Sistemi Operativi

Corso di Laurea in Informatica

a.a. 2020-2021

Gabriele Tolomei

Dipartimento di Informatica

Sapienza Università di Roma

tolomei@di.uniroma1.it

SAPIENZA UNIVERSITÀ DI ROMA

Process/Thread Synchronization

- We already mentioned that processes/threads can cooperate with each other in order to achieve a common task

Process/Thread Synchronization

- We already mentioned that processes/threads can cooperate with each other in order to achieve a common task
- However, cooperation may require synchronization between threads due to the presence of so-called critical sections (critical regions)

Process/Thread Synchronization

- We already mentioned that processes/threads can cooperate with each other in order to achieve a common task
- However, cooperation may require synchronization between threads due to the presence of so-called critical sections (critical regions)
- Synchronization primitives are required to ensure that only one thread at a time executes a critical section

Process/Thread Synchronization

- We already mentioned that processes/threads can cooperate with each other in order to achieve a common task
- However, cooperation may require synchronization between threads due to the presence of so-called critical sections (critical regions)
- Synchronization primitives are required to ensure that only one thread at a time executes a critical section

$\boxed{\text{Synchronization as a solution to the critical section problem}}$

The Need for Synchronization: Example

Consider the following real-world scenario, involving 2 roommates: Bob and Carl

The Need for Synchronization: Example

Consider the following real-world scenario, involving 2 roommates: Bob and Carl

Time	Bob	Carl

The Need for Synchronization: Example

Consider the following real-world scenario, involving 2 roommates: Bob and Carl

Time	Bob	Carl
5:00pm	Arrive home	

The Need for Synchronization: Example

Consider the following real-world scenario, involving 2 roommates: Bob and Carl

Time	Bob	Carl
5:00pm	Arrive home	
5:05pm	Look in the fridge → No milk!	

The Need for Synchronization: Example

Consider the following real-world scenario, involving 2 roommates: Bob and Carl

Time	Bob	Carl
5:00pm	Arrive home	
5:05pm	Look in the fridge → No milk!	
5:10pm	Leave home for the grocery	

The Need for Synchronization: Example

Consider the following real-world scenario, involving 2 roommates: Bob and Carl

Time	Bob	Carl
5:00pm	Arrive home	
5:05pm	Look in the fridge → No milk!	
5:10pm	Leave home for the grocery	
5:20pm		Arrive home

The Need for Synchronization: Example

Consider the following real-world scenario, involving 2 roommates: Bob and Carl

Time	Bob	Carl
5:00pm	Arrive home	
5:05pm	Look in the fridge → No milk!	
5:10pm	Leave home for the grocery	
5:20pm		Arrive home
5:25pm	Arrive at the grocery	Look in the fridge → No milk!

The Need for Synchronization: Example

Consider the following real-world scenario, involving 2 roommates: Bob and Carl

Time	Bob	Carl
5:00pm	Arrive home	
5:05pm	Look in the fridge → No milk!	
5:10pm	Leave home for the grocery	
5:20pm		Arrive home
5:25pm	Arrive at the grocery	Look in the fridge → No milk!
5:30pm	Buy milk	Leave home for the grocery

The Need for Synchronization: Example

Consider the following real-world scenario, involving 2 roommates: Bob and Carl

Time	Bob	Carl
5:00pm	Arrive home	
5:05pm	Look in the fridge → No milk!	
5:10pm	Leave home for the grocery	
5:20pm		Arrive home
5:25pm	Arrive at the grocery	Look in the fridge → No milk!
5:30pm	Buy milk	Leave home for the grocery
5:45pm	Arrive home, put the milk in the fridge	Arrive at the grocery

The Need for Synchronization: Example

Consider the following real-world scenario, involving 2 roommates: Bob and Carl

Time	Bob	Carl
5:00pm	Arrive home	
5:05pm	Look in the fridge → No milk!	
5:10pm	Leave home for the grocery	
5:20pm		Arrive home
5:25pm	Arrive at the grocery	Look in the fridge → No milk!
5:30pm	Buy milk	Leave home for the grocery
5:45pm	Arrive home, put the milk in the fridge	Arrive at the grocery
5:50pm		Buy milk

The Need for Synchronization: Example

Consider the following real-world scenario, involving 2 roommates: Bob and Carl

Time	Bob	Carl
5:00pm	Arrive home	
5:05pm	Look in the fridge → No milk!	
5:10pm	Leave home for the grocery	
5:20pm		Arrive home
5:25pm	Arrive at the grocery	Look in the fridge → No milk!
5:30pm	Buy milk	Leave home for the grocery
5:45pm	Arrive home, put the milk in the fridge	Arrive at the grocery
5:50pm		Buy milk
6:05pm		Arrive home, put the milk in the fridge

The Need for Synchronization: Example

Consider the following real-world scenario, involving 2 roommates: Bob and Carl

Time	Bob	Carl
5:00pm	Arrive home	
5:05pm	Look in the fridge → No milk!	
5:10pm	Leave home for the grocery	
5:20pm		Arrive home
5:25pm	Arrive at the grocery	Look in the fridge → No milk!
5:30pm	Buy milk	Leave home for the grocery
5:45pm	Arrive home, put the milk in the fridge	Arrive at the grocery
5:50pm		Buy milk
6:05pm		Arrive home, put the milk in the fridge
6:05pm		Oh f*%#k!

The Need for Synchronization: Example

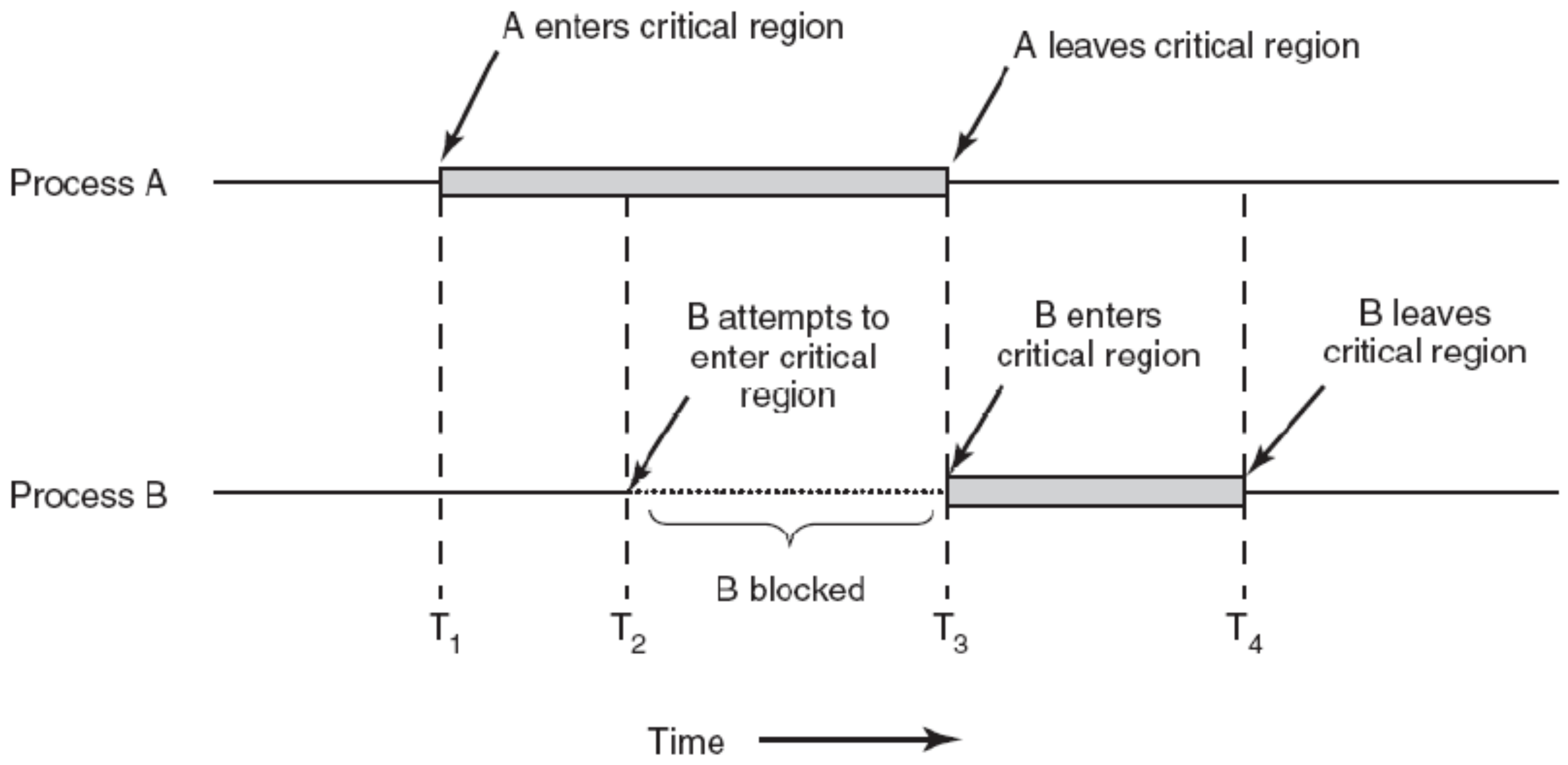
- In the example Bob and Carl represents 2 processes/threads
- Theoretically, they should cooperate to achieve a common task (e.g., buying some milk)
- In practice, though, they might incur in unpleasant situations (e.g., buying too much milk!)

The Need for Synchronization: Example

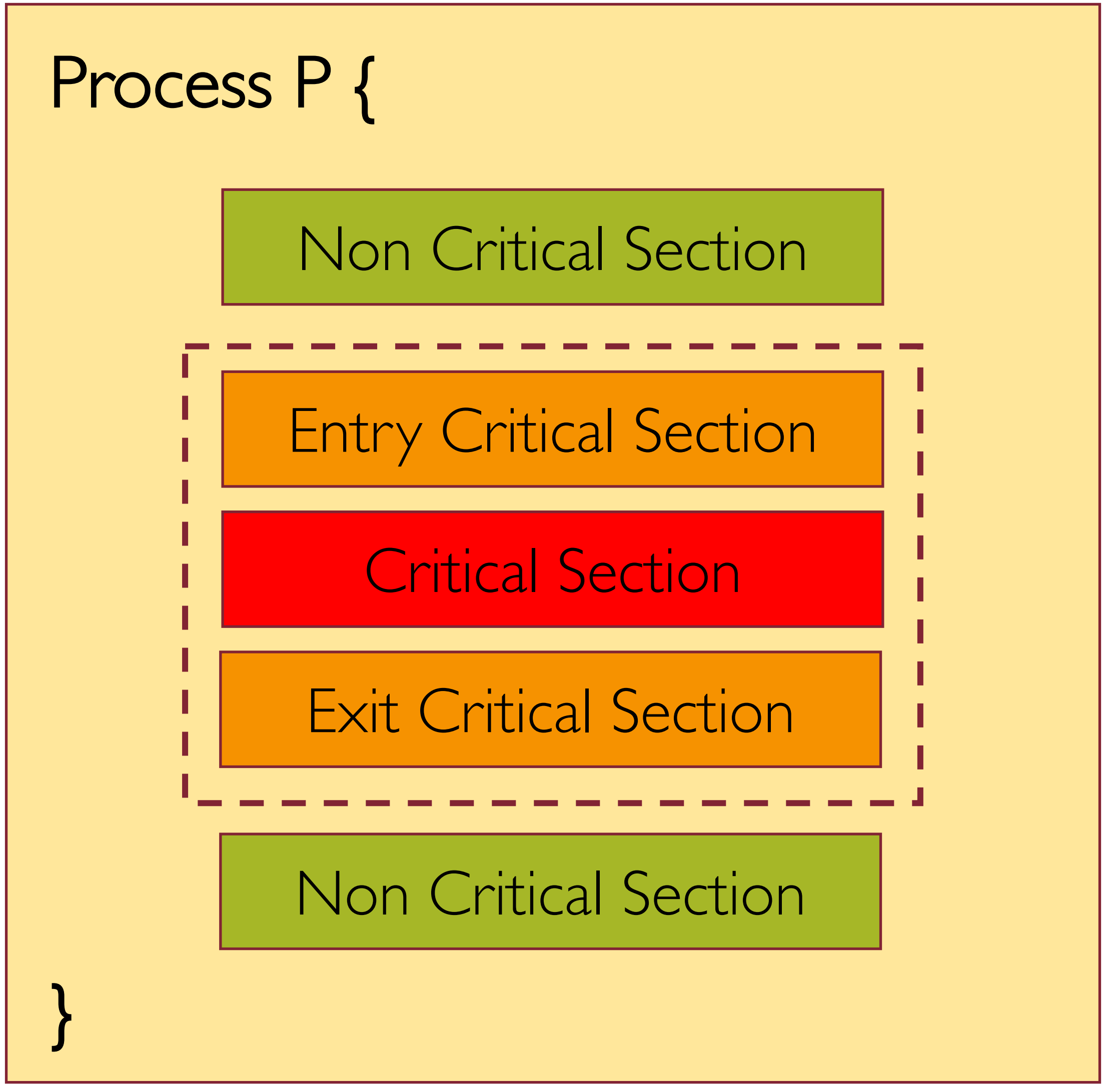
- In the example Bob and Carl represents 2 processes/threads
- Theoretically, they should cooperate to achieve a common task (e.g., buying some milk)
- In practice, though, they might incur in unpleasant situations (e.g., buying too much milk!)

What kind of mechanisms do we need in order to get independent yet cooperating processes to communicate
and have a consistent view of the "world" (i.e., computational state)?

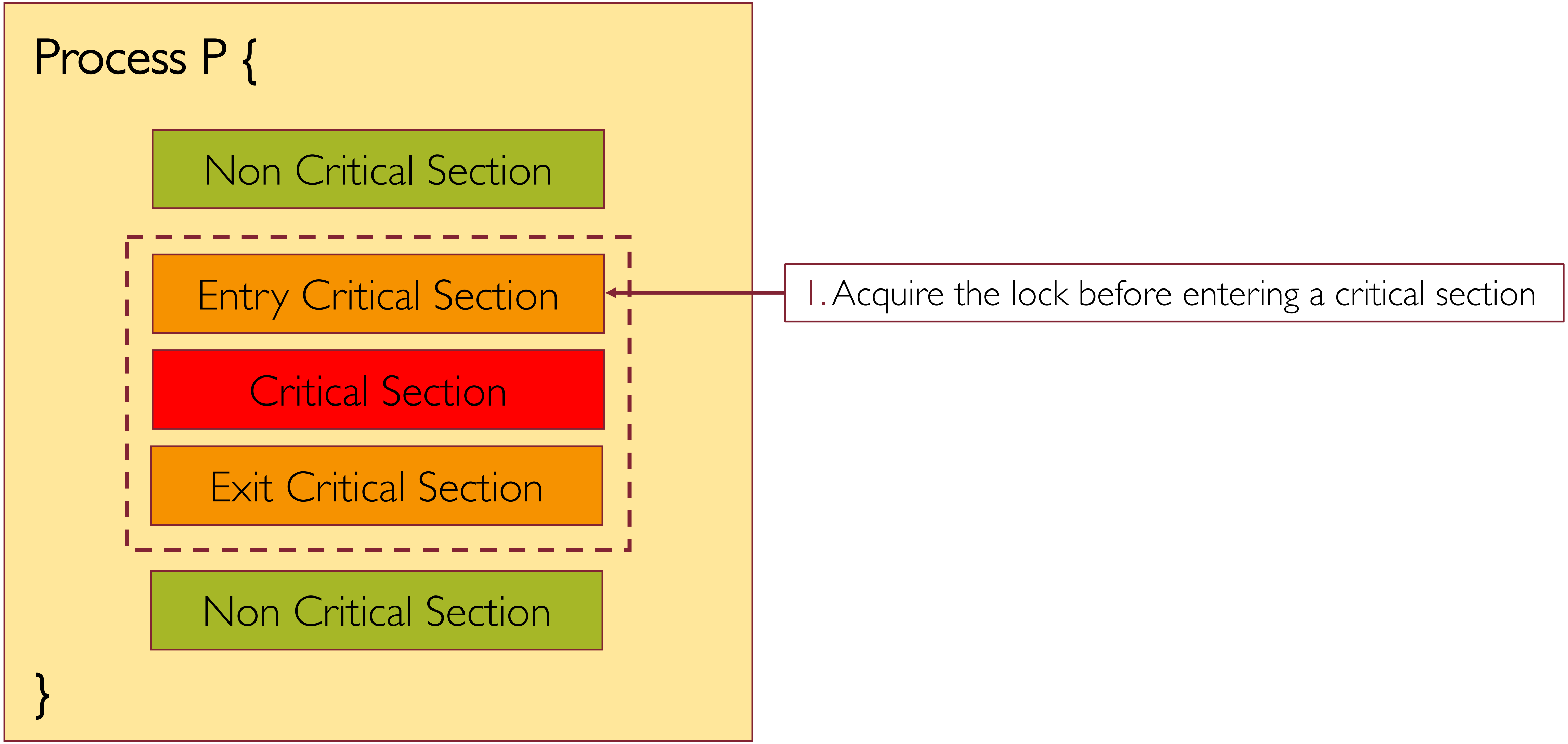
The Critical Section Problem



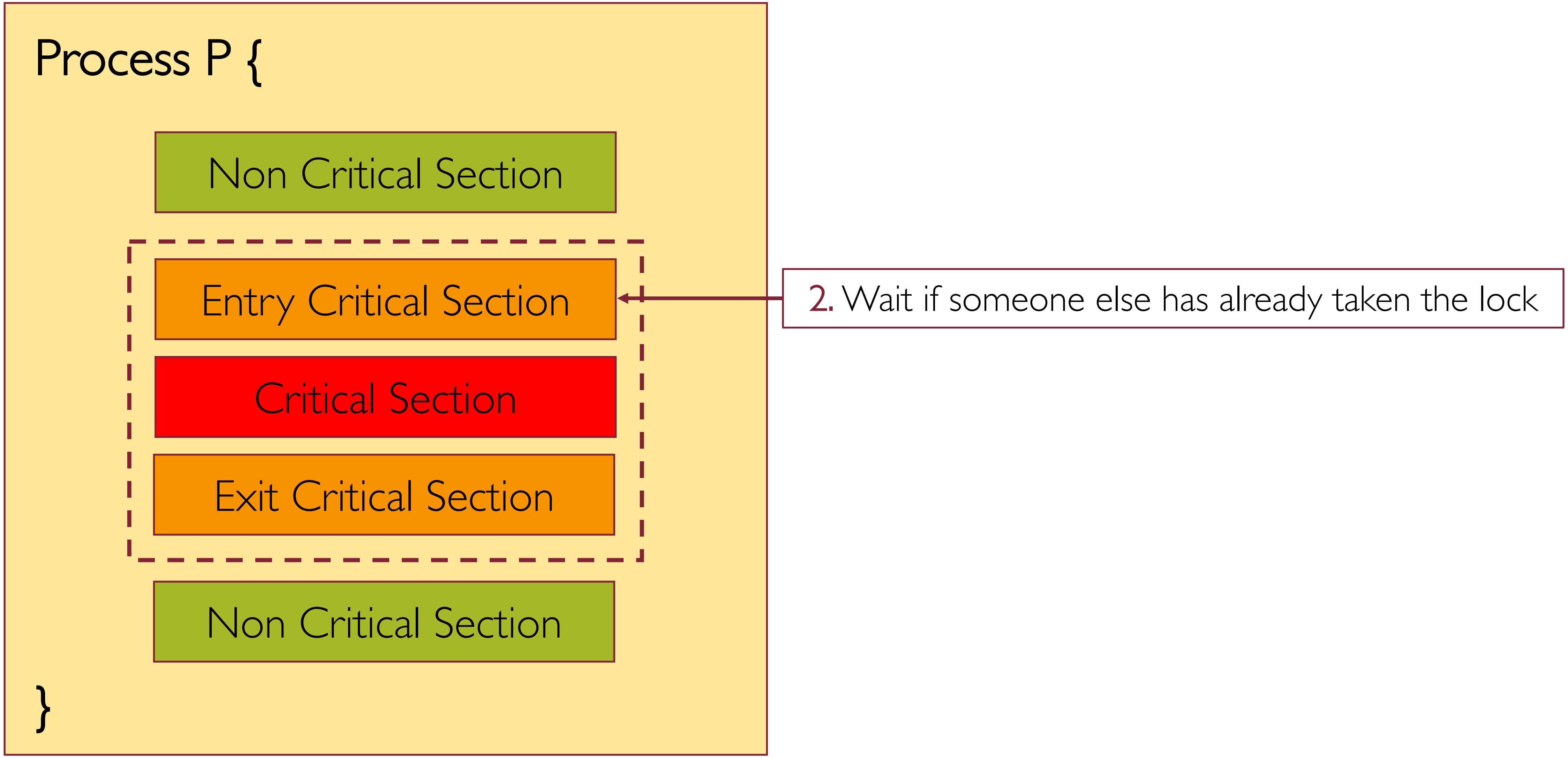
The Anatomy of a Critical Section



Locking Critical Section



Locking Critical Section



Locking Critical Section

Process P {

Non Critical Section

Entry Critical Section

Critical Section

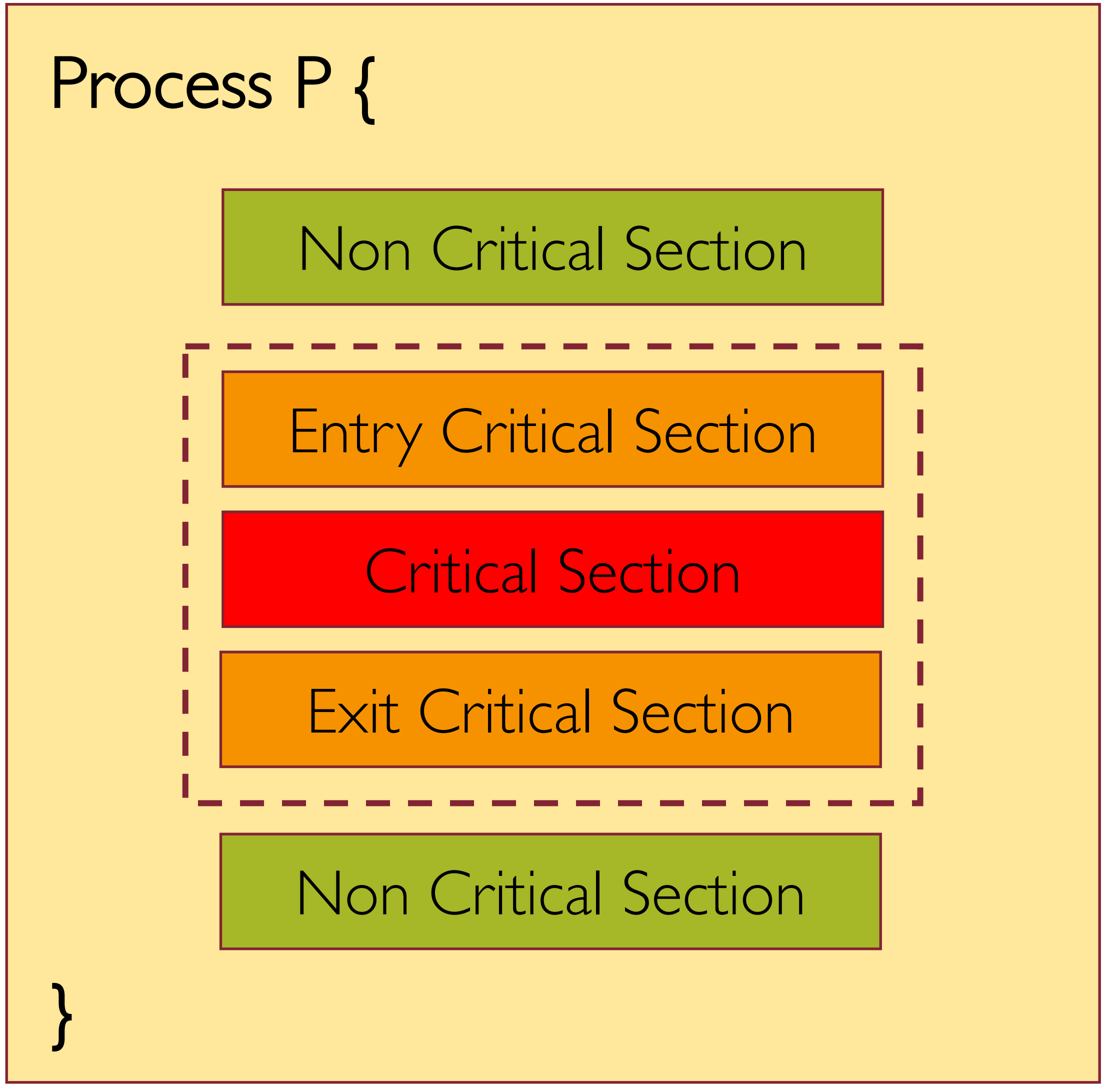
Exit Critical Section

Non Critical Section

}

3. Release the lock after leaving a critical section

Locking Critical Section



All synchronization involves waiting!

Synchronization: Goals

- Any synchronization solution to the critical section problem must satisfy 3 properties:
  - Mutual Exclusion → only one process/thread can be in its critical section at a time!

Synchronization: Goals

- Any synchronization solution to the critical section problem must satisfy 3 properties:
  - Mutual Exclusion → only one process/thread can be in its critical section at a time!
  - Liveness → If no process is in its critical section, and one or more want to execute it then any one of these must be able to get into its critical section

Synchronization: Goals

- Any synchronization solution to the critical section problem must satisfy 3 properties:
  - Mutual Exclusion → only one process/thread can be in its critical section at a time!
  - Liveness → If no process is in its critical section, and one or more want to execute it then any one of these must be able to get into its critical section
  - Bounded Waiting → A process requesting entry into its critical section will get a turn eventually, and there is a limit on how many others get to go first

Synchronization: Goals

- In the milk example:
  - Ensuring mutual exclusion means no more milk than what is needed will be bought (i.e., only one between Bob and Carl will buy milk if needed)

Synchronization: Goals

- In the milk example:
  - Ensuring mutual exclusion means no more milk than what is needed will be bought (i.e., only one between Bob and Carl will buy milk if needed)
  - Ensuring liveness means that someone should buy some milk (i.e., the option where both Bob and Carl do not do anything is surely safe but undesirable)

Synchronization: Goals

- In the milk example:
  - Ensuring mutual exclusion means no more milk than what is needed will be bought (i.e., only one between Bob and Carl will buy milk if needed)
  - Ensuring liveness means that someone should buy some milk (i.e., the option where both Bob and Carl do not do anything is surely safe but undesirable)
  - Ensuring bounding waiting means that eventually Bob and Carl will enter their critical section

Too Much Milk: Solution 1

Use a note

```
# Thread Bob

if (!milk and !note):
    leave_note()
    buy_milk()
    remove_note()
```

```
# Thread Carl

if (!milk and !note):
    leave_note()
    buy_milk()
    remove_note()
```

Too Much Milk: Solution 1

Use a **note**

```
# Thread Bob

if (!milk and !note):
    leave_note()
    buy_milk()
    remove_note()
```

```
# Thread Carl

if (!milk and !note):
    leave_note()
    buy_milk()
    remove_note()
```

Does this solution work?

Too Much Milk: Solution 1

Use a **note**

```
# Thread Bob

if (!milk and !note):
    leave_note()
    buy_milk()
    remove_note()
```

```
# Thread Carl

if (!milk and !note):
    leave_note()
    buy_milk()
    remove_note()
```

Does this solution work **regardless of the scheduling**?

Too Much Milk: Solution 1

Use a **note**

```
# Thread Bob

if (!milk and !note):
    leave_note()
    buy_milk()
    remove_note()
```

```
# Thread Carl

if (!milk and !note):
    leave_note()
    buy_milk()
    remove_note()
```

Does this solution work **regardless of the scheduling**?

No! mutual exclusion can be violated

Too Much Milk: Solution 2

Use 2 (labeled) notes

```
# Thread Bob

leave_note(Bob)

if (!note(Carl)):
    if (!milk):
        buy_milk()

remove_note()
```

```
# Thread Carl

leave_note(Carl)

if (!note(Bob)):
    if (!milk):
        buy_milk()

remove_note()
```

Too Much Milk: Solution 2

Use 2 (labeled) notes

```
# Thread Bob

leave_note(Bob)

if (!note(Carl)):
    if (!milk):
        buy_milk()

remove_note()
```

```
# Thread Carl

leave_note(Carl)

if (!note(Bob)):
    if (!milk):
        buy_milk()

remove_note()
```

Does this solution work regardless of the scheduling?

Too Much Milk: Solution 2

Use 2 (labeled) notes

```
# Thread Bob

leave_note(Bob)

if (!note(Carl)):
    if (!milk):
        buy_milk()

remove_note()
```

```
# Thread Carl

leave_note(Carl)

if (!note(Bob)):
    if (!milk):
        buy_milk()

remove_note()
```

Does this solution work regardless of the scheduling?

No! Liveness propery can be violated

Too Much Milk: Solution 3

Use 2 (labeled) notes… more cleverly

```
# Thread Bob

leave_note(Bob)

while (note(Carl)):
    do_nothing()
if (!milk):
    buy_milk()

remove_note()
```

```
# Thread Carl

leave_note(Carl)

if (!note(Bob)):
    if (!milk):
        buy_milk()

remove_note()
```

Too Much Milk: Solution 3

Use 2 (labeled) notes… more cleverly

```
# Thread Bob

leave_note(Bob)

while (note(Carl)):
    do_nothing()
if (!milk):
    buy_milk()

remove_note()
```

```
# Thread Carl

leave_note(Carl)

if (!note(Bob)):
    if (!milk):
        buy_milk()

remove_note()
```

Does this solution work regardless of the scheduling?

Too Much Milk: Solution 3

Use 2 (labeled) notes… more cleverly

```
# Thread Bob

leave_note(Bob)

while (note(Carl)):
    do_nothing()
if (!milk):
    buy_milk()

remove_note()
```

```
# Thread Carl

leave_note(Carl)

if (!note(Bob)):
    if (!milk):
        buy_milk()

remove_note()
```

Does this solution work regardless of the scheduling?

Yes!

Too Much Milk: Solution 3

```
# Thread Bob

leave_note(Bob)

while (note(Carl)):
    do_nothing()
if (!milk):
    buy_milk()

remove_note()
```

Y:

```
# Thread Carl

leave_note(Carl)

if (!note(Bob)):
    if (!milk):
        buy_milk()

remove_note()
```

Too Much Milk: Solution 3

```
# Thread Bob

leave_note(Bob)

while (note(Carl)):
    do_nothing()
if (!milk):
    buy_milk()

remove_note()
```

Y: →

```
# Thread Carl

leave_note(Carl)

if (!note(Bob)):
    if (!milk):
        buy_milk()

remove_note()
```

Case 1: no note from Bob

Too Much Milk: Solution 3

```
# Thread Bob

leave_note(Bob)

while (note(Carl)):
    do_nothing()
if (!milk):
    buy_milk()

remove_note()
```

Y:

```
# Thread Carl

leave_note(Carl)

if (!note(Bob)):
    if (!milk):
        buy_milk()

remove_note()
```

Case 1: no note from Bob

Thread Bob must be executing different code

Too Much Milk: Solution 3

```
# Thread Bob

leave_note(Bob)

while (note(Carl)):
    do_nothing()
if (!milk):
    buy_milk()

remove_note()
```

Y: →

```
# Thread Carl

leave_note(Carl)

if (!note(Bob)):
    if (!milk):
        buy_milk()

remove_note()
```

Case 1: no note from Bob

↓

Thread Bob must be executing different code

↓

Carl will buy milk only if needed

Too Much Milk: Solution 3

```
# Thread Bob

leave_note(Bob)

while (note(Carl)):
    do_nothing()
if (!milk):
    buy_milk()

remove_note()
```

Y: →

```
# Thread Carl

leave_note(Carl)

if (!note(Bob)):
    if (!milk):
        buy_milk()

remove_note()
```

Case 2: Bob has left a note

Too Much Milk: Solution 3

```
# Thread Bob

leave_note(Bob)

while (note(Carl)):
    do_nothing()
if (!milk):
    buy_milk()

remove_note()
```

Y: →

```
# Thread Carl

leave_note(Carl)

if (!note(Bob)):
    if (!milk):
        buy_milk()

remove_note()
```

Case 2: Bob has left a note

So has Carl, therefore Bob will be waiting (loop)

Too Much Milk: Solution 3

```
# Thread Bob

leave_note(Bob)

while (note(Carl)):
    do_nothing()
if (!milk):
    buy_milk()

remove_note()
```

Y: →

```
# Thread Carl

leave_note(Carl)

if (!note(Bob)):
    if (!milk):
        buy_milk()

remove_note()
```

Case 2: Bob has left a note

So has Carl, therefore Bob will be waiting (loop)

Carl will remove his note and Bob will buy milk if needed

Too Much Milk: Solution 3

```
# Thread Bob

leave_note(Bob)

while (note(Carl)):
    do_nothing()
if (!milk):
    buy_milk()

remove_note()
```

X:

Case 1: no note from Carl

```
# Thread Carl

leave_note(Carl)

if (!note(Bob)):
    if (!milk):
        buy_milk()

remove_note()
```

Too Much Milk: Solution 3

```
# Thread Bob

leave_note(Bob)

while (note(Carl)):
    do_nothing()
if (!milk):
    buy_milk()

remove_note()
```

X:

Case 1: no note from Carl

Thread Carl must be executing different code

```
# Thread Carl

leave_note(Carl)

if (!note(Bob)):
    if (!milk):
        buy_milk()

remove_note()
```

Too Much Milk: Solution 3

X:

```
# Thread Bob

leave_note(Bob)

while (note(Carl)):
    do_nothing()
if (!milk):
    buy_milk()

remove_note()
```

Case 1: no note from Carl

Thread Carl must be executing different code

Bob will buy milk only if needed

```
# Thread Carl

leave_note(Carl)

if (!note(Bob)):
    if (!milk):
        buy_milk()

remove_note()
```

Too Much Milk: Solution 3

Case 2: Carl has left a note

X: →

```
# Thread Bob

leave_note(Bob)

while (note(Carl)):
    do_nothing()
if (!milk):
    buy_milk()

remove_note()
```

```
# Thread Carl

leave_note(Carl)

if (!note(Bob)):
    if (!milk):
        buy_milk()

remove_note()
```

Too Much Milk: Solution 3

```
# Thread Bob

leave_note(Bob)

while (note(Carl)):
    do_nothing()
if (!milk):
    buy_milk()

remove_note()
```

X: →

Case 2: Carl has left a note

Bob will wait doing nothing until Carl removes his note

```
# Thread Carl

leave_note(Carl)

if (!note(Bob)):
    if (!milk):
        buy_milk()

remove_note()
```

Too Much Milk: Solution 3

```
# Thread Bob

leave_note(Bob)

while (note(Carl)):
    do_nothing()
if (!milk):
    buy_milk()

remove_note()
```

X:

Case 2: Carl has left a note

Bob will wait doing nothing until Carl removes his note

Carl will buy milk only if needed

```
# Thread Carl

leave_note(Carl)

if (!note(Bob)):
    if (!milk):
        buy_milk()

remove_note()
```

Is Solution 3 Good?

Not exactly!

Is Solution 3 Good?

Not exactly!

- 3 main reasons why this solution is not viable:

Is Solution 3 Good?

Not exactly!

- 3 main reasons why this solution is not viable:
  - too complicated → it is quite hard to see that it actually works

Is Solution 3 Good?

Not exactly!

- 3 main reasons why this solution is not viable:
  - too complicated → it is quite hard to see that it actually works
  - asymmetrical → thread Bob and Carl are different (adding more threads will mess up things even more!)

Is Solution 3 Good?

Not exactly!

- 3 main reasons why this solution is not viable:
  - too complicated → it is quite hard to see that it actually works
  - asymmetrical → thread Bob and Carl are different (adding more threads will mess up things even more!)
  - busy waiting → thread Bob is consuming CPU cycles doing nothing

Is Solution 3 Good?

Not exactly!

- 3 main reasons why this solution is not viable:
  - too complicated → it is quite hard to see that it actually works
  - asymmetrical → thread Bob and Carl are different (adding more threads will mess up things even more!)
  - busy waiting → thread Bob is consuming CPU cycles doing nothing

This solution assumes loads and stores being atomic (i.e., non-interruptable)

So? How Do We Implement Synchronization?

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

So? How Do We Implement Synchronization?

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

So? How Do We Implement Synchronization?

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

So? How Do We Implement Synchronization?

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

So? How Do We Implement Synchronization?

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 Carl

Lock.acquire()

if (!milk):
    buy_milk()

Lock.release()
```

Too Much Milk: Solution Using Locks

Use lock primitives

```
# Thread Bob

Lock.acquire()

if (!milk):
    buy_milk()

Lock.release()
```

```
# Thread Carl

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 Carl

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

HW Support for Synchronization

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

High-level atomic operations (SW)	lock, monitor, semaphore, send/receive

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 interrupt, 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 interrupt, atomic instructions (test&set)

Implementing Locks: Disabling Interrupts

- If we think about it, the reason why we care about synchronization is because context switches may occur unexpectedly

Implementing Locks: Disabling Interrupts

- If we think about it, the reason why we care about 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

- If we think about it, the reason why we care about 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

- If we think about it, the reason why we care about 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:
  - 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 interrupt, 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

- 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
- Examples:
  - **`test&set`** (most architectures) → reads a value, writes **`1`** back to memory
  - **`exchange`** (x86) → swaps values between register and memory

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`**

Summary

- Communication among threads is usually done via shared variables

Summary

- Communication among threads is usually done via shared variables
- Access or modification to those shared variable often identifies critical sections

Summary

- Communication among threads is usually done via shared variables
- Access or modification to those shared variable often identifies critical sections
- A critical section is a piece of code that cannot be executed in parallel or concurrently by multiple threads

Summary

- Communication among threads is usually done via shared variables
- Access or modification to those shared variable often identifies critical sections
- A critical section is a piece of code that cannot be executed in parallel or concurrently by multiple threads
- Synchronization primitives are required to ensure only one thread at a time executes a critical section (mutual exculsion)
  - High-level primitives: locks, semaphores, and monitors

Summary

- Communication among threads is usually done via shared variables
- Access or modification to those shared variable often identifies critical sections
- A critical section is a piece of code that cannot be executed in parallel or concurrently by multiple threads
- Synchronization primitives are required to ensure only one thread at a time executes a critical section (mutual exculsion)
    - High-level primitives: locks, semaphores, and monitors