Systems and Networking I

Applied Computer Science and Artificial Intelligence

2023-2024

Gabriele Tolomei

Dipartimento di Informatica

Sapienza Università di Roma

tolomei@di.uniroma1.it

SAPIENZA
UNIVERSITÀ DI ROMA

Recap from Last Lecture

- Concurrent accesses to shared resources by multiple cooperating processes/threads can lead to unexpected behavior

Recap from Last Lecture

- Concurrent accesses to shared resources by multiple cooperating processes/threads can lead to unexpected behavior
- Process/Thread cooperation must guarantee consistency of any shared data/resource, **regardless of CPU scheduling**

Recap from Last Lecture

- Concurrent accesses to shared resources by multiple cooperating processes/threads can lead to unexpected behavior
- Process/Thread cooperation must guarantee consistency of any shared data/resource, **regardless of CPU scheduling**
- Maintaining shared data consistency requires mechanisms to ensure synchronized execution of critical sections by processes/threads

Recap from Last Lecture

- Concurrent accesses to shared resources by multiple cooperating processes/threads can lead to unexpected behavior
- Process/Thread cooperation must guarantee consistency of any shared data/resource, **regardless of CPU scheduling**
- Maintaining shared data consistency requires mechanisms to ensure synchronized execution of critical sections by processes/threads
- Critical sections are specific pieces of code which contain shared resources that need to be "protected"

Recap from Last Lecture

We need to have appropriate "tools" (i.e., primitive constructs) provided by programming languages used as atomic building blocks for synchronization

- **Locks** → At each time, only one process holds a lock, executes its critical section, and finally releases the lock
- **Semaphores** → A generalization of locks
- **Monitors** → To connect shared data to synchronization primitives

Require some HW support and waiting

Locks

- Provide **mutual exclusion** to shared data using 2 atomic primitives:

Locks

- Provide **mutual exclusion** to shared data using 2 atomic primitives:
  - Lock.acquire() → wait until the lock is free, then grab it

Locks

- Provide **mutual exclusion** to shared data using 2 atomic primitives:
  - Lock.acquire() → wait until the lock is free, then grab it
  - Lock.release() → unlock and wake up any thread waiting in acquire()

Locks

- Provide **mutual exclusion** to shared data using 2 atomic primitives:
  - Lock.acquire() → wait until the lock is free, then grab it
  - Lock.release() → unlock and wake up any thread waiting in acquire()
- Rules for using a lock:
  - Always acquire the lock **before** accessing shared data
  - Always release the lock **after** finishing with shared data
  - Lock must be **initially free**

Locks

- Provide **mutual exclusion** to shared data using 2 atomic primitives:
  - Lock.acquire() → wait until the lock is free, then grab it
  - Lock.release() → unlock and wake up any thread waiting in acquire()
- Rules for using a lock:
  - Always acquire the lock **before** accessing shared data
  - Always release the lock **after** finishing with shared data
  - Lock must be **initially free**
- Only one process/thread can acquire the lock, others will wait!

Too Much Milk: Solution Using Locks

Use lock primitives

```
# Thread Bob

Lock.acquire()

if (!milk):
    buy_milk()

Lock.release()
```

```
# Thread Carla

Lock.acquire()

if (!milk):
    buy_milk()

Lock.release()
```

This solution is clean and symmetric

Too Much Milk: Solution Using Locks

Use lock primitives

```
# Thread Bob

Lock.acquire()

if (!milk):
    buy_milk()

Lock.release()
```

```
# Thread Carla

Lock.acquire()

if (!milk):
    buy_milk()

Lock.release()
```

This solution is clean and symmetric

Q: How do we make acquire() and release() atomic?

HW Support for Synchronization

Implementing high-level synchronization primitives requires low-level hardware support

High-level atomic operations (SW)	lock, monitor, semaphore, send/receive
Low-level atomic operations (HW)	disabling interrupts, atomic instructions (test&set)

HW Support for Synchronization

Implementing high-level synchronization primitives requires low-level hardware support

High-level atomic operations (SW)	lock, monitor, semaphore, send/receive
Low-level atomic operations (HW)	disabling interrupts, atomic instructions (test&set)

Implementing Locks: Disabling Interrupts

- The reason why we care of synchronization is because context switches may occur **unexpectedly**

Implementing Locks: Disabling Interrupts

- The reason why we care of synchronization is because context switches may occur **unexpectedly**
- The CPU scheduler takes control due to 2 possible situations:

Implementing Locks: Disabling Interrupts

- The reason why we care of synchronization is because context switches may occur **unexpectedly**
- The CPU scheduler takes control due to 2 possible situations:
  - **internal events** → the current thread voluntarily relinquishes control of the CPU (e.g., via an I/O system call)

Implementing Locks: Disabling Interrupts

- The reason why we care of synchronization is because context switches may occur **unexpectedly**
- The CPU scheduler takes control due to 2 possible situations:
  - **internal events** → the current thread voluntarily relinquishes control of the CPU (e.g., via an I/O system call)
  - **external events** → interrupts (e.g., time slice) cause the scheduler to take over the currently running thread

Implementing Locks: Disabling Interrupts

- The reason why we care of synchronization is because context switches may occur **unexpectedly**
- The CPU scheduler takes control due to 2 possible situations:
  - **internal events** → the current thread voluntarily relinquishes control of the CPU (e.g., via an I/O system call)
  - **external events** → interrupts (e.g., time slice) cause the scheduler to take over the currently running thread

We want to prevent the CPU scheduler to take control **while** an acquire() operation is ongoing

Implementing Locks: Disabling Interrupts

- On single-CPU systems, we can prevent the scheduler to take over by:

Implementing Locks: Disabling Interrupts

- On single-CPU systems, we can prevent the scheduler to take over by:
  - **internal events** → enforcing threads not to request any I/O operation during a critical section

Implementing Locks: Disabling Interrupts

- On single-CPU systems, we can prevent the scheduler to take over by:
  - **internal events** → enforcing threads not to request any I/O operation during a critical section
  - **external event** → disabling interrupts (i.e., telling the HW to delay the handling of any external event until the current thread is done with the critical section)

Implementing Locks: Disabling Interrupts

- On single-CPU systems, we can prevent the scheduler to take over by:
  - **internal events** → enforcing threads not to request any I/O operation during a critical section
  - **external event** → disabling interrupts (i.e., telling the HW to delay the handling of any external event until the current thread is done with the critical section)

> We cover all the possible cases where the current thread might loose control of the CPU, either voluntarily (due to internal events) or involuntarily (due to external events)

Implementing Locks: Disabling Interrupts

```
Class Lock {
  public void acquire(Thread t);
  public void release();
  private int value; // 0=FREE, 1=BUSY
  private Queue q;

  Lock() {
    // lock is initially FREE
    this.value = 0;
    this.q = null;
  }
}
```

Implementing Locks: Disabling Interrupts

```
Class Lock {
  public void acquire(Thread t);
  public void release();
  private int value; // 0=FREE, 1=BUSY
  private Queue q;

  Lock() {
    // lock is initially FREE
    this.value = 0;
    this.q = null;
  }
}
```

```
public void acquire(Thread t) {
  disable_interrupts();
  if(this.value) { // lock is held by someone
    q.push(t); // add t to waiting queue
    t.sleep(); // put t to sleep
  }
  else {
    this.value = 1;
  }
  enable_interrupts();
}
```

Implementing Locks: Disabling Interrupts

```
Class Lock {
  public void acquire(Thread t);
  public void release();
  private int value; // 0=FREE, 1=BUSY
  private Queue q;

  Lock() {
    // lock is initially FREE
    this.value = 0;
    this.q = null;
  }
}
```

```
public void acquire(Thread t) {
  disable_interrupts();
  if(this.value) { // lock is held by someone
    q.push(t); // add t to waiting queue
    t.sleep(); // put t to sleep
  }
  else {
    this.value = 1;
  }
  enable_interrupts();
}
```

```
public void release() {
  disable_interrupts();
  if(!q.is_empty()) {
    t = q.pop(); // extract a waiting thread from q
    push_onto_ready_queue(t); // put t on ready queue
  }
  else {
    this.value = 0;
  }
  enable_interrupts();
}
```

Implementing Locks: Disabling Interrupts

```
Class Lock {
  public void acquire(Thread t);
  public void release();
  private int value; // 0=FREE, 1=BUSY
  private Queue q;

  Lock() {
    // lock is initially FREE
    this.value = 0;
    this.q = null;
  }
}
```

We need both acquire and release being implemented as system calls

```
public void acquire(Thread t) {
  disable_interrupts();
  if(this.value) { // lock is held by someone
    q.push(t); // add t to waiting queue
    t.sleep(); // put t to sleep
  }
  else {
    this.value = 1;
  }
  enable_interrupts();
}
```

```
public void release() {
  disable_interrupts();
  if(!q.is_empty()) {
    t = q.pop(); // extract a waiting thread from q
    push_onto_ready_queue(t); // put t on ready queue
  }
  else {
    this.value = 0;
  }
  enable_interrupts();
}
```

Implementing Locks: Disabling Interrupts

```
Class Lock {
  public void acquire(Thread t);
  public void release();
  private int value; // 0=FREE, 1=BUSY
  private Queue q;

  Lock() {
    // lock is initially FREE
    this.value = 0;
    this.q = null;
  }
}
```

We need both acquire and release being implemented as system calls

Why?

```
public void acquire(Thread t) {
  disable_interrupts();
  if(this.value) { // lock is held by someone
    q.push(t); // add t to waiting queue
    t.sleep(); // put t to sleep
  }
  else {
    this.value = 1;
  }
  enable_interrupts();
}
```

```
public void release() {
  disable_interrupts();
  if(!q.is_empty()) {
    t = q.pop(); // extract a waiting thread from q
    push_onto_ready_queue(t); // put t on ready queue
  }
  else {
    this.value = 0;
  }
  enable_interrupts();
}
```

HW Support for Synchronization

Implementing high-level synchronization primitives requires low-level hardware support

High-level atomic operations (SW)	lock, monitor, semaphore, send/receive
Low-level atomic operations (HW)	disabling interrupts, atomic instructions (test&set)

Implementing Locks: Atomic Instructions

- An atomic **read-modify-write** instruction reads a value from memory into a register and writes a new value in one shot!

Implementing Locks: Atomic Instructions

- An atomic **read-modify-write** instruction reads a value from memory into a register and writes a new value in one shot!
  - On a **uniprocessor** → straightforward to implement adding a new instruction

Implementing Locks: Atomic Instructions

- An atomic **read-modify-write** instruction reads a value from memory into a register and writes a new value in one shot!
  - On a **uniprocessor** → straightforward to implement adding a new instruction
  - On a **multiprocessor** → the processor issuing the instruction must also be able to invalidate any copies of the value other processes may have in their cache

Implementing Locks: Atomic Instructions

- Examples:
  - test&set → writes (sets) 1 to a memory location and returns its old value

Implementing Locks: Atomic Instructions

- Examples:
  - test&set → writes (sets) 1 to a memory location and returns its old value
  - fetch&add → increments the contents of a memory location by a specified value

Implementing Locks: Atomic Instructions

- Examples:
  - test&set → writes (sets) 1 to a memory location and returns its old value
  - fetch&add → increments the contents of a memory location by a specified value
  - compare&swap (x86: compare&exchange) → compares the content of a memory location with a given value and, if they are the same, modifies the content of that location to a new value

Implementing Locks: Atomic Instructions

- Examples:
  - test&set → writes (sets) 1 to a memory location and returns its old value
  - fetch&add → increments the contents of a memory location by a specified value
  - compare&swap (x86: compare&exchange) → compares the content of a memory location with a given value and, if they are the same, modifies the content of that location to a new value

Implementing Locks: test&set

```
Class Lock {
  public void acquire();
  public void release();
  private int value;

  Lock() {
    // lock is initially free
    this.value = 0;
  }
}
```

Implementing Locks: test&set

```
Class Lock {
  public void acquire();
  public void release();
  private int value;

  Lock() {
    // lock is initially free
    this.value = 0;
  }
}
```

```
public void acquire() {
  while(test&set(this.value) == 1) {
    // while busy do nothing
  }
}
```

Implementing Locks: test&set

```
Class Lock {
  public void acquire();
  public void release();
  private int value;

  Lock() {
    // lock is initially free
    this.value = 0;
  }
}
```

```
public void acquire() {
  while(test&set(this.value) == 1) {
    // while busy do nothing
  }
}
```

```
public void release() {
  this.value = 0;
}
```

Implementing Locks: test&set

```
Class Lock {
  public void acquire();
  public void release();
  private int value;

  Lock() {
    // lock is initially free
    this.value = 0;
  }
}
```

```
public void acquire() {
  while(test&set(this.value) == 1) {
    // while busy do nothing
  }
}
```

```
public void release() {
  this.value = 0;
}
```

Case 1: if lock is free (value = 0) test&set(value) will read 0, set it to 1 and return 0

Implementing Locks: test&set

```
Class Lock {
  public void acquire();
  public void release();
  private int value;

  Lock() {
    // lock is initially free
    this.value = 0;
  }
}
```

```
public void acquire() {
  while(test&set(this.value) == 1) {
    // while busy do nothing
  }
}
```

```
public void release() {
  this.value = 0;
}
```

Case 1: if lock is free (value = 0) test&set(value) will read 0, set it to 1 and return 0

The lock is now busy, the boolean expression in the while guard is false and acquire terminates

Implementing Locks: test&set

```
Class Lock {
  public void acquire();
  public void release();
  private int value;

  Lock() {
    // lock is initially free
    this.value = 0;
  }
}
```

```
public void acquire() {
  while(test&set(this.value) == 1) {
    // while busy do nothing
  }
}
```

```
public void release() {
  this.value = 0;
}
```

Case 2: if lock is busy (value = 1) test&set(value) will read 1, set it to 1 and return 1

Implementing Locks: test&set

```
Class Lock {
  public void acquire();
  public void release();
  private int value;

  Lock() {
    // lock is initially free
    this.value = 0;
  }
}
```

```
public void acquire() {
  while(test&set(this.value) == 1) {
    // while busy do nothing
  }
}
```

```
public void release() {
  this.value = 0;
}
```

Case 2: if lock is busy (value = 1) test&set(value) will read 1, set it to 1 and return 1

The lock is still busy, the boolean expression in the while guard is true and acquire continues to loop until release executes

Atomic Instructions: Any Issue?

```
public void acquire() {
  while(test&set(this.value) == 1) {
    // while busy do nothing
  }
}
```

- What's wrong with the above implementation?

Atomic Instructions: Any Issue?

```
public void acquire() {
  while(test&set(this.value) == 1) {
    // while busy do nothing
  }
}
```

- What's wrong with the above implementation?
  - What is the CPU doing?

Atomic Instructions: Any Issue?

```
public void acquire() {
  while(test&set(this.value) == 1) {
    // while busy do nothing
  }
}
```

busy waiting

- What's wrong with the above implementation?
  - What is the CPU doing?

Atomic Instructions: Any Issue?

```
public void acquire() {
  while(test&set(this.value) == 1) {
    // while busy do nothing
  }
}
```

- What's wrong with the above implementation?
  - What is the CPU doing?
  - What could happen to threads with different priorities waiting for the lock?

Atomic Instructions: Any Issue?

```
public void acquire() {
  while(test&set(this.value) == 1) {
    // while busy do nothing
  }
}
```

who is going to take the lock once released?

- What's wrong with the above implementation?
  - What is the CPU doing?
  - What could happen to threads with different priorities waiting for the lock?

Disabling Interrupts vs. Atomic Instructions

- 2 main problems with disabling interrupts:

Disabling Interrupts vs. Atomic Instructions

- 2 main problems with disabling interrupts:
  - overhead as it requires invoking the kernel

Disabling Interrupts vs. Atomic Instructions

- 2 main problems with disabling interrupts:
  - **overhead** as it requires invoking the kernel
  - **unfeasible** with mulitprocessor architectures

Disabling Interrupts vs. Atomic Instructions

- 2 main problems with disabling interrupts:
  - **overhead** as it requires invoking the kernel
  - **unfeasible** with mulitprocessor architectures
- 2 main problems with atomic instructions:

Disabling Interrupts vs. Atomic Instructions

- 2 main problems with disabling interrupts:
  - overhead as it requires invoking the kernel
  - unfeasible with mulitprocessor architectures
- 2 main problems with atomic instructions:
  - busy waiting

Disabling Interrupts vs. Atomic Instructions

- 2 main problems with disabling interrupts:
  - overhead as it requires invoking the kernel
  - unfeasible with mulitprocessor architectures
- 2 main problems with atomic instructions:
  - busy waiting
  - unfairness as there is no queue where threads wait for the lock to be released

Improving test&set To Reduce Busy Waiting

Can we implement locks with test&set without any busy-waiting?

Improving test&set To Reduce Busy Waiting

Can we implement locks with test&set without any busy-waiting?

```
Class Lock {
  public void acquire(Thread t);
  public void release();
  private int value;


  Lock() {
    // lock is initially free
    this.value = 0;
  }
}
```

Improving test&set To Reduce Busy Waiting

Can we implement locks with test&set without any busy-waiting?

```
Class Lock {
  public void acquire(Thread t);
  public void release();
  private int value;
  private int guard;
  private Queue q;

  Lock() {
    // lock is initially free
    this.value = 0;
  }
}
```

Improving test&set To Reduce Busy Waiting

Can we implement locks with test&set without any busy-waiting?

```
Class Lock {
  public void acquire(Thread t);
  public void release();
  private int value;
  private int guard;
  private Queue q;

  Lock() {
    // lock is initially free
    this.value = 0;
  }
}
```

```
public void acquire(Thread t) {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  if(this.value) {
    q.push(t);
    t.sleep_and_reset_guard_to_0();
  }
  else {
    this.value = 1;
    this.guard = 0;
  }
}
```

Improving test&set To Reduce Busy Waiting

Can we implement locks with test&set without any busy-waiting?

```
Class Lock {
  public void acquire(Thread t);
  public void release();
  private int value;
  private int guard;
  private Queue q;

  Lock() {
    // lock is initially free
    this.value = 0;
  }
}
```

```
public void acquire(Thread t) {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  if(this.value) {
    q.push(t);
    t.sleep_and_reset_guard_to_0();
  }
  else {
    this.value = 1;
    this.guard = 0;
  }
}
```

```
public void release() {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  if(!q.is_empty()) {
    t = q.pop();
    push_onto_ready_queue(t);
  }
  else {
    this.value = 0;
  }
  this.guard = 0;
}
```

Improving test&set To Reduce Busy Waiting

Can we implement locks with test&set without any busy-waiting?

```
Class Lock {
  public void acquire(Thread t);
  public void release();
  private int value;
  private int guard;
  private Queue q;

  Lock() {
    // lock is initially free
    this.value = 0;
  }
}
```

```
public void acquire(Thread t) {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  if(this.value) {
    q.push(t);
    t.sleep_and_reset_guard_to_0();
  }
  else {
    this.value = 1;
    this.guard = 0;
  }
}
```

```
public void release() {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  if(!q.is_empty()) {
    t = q.pop();
    push_onto_ready_queue(t);
  }
  else {
    this.value = 0;
  }
  this.guard = 0;
}
```

No, but we can minimize busy-waiting time by atomically checking the lock value and giving up the CPU if the lock is busy

Improving test&set To Reduce Busy Waiting

Can we implement locks with test&set without any busy-waiting?

```
Class Lock {
  public void acquire(Thread t);
  public void release();
  private int value;
  private int guard;
  private Queue q;

  Lock() {
    // lock is initially free
    this.value = 0;
  }
}
```

```
public void acquire(Thread t) {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  if(this.value) {
    q.push(t);
    t.sleep_and_reset_guard_to_0();
  }
  else {
    this.value = 1;
    this.guard = 0;
  }
}
```

```
public void release() {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  if(!q.is_empty()) {
    t = q.pop();
    push_onto_ready_queue(t);
  }
  else {
    this.value = 0;
  }
  this.guard = 0;
}
```

We can't totally get rid of busy-waiting but we can make it independent on how long is the critical section delimited by acquire and release

Locks: Wrap Up

- Synchronization primitives ensure that only one process/thread at a time executes in a critical section (**mutual exclusion**)
- Locks allow protection of critical sections by atomically testing and taking/releasing the access to a critical section
- Locks can be implemented leveraging some HW support:
  - **disabling interrupts** (can miss or delay important events)
  - **atomic instructions** (busy waiting/spinlock inefficient)

Higher-Level Synchronization Primitives

- More general synchronization mechanisms
  - Not only for safely accessing critical sections

Higher-Level Synchronization Primitives

- More general synchronization mechanisms
  - Not only for safely accessing critical sections
- 2 common high-level synchronization primitives:
  - Semaphores: binary (mutex) and counting

Higher-Level Synchronization Primitives

- More general synchronization mechanisms
  - Not only for safely accessing critical sections
- 2 common high-level synchronization primitives:
  - Semaphores: binary (mutex) and counting
  - Monitors: mutex and condition variables

Semaphores: Overview

- Another data structure that provides mutual exclusion to critical sections

Semaphores: Overview

- Another data structure that provides mutual exclusion to critical sections
- Can also play the role of an atomic counter

Semaphores: Overview

- Another data structure that provides mutual exclusion to critical sections
- Can also play the role of an atomic counter
- Generalization of locks invented by **Dijkstra** in 1965

Semaphores: Overview

- Another data structure that provides mutual exclusion to critical sections
- Can also play the role of an atomic counter
- Generalization of locks invented by **Dijkstra** in 1965
- Special type of (integer) variable that supports 2 **atomic operations**
  - wait() (also P()): decrement, block until semaphore is open
  - signal() (also V()): increment, allow another thread to enter

Blocking in Semaphores

- Associated with each semaphore is a queue of waiting processes/threads

Blocking in Semaphores

- Associated with each semaphore is a queue of waiting processes/threads
- When wait() is called by a thread:
  - If semaphore is open thread continues, otherwise thread blocks on queue

Blocking in Semaphores

- Associated with each semaphore is a queue of waiting processes/threads
- When wait() is called by a thread:
  - If semaphore is open thread continues, otherwise thread blocks on queue
- Then signal() opens the semaphore:
  - If a thread is waiting on the queue the thread is unblocked, whilst if no threads are waiting on the queue, the signal is remembered for the next thread

Blocking in Semaphores

- Associated with each semaphore is a queue of waiting processes/threads
- When wait() is called by a thread:
  - If semaphore is open thread continues, otherwise thread blocks on queue
- Then signal() opens the semaphore:
  - If a thread is waiting on the queue the thread is unblocked, whilst if no threads are waiting on the queue, the signal is remembered for the next thread
- In other words, signal() is stateful and has "history"

Semaphores: Types

- **Binary Semaphore** a.k.a. Mutex (same as a Lock)
  - Guarantees mutually exclusive access to a resource (i.e., only one process/thread executes in a critical section)
  - Its associated integer variable can only take 2 values: 0/1
  - Initialized to open (e.g., value = 1)

Semaphores: Types

- **Binary Semaphore** a.k.a. Mutex (same as a Lock)
  - Guarantees mutually exclusive access to a resource (i.e., only one process/thread executes in a critical section)
  - Its associated integer variable can only take 2 values: 0/1
  - Initialized to open (e.g., value = 1)
- **Counting Semaphore**
  - To manage multiple shared resources
  - The semaphore is initially set to the number of resources
  - A process can access to a resource as long as at least one is available

Semaphores: Key Ideas

```
// Semaphore S

S.wait(); // wait until S is available

<critical section>

S.signal(); notify other processes that S is open
```

Semaphores: Key Ideas

```
// Semaphore S

S.wait(); // wait until S is available

<critical section>

S.signal(); notify other processes that S is open
```

Each semaphore supports a queue of processes that are waiting to access the critical section (e.g., to buy milk)

Semaphores: Key Ideas

```
// Semaphore S

S.wait(); // wait until S is available

<critical section>

S.signal(); notify other processes that S is open
```

Each semaphore supports a queue of processes that are waiting to access the critical section (e.g., to buy milk)

If a process executes S.wait() and semaphore S is open (non-zero), it continues executing, otherwise the OS puts the process on the wait queue

Semaphores: Key Ideas

```
// Semaphore S

S.wait(); // wait until S is available

<critical section>

S.signal(); notify other processes that S is open
```

Each semaphore supports a queue of processes that are waiting to access the critical section (e.g., to buy milk)

If a process executes S.wait() and semaphore S is open (non-zero), it continues executing, otherwise the OS puts the process on the wait queue

A S.signal() unblocks one process on semaphore S's wait queue

Binary Semaphore: Example

"Too Much Milk" Using
Lock

```
# Thread Bob

Lock.acquire()

if (!milk):
    buy_milk()

Lock.release()
```

```
# Thread Carla

Lock.acquire()

if (!milk):
    buy_milk()

Lock.release()
```

Binary Semaphore: Example

"Too Much Milk" Using Lock		"Too Much Milk" Using Semaphore	
# Thread Bob	# Thread Carla	# Thread Bob	# Thread Carla

"Too Much Milk" Using Lock

```
# Thread Bob

Lock.acquire()

if (!milk):
    buy_milk()

Lock.release()
```

```
# Thread Carla

Lock.acquire()

if (!milk):
    buy_milk()

Lock.release()
```

"Too Much Milk" Using Semaphore

```
# Thread Bob

S.wait()

if (!milk):
    buy_milk()

S.signal()
```

```
# Thread Carla

S.wait()

if (!milk):
    buy_milk()

S.signal()
```

Binary Semaphore: Example

"Too Much Milk" Using Lock

```
# Thread Bob
Lock.acquire()
if (!milk):
    buy_milk()
Lock.release()
```

```
# Thread Carla
Lock.acquire()
if (!milk):
    buy_milk()
Lock.release()
```

"Too Much Milk" Using Semaphore

```
# Thread Bob
S.wait()
if (!milk):
    buy_milk()
S.signal()
```

```
# Thread Carla
S.wait()
if (!milk):
    buy_milk()
S.signal()
```

Semaphore: Implementation

```
Class Semaphore {
  public void wait(Thread t);
  public void signal();
  private int value;
  private int guard;
  private Queue q;

  Semaphore(int val) {
    // initialize semaphore
    // with val and empty queue
    this.value = val;
    this.q = null;
  }
}
```

Semaphore: Implementation

```
Class Semaphore {
  public void wait(Thread t);
  public void signal();
  private int value;
  private int guard;
  private Queue q;

  Semaphore(int val) {
    // initialize semaphore
    // with val and empty queue
    this.value = val;
    this.q = null;
  }
}
```

```
public void wait(Thread t) {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  this.value -= 1;
  if(this.value < 0) {
    q.push(t);
    t.sleep_and_reset_guard_to_0();
  }
  else {
    this.guard = 0;
  }
}
```

Semaphore: Implementation

```
Class Semaphore {
  public void wait(Thread t);
  public void signal();
  private int value;
  private int guard;
  private Queue q;

  Semaphore(int val) {
    // initialize semaphore
    // with val and empty queue
    this.value = val;
    this.q = null;
  }
}
```

```
public void wait(Thread t) {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  this.value -= 1;
  if(this.value < 0) {
    q.push(t);
    t.sleep_and_reset_guard_to_0();
  }
  else {
    this.guard = 0;
  }
}
```

```
public void signal() {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  this.value += 1;
  if(!q.isEmpty()) {
    t = q.pop();
    push_onto_ready_queue(t);
  }
  this.guard = 0;
}
```

Semaphore: Implementation

```
Class Semaphore {
  public void wait(Thread t);
  public void signal();
  private int value;
  private int guard;
  private Queue q;

  Semaphore(int val) {
    // initialize semaphore
    // with val and empty queue
    this.value = val;
    this.q = null;
  }
}
```

```
public void wait(Thread t) {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  this.value -= 1;
  if(this.value < 0) {
    q.push(t);
    t.sleep_and_reset_guard_to_0();
  }
  else {
    this.guard = 0;
  }
}
```

```
public void signal() {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  this.value += 1;
  if(!q.isEmpty()) {        // this.value <= 0
    t = q.pop();
    push_onto_ready_queue(t);
  }
  this.guard = 0;
}
```

Semaphore: Implementation

```
Class Semaphore {
  public void wait(Thread t);
  public void signal();
  private int value;
  private int guard;
  private Queue q;

  Semaphore(int val) {
    // initialize semaphore
    // with val and empty queue
    this.value = val;
    this.q = null;
  }
}
```

```
public void wait(Thread t) {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  this.value -= 1;
  if(this.value < 0) {
    q.push(t);
    t.sleep_and_reset_guard_to_0();
  }
  else {
    t.guard = 0;
  }
}
```

```
public void signal() {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  this.value += 1;
  if(!q.isEmpty()) {      // this.value <= 0
    t = q.pop();
    push_onto_ready_queue(t);
  }
  this.guard = 0;
}
```

wait() and signal() are of course atomic!

either interrupts must be disabled or test&set used

Semaphore: Example

S = 2

A

```
S.wait()
S.wait()
S.signal()
S.signal()
```

B

```
S.wait()
S.signal()
```

Semaphore: Example

S = 2

A

```
S.wait()
S.wait()
S.signal()
S.signal()
```

B

```
S.wait()
S.signal()
```

A possible execution flow

S (value)	Queue	A	B
2	Ø	ready to exec	ready to exec

Semaphore: Example

A

```
S.wait()
S.wait()
S.signal()
S.signal()
```

B

```
S.wait()
S.signal()
```

A possible execution flow

	S (value)	Queue	A	B
	2	∅	ready to exec	ready to exec
A: S.wait()				

Semaphore: Example

A

```
S.wait()
S.wait()
S.signal()
S.signal()
```

B

```
S.wait()
S.signal()
```

A possible execution flow

	S (value)	Queue	A	B
	2	∅	ready to exec	ready to exec
A: S.wait()	1	∅	ready to exec	ready to exec

Semaphore: Example

A

```
S.wait()
S.wait()
S.signal()
S.signal()
```

B

```
S.wait()
S.signal()
```

A possible execution flow

	S (value)	Queue	A	B
	2	∅	ready to exec	ready to exec
A: S.wait()	1	∅	ready to exec	ready to exec
B: S.wait()				

Semaphore: Example

A

```
S.wait()
S.wait()
S.signal()
S.signal()
```

B

```
S.wait()
S.signal()
```

A possible execution flow

	S (value)	Queue	A	B
	2	∅	ready to exec	ready to exec
A: S.wait()	1	∅	ready to exec	ready to exec
B: S.wait()	0	∅	ready to exec	ready to exec

Semaphore: Example

A

```
S.wait()
S.wait()
S.signal()
S.signal()
```

B

```
S.wait()
S.signal()
```

A possible execution flow	S (value)	Queue	A	B
	2	∅	ready to exec	ready to exec
A: S.wait()	1	∅	ready to exec	ready to exec
B: S.wait()	0	∅	ready to exec	ready to exec
A: S.wait()				

Semaphore: Example

A

S.wait()
S.wait()
S.signal()
S.signal()

B

S.wait()
S.signal()

A possible execution flow

	S (value)	Queue	A	B
	2	∅	ready to exec	ready to exec
A: S.wait()	1	∅	ready to exec	ready to exec
B: S.wait()	0	∅	ready to exec	ready to exec
A: S.wait()	-1	A	blocked	ready to exec

Semaphore: Example

A

```
S.wait()
S.wait()
S.signal()
S.signal()
```

B

```
S.wait()
S.signal()
```

A possible execution flow

	S (value)	Queue	A	B
	2	Ø	ready to exec	ready to exec
A: S.wait()	1	Ø	ready to exec	ready to exec
B: S.wait()	0	Ø	ready to exec	ready to exec
A: S.wait()	-1	A	blocked	ready to exec
B: S.signal()				

Semaphore: Example

A

```
S.wait()
S.wait()
S.signal()
S.signal()
```

B

```
S.wait()
S.signal()
```

A possible execution flow

	S (value)	Queue	A	B
	2	∅	ready to exec	ready to exec
A: S.wait()	1	∅	ready to exec	ready to exec
B: S.wait()	0	∅	ready to exec	ready to exec
A: S.wait()	-1	A	blocked	ready to exec
B: S.signal()	0	∅	ready to exec	ready to exec

Semaphore: Example

A

```
S.wait()
S.wait()
S.signal()
S.signal()
```

B

```
S.wait()
S.signal()
```

A possible execution flow	S (value)	Queue	A	B
	2	∅	ready to exec	ready to exec
A: S.wait()	1	∅	ready to exec	ready to exec
B: S.wait()	0	∅	ready to exec	ready to exec
A: S.wait()	-1	A	blocked	ready to exec
B: S.signal()	0	∅	ready to exec	ready to exec
A: S.signal()	1	∅	ready to exec	ready to exec
A: S.signal()	2	∅	ready to exec	ready to exec

Semaphores: Purposes

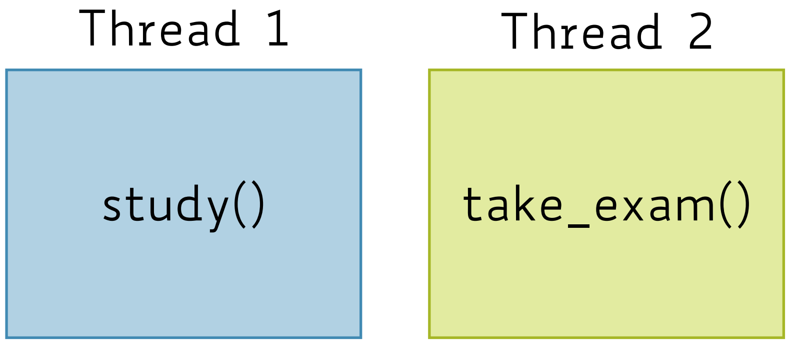
- **Mutual Exclusion:** used to guard critical sections
  - The initial value of the semaphore is set to 1
  - Call wait() before the critical section, signal() after the critical section

Semaphores: Purposes

- **Mutual Exclusion:** used to guard critical sections
  - The initial value of the semaphore is set to 1
  - Call wait() before the critical section, signal() after the critical section
- **Scheduling Constraints:** used to enforce threads to wait
  - The initial value of the semaphore is set to 0
  - Example → join() or waitpid()

Semaphores: Purposes

- **Mutual Exclusion:** used to guard critical sections
  - The initial value of the semaphore is set to 1
  - Call wait() before the critical section, signal() after the critical section
- **Scheduling Constraints:** used to enforce threads to wait
  - The initial value of the semaphore is set to 0
  - Example → join() or waitpid()



Semaphores: Purposes

- **Mutual Exclusion:** used to guard critical sections
  - The initial value of the semaphore is set to 1
  - Call wait() before the critical section, signal() after the critical section
- **Scheduling Constraints:** used to enforce threads to wait
  - The initial value of the semaphore is set to 0
  - Example → join() or waitpid()

S = 0

Thread 1	Thread 2
study() S.signal()	S.wait() take_exam()

Producer-Consumer

Producer Process:

```
while (true)
{
    /* produce an item in nextProduced */
    while (counter == BUFFER_SIZE)
      ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Consumer Process:

```
while (true)
{
    while (counter == 0)
      ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    /* consume the item in nextConsumed */
}
```

Both the producer and the consumer share a common buffer (of items)

Producer-Consumer

Producer Process:

```
while (true)
{
        /* produce an item in nextProduced */
        while (counter == BUFFER_SIZE)
           ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer Process:

```
while (true)
{
        while (counter == 0)
           ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in nextConsumed */
}
```

Both the producer and the consumer share a common buffer (of items)

counter keeps track of the number of items currently in the buffer

Producer-Consumer

Producer Process:

```
while (true)
{
        /* produce an item in nextProduced */
        while (counter == BUFFER_SIZE)
           ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer Process:

```
while (true)
{
        while (counter == 0)
           ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in nextConsumed */
}
```

Both the producer and the consumer share a common buffer (of items)

counter keeps track of the number of items currently in the buffer

possible race condition as counter can be updated by the producer and consumer

Producer-Consumer: Race Condition

Producer:

$$register_1 = \texttt{counter}$$
$$register_1 = register_1 + 1$$
$$\texttt{counter} = register_1$$

Consumer:

$$register_2 = \texttt{counter}$$
$$register_2 = register_2 - 1$$
$$\texttt{counter} = register_2$$

Interleaving:

Assuming the initial value of counter is 5

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

Producer-Consumer: Race Condition

Producer:

$$register_1 = \texttt{counter}$$
$$register_1 = register_1 + 1$$
$$\texttt{counter} = register_1$$

Consumer:

$$register_2 = \texttt{counter}$$
$$register_2 = register_2 - 1$$
$$\texttt{counter} = register_2$$

Interleaving:

Assuming the initial value of counter is 5

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

Q1: What would be the resulting value of counter if the order of statements T4 and T5 were reversed?

Producer-Consumer: Race Condition

Producer:

$$register_1 = \texttt{counter}$$
$$register_1 = register_1 + 1$$
$$\texttt{counter} = register_1$$

Consumer:

$$register_2 = \texttt{counter}$$
$$register_2 = register_2 - 1$$
$$\texttt{counter} = register_2$$

Interleaving:

Assuming the initial value of counter is 5

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{\texttt{counter} = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{\texttt{counter} = 4\}$

Q2: What should the value of counter be after one producer and one consumer, assuming the original value was 5?

Producer-Consumer: Desiderata

- Mutual Exclusion
  - Access to the shared buffer of items must be granted to a single thread at a time (either the producer or the consumer)

Producer-Consumer: Desiderata

- Mutual Exclusion
  - Access to the shared buffer of items must be granted to a single thread at a time (either the producer or the consumer)
- Scheduling Constraints
  - Producer can put a new item iff the buffer is not full
  - Consumer can take an item iff the buffer is not empty

Producer-Consumer in Java

Semaphores: Wrap Up

- Generalization of locks
- Can be used for 3 purposes:
  - To ensure mutually exclusive execution of a critical section as locks do (binary semaphore)
  - To control access to a shared pool of resources (counting semaphore)
  - To enforce scheduling constraints so as to execute threads according to some specific order

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**

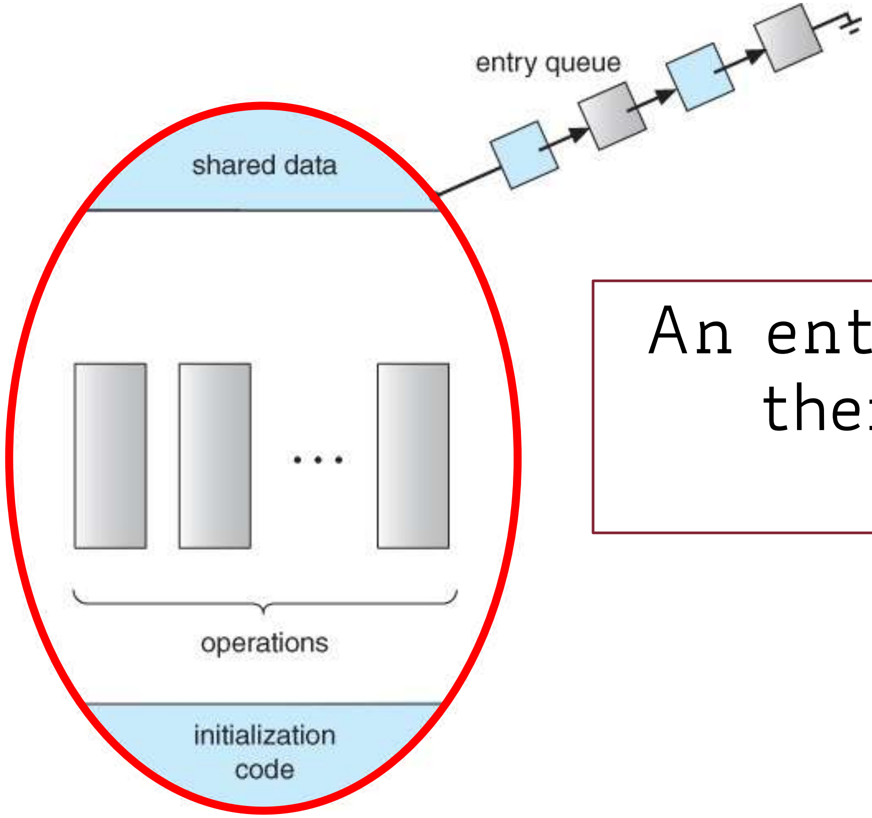
What is a Monitor?

- 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;
        }
    }
}
```

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;
        }
    }
}
```

What happens if a thread tries to remove an element from an empty queue?

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 of 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 of 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 of 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

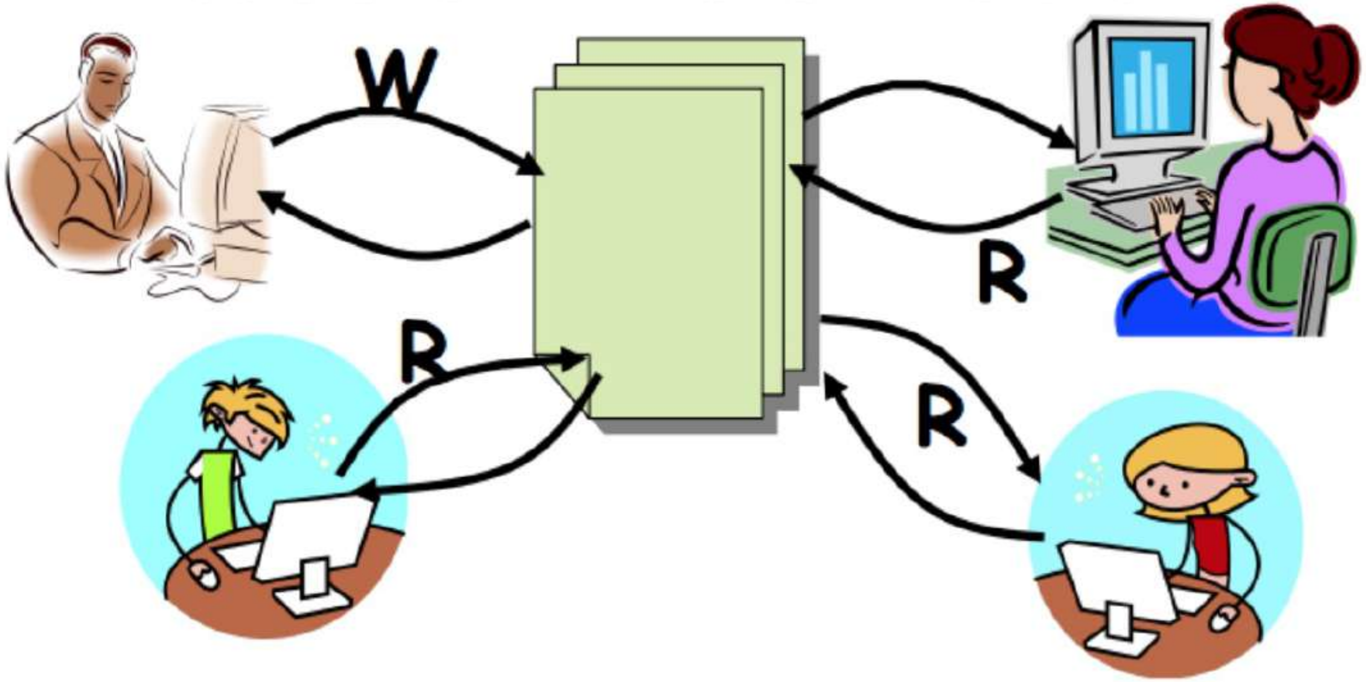
```
class Queue {
    …
    private ArrayList<Item> data;
    …

    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

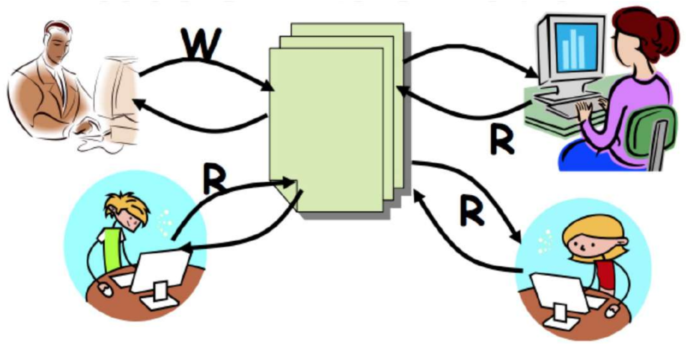
Motivation: Consider a shared database system

(more generally, any shared resource)



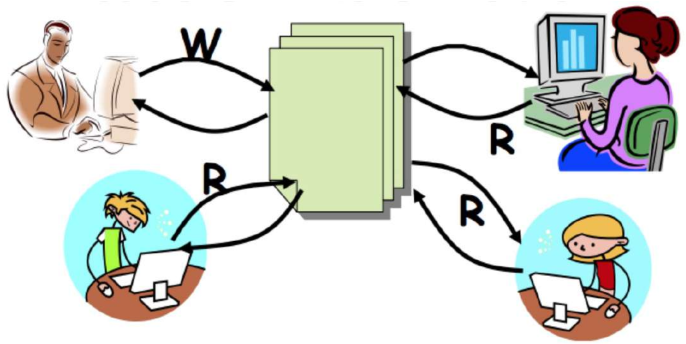
Readers-Writers Problem

Two classes of users:
- **Readers** → never modify the DB



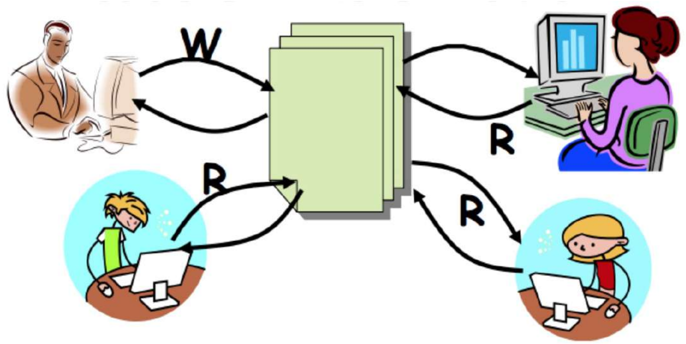
Readers-Writers Problem

Two classes of users:
- **Readers** → never modify the DB
- **Writers** → read and modify the DB



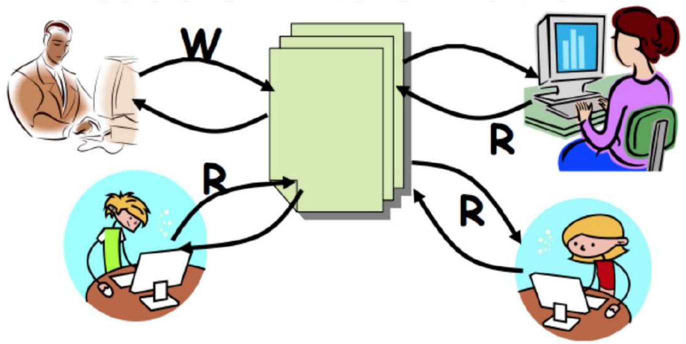
Readers-Writers Problem

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



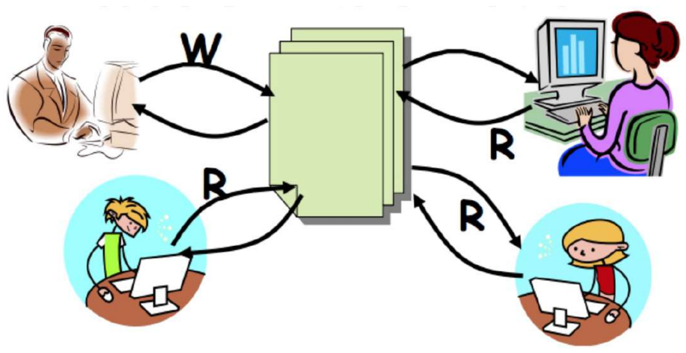
Readers-Writers Problem

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



Readers-Writers Problem

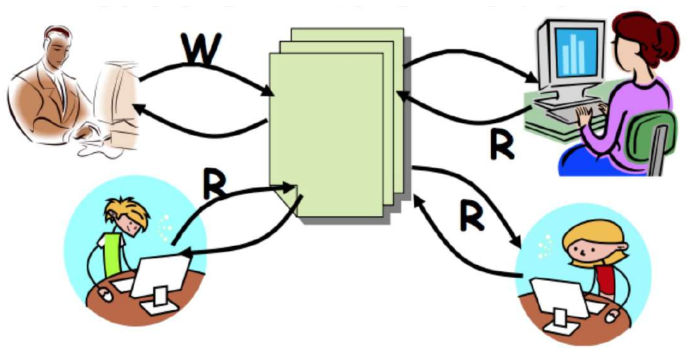
Simplest solution:
- Use a single lock on the data object for each operation
- May be too restrictive!



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

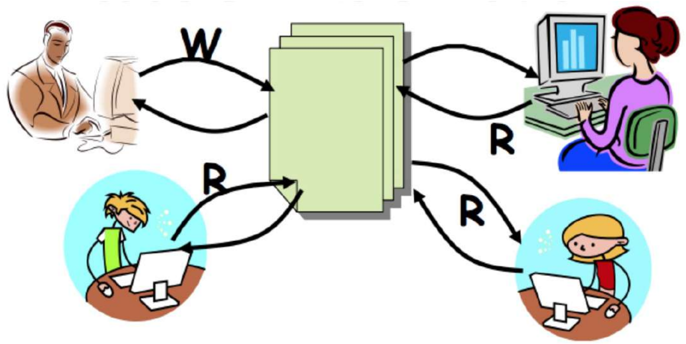
Readers-Writers Problem

<u>Constraints:</u>
- Readers can access DB when no writers

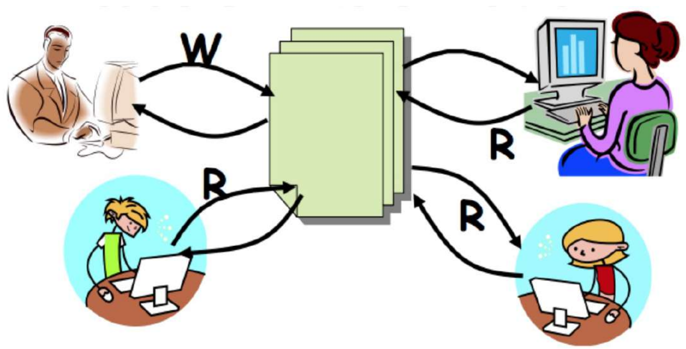


Readers-Writers Problem

<u>Constraints:</u>
- 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

- Piority 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