UF 2: Multi-thread programming

Maribel Madueño

Thread issues

Thread issues

❶ Non-atomic access

❷ Interference

❸ Memory consistency errors

Non-atomic access

Expressions can define complex actions that can decompose into other actions.

For example, c++, does not describe an atomic action:

1. Retrieve the current value of c.
2. Increment the retrieved value by 1.
3. Store the incremented value back in c.

https://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html
https://docs.oracle.com/javase/tutorial/essential/concurrency/interfere.html

Thread interference

It happens when two operations, running in different threads, but acting on the **same data**, interleave.

https://docs.oracle.com/javase/tutorial/essential/concurrency/interfere.html

Thread interference Example

```
class Counter {
    private int c = 0;
    public void increment()
    {
        c++;
    }
    public void decrement()
    {
        c--;
    }
    public int value()
    {
        return c;
    }
}
```

Thread A invokes increment
Thread B invokes decrement

Thread A: Retrieve c=0.

Thread B: Retrieve c=0.

Thread A: Increment retrieved value; result is 1.

Thread B: Decrement retrieved value; result is -1.

Thread A: Store result in c; c is now 1.

Thread B: Store result in c; c is now -1.

Thread interference

Thread interference bugs are **difficult to detect** and fix, since the execution order of Threads is not controlled.

https://docs.oracle.com/javase/tutorial/essential/concurrency/interfere.html

Memory consistency error

It occur when different threads have inconsistent views of what should be the **same data**.

https://docs.oracle.com/javase/tutorial/essential/concurrency/memconsist.html
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html#MemoryVisibility

Memory consistency error Example

```java
class Counter {
    private int c = 0;
    public void increment()
    {
        c++;
    }
    public void decrement()
    {
        c--;
    }
    public int value()
    {
        return c;
    }
}
```

Thread A invokes increment
Thread B invokes value

The obtained value might well be "0", because there's no guarantee that thread A's change to c will be visible to thread B.

https://docs.oracle.com/javase/tutorial/essential/concurrency/memconsist.html
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html#MemoryVisibility

Memory consistency error

Solution: To establish a happens-before relationship between related statements.

https://docs.oracle.com/javase/tutorial/essential/concurrency/memconsist.html
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html#MemoryVisibility

Happens-before relationship

The results of a write by one thread are **guaranteed** to be visible to a read by another thread only if the **write** operation **happens-before** the **read** operation.

https://docs.oracle.com/javase/tutorial/essential/concurrency/memconsist.html
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html#MemoryVisibility

Happens-before relationships

- start()

  A call to start on a thread happens-before any action in the started thread
- join()

  All actions in a thread happen-before any other thread successfully returns from a join on that thread.

https://docs.oracle.com/javase/tutorial/essential/concurrency/memconsist.html
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html#MemoryVisibility

Happens-before relationships

- synchronized
- volatile

  A write to a volatile field happens-before every subsequent read of that same field. It does not entail mutual exclusion locking.

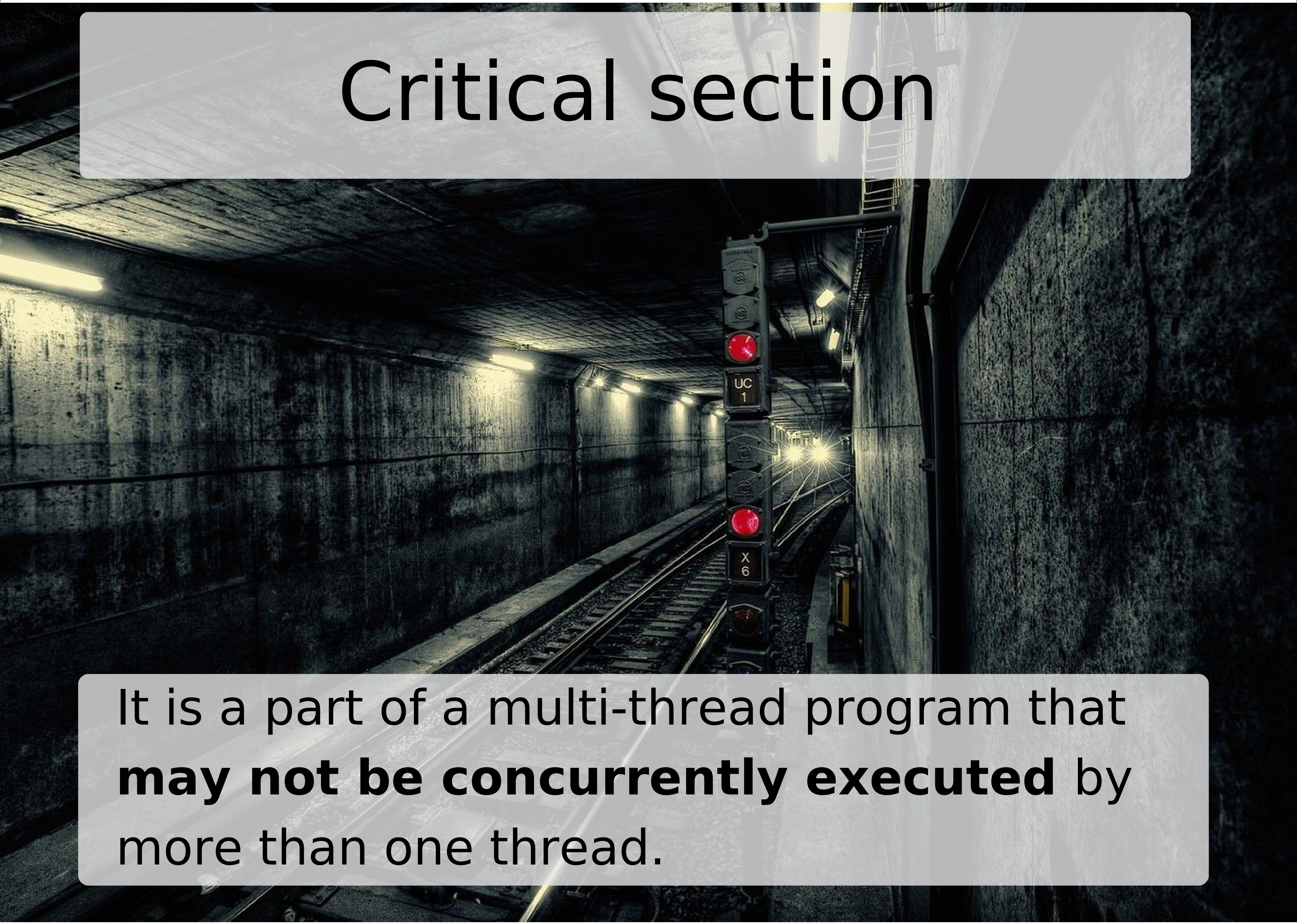
https://docs.oracle.com/javase/tutorial/essential/concurrency/memconsist.html
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html#MemoryVisibility

❶ ❷ ❸ ❹

Synchronization

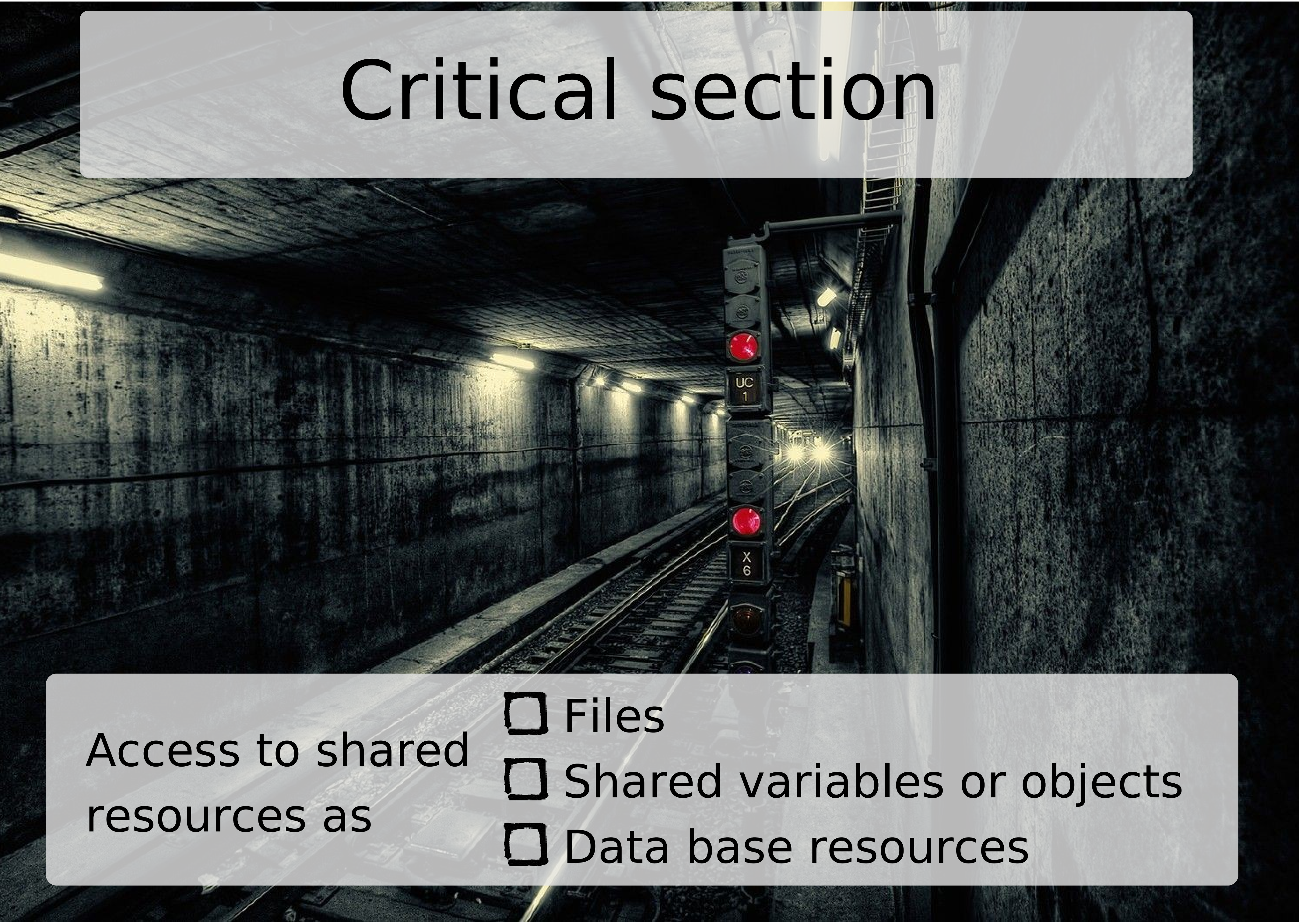
https://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html

Critical section



It is a part of a multi-thread program that **may not be concurrently executed** by more than one thread.

Critical section



Access to shared resources as

- Files
- Shared variables or objects
- Data base resources

Mutual exclusion

It is the requirement of ensuring that no two concurrent Threads are in their critical section at the same time.

Synchronized code

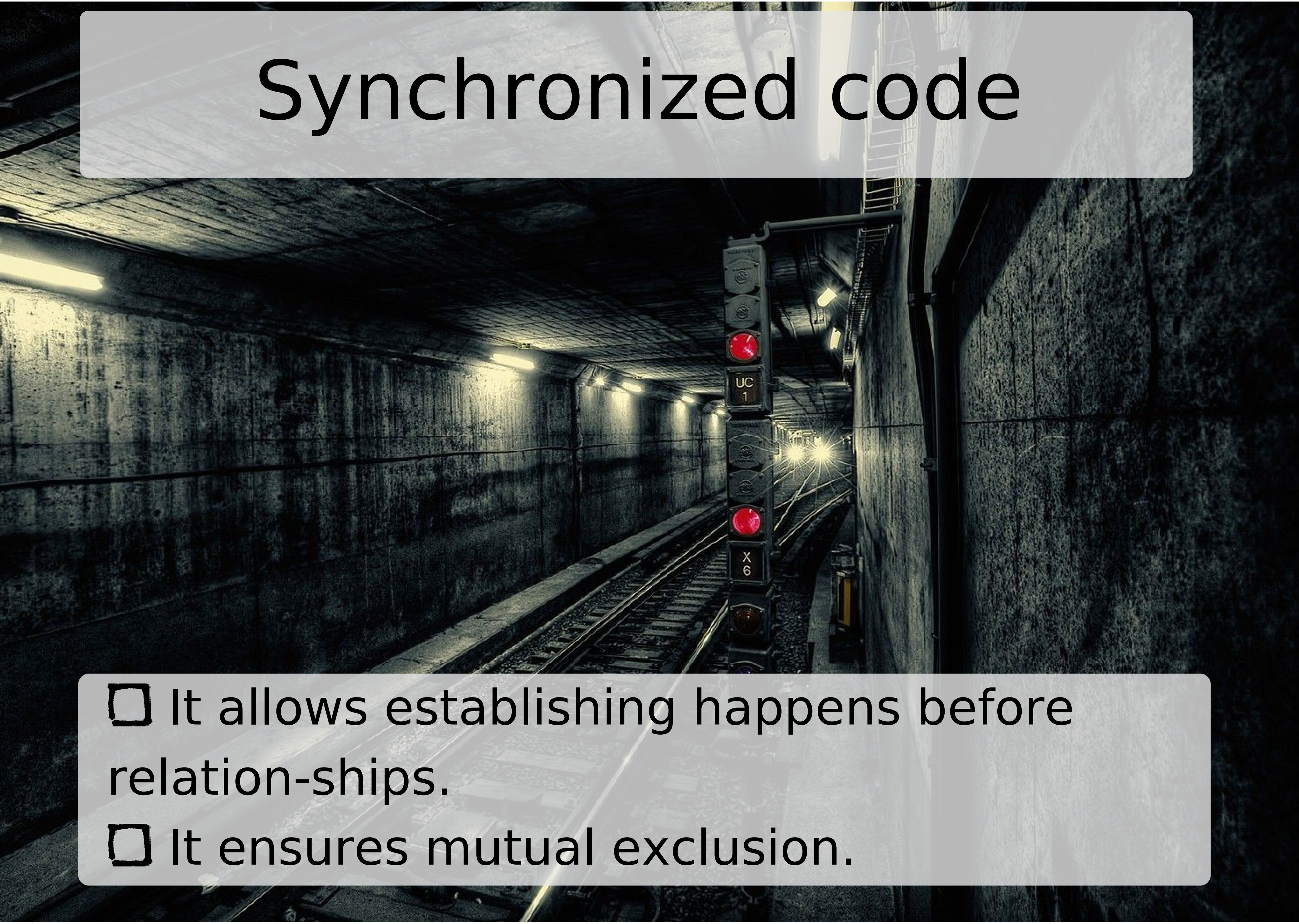
Synchronized method

```
synchronized public void  syncMethod(){
    //...
}
```

Synchronized statement

```
synchronized (this)
{
    //....
}
```

Synchronized code



- It allows establishing happens before relation-ships.
- It ensures mutual exclusion.

Synchronized methods

They enable a simple strategy for **preventing** thread interference and memory consistency **errors**.

https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

Synchronized methods

Reads or writes to an object's variables, visible to more than one thread, are done through synchronized methods.

https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

Synchronized methods

Final fields can be safely read through non-synchronized methods, once the object is constructed.

https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

Synchronized methods

When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object are **blocked** until the first is done.

https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

Synchronized methods

This automatically establishes a **happens-before relationship** with any subsequent invocation of a synchronized method for the same object.

https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

Synchronized methods

A Thread blocked by this means, is set to the **ready state** when the execution of the synchronized method

- ends
- A return occurs
- An Exception occurs

Synchronized methods Example

```java
class Counter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

Synchronized methods Example

Thread A invokes increment
Thread B invokes decrement

Thread A: Retrieve c=0.

Thread A: Increment retrieved value; result is 1.

Thread A: Store result in c; c is now 1.

Thread B: Retrieve c=0.

Thread B: Decrement retrieved value; result is -1.

Thread B: Store result in c; c is now -1.

https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

Intrinsic locks

- Synchronization is built around an internal entity known as the **intrinsic lock** or **monitor lock**.
- Every object has an intrinsic lock associated with it.

https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

Static synchronized methods

When a **static synchronized method** is invoked, the thread acquires the intrinsic lock for the Class object associated with the class.

https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

Synchronized statements

It must be specified the object that provides the **intrinsic lock**

```
synchronized (this)
{
    //....
}
```

https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

Synchronized statements

```java
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

The method needs to synchronize changes to lastName and nameCount, but to avoid synchronizing invocations of other objects' methods.

https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

Synchronized statements

```
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++;}
    }
}
```

They are also useful for improving concurrency with fine-grained synchronization.

https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

Synchronized statements

```
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++;}
    }
}
```

Instead of using synchronized methods or otherwise using the lock associated with this, we create two objects solely to provide locks.

https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

Reentrant synchronization

This describes a situation where synchronized code, directly or indirectly, invokes a method that also contains synchronized code, and both sets of code use the same lock.

https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

Guarded block

```
public void guardedJoy() {
   //Simple loop guard. Wastes processor time.
   //Don't do this!
   while(!joy) {}
      System.out.println("Joy has been achieved!");
}
```

joy is a shared variable
set by another thread

Such a block begins by polling a condition that must be true before the block can proceed.

https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

Guarded block

```
public synchronized void guardedJoy() {
   //This guard only loops once for each special
   //event, which may not be the event we're waiting for.
   while(!joy) {
      try {
         wait();
      }
      catch (InterruptedException e) {}
      }
      System.out.println("Joy and efficiency have been
      achieved!");
}
```

A more efficient guard invokes Object.wait to suspend the current thread.

https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

Guarded block

```
public synchronized notifyJoy(){
    joy = true;
    notifyAll();
}

public synchronized notifyOther(){
    other = true;
    notifyAll();
}
```

The thread which invoked guardedJoy becomes ready

The invocation of wait does not return until another thread has issued a notification that some special event may have occurred — though not necessarily the event this thread is waiting for

https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

➊ ➋ ➌ ➍

Synchronization issues

https://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html

Liveness

It is the ability of a concurrent application to execute in a **timely** manner.

https://docs.oracle.com/javase/tutorial/essential/concurrency/liveness.html

Liveness problems

❶ Deadlock

❷ Starvation

❸ Live-lock

https://docs.oracle.com/javase/tutorial/essential/concurrency/liveness.html

Deadlock

It's a situation where two or more threads are **blocked forever**, waiting for each other.

https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html

Deadlock - Example

```
public class Deadlock {
   static class Friend {

    private final String name;
      public Friend(String name)
      {this.name = name;}
      public String getName()
    {return this.name;}

    public synchronized void bow(Friend bower) {
       System.out.format("%s: %s"+" has bowed to me!%n",
                this.name, bower.getName());
         bower.bowBack(this);}
      public synchronized void bowBack(Friend bower) {
         System.out.format("%s: %s"+" has bowed back to
         me!%n",this.name, bower.getName());
        }
```

https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html

Deadlock - Example

```
public class Deadlock {
   static class Friend {

    private final String name;
      public Friend(String name)
      {this.name = name;}
      public String getName()
    {return this.name;}

    public synchronized void bow(Friend bower) {
       System.out.format("%s: %s"+" has bowed to me!%n",
                this.name, bower.getName());
         bower.bowBack(this);}
      public synchronized void bowBack(Friend bower) {
         System.out.format("%s: %s"+" has bowed back to
         me!%n",this.name, bower.getName());
        }
```

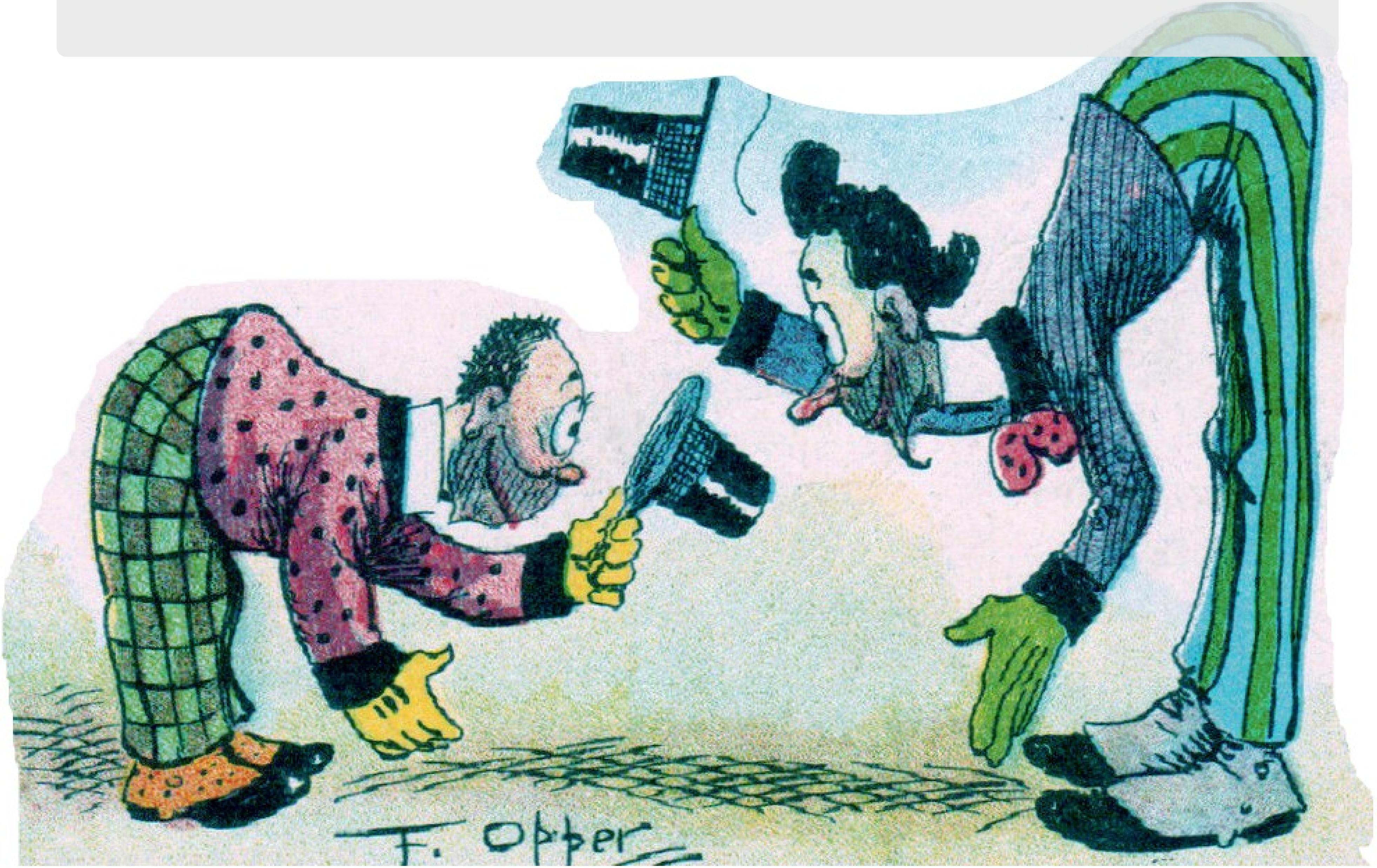
https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html

Deadlock - Example

```
    public static void main(String[] args) {
        final Friend alphonse =
            new Friend("Alphonse");
        final Friend gaston =
            new Friend("Gaston");
        new Thread(new Runnable() {
            public void run() {
           alphonse.bow(gaston); }
           }).start();
        new Thread(new Runnable() {
            public void run() {
           gaston.bow(alphonse); }
           }).start();
    }
}
```

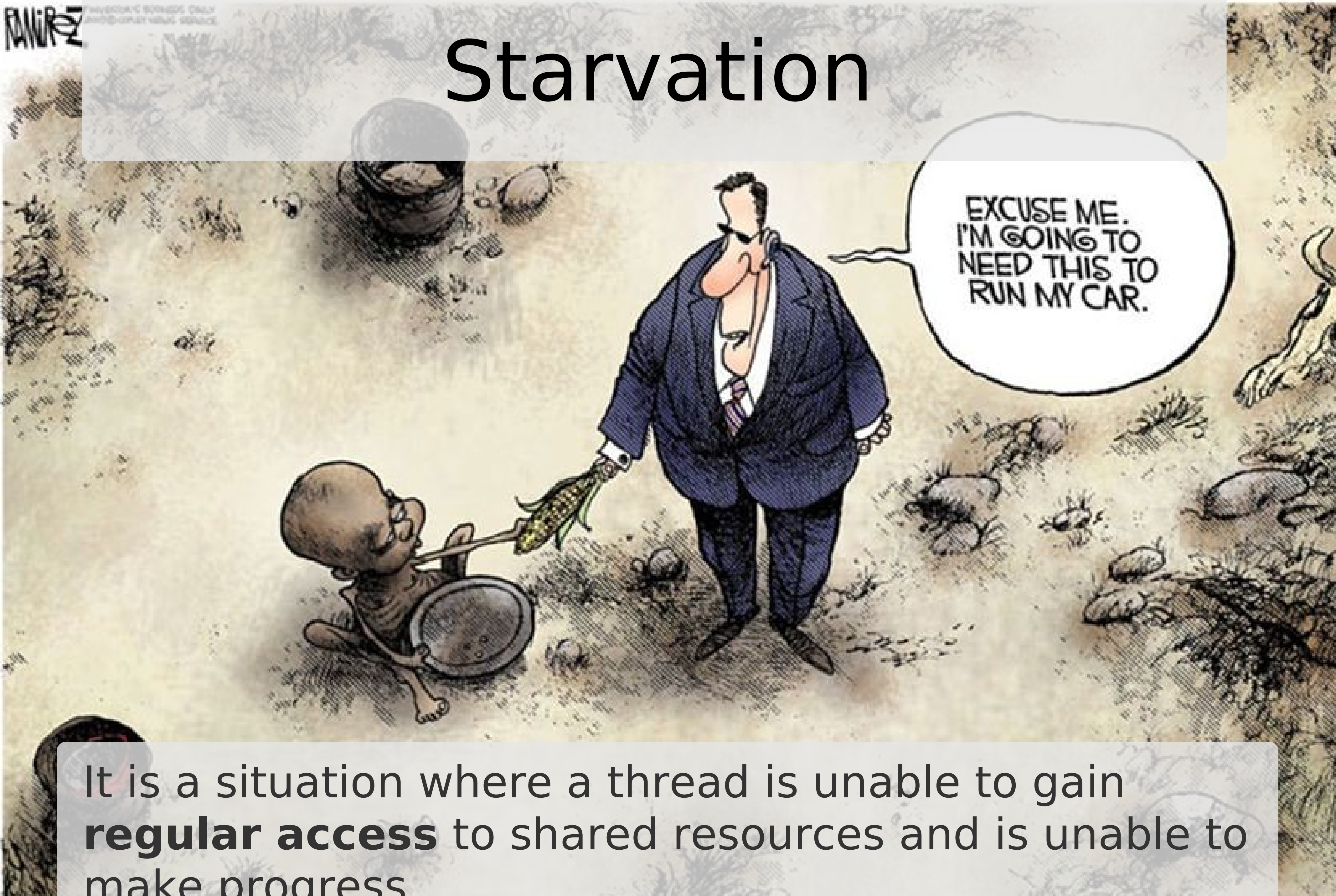
https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html

Deadlock



https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html

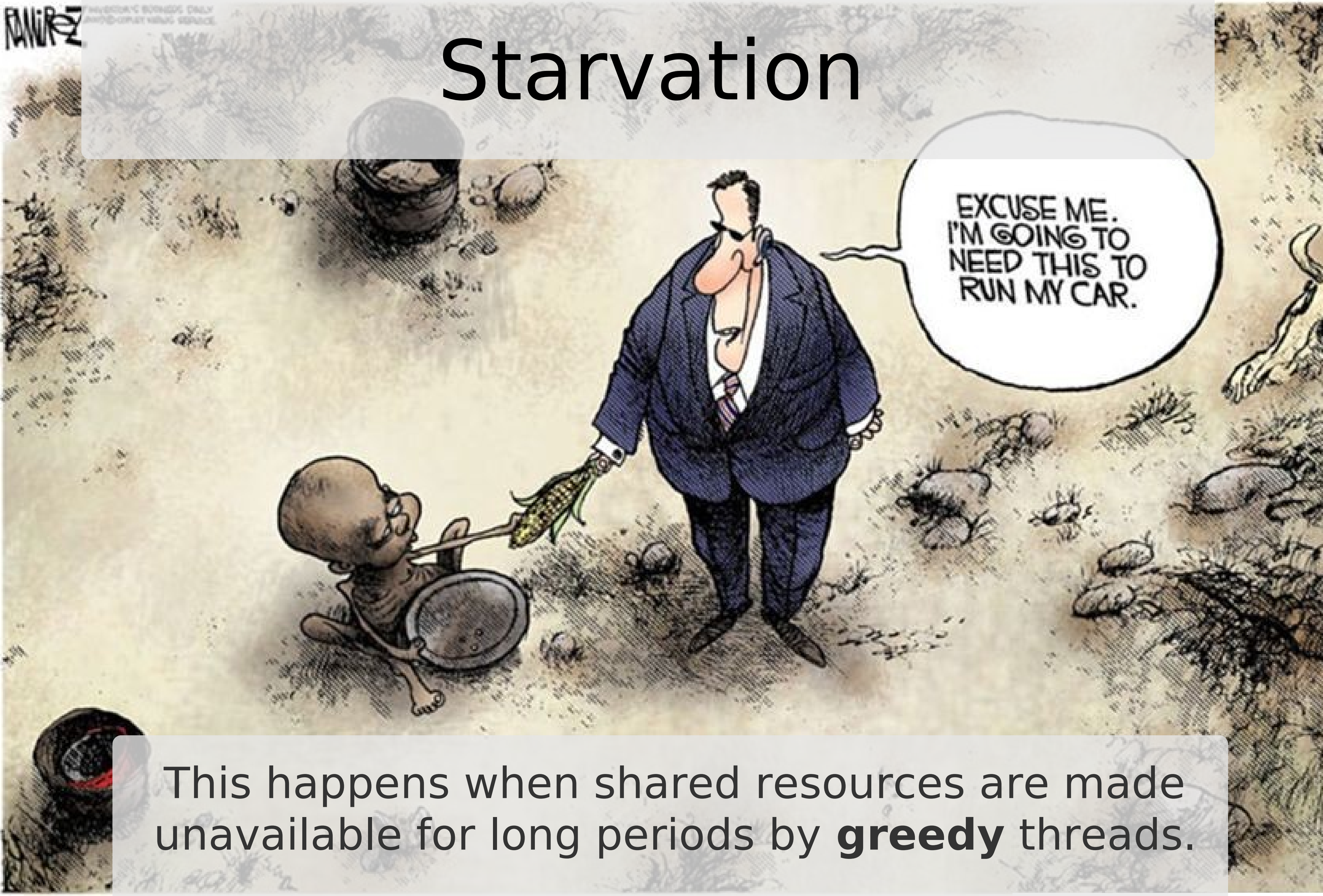
Starvation



It is a situation where a thread is unable to gain **regular access** to shared resources and is unable to make progress.

https://docs.oracle.com/javase/tutorial/essential/concurrency/starvelive.html

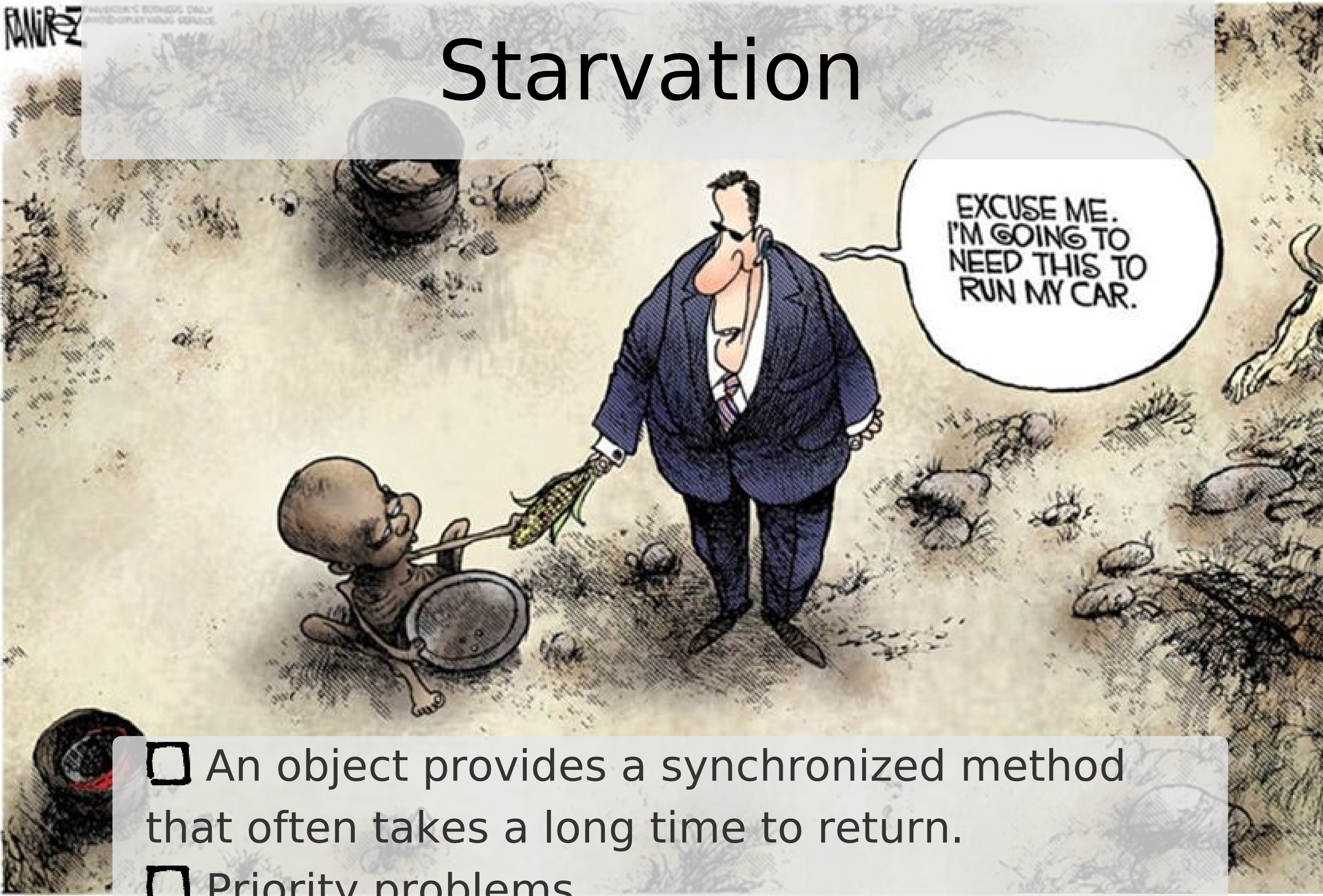
Starvation



This happens when shared resources are made unavailable for long periods by **greedy** threads.

https://docs.oracle.com/javase/tutorial/essential/concurrency/starvelive.html

Starvation



- An object provides a synchronized method that often takes a long time to return.
- Priority problems

https://docs.oracle.com/javase/tutorial/essential/concurrency/starvelive.html

Starvation - Example

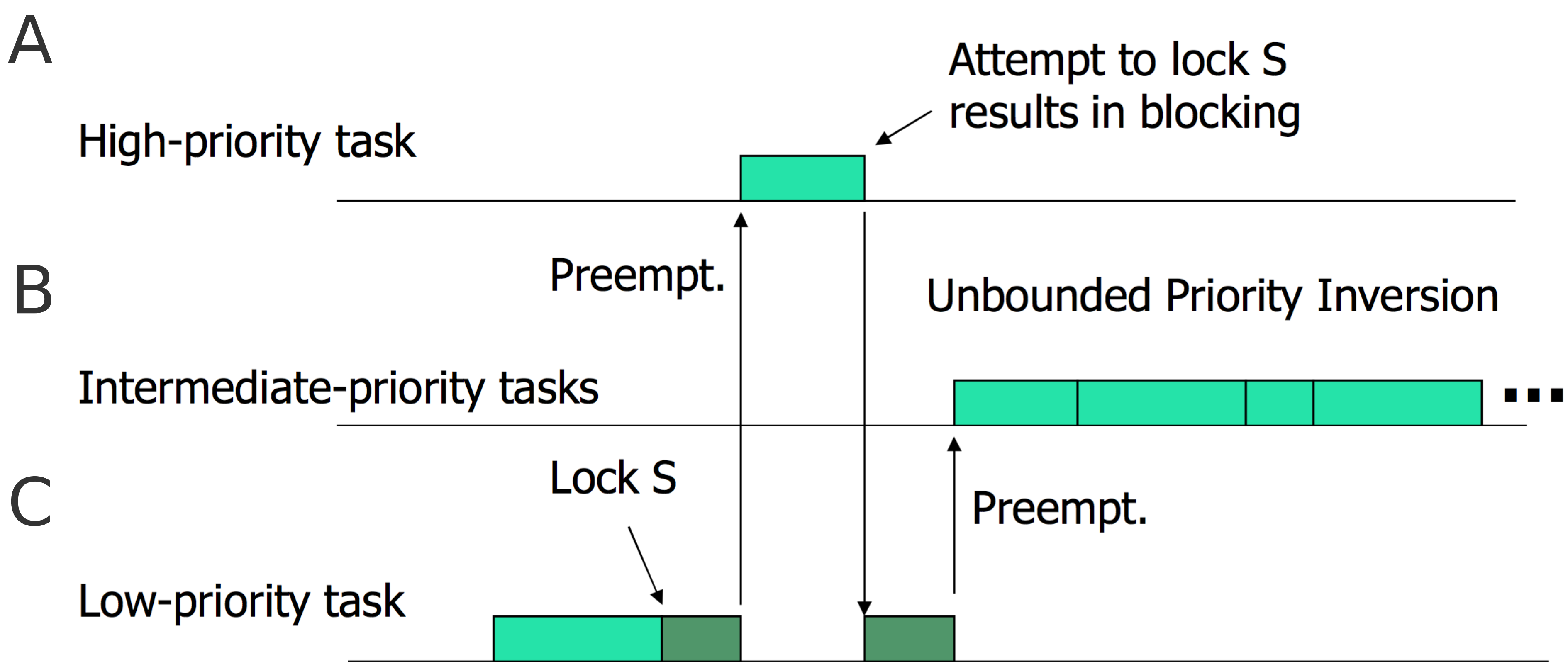
Priority inversion problem

- Thread A is at high priority, waiting for result or resource from Thread C at low priority.
- Thread B at intermediate priority is CPU-bound.
- Thread C never runs, hence thread A never runs.

http://web.cs.ucla.edu/classes/fall10/cs111/scribe/10b/#lec10_3

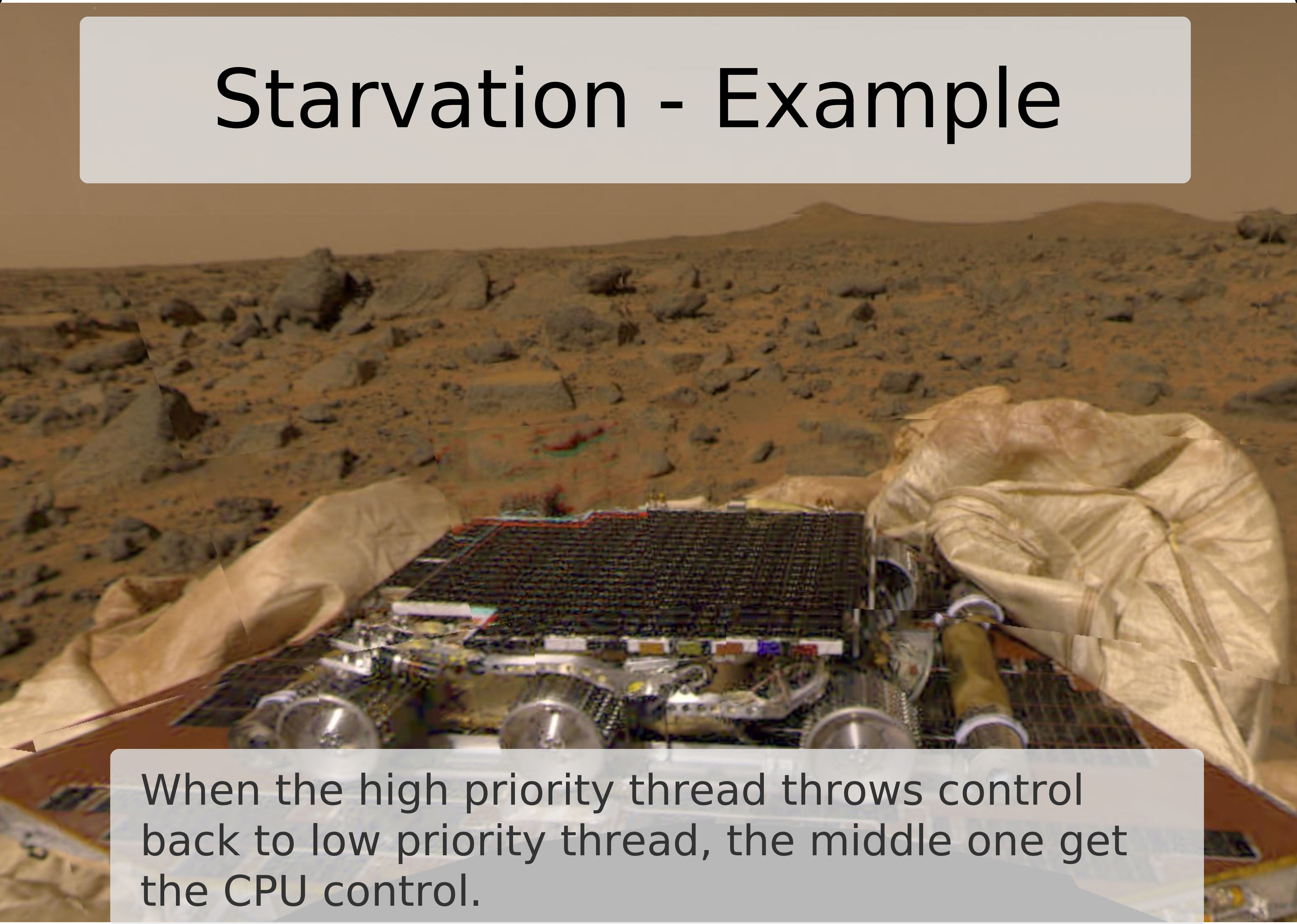
Starvation - Example

Priority inversion problem



http://web.cs.ucla.edu/classes/fall10/cs111/scribe/10b/#lec10_3

Starvation - Example



When the high priority thread throws control back to low priority thread, the middle one get the CPU control.

https://www.youtube.com/watch?v=lyx7kARrGeM

Live-lock

Several threads spend all their time **trying to synchronize** instead of getting any work done.

https://docs.oracle.com/javase/tutorial/essential/concurrency/starvelive.html

Livelock

The threads are **not blocked**, they are too busy responding to each other to resume work.

http://web.cs.ucla.edu/classes/fall10/cs111/scribe/10b/#lec10_4

Classical problems

❶ Producer-consumer

❷ Reader-writer

❸ Dining Philosophers

❹ Sleeping Barber's

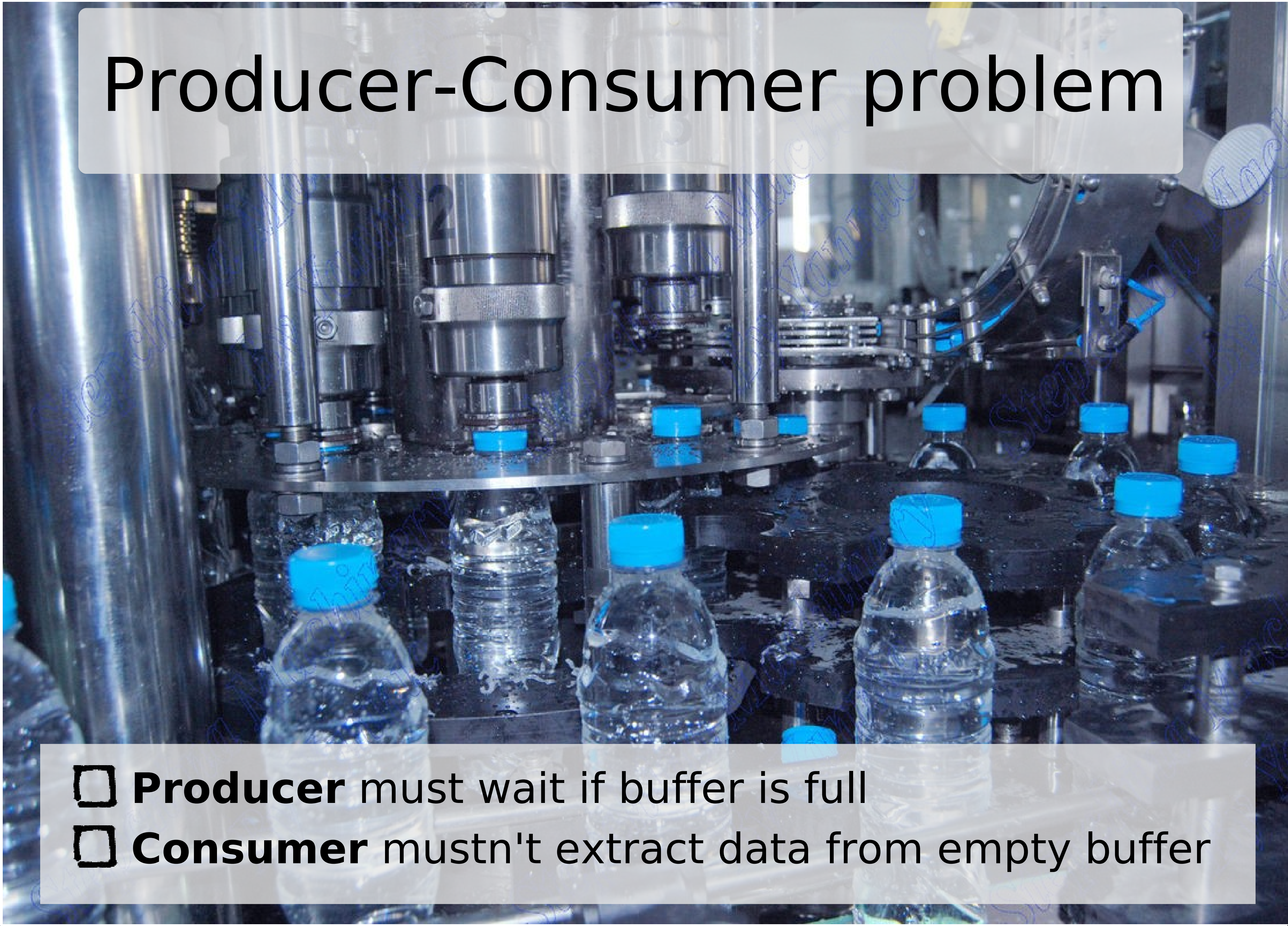
Producer-Consumer problem

Two or more threads **communicate** with some threads “producing” data that others “consume”.

Producer-Consumer problem

- **Producer** generates things into a buffer
- **Consumer** takes those things and uses them

Producer-Consumer problem



- ❑ **Producer** must wait if buffer is full
- ❑ **Consumer** mustn't extract data from empty buffer

Producer-Consumer Guarded block - Example

```
public class Drop {
    // Message sent from producer(P) to consumer(C).
    private String message;
    // true if C should wait for P to send message,
    // false if P should wait for C to retrieve message.
    private boolean empty = true;
    public synchronized String take() {
        // Wait until message is available.
        while (empty) {
            try {wait();}
            catch (InterruptedException e) {}
        }
        // Toggle status.
        empty = true;
        // Notify producer that status has changed.
        notifyAll();
        return message;
    }
```

https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

Producer-Consumer Guarded block - Example

```
    public synchronized void put(String message) {
        // Wait until message has been retrieved.
        while (!empty) {
            try {wait();}
            catch (InterruptedException e) {}
        }
        // Toggle status.
        empty = false;
        // Store message.
        this.message = message;
        // Notify consumer that status has changed.
        notifyAll();
    }
}
```

https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

Producer-Consumer Guarded block - Example

```java
import java.util.Random;
public class Producer implements Runnable {
    private Drop drop;
    public Producer(Drop drop) {this.drop = drop;}
    public void run() {
        String importantInfo[] = {
        "Mares eat oats","Does eat oats",
        "Little lambs eat ivy","A kid will eat ivy too"};
        Random random = new Random();
        for (int i = 0;i < importantInfo.length;i++) {
            drop.put(importantInfo[i]);
            try {
                Thread.sleep(random.nextInt(5000));
            } catch (InterruptedException e) {}
        }
        drop.put("DONE");
    }
}
```

https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

Producer-Consumer Guarded block - Example

```
import java.util.Random;
public class Consumer implements Runnable {
    private Drop drop;
    public Consumer(Drop drop) {this.drop = drop;}
    public void run() {
        Random random = new Random();
        for (String message = drop.take();
             !message.equals("DONE");
             message = drop.take()) {
            System.out.format("MESSAGE RECEIVED: %s%n",
             message);
            try {
                Thread.sleep(random.nextInt(5000));
            } catch (InterruptedException e) {}
        }
    }
}
```

https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

Producer-Consumer Guarded block - Example

```
public class ProducerConsumerExample {
    public static void main(String[] args) {
        Drop drop = new Drop();
        (new Thread(new Producer(drop))).start();
        (new Thread(new Consumer(drop))).start();
    }
}
```