

Sistemi Operativi I

Corso di Laurea in Informatica
2024-2025



SAPIENZA
UNIVERSITÀ DI ROMA

Gabriele Tolomei

Dipartimento di Informatica

Sapienza Università di Roma

tolomei@di.uniroma1.it

What's Wrong with Semaphores?

- Not easy to get the meaning of waiting/signaling on a semaphore
- They are essentially shared global variables
- There is no direct connection between the semaphore and the data which the semaphore controls access to
- They serve multiple purposes (e.g., mutex, scheduling constraints, etc.)
- Their correctness depends on the programmer's ability

What's Wrong with Semaphores?

- Not easy to get the meaning of waiting/signaling on a semaphore
- They are essentially shared global variables
- There is no direct connection between the semaphore and the data which the semaphore controls access to
- They serve multiple purposes (e.g., mutex, scheduling constraints, etc.)
- Their correctness depends on the programmer's ability

Solution: Use a higher level primitive called **monitors**

What is a Monitor?

- A monitor is a programming language construct that controls access to shared data

What is a Monitor?

- A monitor is a programming language construct that controls access to shared data
- Similar to a (Java/C++) class that embodies all together: **data**, **operations**, and **synchronization**

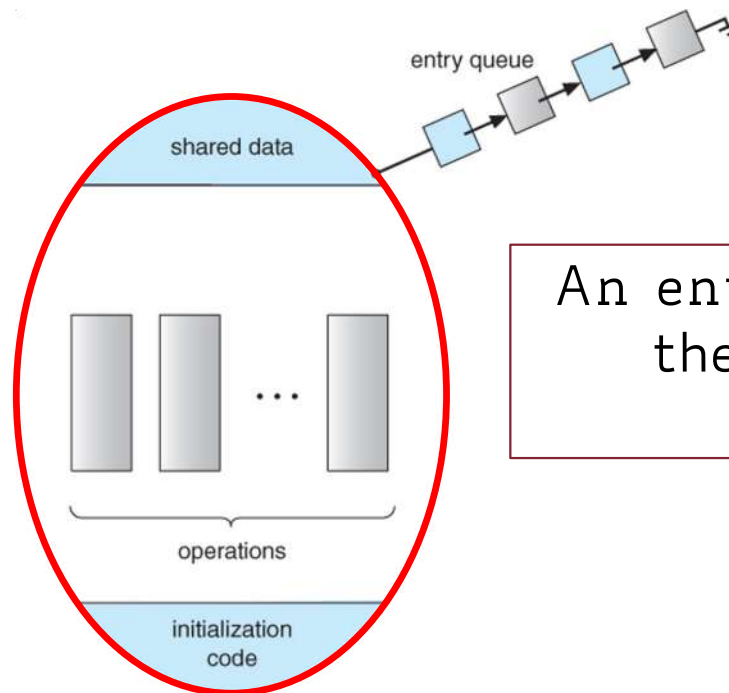
What is a Monitor?

- A monitor is a programming language construct that controls access to shared data
- Similar to a (Java/C++) class that embodies all together: **data**, **operations**, and **synchronization**
- Synchronization code added by compiler, enforced at runtime

What is a Monitor?

- Unlike classes, monitors:
 - guarantee mutual exclusion, i.e., only one thread may execute a monitor's method at a time
 - require all data to be private

Monitor: A Schematic Overview



An entry queue of processes waiting their turn to execute monitor operations (methods)

Monitor: A Formal Definition

- Defines a lock and zero or more **condition variables** for managing concurrent access to shared data

Monitor: A Formal Definition

- Defines a lock and zero or more **condition variables** for managing concurrent access to shared data
- Uses the lock to ensure that only a single thread is active within the monitor at any time

Monitor: A Formal Definition

- Defines a lock and zero or more **condition variables** for managing concurrent access to shared data
- Uses the lock to ensure that only a single thread is active within the monitor at any time
- The lock provides of course mutual exclusion for shared data

Monitor: Java Implementation Example

- It is straightforward to turn a Java class into a monitor by just:
 - Making all the data private
 - Making all methods (or non-private ones) **synchronized**

Monitor: Java Implementation Example

- It is straightforward to turn a Java class into a monitor by just:
 - Making all the data private
 - Making all methods (or non-private ones) `synchronized`
- The `synchronized` keyword indicates the method is subject to mutual exclusion

Monitor: Java Implementation Example

```
class Queue {  
    ...  
    private ArrayList<Item> data;  
    ...  
  
    public void synchronized add(Item i) {  
        data.add(i);  
    }  
  
    public Item synchronized remove() {  
        if (!data.isEmpty()) {  
            Item i = data.remove(0);  
            return i;  
        }  
    }  
}
```

Condition Variables

- In the previous example, the `remove()` method should wait until something is available on the queue

Condition Variables

- In the previous example, the `remove()` method should wait until something is available on the queue
 - Intuitively, the thread should sleep inside the critical section

Condition Variables

- In the previous example, the `remove()` method should wait until something is available on the queue
 - Intuitively, the thread should sleep inside the critical section
 - But if the thread sleeps while still holding a lock then no other threads can access the queue, add an item to it, and eventually wake up the sleeping thread

Condition Variables

- In the previous example, the `remove()` method should wait until something is available on the queue
 - Intuitively, the thread should sleep inside the critical section
 - But if the thread sleeps while still holding a lock then no other threads can access the queue, add an item to it, and eventually wake up the sleeping thread
 - **Deadlock** (more on this later...)

Condition Variables

- Solution: `condition variables`

Condition Variables

- Solution: `condition variables`
 - Conceptually a queue of threads, associated with a lock, on which a thread may wait for some condition to become true

Condition Variables

- Solution: **condition variables**
 - Conceptually a queue of threads, associated with a lock, on which a thread may wait for some condition to become true
 - Enable a thread to sleep **within** a critical section

Condition Variables

- Solution: **condition variables**
 - Conceptually a queue of threads, associated with a lock, on which a thread may wait for some condition to become true
 - Enable a thread to sleep **within** a critical section
 - Any lock held by the thread is **atomically** released before going to sleep

Condition Variables: Operations

- Each condition variable supports 3 operations:

Condition Variables: Operations

- Each condition variable supports 3 operations:
 - `wait` → release lock and go to sleep atomically (queue of waiters)

Condition Variables: Operations

- Each condition variable supports 3 operations:
 - `wait` → release lock and go to sleep atomically (queue of waiters)
 - `signal` → wake up a waiting thread if one exists, otherwise it does nothing

Condition Variables: Operations

- Each condition variable supports 3 operations:
 - `wait` → release lock and go to sleep atomically (queue of waiters)
 - `signal` → wake up a waiting thread if one exists, otherwise it does nothing
 - `broadcast` → wake up all waiting threads

Condition Variables: Operations

- Each condition variable supports 3 operations:
 - **wait** → release lock and go to sleep atomically (queue of waiters)
 - **signal** → wake up a waiting thread if one exists, otherwise it does nothing
 - **broadcast** → wake up all waiting threads
- **Rule:** thread must hold the lock when doing condition variable operations

Condition Variables: Operations

- Each condition variable supports 3 operations:
 - **wait** → release lock and go to sleep atomically (queue of waiters)
 - **signal** → wake up a waiting thread if one exists, otherwise it does nothing
 - **broadcast** → wake up all waiting threads
- **Rule:** thread must hold the lock when doing condition variable operations
- **Note:** condition variables are not boolean objects!

Condition Variables in Java

- Use `wait()` to give up the lock

Condition Variables in Java

- Use `wait()` to give up the lock
- Use `notify()` to signal that the condition a thread is waiting on is satisfied

Condition Variables in Java

- Use `wait()` to give up the lock
- Use `notify()` to signal that the condition a thread is waiting on is satisfied
- Use `notifyAll()` to wake up all waiting threads

Condition Variables in Java

- Use `wait()` to give up the lock
- Use `notify()` to signal that the condition a thread is waiting on is satisfied
- Use `notifyAll()` to wake up all waiting threads
- Concretely, one condition variable per object

Monitor: Java Implementation Example

```
class Queue {  
    ...  
    private ArrayList<Item> data;  
    ...  
  
    public void synchronized add(Item i) {  
        data.add(i);  
        notify();  
    }  
  
    public Item synchronized remove() {  
        while (data.isEmpty()) {  
            wait(); // give up the lock and sleep  
        }  
        Item i = data.remove(0);  
        return i;  
    }  
}
```

Condition Variables != Semaphores

- Same operations yet entirely different semantics

Condition Variables != Semaphores

- Same operations yet entirely different semantics
- Access to the monitor is controlled by a lock

Condition Variables != Semaphores

- Same operations yet entirely different semantics
- Access to the monitor is controlled by a lock
- `wait()` blocks the calling thread, and gives up the lock
 - to call `wait()`, the thread has to be in the monitor (hence, it has the lock!)
 - on a semaphore, `wait()` just blocks the thread on the queue

Condition Variables != Semaphores

- Same operations yet entirely different semantics
- Access to the monitor is controlled by a lock
- `wait()` blocks the calling thread, and gives up the lock
 - to call `wait()`, the thread has to be in the monitor (hence, it has the lock!)
 - on a semaphore, `wait()` just blocks the thread on the queue
- `signal()` causes a waiting thread to wake up
 - If there is no waiting thread, the signal is lost though!
 - on a semaphore, signal increases the counter, allowing future entry even if no thread is currently waiting

signal(): Mesa- vs. Hoare-style

- **Mesa-style** (Nachos, Java, and most real OSs)
 - The signaling thread places a waiter on the ready queue, but signaler continues inside monitor
 - Condition is not necessarily true when waiter runs again
 - Returning from **wait()** is only a hint that something changed
 - Must re-check the conditional case

signal(): Mesa- vs. Hoare-style

- Hoare-style (most textbooks)
 - The signaling thread immediately switches to a waiting thread
 - The condition that the waiter was anticipating is guaranteed to hold when waiter executes

Mesa vs. Hoare Monitors

- Mesa-style

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


Mesa vs. Hoare Monitors

- Mesa-style

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

- Hoare-style

```
if (empty) {  
    wait(condition);  
}
```

Mesa vs. Hoare Monitors

- Mesa-style

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

Easier to use and more efficient

- Hoare-style

```
if (empty) {  
    wait(condition);  
}
```

Easier to reason about the program's behaviour

Mesa vs. Hoare

Mesa

```
class Queue {  
    ...  
    private ArrayList<Item> data;  
    ...  
  
    public void synchronized add(Item i) {  
        data.add(i);  
        notify();  
    }  
  
    public Item synchronized remove() {  
        while (data.isEmpty()) {  
            wait(); // give up the lock and sleep  
        }  
        Item i = data.remove(0);  
        return i;  
    }  
}
```

The waiting thread may need to wait again after it is awakened, because some other thread could grab the lock and remove the item before it gets to run

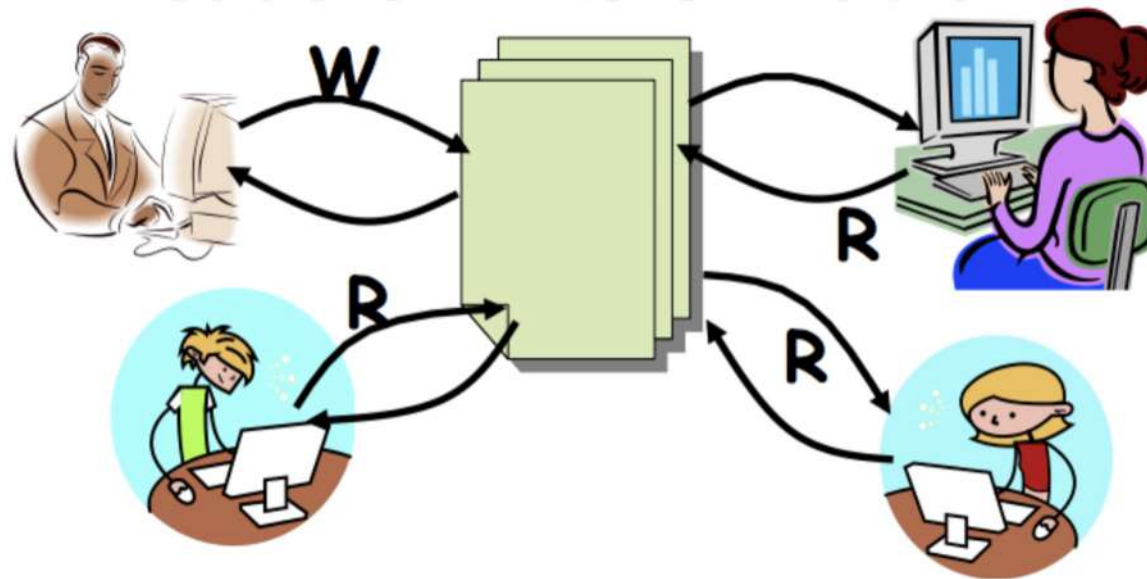
Hoare

```
class Queue {  
    ...  
    private ArrayList<Item> data;  
    ...  
  
    public void synchronized add(Item i) {  
        data.add(i);  
        notify();  
    }  
  
    public Item synchronized remove() {  
        if (data.isEmpty()) {  
            wait(); // give up the lock and sleep  
        }  
        Item i = data.remove(0);  
        return i;  
    }  
}
```

The waiting thread runs immediately after an item is added to the queue

Readers-Writers Problem

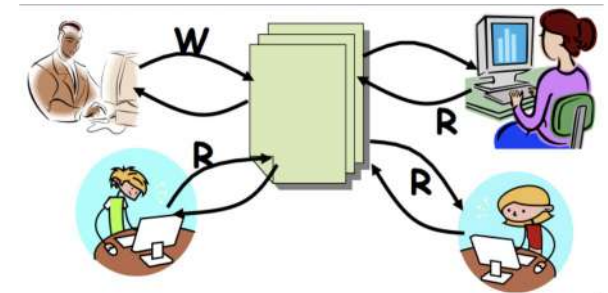
Motivation: Consider a shared database system
(more generally, any shared resource)



Readers-Writers Problem

Two classes of users:

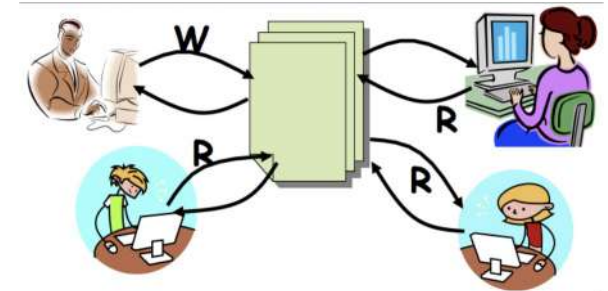
- **Readers** → never modify the DB



Readers-Writers Problem

Two classes of users:

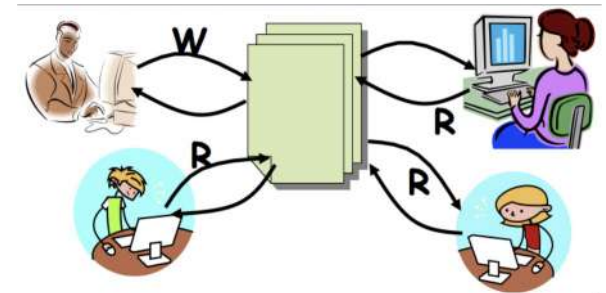
- **Readers** → never modify the DB
- **Writers** → read and modify the DB



Readers-Writers Problem

Simplest solution:

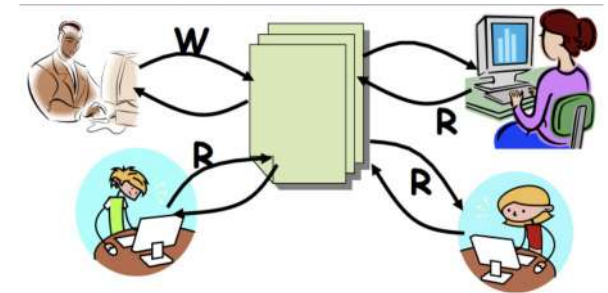
- Use a single lock on the data object for each operation



Readers-Writers Problem

Simplest solution:

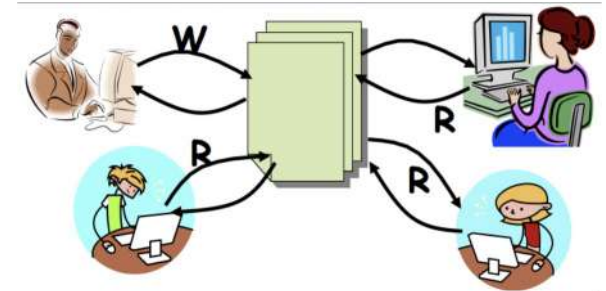
- Use a single lock on the data object for each operation
- May be too restrictive!



Readers-Writers Problem

Simplest solution:

- Use a single lock on the data object for each operation
- May be too restrictive!

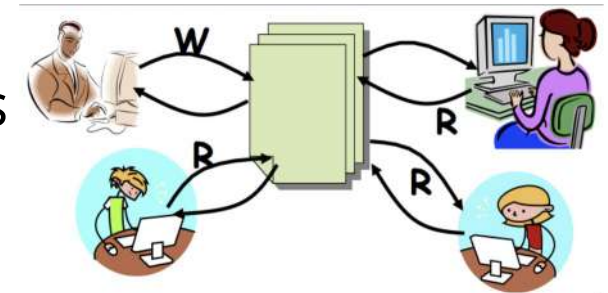


Only one writer at a time but, possibly, multiple readers

Readers-Writers Problem

Constraints:

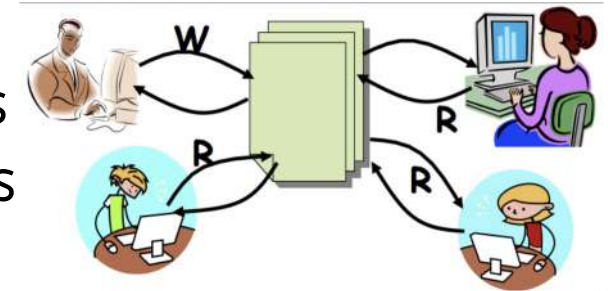
- Readers can access DB when no writers



Readers-Writers Problem

Constraints:

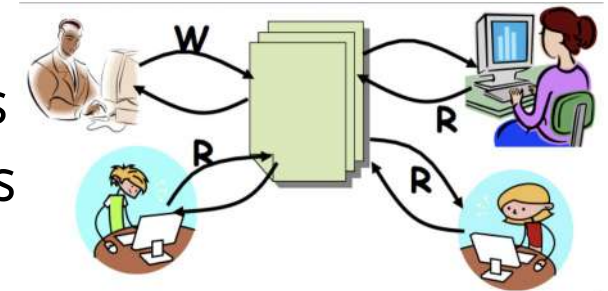
- Readers can access DB when no writers
- Writers can access DB when no readers or writers



Readers-Writers Problem

Constraints:

- Readers can access DB when no writers
- Writers can access DB when no readers or writers
- Only one thread manipulates state variables at a time



Readers-Writers Problem

- 2 variations of the problem depending on whether priority is on readers or writers:

Readers-Writers Problem

- 2 variations of the problem depending on whether priority is on readers or writers:
 - first readers-writers problem (priority to the readers)

Readers-Writers Problem

- 2 variations of the problem depending on whether priority is on readers or writers:
 - first readers-writers problem (priority to the readers)
 - second readers-writers problem (priority to the writers)

First Readers-Writers Problem

- Priority to the readers
- If a reader wants access to the data, and there is not already a writer accessing it, then access is granted to the reader
- Possible starvation of the writers, as there could always be more readers coming along to access the data

Second Readers-Writers Problem

- Priority to the writers
- When a writer wants access to the data it jumps to the head of the queue
- Possible starvation of the readers, as they are all blocked as long as there are writers

Readers-Writers in Java Using Lock

Readers-Writers in Java Using Monitors

Summary

- 3 synchronization primitives:

Summary

- 3 synchronization primitives:
 - Locks → the simplest one

Summary

- 3 synchronization primitives:
 - Locks → the simplest one
 - Semaphores → a generalization of locks

Summary

- 3 synchronization primitives:
 - Locks → the simplest one
 - Semaphores → a generalization of locks
 - Monitors → highest-level primitives that wrap methods with mutex

Summary

- 3 synchronization primitives:
 - Locks → the simplest one
 - Semaphores → a generalization of locks
 - Monitors → highest-level primitives that wrap methods with mutex
- Examples of typical synchronization problems:

Summary

- 3 synchronization primitives:
 - Locks → the simplest one
 - Semaphores → a generalization of locks
 - Monitors → highest-level primitives that wrap methods with mutex
- Examples of typical synchronization problems:
 - Producers-Consumers

Summary

- 3 synchronization primitives:
 - Locks → the simplest one
 - Semaphores → a generalization of locks
 - Monitors → highest-level primitives that wrap methods with mutex
- Examples of typical synchronization problems:
 - Producers-Consumers
 - Readers-Writers