Systems and Networking I

Applied Computer Science and Artificial Intelligence

2024-2025

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:
  - 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 → discouraging threads from requesting any I/O operation within a critical section

Implementing Locks: Disabling Interrupts

- On single-CPU systems, we can prevent the scheduler to take over by:
  - internal events → discouraging threads from requesting any I/O operation within 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 → discouraging threads from requesting any I/O operation within 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();

  Lock() {}
}
```

Implementing Locks: Disabling Interrupts

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

  Lock() {}
}
```

```
public void acquire(Thread t) {
    disable_interrupts();
}
```

Implementing Locks: Disabling Interrupts

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

  Lock() {}
}
```

```
public void acquire(Thread t) {
    disable_interrupts();
}
```

```
public void release() {
    enable_interrupts();
}
```

Implementing Locks: Disabling Interrupts

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

  Lock() {}
}
```

We need both acquire and release being implemented as system calls

```
public void acquire(Thread t) {
    disable_interrupts();
}
```

```
public void release() {
    enable_interrupts();
}
```

Implementing Locks: Disabling Interrupts

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

  Lock() {}
}
```

We need both acquire and release being implemented as system calls

Why?

```
public void acquire(Thread t) {
    disable_interrupts();
}
```

```
public void release() {
    enable_interrupts();
}
```

Implementing Locks: Disabling Interrupts

- PROs:
  - Very simple!

Implementing Locks: Disabling Interrupts

- PROs:
  - Very simple!
- CONs:
  - Privileged instructions
  - Trust no one abuses this capability, e.g.:
    - A greedy program that gets the lock and hog the CPU
    - A malicious program that gets the lock and goes into an infinite loop
  - Does not work on multiprocessors!
  - May loose relevant interrupts
  - Masking/Unmasking interrupts is inefficient

Implementing Locks: A First Attempt

```
Class Lock {
  public void acquire(Thread t);
  public void release();
  private int flag; // 0=free; 1=busy

  Lock() {
     this.flag = 0; // initially free
  }
}
```

Implementing Locks: A First Attempt

```
Class Lock {
  public void acquire(Thread t);
  public void release();
  private int flag; // 0=free; 1=busy

  Lock() {
     this.flag = 0; // initially free
  }
}
```

```
public void release() {
  this.flag = 0; // set the flag to free
}
```

Implementing Locks: A First Attempt

```
Class Lock {
  public void acquire(Thread t);
  public void release();
  private int flag; // 0=free; 1=busy

  Lock() {
     this.flag = 0; // initially free
  }
}
```

```
public void acquire(Thread t) {
    while(this.flag == 1) { // test flag
       // do nothing (spin-wait)
    }
    this.flag = 1; // set the flag to busy
}
```

```
public void release() {
  this.flag = 0; // set the flag to free
}
```

Does this solution work?

Implementing Locks: A First Attempt

An unlucky (yet plausible) trace

Implementing Locks: A First Attempt

An unlucky (yet plausible) trace

time

Thread 1

Thread 2

Implementing Locks: A First Attempt

An unlucky (yet plausible) trace

time

Thread 1

Thread 2

call **acquire()**
while(this.flag == 1)
interrupt: switch to Thread 2

Implementing Locks: A First Attempt

An unlucky (yet plausible) trace

time

Thread 1

flag = 0

Thread 2

call **acquire()**
while(this.flag == 1)
interrupt: switch to Thread 2

Implementing Locks: A First Attempt

An unlucky (yet plausible) trace

time

Thread 1	Thread 2
call **acquire()** while(this.flag == 1) *interrupt:* switch to Thread 2	
	call **acquire()** while(this.flag == 1) this.flag = 1 *interrupt:* switch to Thread 1

Implementing Locks: A First Attempt

An unlucky (yet plausible) trace

time

Thread 1

flag = 0

Thread 2

call **acquire()**
while(this.flag == 1)
interrupt: switch to Thread 2

call **acquire()**
while(this.flag == 1)
this.flag = 1
interrupt: switch to Thread 1

Implementing Locks: A First Attempt

An unlucky (yet plausible) trace

time

Thread 1

flag = 1

Thread 2

call **acquire()**
while(this.flag == 1)
interrupt: switch to Thread 2

call **acquire()**
while(this.flag == 1)
this.flag = 1
interrupt: switch to Thread 1

this.flag = 1 // set to busy too!

Violation of mutual exclusion!

Implementing Locks: A First Attempt

- Spin-Waiting is bad!
- The waiter thread will waste CPU cycles doing nothing
- On uniprocessors such a waste could be even worse
- The only thread holding the lock must take its turn on the CPU, otherwise the other(s) spin-waiting will never take it!
- What if the we have a single CPU with a non-preemptive scheduler?

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 flag;

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

Implementing Locks: test&set

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

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

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

Implementing Locks: test&set

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

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

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

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

Implementing Locks: test&set

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

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

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

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

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

Implementing Locks: test&set

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

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

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

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

Case 1: if lock is free (flag = 0) test&set(flag) 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 flag;

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

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

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

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

Implementing Locks: test&set

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

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

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

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

Case 2: if lock is busy (flag = 1) test&set(flag) 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
(*spin-lock*)

- 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?

Masking Interrupts vs. Atomic Instructions

- 3 main problems with disabling interrupts:
  - overhead as it requires kernel privileges
  - trusted code (only OS kernel code can use this mechanism)
  - unfeasible with mulitprocessor architectures

Masking Interrupts vs. Atomic Instructions

- 3 main problems with disabling interrupts:
  - overhead as it requires kernel privileges
  - trusted code (only OS kernel code can use this mechanism)
  - unfeasible with mulitprocessor architectures
- 3 main problems with atomic instructions:
  - busy waiting (*spin-lock*)
  - unfairness as there is no queue where threads wait for the lock to be released
  - performance as N-1 waiter threads may spin for one round each before the thread holding the lock gets its turn on the CPU

Reduce Busy-Waiting: Yield!

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

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

```
public void acquire() {
  while(test&set(this.flag) == 1) {
    yield(); // give up the CPU
  }
}
```

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

An OS primitive (**yield**) that allows threads to give up the CPU

Reduce Busy-Waiting: Yield!

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

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

```
public void acquire() {
  while(test&set(this.flag) == 1) {
    yield(); // give up the CPU
  }
}
```

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

An OS primitive (**yield**) that allows threads to give up the CPU

In this way, the waiter thread releases the CPU immediately

Reduce Busy-Waiting: Yield!

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

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

```
public void acquire() {
  while(test&set(this.flag) == 1) {
    yield(); // give up the CPU
  }
}
```

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

An OS primitive (**yield**) that allows threads to give up the CPU

In this way, the waiter thread releases the CPU immediately

Still, some threads may starve in an endlessly yielding!

Reduce Busy-Waiting and Improve Fairness: Sleeping + Queues

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

  Lock() {
    // lock is initially free
    this.flag = 0;
    this.guard = 0;
    this.q = new Queue();
  }
}
```

```
public void acquire(Thread t) {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  if(this.flag == 0) {
    this.flag = 1; // lock is taken
    this.guard = 0;
  }
  else {
    this.q.push(t);
    this.guard = 0;
    park(); // Solaris primitive
  }
}
```

```
public void release() {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  if(q.is_empty()) {
    this.flag = 0; // lock is free
  }
  else {
    t = q.pop();
    unpark(t); // Solaris primitive
  }
  this.guard = 0;
}
```

park → Put the caller thread to sleep if it tries to acquire a busy lock

unpark → Wake up the waiter when the lock is free

Reduce Busy-Waiting and Improve Fairness: Sleeping + Queues

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

  Lock() {
    // lock is initially free
    this.flag = 0;
    this.guard = 0;
    this.q = new Queue();
  }
}
```

```
public void acquire(Thread t) {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  if(this.flag == 0) {
    this.flag = 1; // lock is taken
    this.guard = 0;
  }
  else {
    this.q.push(t);
    this.guard = 0;
    park(); // Solaris primitive
  }
}
```

```
public void release() {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  if(q.is_empty()) {
    this.flag = 0; // lock is free
  }
  else {
    t = q.pop();
    unpark(t); // Solaris primitive
  }
  this.guard = 0;
}
```

NOTE: The flag is not set to 0 when another thread is woken up

The thread releasing the lock passes it directly to the next thread

Reduce Busy-Waiting and Improve Fairness: Sleeping + Queues

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

  Lock() {
    // lock is initially free
    this.flag = 0;
    this.guard = 0;
    this.q = new Queue();
  }
}
```

```
public void acquire(Thread t) {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  if(this.flag == 0) {
    this.flag = 1; // lock is taken
    this.guard = 0;
  }
  else {
    this.q.push(t);
    this.guard = 0;
    park(); // Solaris primitive
  }
}
```

```
public void release() {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  if(q.is_empty()) {
    this.flag = 0; // lock is free
  }
  else {
    t = q.pop();
    unpark(t); // Solaris primitive
  }
  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

Reduce Busy-Waiting and Improve Fairness: Sleeping + Queues

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

  Lock() {
    // lock is initially free
    this.flag = 0;
    this.guard = 0;
    this.q = new Queue();
  }
}
```

```
public void acquire(Thread t) {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  if(this.flag == 0) {
    this.flag = 1; // lock is taken
    this.guard = 0;
  }
  else {
    this.q.push(t);
    this.guard = 0;
    park(); // Solaris primitive
  }
}
```

```
public void release() {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  if(q.is_empty()) {
    this.flag = 0; // lock is free
  }
  else {
    t = q.pop();
    unpark(t); // Solaris primitive
  }
  this.guard = 0;
}
```

What would happen if an unlucky thread switch happens right before calling park() and the incoming thread releases the lock?

Reduce Busy-Waiting and Improve Fairness: Sleeping + Queues

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

  Lock() {
    // lock is initially free
    this.flag = 0;
    this.guard = 0;
    this.q = new Queue();
  }
}
```

```
public void acquire(Thread t) {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  if(this.flag == 0) {
    this.flag = 1; // lock is taken
    this.guard = 0;
  }
  else {
    this.q.push(t);
    setpark(); // Solaris primitive
    this.guard = 0;
    park(); // Solaris primitive
  }
}
```

```
public void release() {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  if(q.is_empty()) {
    this.flag = 0; // lock is free
  }
  else {
    t = q.pop();
    unpark(t); // Solaris primitive
  }
  this.guard = 0;
}
```

Solution:

Use another system call to tell the OS the thread is *about* to park

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/spin-lock 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

```
# 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		"Too Much Milk" Using Semaphore	
# Thread Bob	# Thread Carla	# Thread Bob	# Thread Carla
lock.acquire()	lock.acquire()	S.wait()	S.wait()

if (!milk):
    buy_milk()
	if (!milk):
	    buy_milk()
		if (!milk):
		    buy_milk()
			if (!milk):
			    buy_milk()

lock.release()	lock.release()	S.signal()	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 = new Queue();
  }
}
```

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 = new Queue();
  }
}
```

```
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);
    park();
  }
  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 = new Queue();
  }
}
```

```
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);
    park();
  }
  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();
    unpark(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 = new Queue();
  }
}
```

```
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);
    park();
  }
  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();
    unpark(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 = new Queue();
  }
}
```

```
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);
    park();
  }
  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();
    unpark(t);
  }
  this.guard = 0;
}
```

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

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	Ø	ready to exec	ready to exec
A: S.wait()	1	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

A: S.wait()

B: S.wait()

S (value)	Queue	A	B
2	Ø	ready to exec	ready to exec
1	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	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

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	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

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()				

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

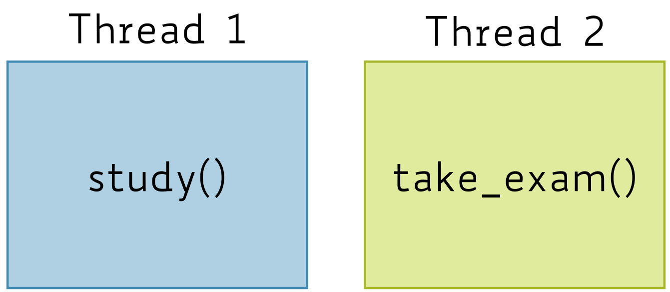
- Mutual Exclusion: used to guard critical sections
  - The initial value of the semaphore is set to 1
  - Call wait() before the critical section, signal() after the critical section

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()

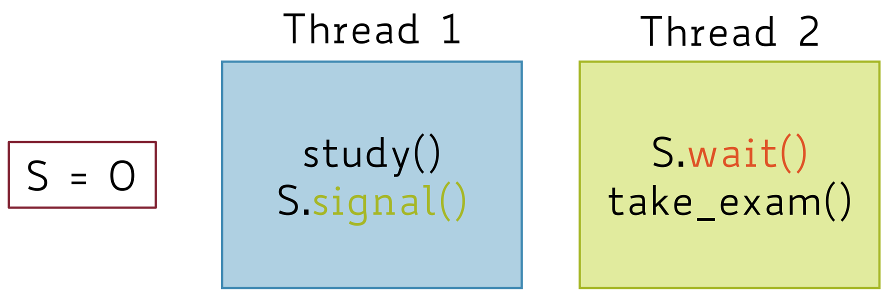
Semaphores: Purposes

- 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()



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:

$$\begin{aligned} register_1 &= \texttt{counter} \\ register_1 &= register_1 + 1 \\ \texttt{counter} &= register_1 \end{aligned}$$

Consumer:

$$\begin{aligned} register_2 &= \texttt{counter} \\ register_2 &= register_2 - 1 \\ \texttt{counter} &= register_2 \end{aligned}$$

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	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

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