitor m = new ReadWriteMonitor();
Semaphore semReaders = new Semaphore(5,true);
Semaphore semWriters = new Semaphore(5,true);
for (int i = 0; i < 10; i++) {
    // start a reader
    new Thread(() -> {
          m.readLock();
          semReaders.acquire();
          // read
          semReaders.release();
          m.readUnlock();
    }).start();

    // start a writer
    new Thread(() -> {
          m.writeLock();
          semWriters.acquire();
          // write
          semWriters.acquire();
          m.writeUnlock();
    }).start();
}
```

Java semaphores have a fair flag so that their entry queue prioritizes the longest waiting thread

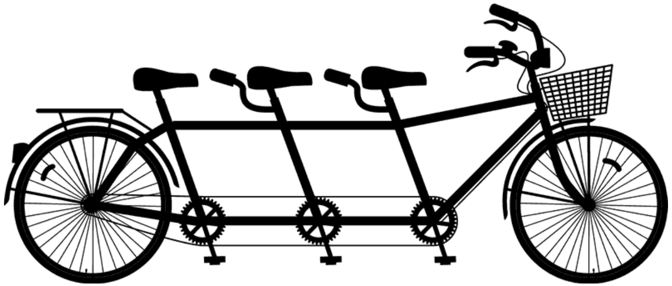
See **`ReadersWritersSemaphore.java`**

Barriers

- *Barriers* are synchronization primitives used to wait until several thread reach some point in their computation

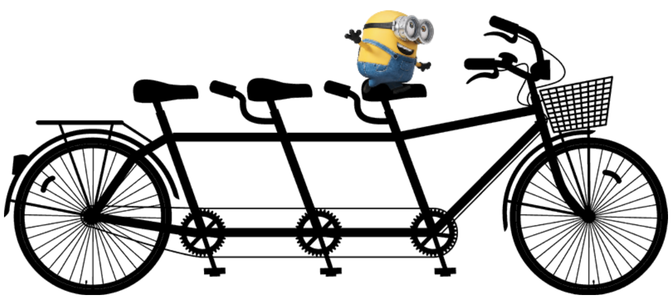
Barriers | Intuition

- *Barriers* are synchronization primitives used to wait until several thread reach some point in their computation



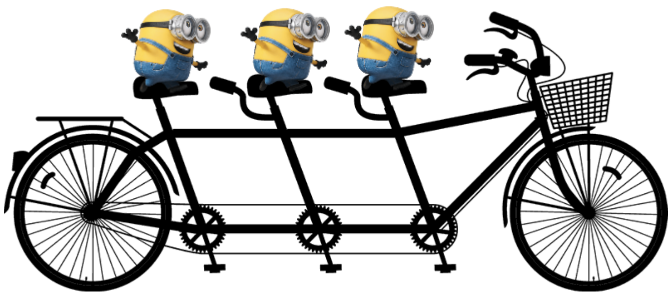
Barriers | Intuition

- *Barriers* are synchronization primitives used to wait until several thread reach some point in their computation



Barriers | Intuition

- *Barriers* are synchronization primitives used to wait until several thread reach some point in their computation



Barriers

- *Barriers* are synchronization primitives used to wait until several thread reach some point in their computation

- Barriers consists of
  - A number *parties* to wait for
  - A method **`await()`**
    – If the number of waiting threads is less than *parties*, then the calling thread blocks, otherwise all waiting threads wake up and the calling thread is allowed to make progress

- Java includes the class **`CyclicBarrier`**
  - After *parties* called **`await()`**, then the state is reset and the barrier behaves as initially

Barrier Example | Parallel initialization

- Several threads are used to initialize an array (each a different position), the barrier is used for threads to know when the initialization is finished
  - This example is a bit artificial, but it illustrates the use of barriers.

```
int parties        = 10;
CyclicBarrier cb   = new CyclicBarrier(parties);
int[] shared_array = new int[parties];
…
for (int i = 0; i < parties; i++) {
  new SetterClass(i).start();
}
…
public class SetterClass extends Thread {
  int index;
  public SetterClass(int index) {this.index = index;}

  public void run() {
    shared_array[index] = index+1;
    cb.await();
    // After this point the array is initialized and it is safe to read it
  }
}
```

Barrier Example | Parallel initialization

- Several threads are used to initialize an array (each a different position), the barrier is used for threads to know when the initialization is finished
  - This example is a bit artificial, but it illustrates the use of barriers.

See `BarrierExample.java`

```
int parties        = 10;
CyclicBarrier cb   = new CyclicBarrier(parties);
int[] shared_array = new int[parties];
…
for (int i = 0; i < parties; i++) {
  new SetterClass(i).start();
}
…
public class SetterClass extends Thread {
  int index;
  public SetterClass(int index) {this.index = index;}

  public void run() {
    shared_array[index] = index+1;
    cb.await();
    // After this point the array is initialized and it is safe to read it
  }
}
```

Producer-consumer problem

- Consider a shared data structure of fixed size from which threads may add and remove elements

- *Producer* threads may add elements to the structure as long as it is not full
  - If the structure is full and a producer tries to add an element, it must block until there an element is removed

- *Consumer* threads remove elements to the structure as long as it is not empty
  - If the structure is empty and a consumer tries to remove an element, then it must block until an element is added

- A good solution to the problem must be deadlock free and (possibly) starvation free

Producer-consumer problem | Intuition

- Perhaps more intuitive example

Producer-consumer problem

- The producer-consumer problem appears in many multi-threaded situations
  - Handling access to a shared bounded data structure
  - Controlling access to limited computational resources
    – E.g., thread pools
  - Asynchronous I/O operations
    – External devices may act as producers providing data to the system (keyboard, mouse, etc…), or consumer obtaining tasks to perform (IoT devices)

Agenda

- Definitions of thread-safety
  - Classes
  - Programs
- Safe publication
- Immutability
- Instance confinement
- Synchronization primitives (synchronizers)
  - Semaphores
  - Barriers
- Producer-consumer problem