ECS518U - Operating Systems
Week 10

Synchronisation, Deadlocks, Conditional Suspension

Tassos Tombros

Things you will learn today

- Important concepts in concurrency:
  - **Race condition**
  - **Critical region**
  - **Mutual exclusion**
- Locks & **Synchronized** methods in Java
- **Deadlocks**
  - Dining philosophers example
- Conditional suspension
  - **wait / notify**
- Thread **termination**

Reading: Java Concurrency tutorial (link on QMPlus)
Your favourite (!!!) Java book
Tanenbaum: Chapter 2, section 2.3
Stallings: Parts of Chapters 5 & 6

Race Conditions between threads

Race condition: a form of **error** in concurrent programs. Occurs if effect of threads on shared data **depends on the order in which the threads are scheduled**.

Behaviour becomes unpredictable – as it differs for different interleavings and is not controlled

- Two threads, A and B, compete with each other
  - One tries to increment a shared counter, the other tries to decrement it

Thread A

```
i = 0;
while (i < 10)
   i = i + 1;
print("A wins!");
```

Thread B

```
i = 0;
while (i > -10)
   i = i - 1;
print("B wins!");
```

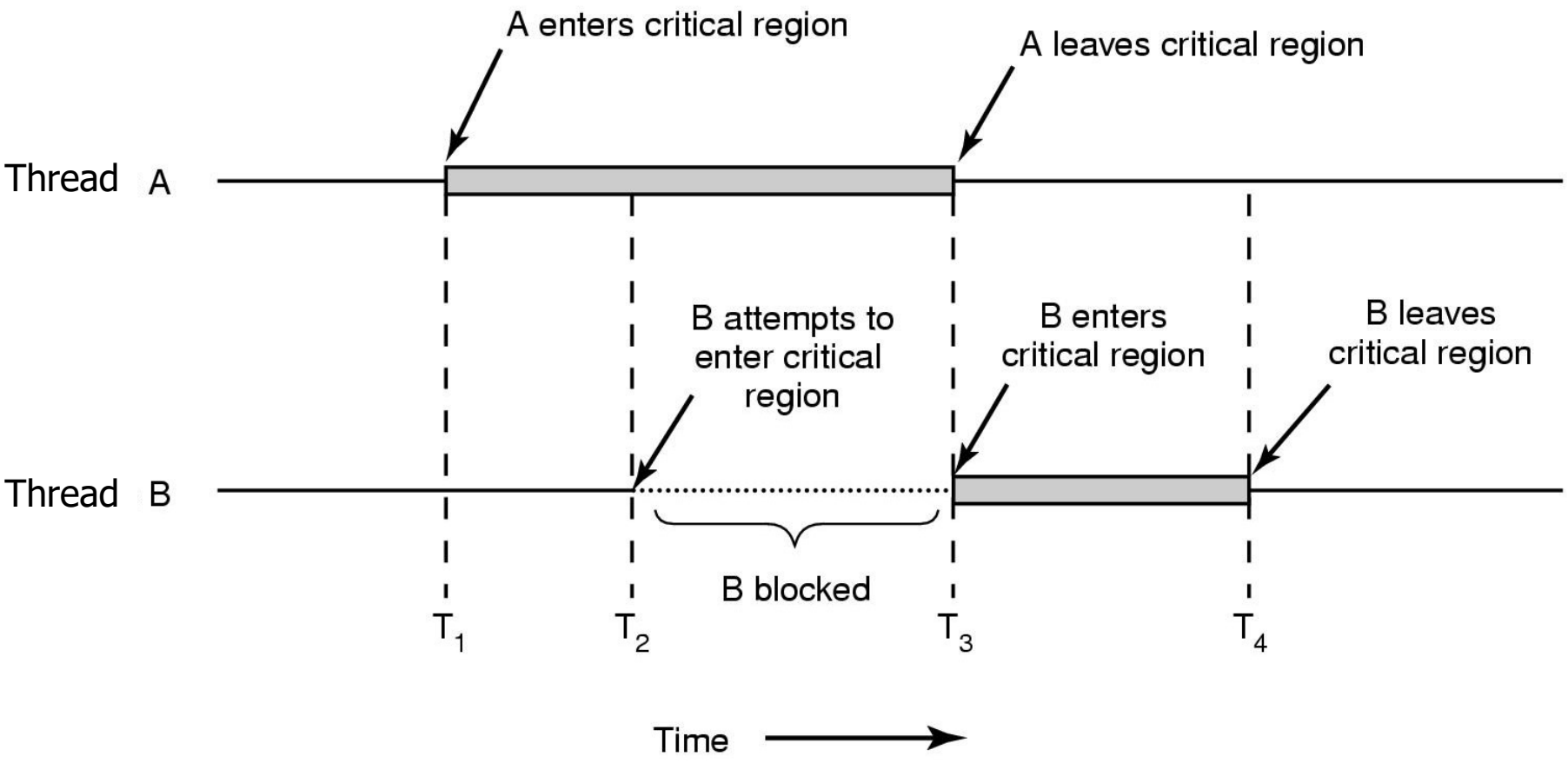
- Who wins? (no guarantee)
- Is it guaranteed that someone wins? Why or why not? (No it is not, it is entirely possible that threads increment/decrement the variable i without ever exiting the while loop)
- What if both threads have their own CPU running at same speed? Is it guaranteed that it goes on forever? (no, but it is more likely than in other scenarios)
- What about if they run on a uniprocessor system? (it is more likely that we will have a winner, but we don't know that for sure)

How to avoid Race Conditions

- **Mutual Exclusion**
  - Prevent two threads using a shared resource (variable, memory, file) at the same time
  - One thread **excludes** the other while doing its task
- **Critical Region**
  - Part of the program over which mutual exclusion is needed
- **Requirements** for critical regions:
  - No two threads may be simultaneously inside their critical regions (**MUTUAL EXCLUSION)**
  - No assumptions may be made about speeds or the number of CPUs
    - program has to work correctly for any **arbitrary interleavings**
  - No thread running outside its critical region may block other threads
  - No thread should have to wait forever to enter its critical region

Critical Region: What is Needed

- Mutual exclusion using critical regions



Critical Regions and Locks

- Java **Synchronized** methods
- '**Synchronized**' code is a 'critical region'
  - Only one thread at a time
  - Access is controlled by a 'lock'
- All objects have a lock – **lock is for an object NOT class**
  - when we call a synchronized method, we take a lock at the start of the method, we release it at the end
  - when one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done
- Synchronized method – simply add the **synchronized** keyword to its declaration

```
public synchronized int myMethod() {
   code
}
```

Alternative use of Synchronized

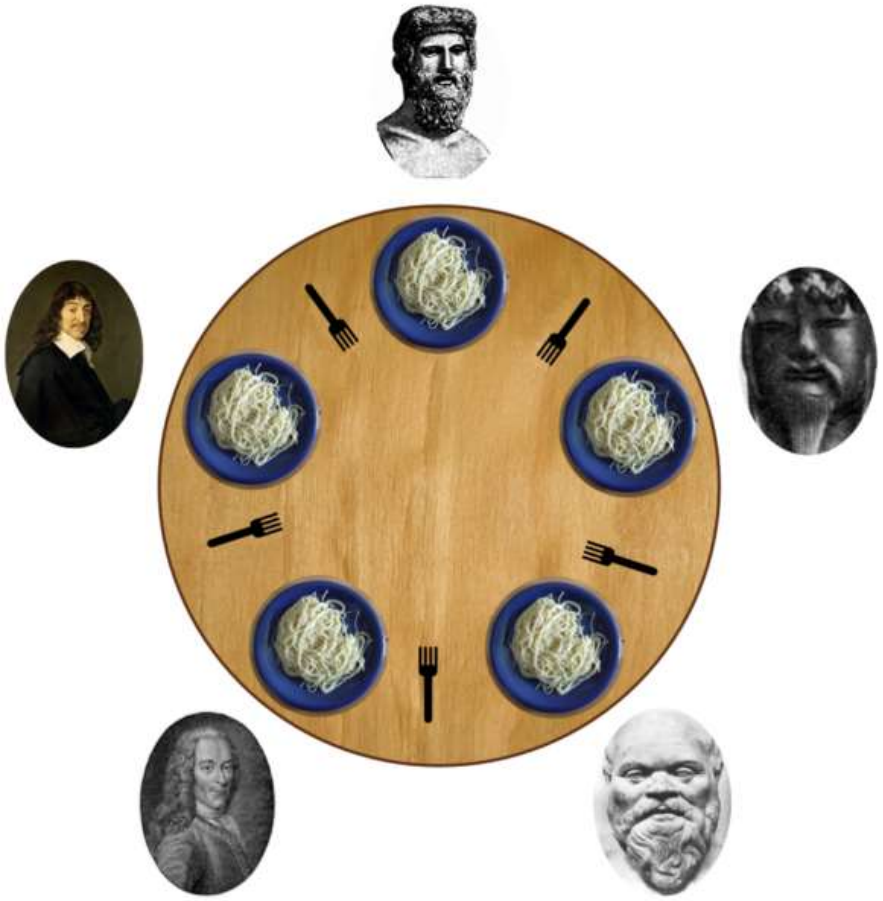
- Alternative way: **Synchronized statements**
  - synchronized statements must specify the object that provides the lock

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

Checkpoint: Concepts

1. **Mutual exclusion**
   - One user excludes another: multiple users not allowed
   - Applies to resources of all types
2. **Critical region**
   - Region of the program needing mutual exclusion
   - Thread must enter eventually; no timing assumptions
3. **Lock**
   - Controls access to critical region
   - All Java objects have lock
4. **Synchronized method**
   - Lock held; block otherwise

Dining Philosophers



- Classic problem to abstract concurrency issues & **resource sharing**
- These philosophers only eat or think
- To eat, must acquire two forks - one from their left and one from their right
- When a philosopher thinks, he puts down both forks in their original locations

Dining Philosophers – 'Solution'

- **Each philosopher a thread**
  - Pickup left fork
  - Pickup right fork
  - Eat
  - Putdown left fork
  - Put down right fork
  - Think (sleep)
- Fork (eating implement, not fork()!!!)
  - Synchronized methods to prevent more than one philosopher using the fork at once

- Problem is of shared resources
- Fork is a resource, acquired dynamically
- Mutual exclusion required for forks

Philosopher – a thread

```
public class Philosopher implements Runnable {

  public Philosopher(String n, Fork l, Fork r) { … }

  public void run() {
    System.out.println(name + " starting") ;
    try {
      while (!Thread.interrupted()) {
        Thread.sleep(random.nextInt(700)) ;
        left.take(name) ;
        right.take(name) ;
        eat() ;
        left.release(name) ;
        right.release(name) ;
      }
    } catch (InterruptedException e) {}
  }
```

Fork – Resource

```
public class Fork {
  private boolean onTable ;

  public Fork() {
    onTable = true ;
  }

  public synchronized void take(String phil) {
     while (!onTable) { wait(); }
     onTable = false;
  }

  public synchronized void release(String phil) {
     onTable = false;
     notifyAll();
  }
}
```

- Lock prevents race between philosophers
- Only one gets fork
- Other blocks
- wait/notify explained shortly

Deadlock on the table

- (Run demo)
- How do the philosophers deadlock?
  - All philosophers decide to eat at same time
  - They all pick up one fork
  - None of them can eat hence the system comes to a halt
  - **Circular waiting for resources**
- **Remember:** You can't test for correctness with concurrent programs, you have to **design for correctness**
- There are 2 types of programs:
  - Those that have no obvious errors
  - And those that obviously have no errors

  (Tony Hoare)

Definition of Deadlocks

A set of processes (threads) is deadlocked if each process (thread) in the set is waiting for an event that only another process (thread) in the set can cause

- **Circular wait**
- Deadlock is **created by using locks** to ensure mutual exclusion

Perspectives on Deadlock

- **Programmers' point of view**
  - A program in which deadlock is possible for some interleaving is incorrect
  - We need to learn about algorithms that are deadlock-free
- **Operating System's point of view**
  - Resource deadlock: processes can have a circular wait for resources
  - **What should an OS do to prevent user programs from creating deadlocks ?**
    - We will look at this issue next week

A Deadlock-Free Solution

- One philosopher S picks up 'right' fork first
- Others pick up 'left' fork first
- Argument
  1. Suppose that S has his right fork then
     - Either the philosopher to his left has his right fork, in which case he has both forks
     - Or this fork is available to S
     - No deadlock
  2. Suppose that S does not have his right fork then
     - S does not have his left fork either, so
     - It is available to the philosopher on his left
     - No deadlock
- Not the end of the story – other problems can occur but out of scope for now

The Thread Termination Problem

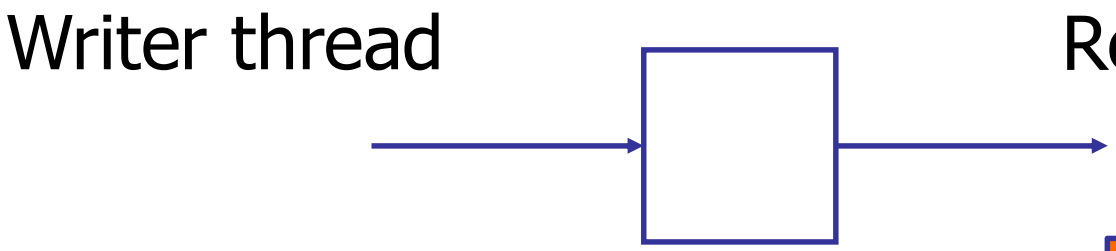
- A thread terminates 'normally' when the run method returns
- In many cases however we need to terminate a thread 'earlier'
  - e.g. many threads working on a problem, one thread 'solves' the problem, others need to stop
- Deprecated methods
  - **public final void destroy()**: stop, releasing locks ➔ unsafe
  - **public final void suspend():** stop without releasing locks ➔ deadlock
- **Co-operative termination**
  - Thread requests to terminate
  - Thread must monitor request
  - Thread must co-operate by releasing any resources that it is currently using

Termination using 'Interrupt'

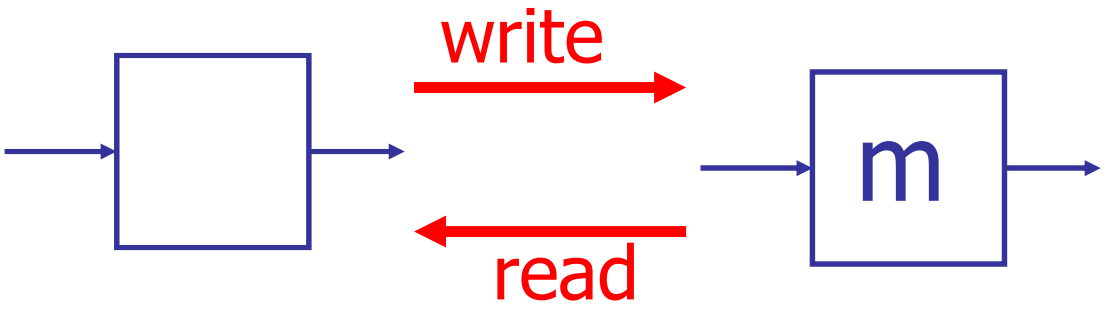
```
InterruptedException
public boolean isInterrupted()
public static boolean interrupted()
public void interrupt()
```

- Each thread has an 'interrupted' flag
- Calling 'interrupt'
  - Sets the flag
  - If the thread is sleeping or waiting, `InterruptedException` is thrown
- Threads must:
  - **test** the flag using '`interrupted`' and
  - **catch** the exception

Conditional Suspension (wait/notify): Mailbox Example



- Mailbox can be **empty** or **full**:



The mailbox is a critical region. Only one of the reader or the writer can access mailbox at any time. The writer puts a message in if the box is empty, the reader takes the message out if the box is full.

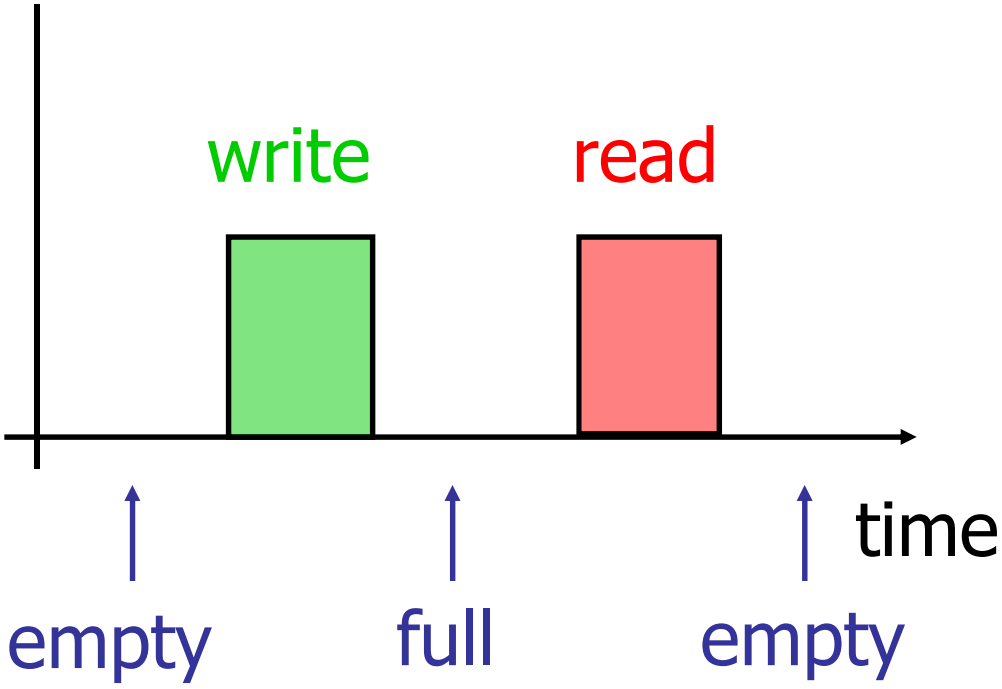
Mailbox Code – Solution attempt I

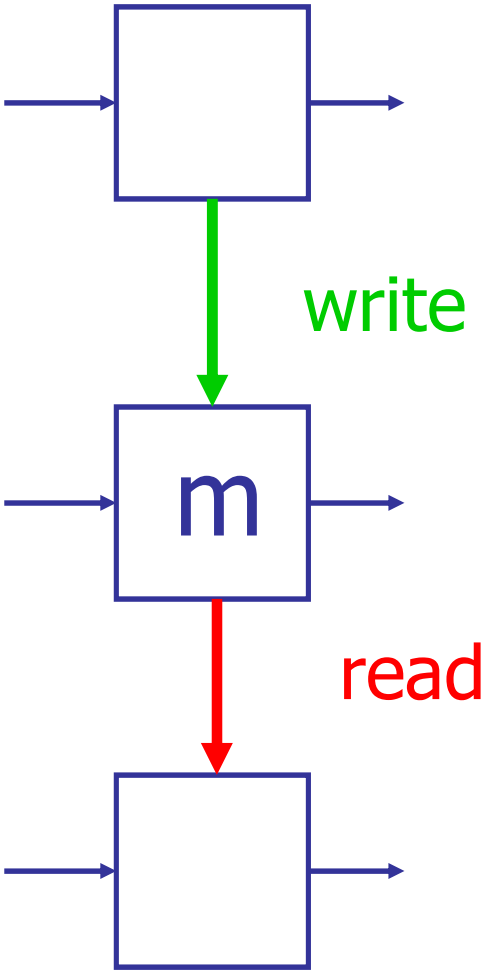
```
private T content ;

public synchronized void write(T message) {
  content = message ;
}

public synchronized T read() {
  return content ;
}
```

Using synchronized achieves mutual exclusion.
Also ensures changes are visible.

Scenario I – Write First

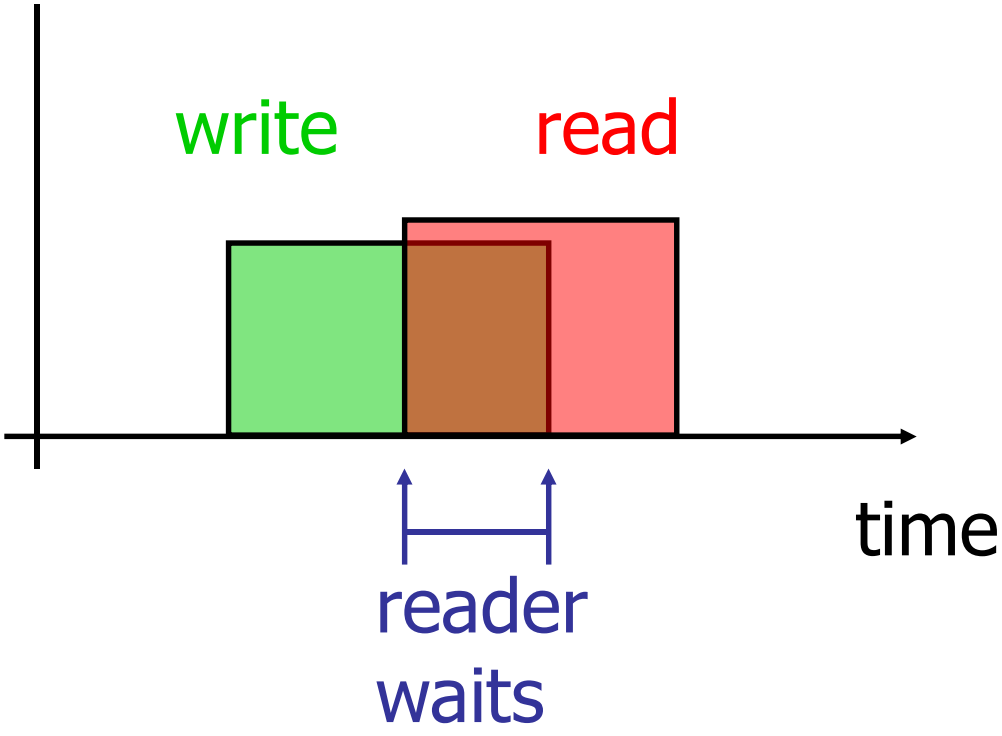
- Write before read
- Ok

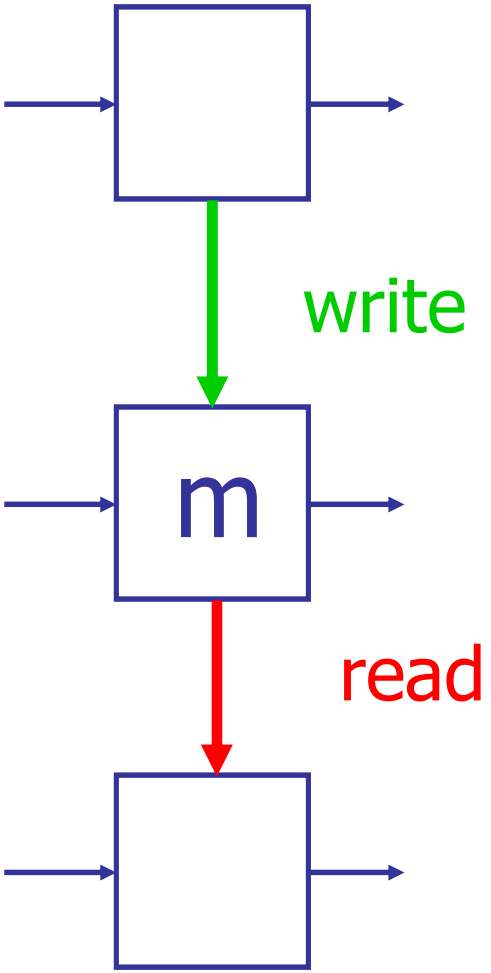




Scenario II – Read while Writing

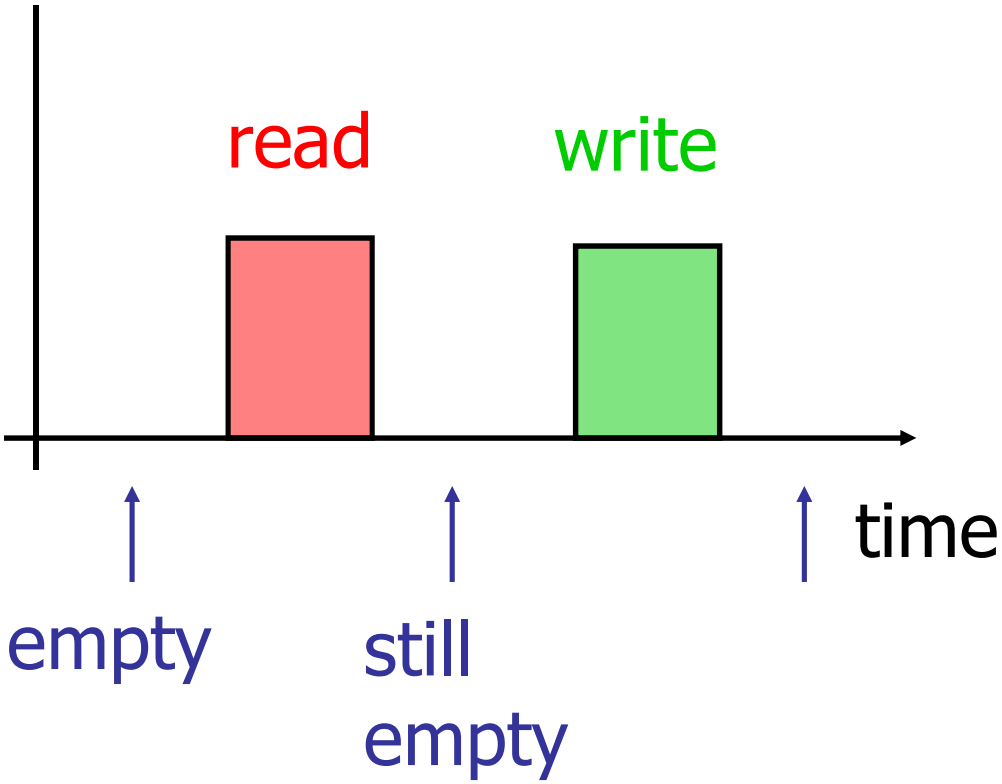
- Write
- Read while still writing
- Synchronized – ok





Scanario III – Read First

- If read first – nothing to read
- **Problem**



- The Reader should not read an empty mailbox

… symmetrically

- The Writer should not write to a full mailbox
- Thread must wait when mailbox is in the wrong state

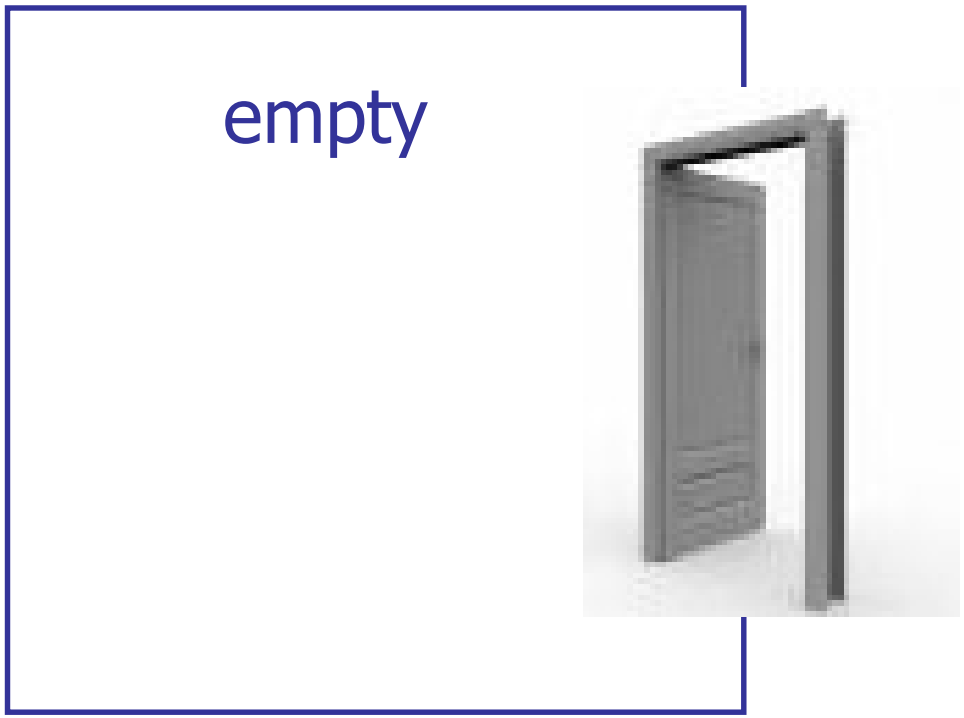
How can we do it?

Mailbox Code – Attempt II

```
private int content ;
private boolean empty ;

public synchronized int read() {
   while (empty) {Thread.sleep(1);}
   // mailbox now NOT empty
   empty = true ;   // mailbox now empty again
   return content ;
}
```

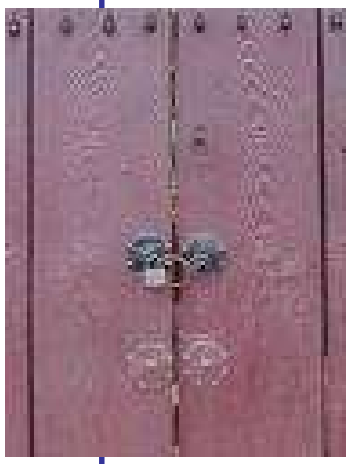
Does this code work?

1. Mailbox empty



1. Mailbox empty
2. Reader enters and locks door

empty



1. Mailbox empty
2. Reader enters and locks door
3. Reader asleep with the key

Deadlock

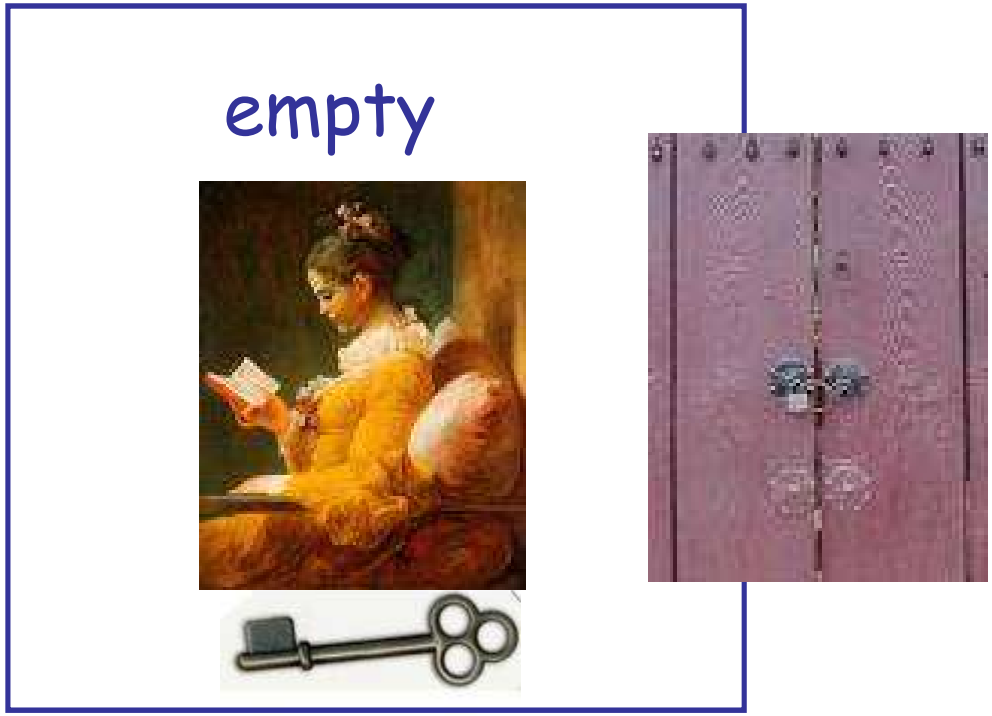
1. Mailbox empty
2. Reader enters and locks door
3. Reader asleep with the key
4. Writer locked out

Mailbox Code – Solution

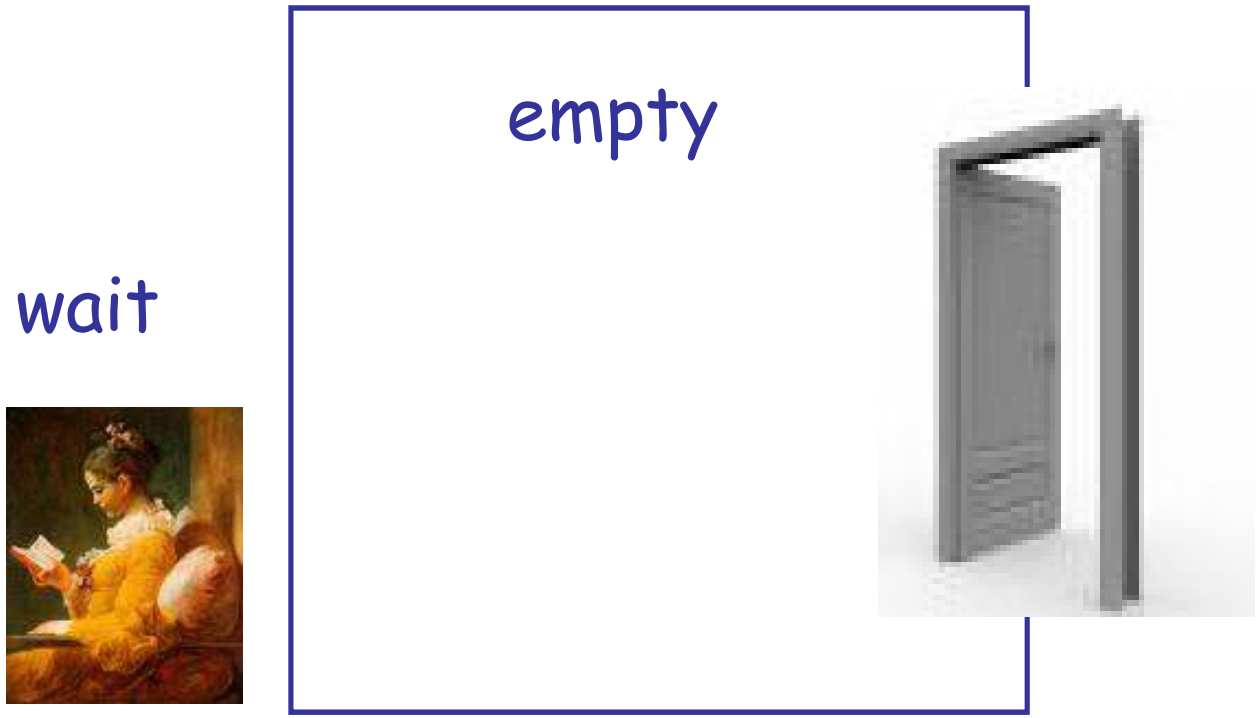
```
private int content ;
private boolean empty ;

public synchronized int read() {
   while (empty) {wait() ;}
   // mailbox now NOT empty

   empty = true ;   // mailbox now empty again
   notify() ;
   return content ;
}
```

… solution not complete yet!

1. Mailbox empty
2. Reader enters and locks door

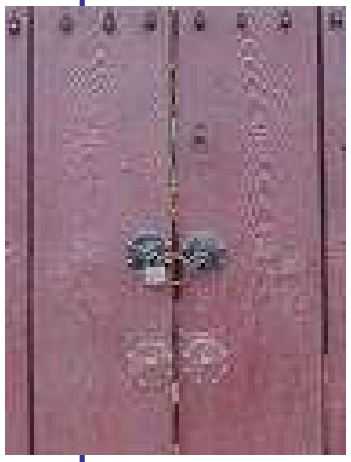


1. Mailbox empty
2. Reader enters and locks door
3. … finding mailbox empty, waits outside

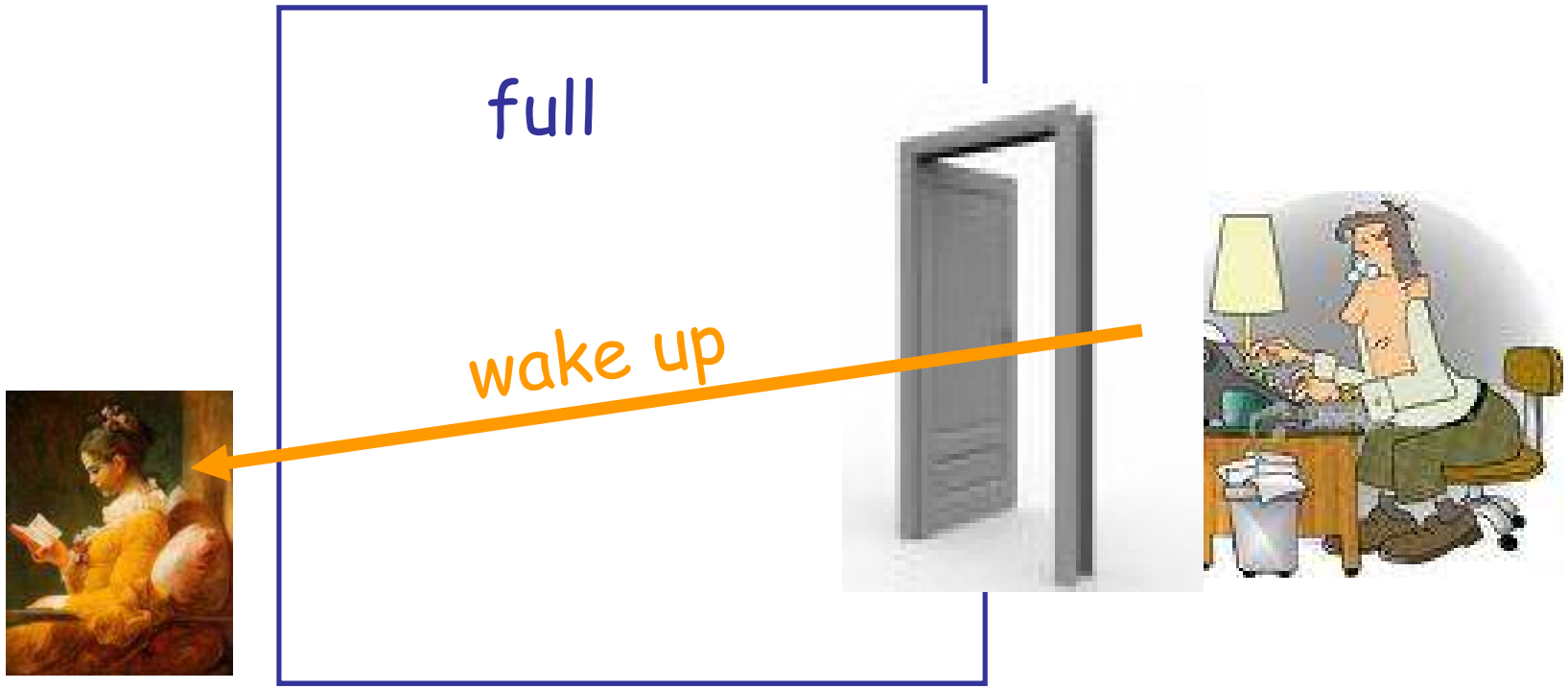


1. Mailbox empty
2. Reader enters and locks door
3. … finding mailbox empty, waits outside
4. Writer enters, and leaves message

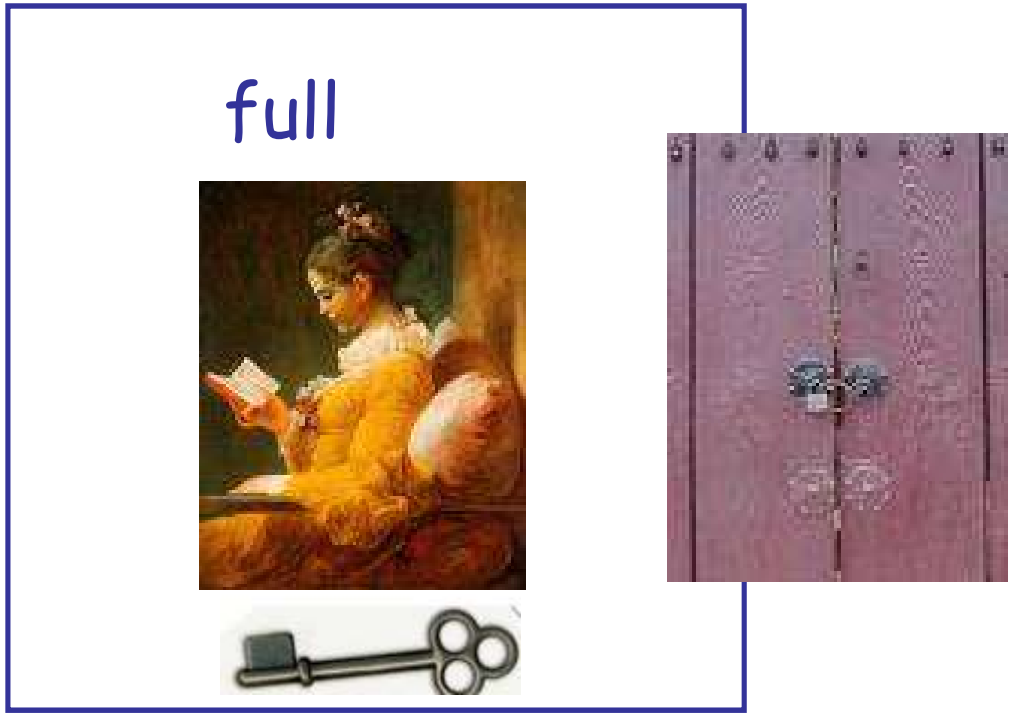
full



1. Mailbox empty
2. Reader enters and locks door
3. … finding mailbox empty, waits outside
4. Writer enters, and leaves message
5. … notifies reader on way out



1. Mailbox empty
2. Reader enters and locks door
3. … finding mailbox empty, waits outside
4. Writer enters, and leaves message
5. … notifies reader on way out
6. Reader goes back to get message



Mailbox Code – Read

```
private int content ;
private boolean empty ;

public synchronized int read() {
   while (empty) {wait() ;}
   // mailbox now NOT empty

   empty = true ;
   // mailbox now empty again
   notify() ;
   return content ;
}
```

1. Variable – model the condition

2. Wait in a loop – while condition false

3. Update condition

4. Notify other threads

... solution not complete yet!

Mailbox Code – Write Solution

```
private int content ;
private boolean empty ;

public synchronized void write(int message) {
  while (!empty) {wait() ;}
  // mailbox now empty

  content = message ;
  empty = false ;  // mailbox now full again
  notify() ;
}
```

... solution not complete yet!

wait() and notify()

public class **Object**

void	**notify**() Wakes up a single thread that is waiting on this object's monitor.
void	**notifyAll**() Wakes up all threads that are waiting on this object's monitor.
void	**wait**() Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.

Rules for Using Wait/Notify

1. Create variables for condition
2. Wait called in a synchronized method
3. Always call wait inside a loop
   - Many threads may wake-up
   - Don't assume that the interrupt was for the condition you were waiting for, or that the condition is still true

```
while (!empty) {
  wait() ;
}
```

4. notify / notifyAll before exit

   else we deadlock…

Wait can cause deadlock

At least one condition must be true

Must use notify correctly

Alternative Termination of Mailbox Example

- Threads
  - Writer: writes items into the mailbox
  - Reader: reads items from the mailbox
- When writer has no more items
  - Interrupts reader
  - Exits

Example: Mailbox Writer

- After a fixed number MAX of messages, the writer interrupts the reader and returns

```
Thread reader_thread ; // the reader thread
public void run() {
  int x = 0 ;
  try {
    while (!Thread.interrupted() & x <= MAX) {
      if (x < MAX) {
        m.write(x) ;
      } else {
        reader_thread.interrupt() ; break ; }
      x++ ;
    }
  } catch (InterruptedException e) {} ;
  System.out.println("Writer thread completed") ;
}
```

Static method

Ask other thread to end

Example: Mailbox Reader

- The interrupt status is tested on each iteration
- Interrupt is caught if the thread is waiting

```
public void run() {
  int x = 0 ;
    try {
      while (!Thread.interrupted()) {
        x = m.read() ;
        System.out.println("Read " + x) ;
      }
    } catch (InterruptedException e) {} ;
    System.out.println("Reader thread completed") ;
}
```

Outside loop

Example: Mailbox

- Mailbox read and write throw the exception

```
public synchronized void write(int message)
                                        throws InterruptedException {
    while (! empty) { wait() ; }
    content = message ;
    empty = false ;  // the mailbox is now full
    notifyAll() ;
}

public synchronized int read() throws InterruptedException {
    while (empty) { wait() ; }
    empty = true ;  // the mailbox is now empty
    notifyAll() ;
    return content ;
}
```

Thrown by wait

Summary

- Need locks to avoid **race conditions**
- Locks introduce '**deadlock**'
  - New kind of error
  - Testing is no good for finding deadlock
- Locks introduce need for **co-operative thread termination**
  - Interrupt wait and sleep
  - Test the 'interrupt' status
  - Catch interrupts
- **wait()** and **notify()** are used to
  - Wait for a condition inside a synchronized method
  - Without block access to other threads