Unit 2.- Task synchronization

Concurrency and Distributed Systems

Teaching Unit Objectives

- Review the concept of concurrent programming
  - Identify the problems related to concurrent programming
  - Understand the cooperation mechanisms needed to implement a concurrent application.
    - Mechanisms for communication and synchronization
  - Identify the parts of the code that can cause problems (critical sections) and how to protect them
- Know a programming language that supports concurrent programming: Concurrency in Java
  - Introduce the Java basic mechanisms for supporting concurrent programming

Content

- **Task synchronization**
  - Communication mechanisms
    - Shared memory
    - Message exchange
  - Types of synchronization
    - Mutual Exclusion
      - Critical Section
      - Locks
    - Conditional Synchronization

Cooperation among threads: requirements

- Concurrency implies parallelism and cooperation among threads
- **Cooperation** requires:
  - a) communication (information interchange)
  - b) synchronization (setting certain order in specific cases)

Communication Mechanisms

- **Communication Mechanisms:**
  - a) **Shared Memory:** threads share space in memory (variables or shared objects)
  - b) **Message Exchange** (or **Message Passing**): sender/receiver are potentially in disjoint spaces of memory

- We focus here on **shared memory (option 'a')**
  - It requires synchronization mechanisms to coordinate tasks
    - Employed in languages such as Java and C#, and libraries such as *pthreads*
    - We will see the classical concurrency model

- We will see *option b* in Distributed Systems
  - The current trend is to implement distributed applications

Shared memory: problems that can appear

- **Race Conditions:** inconsistent modification of shared memory
  - Errors may occur when multiple threads access shared data.
  - Inconsistent views of the shared memory may result in errors.

Synchronization prevents these problems

Problems with the use of shared memory → Example 1

- Example: **c++** can be decomposed into three steps:
  1. Retrieve the current value of **c**
  2. Increment the retrieved value by 1
  3. Store the incremented value back in **c**

  (the same for **c--** but decrementing)

- Important! Even simple statements can be translated to multiple steps by the Java Virtual Machine

```
public class Counter {
        protected long count = 0;
        public void add(long x) {
                count += x;
        }

        public long getCount() {
                return count;
        }
}
...
Counter c= new Counter();
// initialized to 0
```

Example 1

- EXAMPLE: we have an object *c* of type *Counter* and two threads A and B
  - c initially has value 0
  - A executes *c.add(3)*
  - B executes *c.add(2)* concurrently
- The final value of *c* should be 5

```
public class Counter {
        protected long count = 0;
        public void add(long x) {
                count += x;
        }

        public long getCount() {
                return count;
        }
}
...
Counter c= new Counter();
// initialized to 0
```

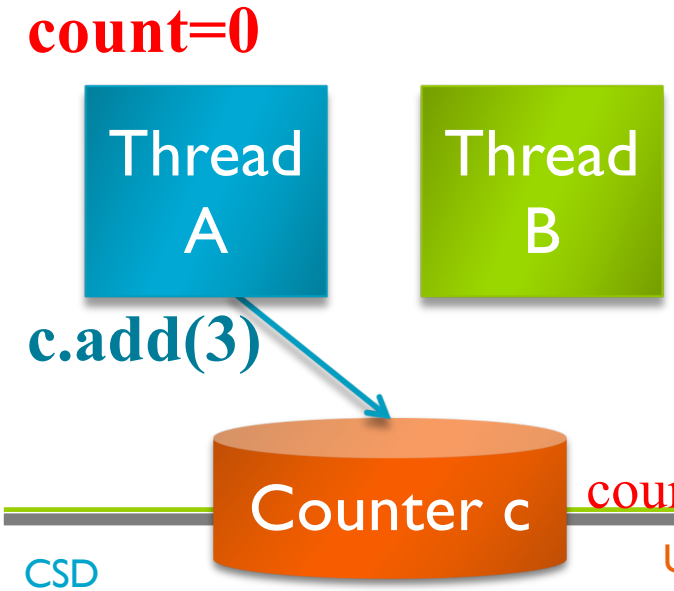
- But the final value depends of the **exact interleaving** (scheduling) when running A and B
  - There are **thread interferences** when two operations, running in different threads, but acting on the same data, ***interleave***.

Example 1

- Interleaving example:

Thread A executes c.add(3)

A loads c.count (local copy=0)




```
public class Counter {
      protected long count = 0;
      public void add(long x) {
            count += x;
      }

      public long getCount() {
            return count;
      }
}
...
Counter c= new Counter();  //initialized to 0
```

Example 1

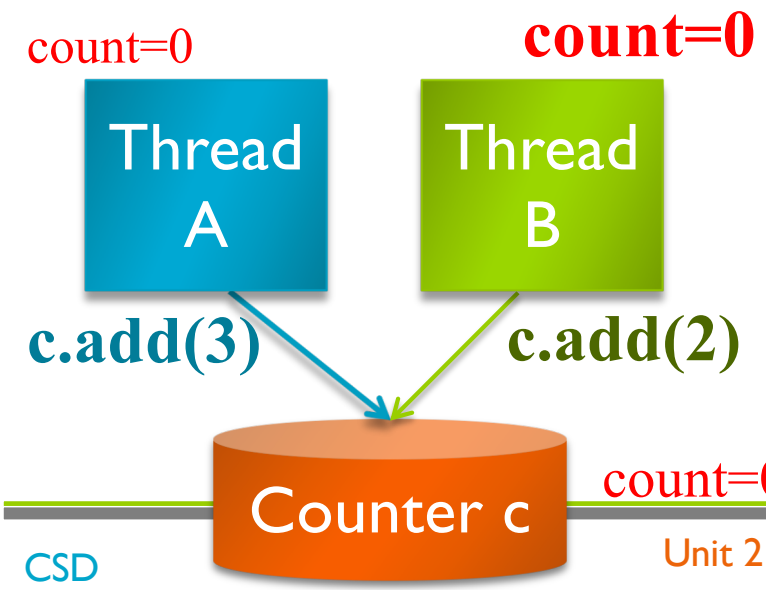
- Interleaving example:

Thread A executes c.add(3)

A loads c.count (local copy=0)

Thread B executes c.add(2)

B loads c.count (local copy=0)




```
public class Counter {
        protected long count = 0;
        public void add(long x) {
                count += x;
        }

        public long getCount() {
                return count;
        }
}
...
Counter c= new Counter();
```

Example 1

- Interleaving example:

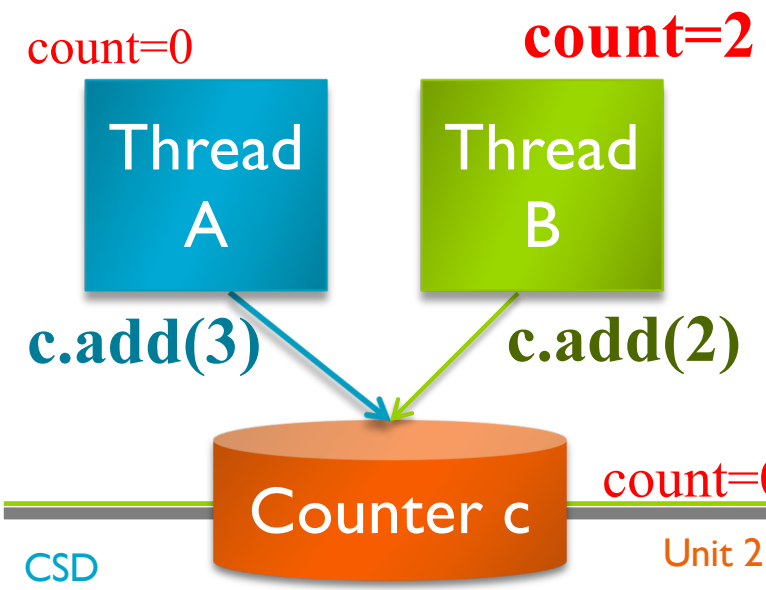
Thread A executes c.add(3)

A loads c.count (local copy=0)

Thread B executes c.add(2)

B loads c.count (local copy=0)

B adds 2 (local copy=2)




```
public class Counter {
      protected long count = 0;
      public void add(long x) {
            count += x;
      }

      public long getCount() {
            return count;
      }
}
...
Counter c= new Counter();
```

Example 1

- Interleaving example:

Thread A executes c.add(3)

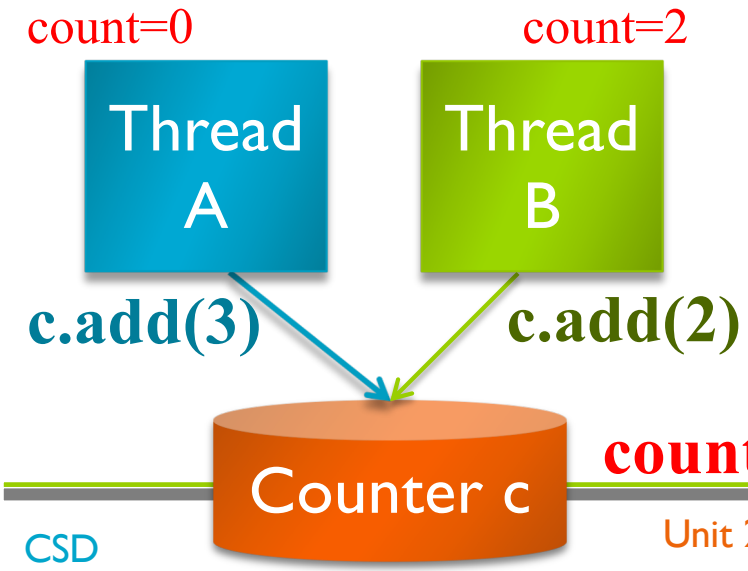
A loads c.count (local copy=0)

Thread B executes c.add(2)

B loads c.count (local copy=0)

B adds 2 (local copy=2)

B updates the heap (c.count=2)




```
public class Counter {
      protected long count = 0;
      public void add(long x) {
            count += x;
      }

      public long getCount() {
            return count;
      }
}
...
Counter c= new Counter();
```

Example 1

- Interleaving example:

Thread A executes c.add(3)

A loads c.count (local copy=0)

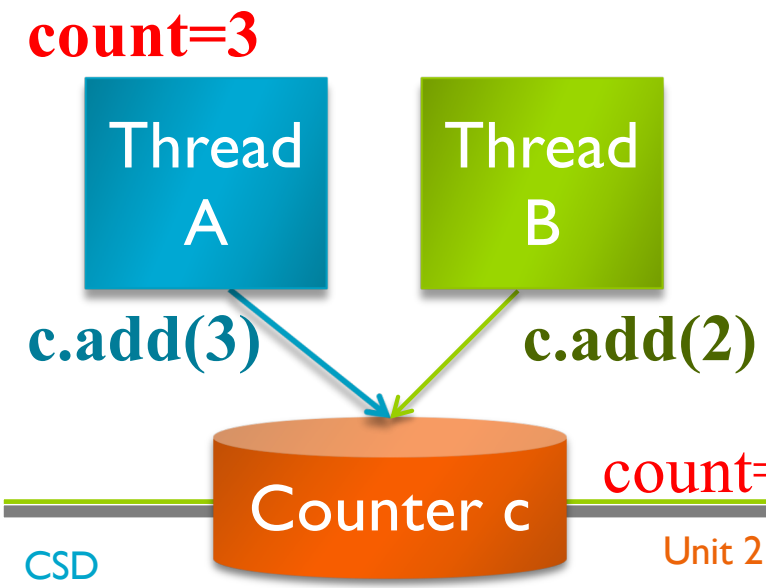
A adds 3 (local copy=3)

Thread B executes c.add(2)

B loads c.count (local copy=0)

B adds 2 (local copy=2)

B updates the heap (c.count=2)




```
public class Counter {
      protected long count = 0;
      public void add(long x) {
            count += x;
      }

      public long getCount() {
            return count;
      }
}
...
Counter c= new Counter();
```

Example 1

- Interleaving example:

Thread A executes c.add(3))

A loads c.count (local copy=0)

A adds 3 (local copy=3)

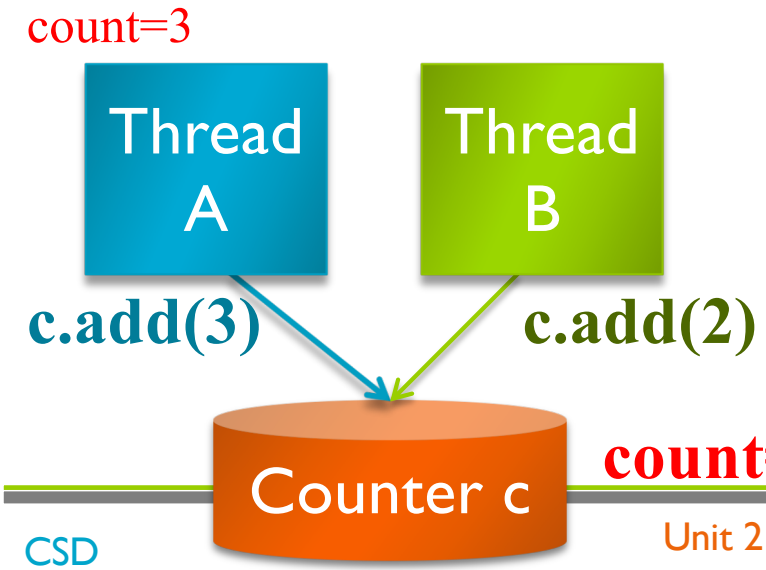
A updates the heap (c.count=3)

Thread B executes c.add(2)

B loads c.count (local copy=0)

B adds 2 (local copy=2)

B updates the heap (c.count=2)




```
public class Counter {
      protected long count = 0;
      public void add(long x) {
            count += x;
      }

      public long getCount() {
            return count;
      }
}
...
Counter c= new Counter();
```

Example 1

- Interleaving example:

Thread A executes c.add(3)

A loads c.count (local copy=0)

A adds 3 (local copy=3)

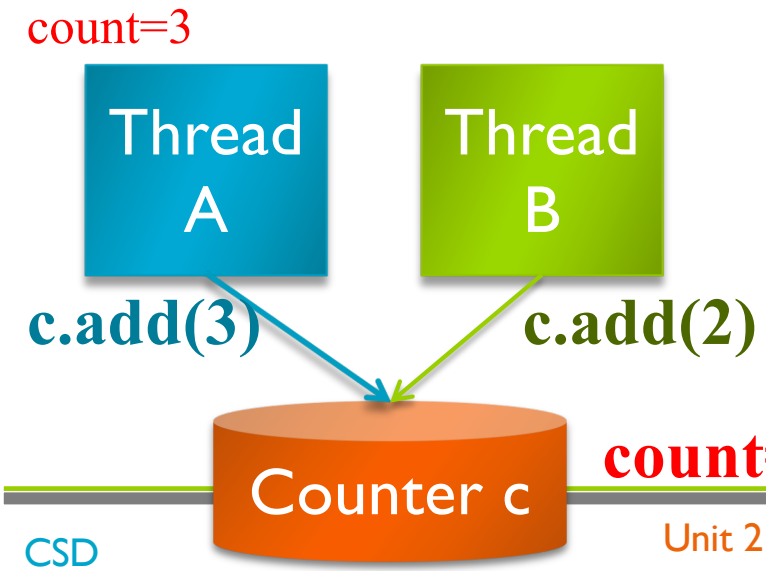
A updates the heap (c.count=3)

Thread B executes c.add(2)

B loads c.count (local copy=0)

B adds 2 (local copy=2)

B updates the heap (c.count=2)



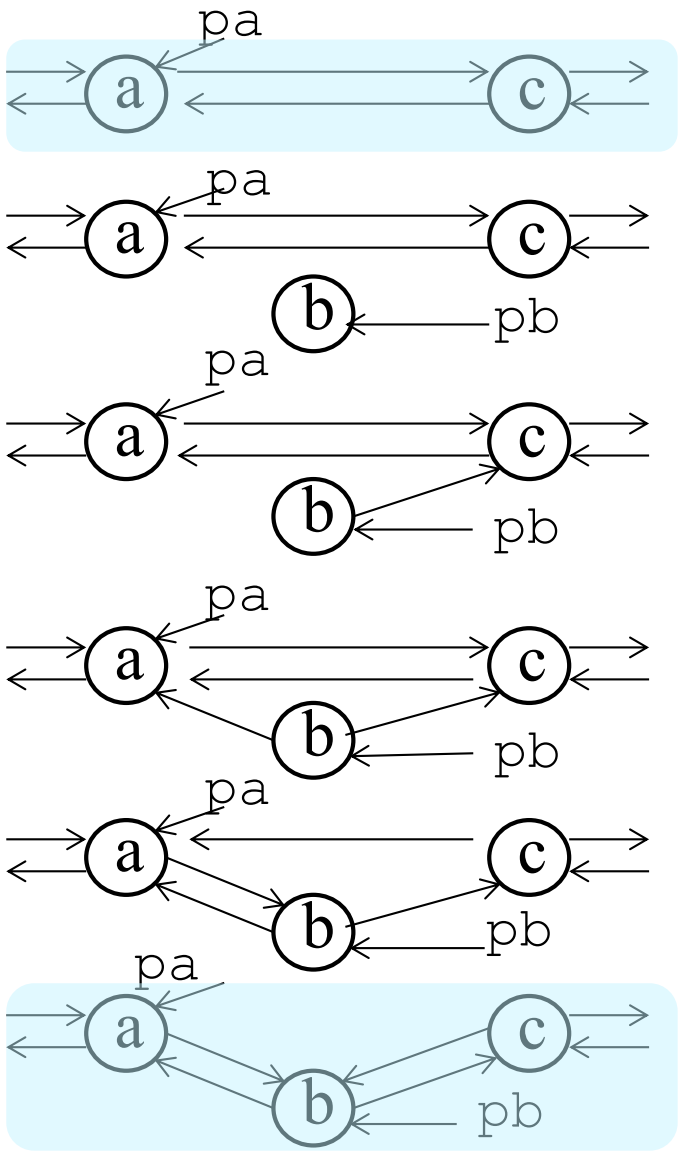
Final value: 3
We expected 5 !!!

There has been a **race condition**

Example 2 - Inserts **b** between **a** and **c** in a double-linked list

As atomic action

```
Node pa=.. //insert after pa
Node pb = new Node(b);
pb.next = pa.next;
pb.prev = pa;
pa.next = pb;
(pb.next).prev = pb;
```



Content

- **Task synchronization**
  - Communication mechanisms
    - Shared memory
    - Message exchange
  - Types of synchronization
    - Mutual Exclusion
      - Critical Section
      - Locks
    - Conditional Synchronization

Goals of synchronization

Goals of Synchronization Mechanisms

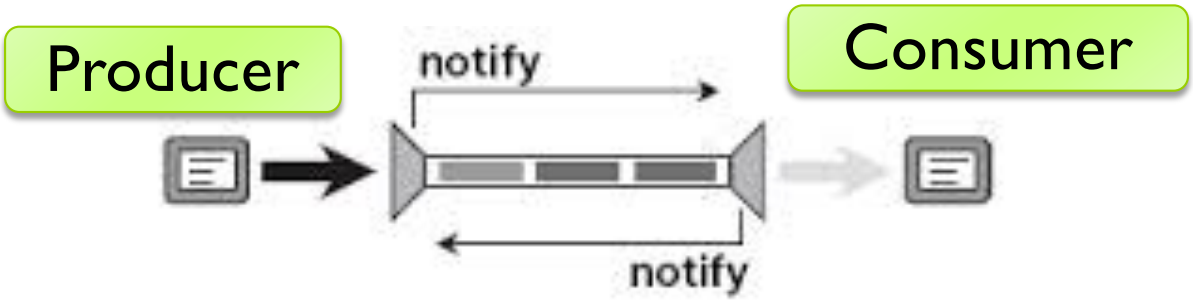
- Ensure execution order of sentences
- Respect restrictions on code execution

- **Types of synchronization:**
  - **Mutual Exclusion**
  - **Conditional Synchronization**

Types of Synchronization

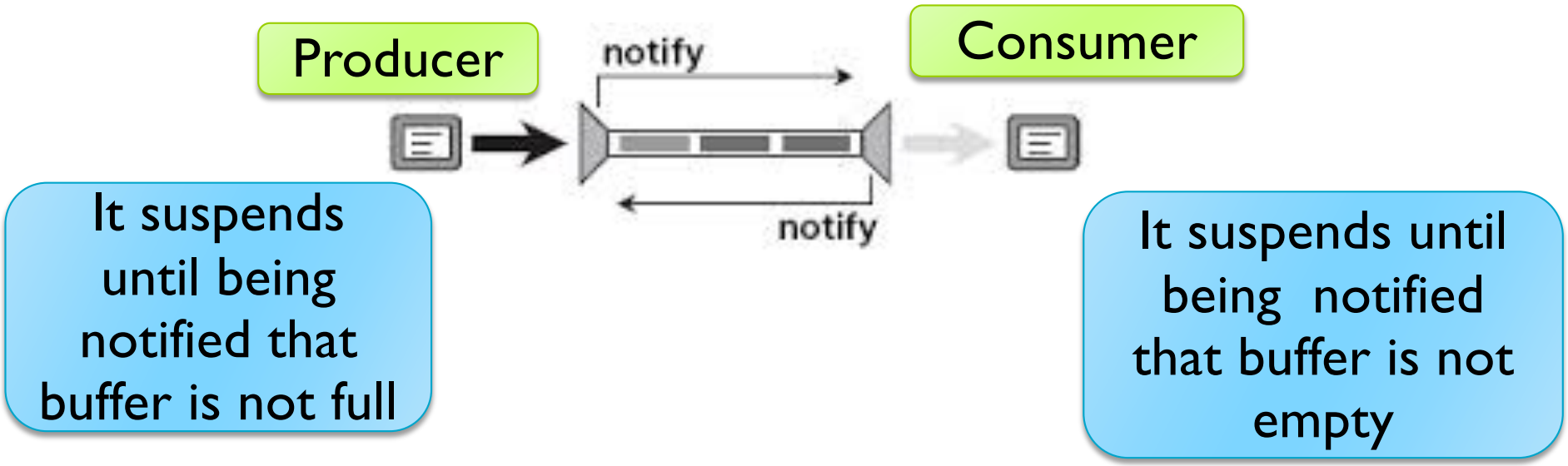
- **Mutual exclusion**
  - A section of code is executed in ***mutual exclusion*** when **only one thread** can run this section at any time.
  - Needed to avoid interferences among threads.
  - Required for shared variables/objects

Access in **mutual exclusion** to the buffer



Conditional Synchronization

- A thread must delay its execution until a specific condition holds.
- The condition often depends on the value of some shared variables.
- Other threads, when modifying these variables, will get the condition to be fulfilled, thus reactivating the suspended threads



Critical Section → What is it?

Critical Section (CS) : fragment of code that can cause race conditions

- Access to *local variables* → *it is not critical*
- *Access to immutable shared objects* → not critical
- Access to ***shared variables/objects*** → **critical**

Critical Section (CS): fragment that accesses to variables or objects shared by more than one thread

Critical Section→ How can you protect it?

...
Rest of code
....

input protocol

Critical Section

output protocol

...
Rest of code
(Non-critical Section)
....

Input/Output protocols that ensure (all together):

- **Mutual Exclusion**
- **Progress:** every service requested is completed at some point
- **Bounded Waiting:** there exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has **made a request to enter its critical section** and before that request is granted.

Thread-safe code: code that can be run concurrently by different threads safely (i.e. without race-condition problems)

Critical Section→ How can you protect it?

- Solution strategies depending on the level:

Hardware
- Disabling interruptions
- Only for encoding the OS kernel

Operating System
- System calls for using semaphores, mutexes, events, conditions

Programming Language
- *Locks, conditions, monitors, critical sections, protected objects*
- Examples: Concurrent Pascal, Modula-2, Modula-3, Ada, Java, C#

Locks → What is it? How does it work?

- A **lock** is an object with two states (*open/closed*) and two operations (*to open/to close*)
- When creating the lock, it will be initially open.
- Usage of *lock*:

close lock

CS

open lock

Non-critical section

Locks → How does it work?

- **Close** lock:
  - If open → it closes the lock
  - If closed by another thread → the current thread waits
  - If closed by itself → no effect

- **Open** lock:
  - If already open → no effect
  - If closed by another thread → no effect
  - If closed by itself → the lock opens
    - If anyone is waiting, it gets closed by one of the threads that are waiting (who will keep on running)
      - The choice depends on the implementation, but it normally will meet equity

Locks → How does it work?

- Using *locks* we convert a critical section into an ***atomic action***
  - **Atomicity**: only one thread can execute the protected code at any given time
  - Ensures reliable update of shared variables or shared objects
    - Free from race conditions or corruption of states
    - Only the steady states are visible
    - Ensures that every thread always access to the most recent value of each shared variable or shared object.

close lock

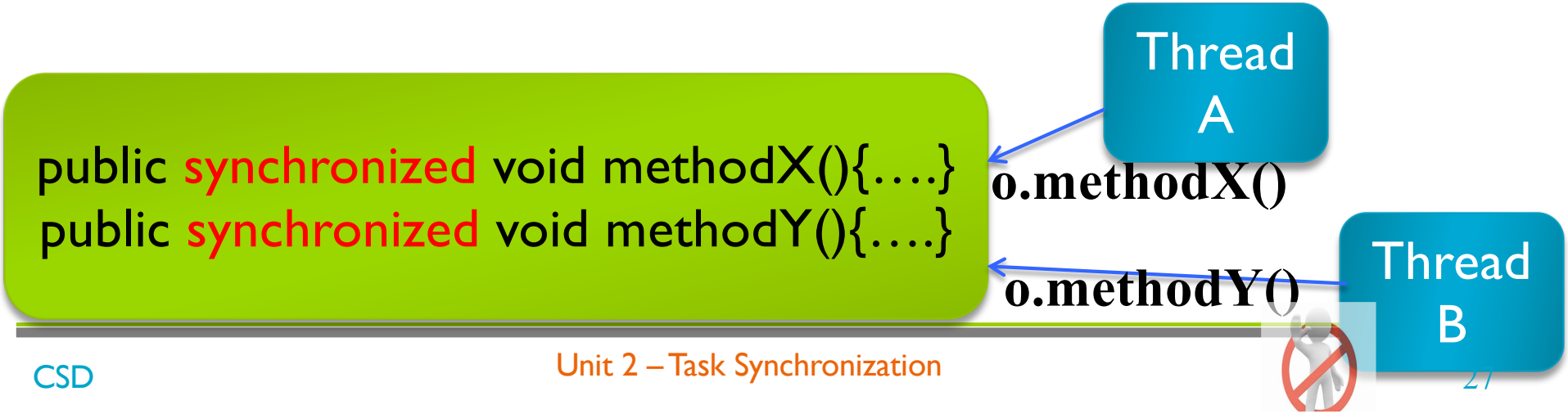
CS

open lock

Non-critical Section

Locks in **Java** → How can you use them?

- Every object has an **implicit** associated *lock*
  - You must label with ***synchronized*** all methods that are part of the critical section
  - When Java enters a method labeled as '***synchronized***', the lock gets closed, and it opens when leaving the method.
    - When a thread releases an intrinsic lock, a *happens-before* relationship is established with that action and any subsequent acquisition of the same lock.
  - For the same object, all its methods labeled with *synchronized* are executed in **mutual exclusion among them**



Locks in **Java** → How can you implement them?

- **Implementation**→ add the ***synchronized*** label to every method that:
  - Modifies an attribute which another thread has to read later
  - Reads an attribute updated by another thread

Locks in **Java** → How can you implement them?

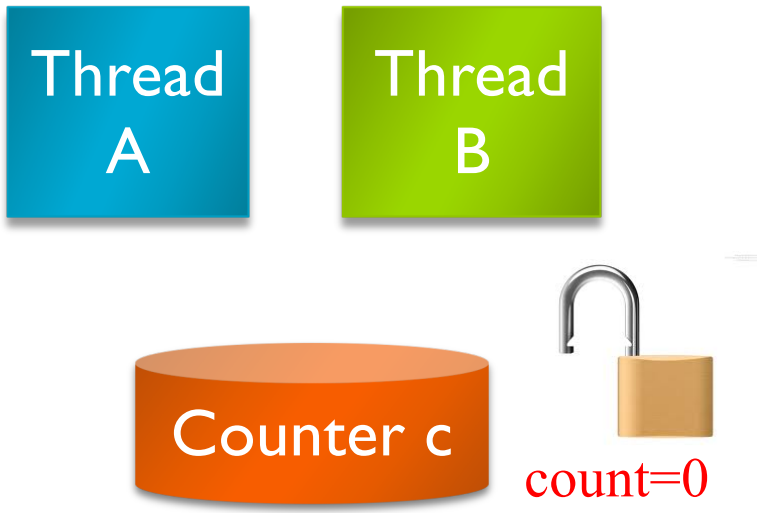
```
public class Counter {
        private long count = 0;
        public synchronized void add(long x) {
                count += x;
        }

        public synchronized long getCount() {
                return count;
        }
}
...
Counter c= new Counter(); // initialized to 0
```

Example:
Counter Version
without race
conditions

Locks in **Java** → Let's see an example

- Interleaving example:

Initially: implicit lock of **Counter** is open




```
public class Counter {
        protected long count = 0;
        public synchronized void add(long x) {
                count += x;
        }

        public synchronized  long getCount() {
                return count;
        }
}
...
Counter c= new Counter();
```

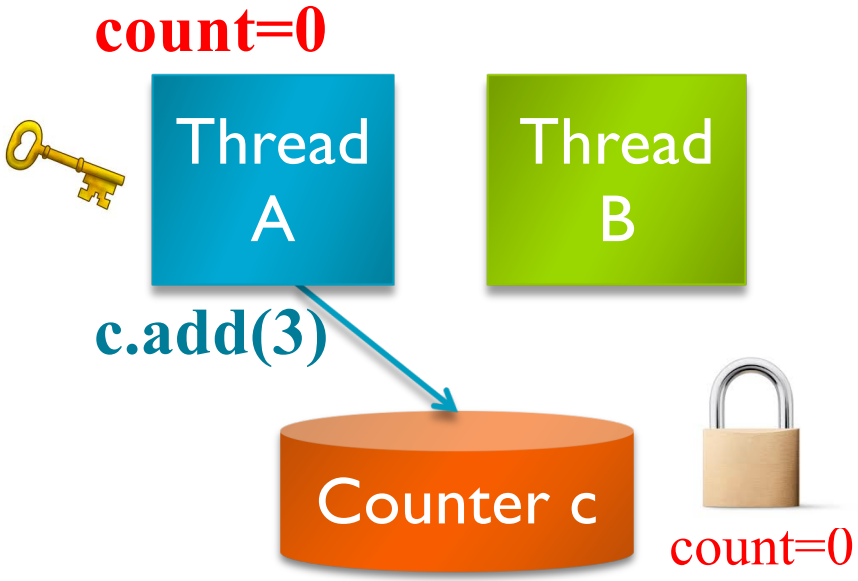
Locks in **Java** → Let's see an example

- Interleaving example:

Thread A executes c.add(3)

A loads c.count (local copy=0)

And Java closes the implicit **lock**




```
public class Counter {
        protected long count = 0;
        public synchronized void add(long x) {
                count += x;
        }

        public synchronized  long getCount() {
                return count;
        }
}
...
Counter c= new Counter();
```

Locks in **Java** → Let's see an example

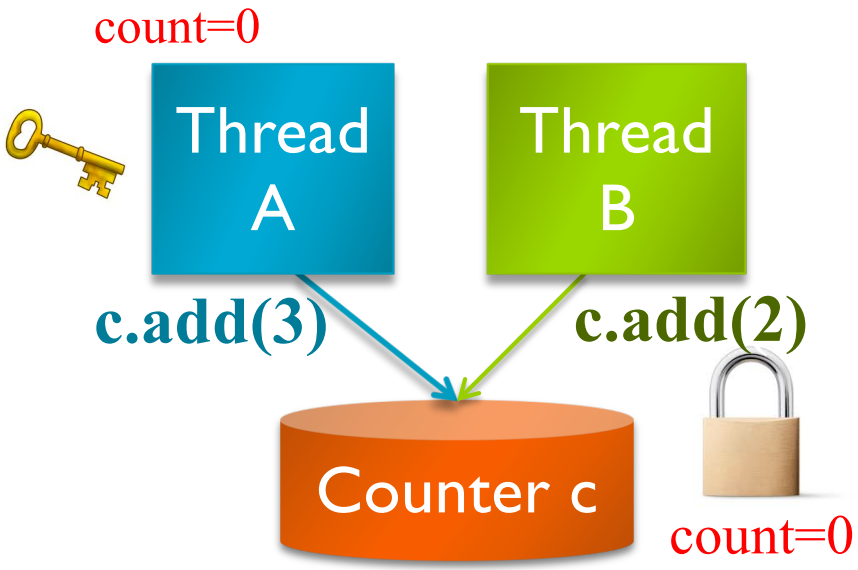
- Interleaving example:

Thread A executes c.add(3)

A loads c.count (local copy=0)

Thread B executes c.add(2)

B has to wait (lock already closed by another thread)




```
public class Counter {
        protected long count = 0;
        public synchronized void add(long x) {
                count += x;
        }

        public synchronized  long getCount() {
                return count;
        }
}
...
Counter c= new Counter();
```

Locks in **Java** → Let's see an example

- Interleaving example:

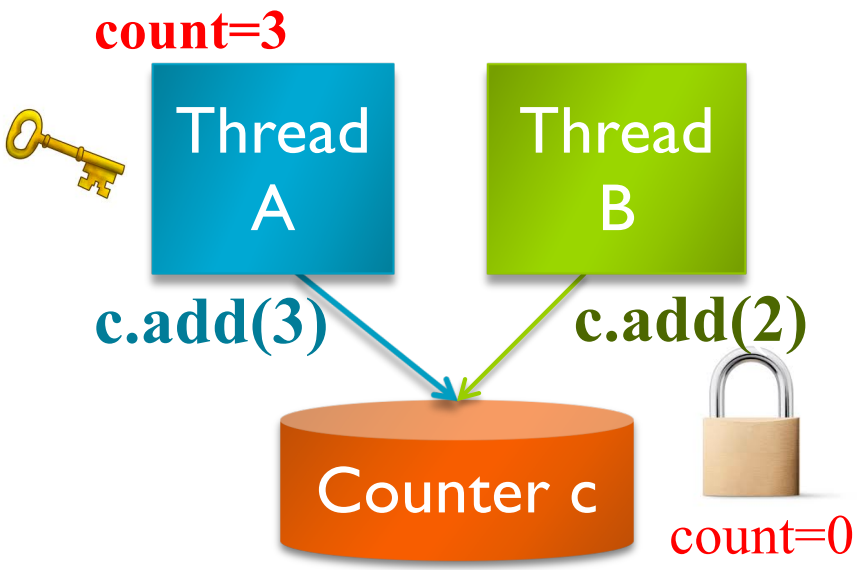
Thread A executes c.add(3)

A loads c.count (local copy=0)

A adds 3 (local copy=3)

Thread B executes c.add(2)

B has to wait (lock already closed by another thread)




```
public class Counter {
      protected long count = 0;
      public synchronized void add(long x) {
            count += x;
      }

      public synchronized  long getCount() {
            return count;
      }
}
...
Counter c= new Counter();
```

Locks in **Java** → Let's see an example

- Interleaving example:

Thread A executes c.add(3)

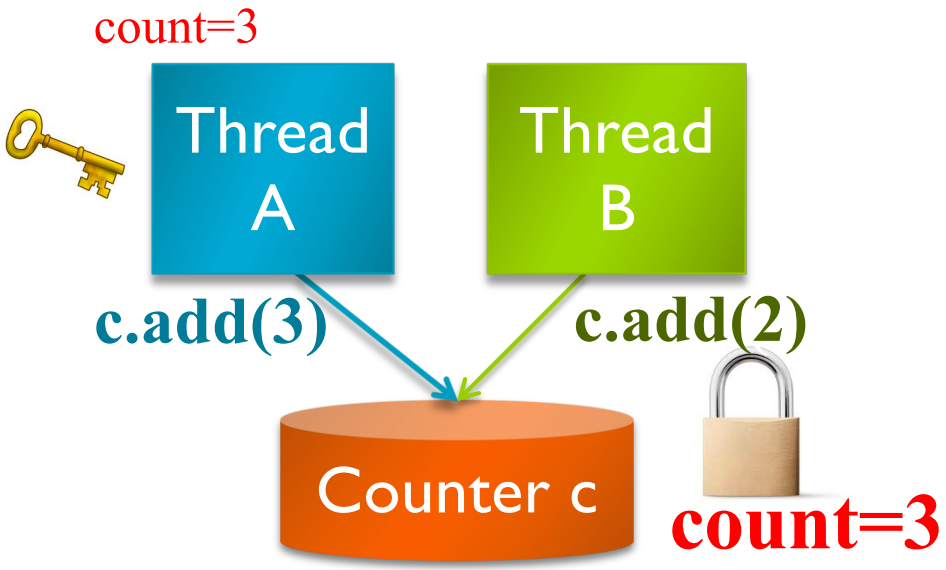
A loads c.count (local copy=0)

A adds 3 (local copy=3)

A updates the heap (c.count=3)

Thread B executes c.add(2)

B has to wait (lock already closed by another thread)




```
public class Counter {
      protected long count = 0;
      public synchronized void add(long x) {
            count += x;
      }

      public synchronized  long getCount() {
            return count;
      }
}
...
Counter c= new Counter();
```

Locks in **Java** → Let's see an example

- Interleaving example:

Thread A executes c.add(3)

A loads c.count (local copy=0)

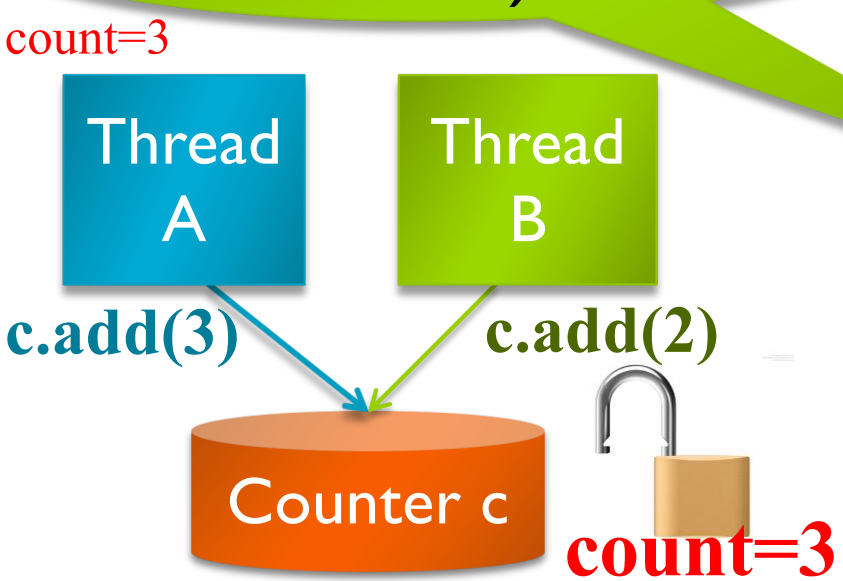
A adds 3 (local copy=3)

A updates the heap (c.count=3)

Thread B executes c.add(2)

B has to wait (lock already closed by another thread)

A leaves the synchronized method (and Java opens the lock)




```
public class Counter {
      protected long count = 0;
      public synchronized void add(long x) {
            count += x;
      }

      public synchronized  long getCount() {
            return count;
      }
}
…
Counter c= new Counter();
```

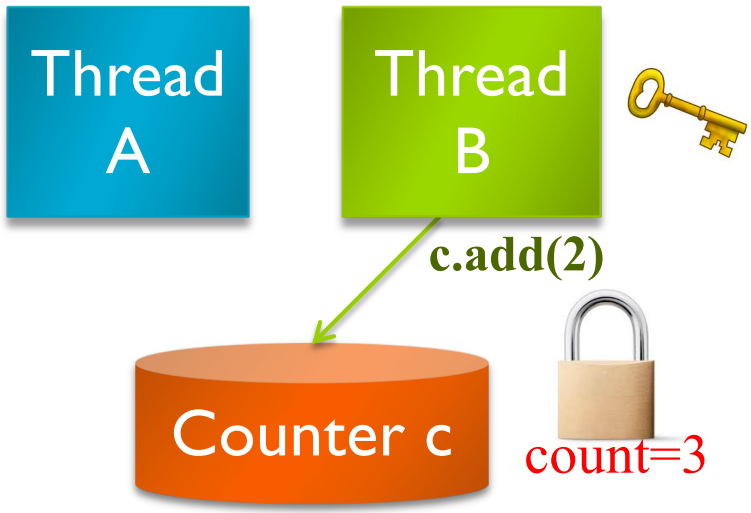
Locks in **Java** → Let's see an example

- Interleaving example:

Thread A executes c.add(3)

Thread B executes c.add(2)

B awakes and Java closes the lock




```
public class Counter {
        protected long count = 0;
        public synchronized void add(long x) {
                count += x;
        }

        public synchronized  long getCount() {
                return count;
        }
}
...
Counter c= new Counter();
```

Locks in **Java** → Let's see an example

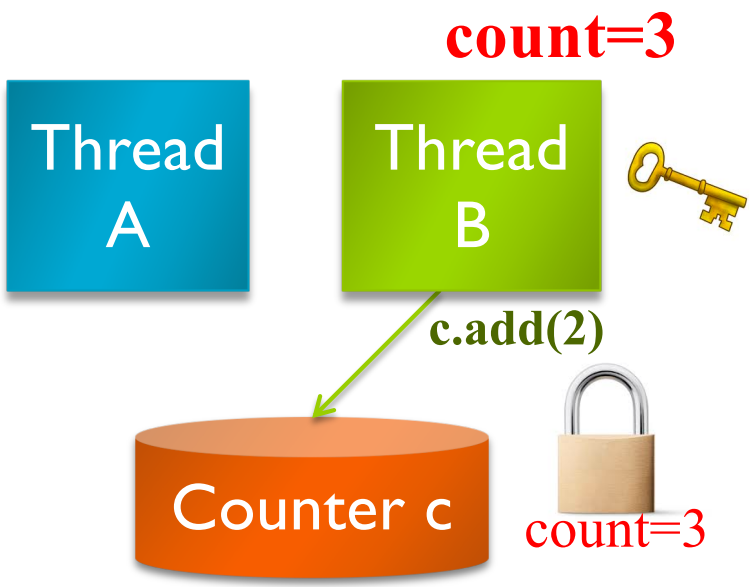
- Interleaving example:

Thread A executes c.add(3)

Thread B executes c.add(2)

B awakes and Java closes the lock

B loads c.count (local copy=3)




```
public class Counter {
        protected long count = 0;
        public synchronized void add(long x) {
                count += x;
        }

        public synchronized  long getCount() {
                return count;
        }
}
...
Counter c= new Counter();
```

Locks in **Java** → Let's see an example

- Interleaving example:

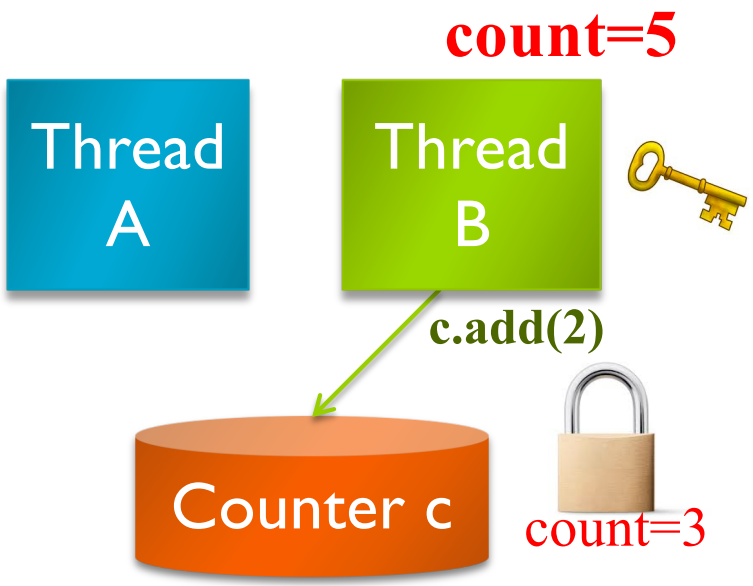
Thread A executes c.add(3)

Thread B executes c.add(2)

B awakes and Java closes the lock

B loads c.count (local copy=3)

B adds 2 (local copy=5)




```
public class Counter {
        protected long count = 0;
        public synchronized void add(long x) {
                count += x;
        }

        public synchronized  long getCount() {
                return count;
        }
}
…
Counter c= new Counter();
```

Locks in **Java** → Let's see an example

- Interleaving example:

Thread A executes c.add(3)

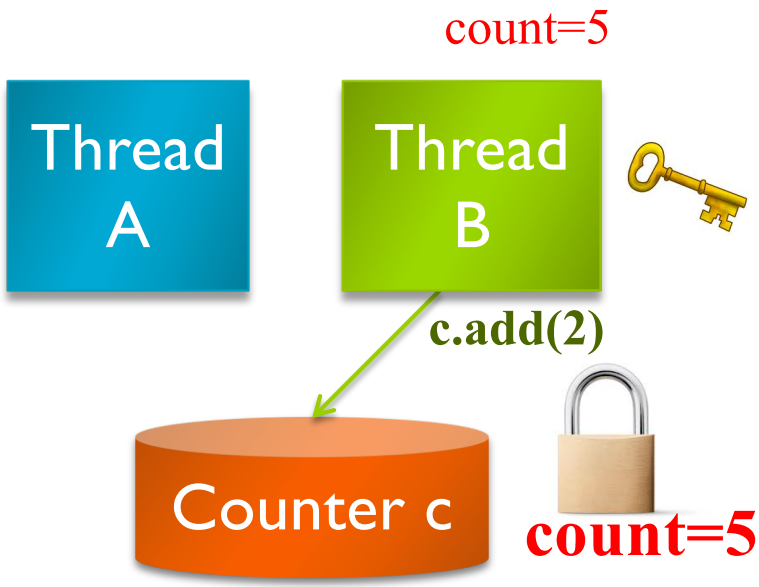
Thread B executes c.add(2)

B awakes and Java closes the lock

B loads c.count (local copy=3)

B adds 2 (local copy=5)

B updates the heap (c.count=5)




```
public class Counter {
      protected long count = 0;
      public synchronized void add(long x) {
            count += x;
      }

      public synchronized  long getCount() {
            return count;
      }
}
...
Counter c= new Counter();
```

Locks in **Java** → Let's see an example

- Interleaving example:

Thread A executes c.add(3)

Thread B executes c.add(2)

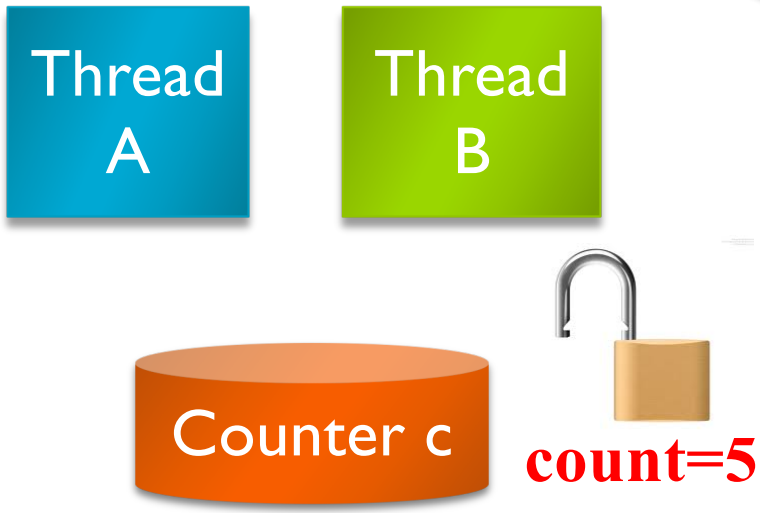
B awakes and Java closes the lock

B loads c.count (local copy=3)

B adds 2 (local copy=5)

B updates the heap (c.count=5)

B leaves the synchronized method (and Java opens the lock)




```
public class Counter {
      protected long count = 0;
      public synchronized void add(long x) {
            count += x;
      }

      public synchronized  long getCount() {
            return count;
      }
}
...
Counter c= new Counter();
```

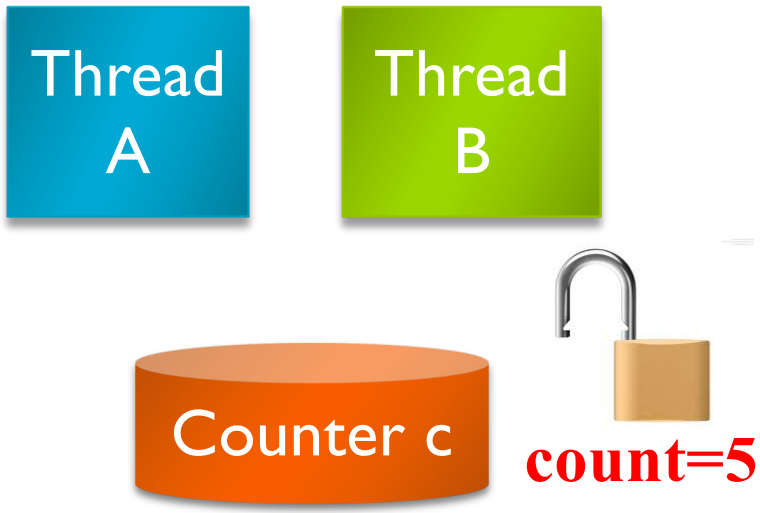
Locks in **Java** → Let's see an example

- Independent of interleaving:

Thread A executes c.add(3) *Thread B executes c.add(2)*

Final value: ALWAYS 5!!

Code free of **race conditions**




```
public class Counter {
        protected long count = 0;
        public synchronized void add(long x) {
                count += x;
        }

        public synchronized  long getCount() {
                return count;
        }
}
...
Counter c= new Counter();
```

Java synchronization

- Java provides two basic synchronization idioms:
  - **Synchronized methods:** using **synchronized** label at method declaration.
  - **Synchronized statements:** specifying the object that provides the intrinsic lock
    - Allows using more than one lock inside the same method.

- Methods *inc1* & *inc2* can interleave
- But updating **c1** is done in mutual exclusion
- Same for **c2**

```
public class MsLunch{
      private long c1 = 0;
      private long c2 = 0;
      private  Object lock1 = new Object();
      private  Object lock2 = new Object();

      public void inc1() {
            synchronized(lock1){ c1++; }
       }

      public void inc2() {
            synchronized(lock2){ c2++; }
      }
}
```

Conditional Synchronization

- The concept of **critical section** prevents interferences when accessing to shared variables
- But we must also solve the problem of *conditional synchronization:*
  - Make a thread wait until certain condition is met
- In next unit we will study the concept of *monitor*
  - This solves both problems (i.e. mutual exclusion and conditional synchronization) using a single construction

Learning results of this Teaching Unit

- At the end of this unit, the student should be able to:
  - Identify the sections of an application that can be executed concurrently by different activities.
  - Identify the problems related with communication using shared memory. Describe the determinism problem.
  - Identify the different types of synchronization.
  - Distinguish the critical sections of an application and protect them using the suitable mechanisms.