

# Systems and Networking I

Applied Computer Science and Artificial Intelligence  
2024-2025



**SAPIENZA**  
UNIVERSITÀ DI ROMA

Gabriele Tolomei

Dipartimento di Informatica

Sapienza Università di Roma

[tolomei@di.uniroma1.it](mailto:tolomei@di.uniroma1.it)

# 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

Synchronization as a solution to the critical section problem

# Part III:

# Process Synchronization

# The Need for Synchronization

Consider the following scenario, involving 2 roommates: Bob and Carla

# The Need for Synchronization

Consider the following scenario, involving 2 roommates: Bob and Carla

Time	Bob	Carla
5:00pm	Gets home	



# The Need for Synchronization

Consider the following scenario, involving 2 roommates: Bob and Carla

Time	Bob	Carla
5:00pm	Gets home	
5:05pm	Looks in the fridge → No milk!	

# The Need for Synchronization

Consider the following scenario, involving 2 roommates: Bob and Carla

Time	Bob	Carla
5:00pm	Gets home	
5:05pm	Looks in the fridge → No milk!	
5:10pm	Leaves home for the grocery	

# The Need for Synchronization

Consider the following scenario, involving 2 roommates: Bob and Carla

Time	Bob	Carla
5:00pm	Gets home	
5:05pm	Looks in the fridge → No milk!	
5:10pm	Leaves home for the grocery	
5:20pm		Gets home

# The Need for Synchronization

Consider the following scenario, involving 2 roommates: Bob and Carla

Time	Bob	Carla
5:00pm	Gets home	
5:05pm	Looks in the fridge → No milk!	
5:10pm	Leaves home for the grocery	
5:20pm		Gets home
5:25pm	Gets at the grocery	Looks in the fridge → No milk!

# The Need for Synchronization

Consider the following scenario, involving 2 roommates: Bob and Carla

Time	Bob	Carla
5:00pm	Gets home	
5:05pm	Looks in the fridge → No milk!	
5:10pm	Leaves home for the grocery	
5:20pm		Gets home
5:25pm	Gets at the grocery	Looks in the fridge → No milk!
5:30pm	Buys milk	Leaves home for the grocery

# The Need for Synchronization

Consider the following scenario, involving 2 roommates: Bob and Carla

Time	Bob	Carla
5:00pm	Gets home	
5:05pm	Looks in the fridge → No milk!	
5:10pm	Leaves home for the grocery	
5:20pm		Gets home
5:25pm	Gets at the grocery	Looks in the fridge → No milk!
5:30pm	Buys milk	Leaves home for the grocery
5:45pm	Gets home, puts the milk in the fridge	Gets at the grocery

# The Need for Synchronization

Consider the following scenario, involving 2 roommates: Bob and Carla

Time	Bob	Carla
5:00pm	Gets home	
5:05pm	Looks in the fridge → No milk!	
5:10pm	Leaves home for the grocery	
5:20pm		Gets home
5:25pm	Gets at the grocery	Looks in the fridge → No milk!
5:30pm	Buys milk	Leaves home for the grocery
5:45pm	Gets home, puts the milk in the fridge	Gets at the grocery
5:50pm		Buys milk

# The Need for Synchronization

Consider the following scenario, involving 2 roommates: Bob and Carla

Time	Bob	Carla
5:00pm	Gets home	
5:05pm	Looks in the fridge → No milk!	
5:10pm	Leaves home for the grocery	
5:20pm		Gets home
5:25pm	Gets at the grocery	Looks in the fridge → No milk!
5:30pm	Buys milk	Leaves home for the grocery
5:45pm	Gets home, puts the milk in the fridge	Gets at the grocery
5:50pm		Buys milk
6:05pm		Gets home, puts the milk in the fridge



# The Need for Synchronization

Consider the following scenario, involving 2 roommates: Bob and Carla

Time	Bob	Carla
5:00pm	Gets home	
5:05pm	Looks in the fridge → No milk!	
5:10pm	Leaves home for the grocery	
5:20pm		Gets home
5:25pm	Gets at the grocery	Looks in the fridge → No milk!
5:30pm	Buys milk	Leaves home for the grocery
5:45pm	Gets home, puts the milk in the fridge	Gets at the grocery
5:50pm		Buys milk
6:05pm		Gets home, puts the milk in the fridge
6:05pm	Oh f*%#k!	Oh f*%#k!

# The Need for Synchronization:

- In the example, **Bob** and **Carla** represents 2 processes/threads

# The Need for Synchronization:

- In the example, **Bob** and **Carla** represents 2 processes/threads
- Theoretically, they should cooperate to achieve a common task (e.g., buying some milk)

# The Need for Synchronization:

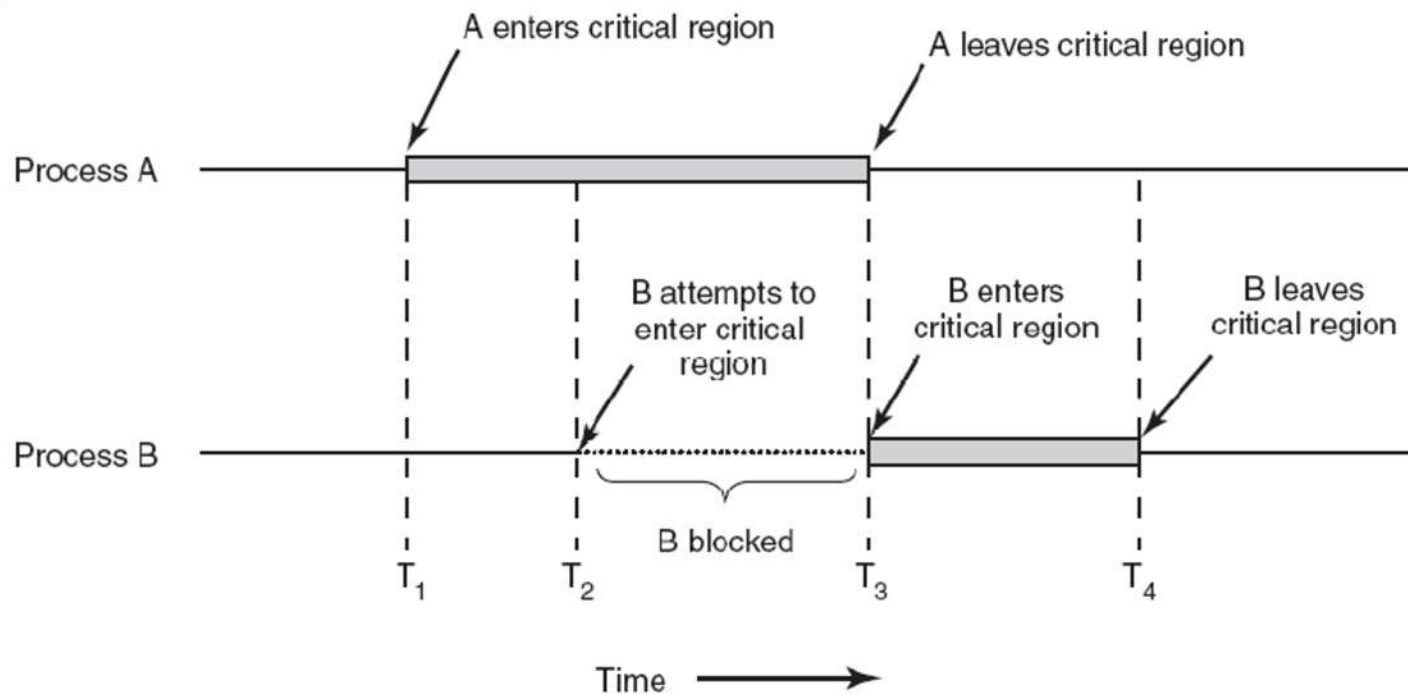
- In the example, **Bob** and **Carla** represents 2 processes/threads
- Theoretically, they should cooperate to achieve a common task (e.g., buying some milk)
- In practice, though, they might incur in unpleasant situations (e.g., buying too much milk!)

# The Need for Synchronization:

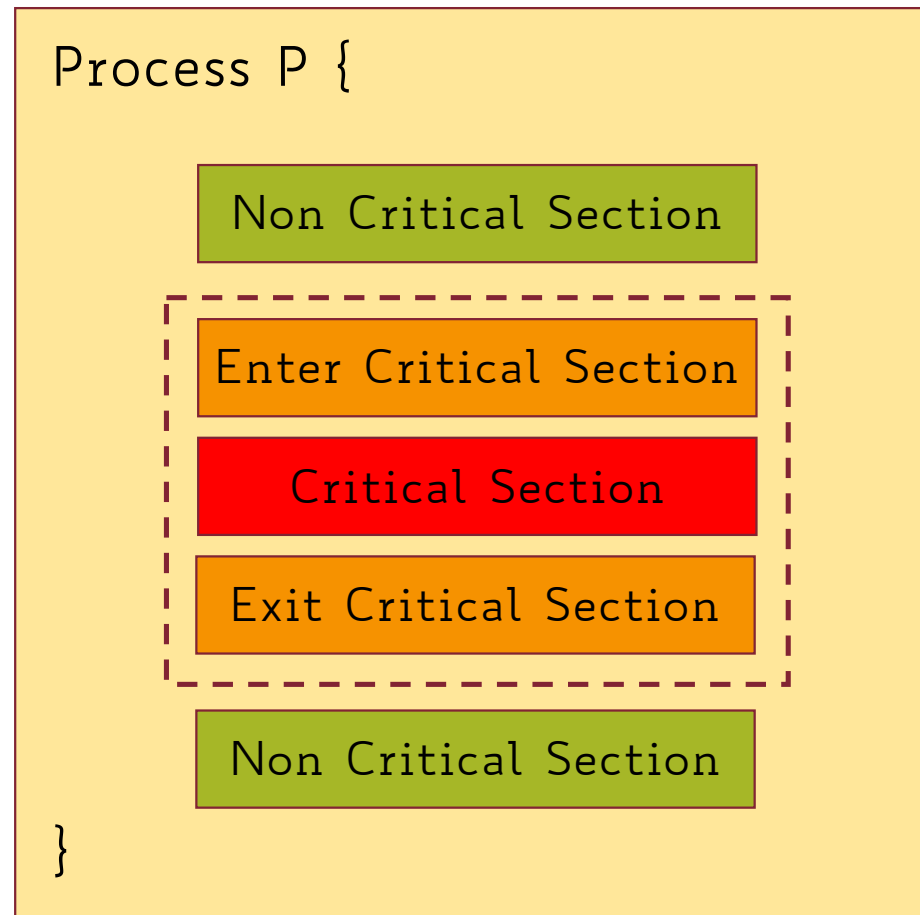
- In the example, **Bob** and **Carla** represents 2 processes/threads
- Theoretically, they should cooperate to achieve a common task (e.g., buying some milk)
- In practice, though, they might incur in unpleasant situations (e.g., buying too much milk!)

What mechanism do we need to get independent yet cooperating processes to communicate with each other and have a consistent view of the "world" (i.e., computational state)?

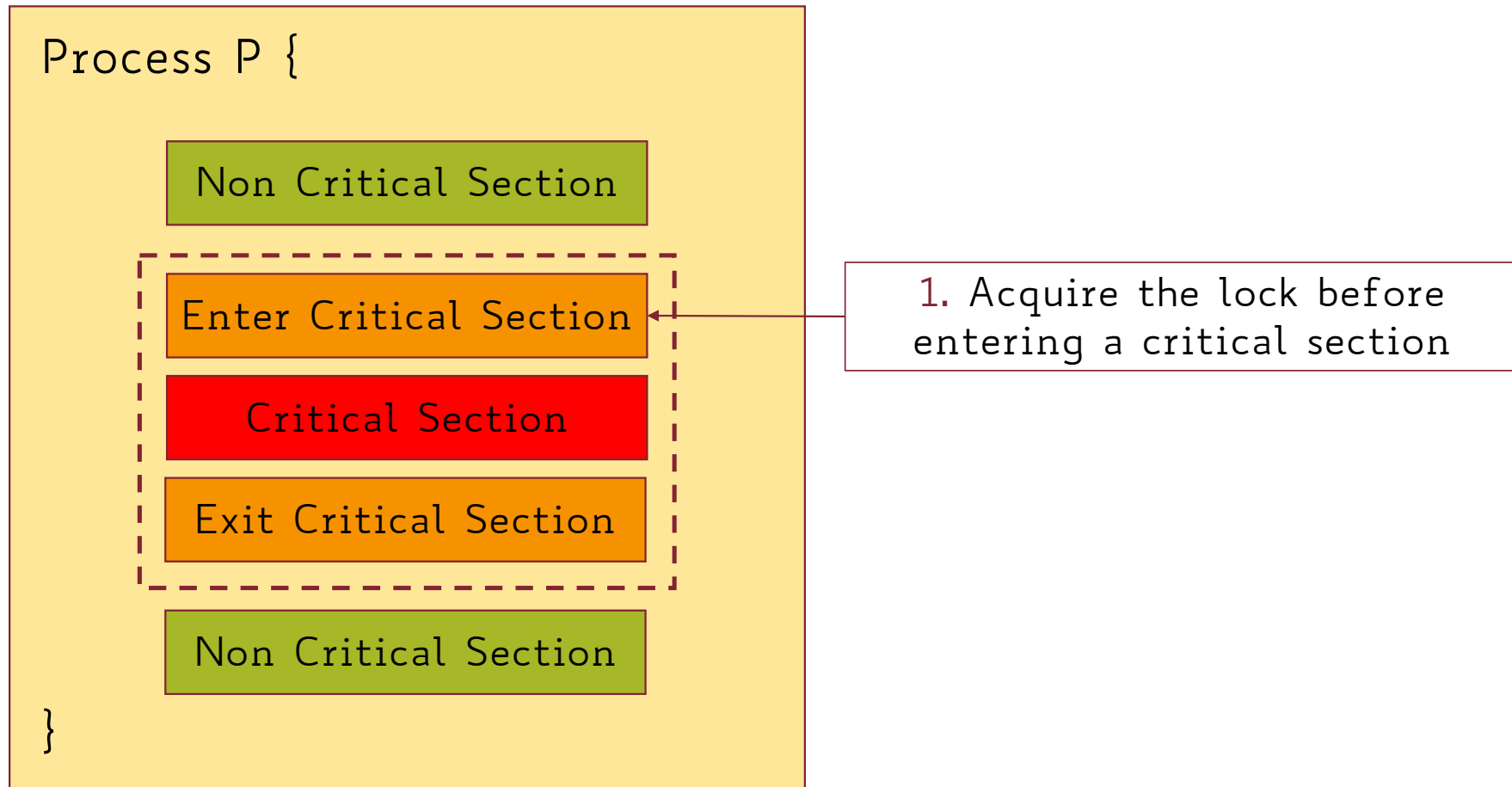
# The Critical Section Problem



# The Anatomy of a Critical Section

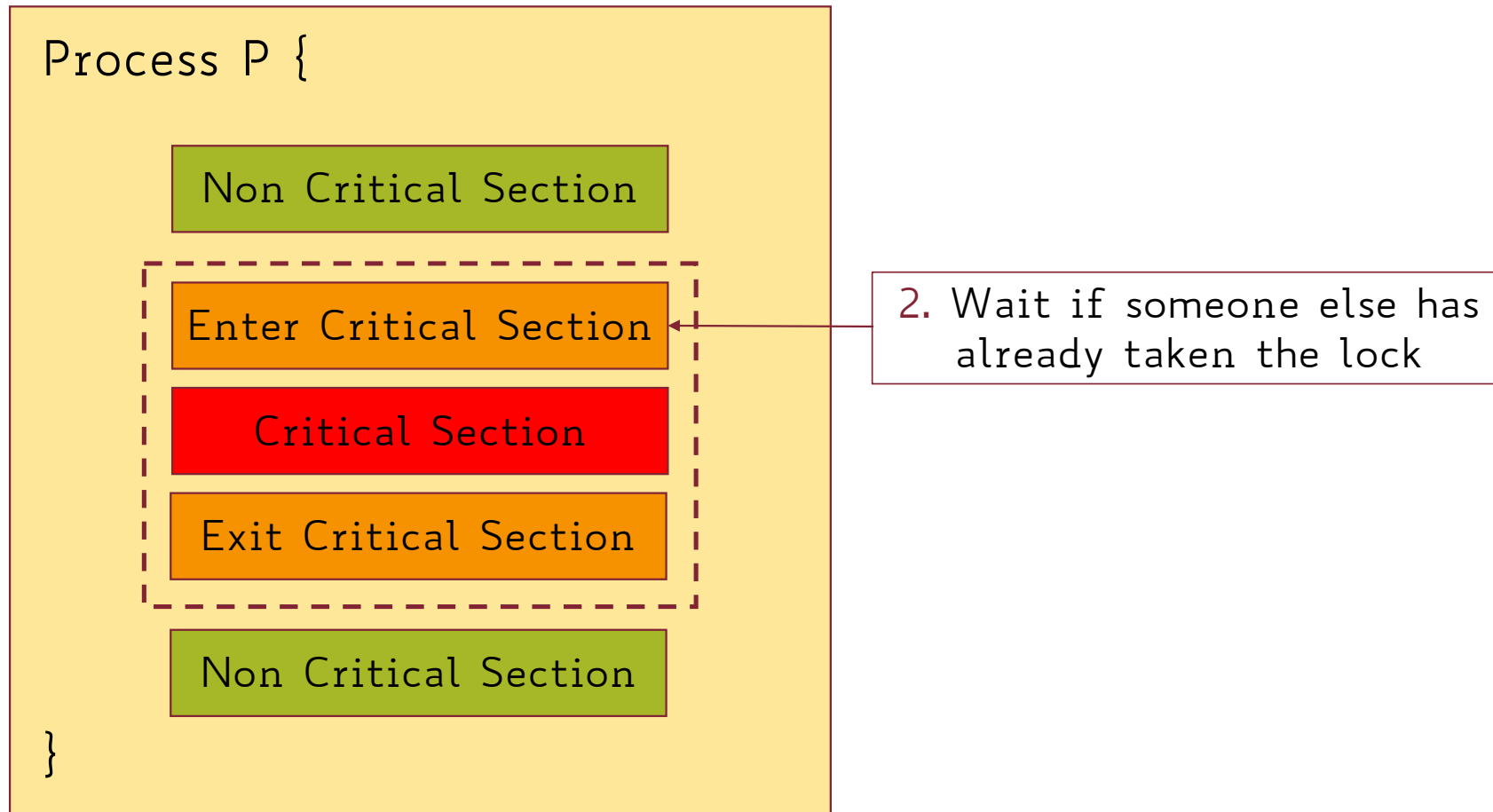


# Locking Critical Section

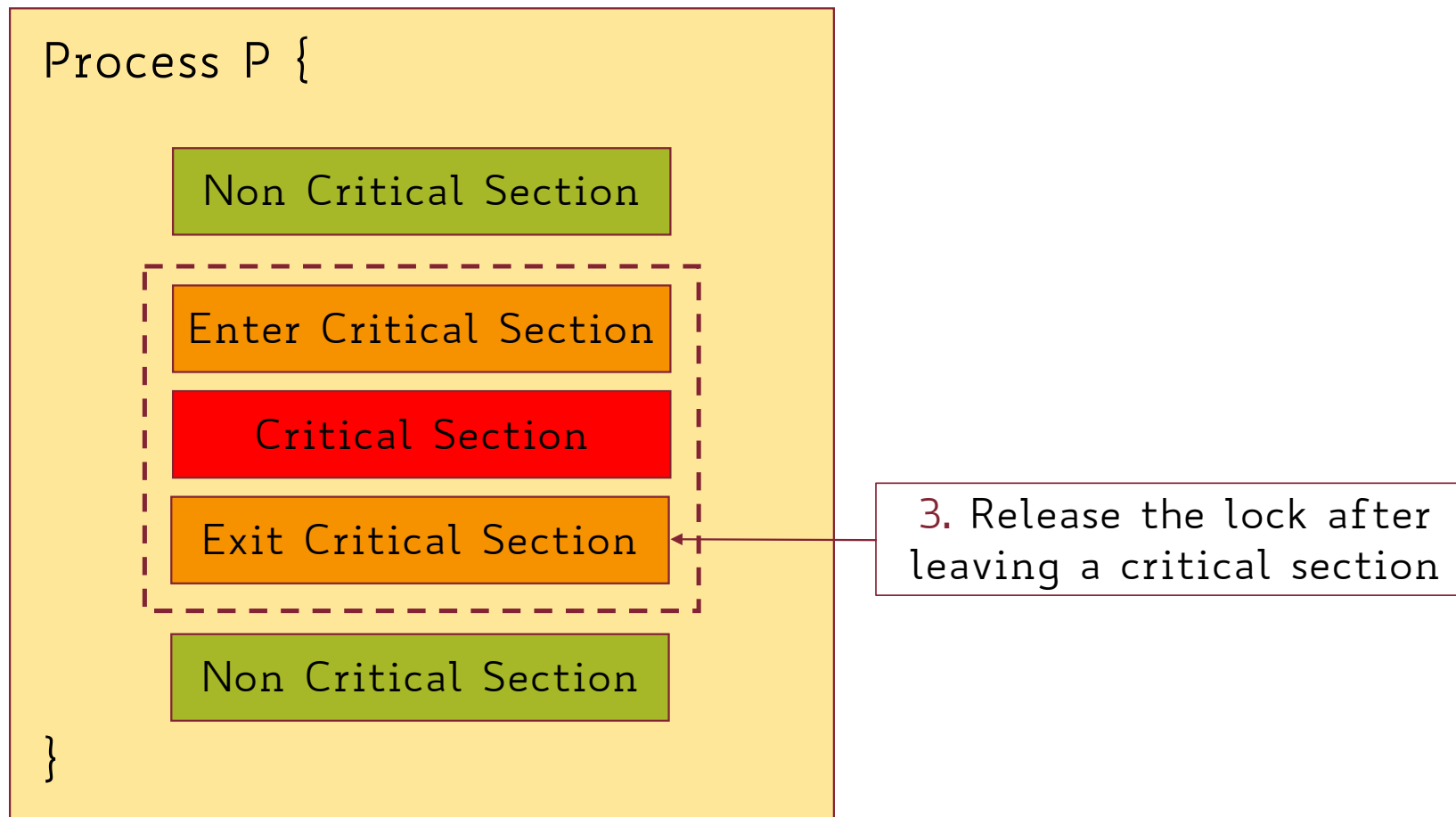




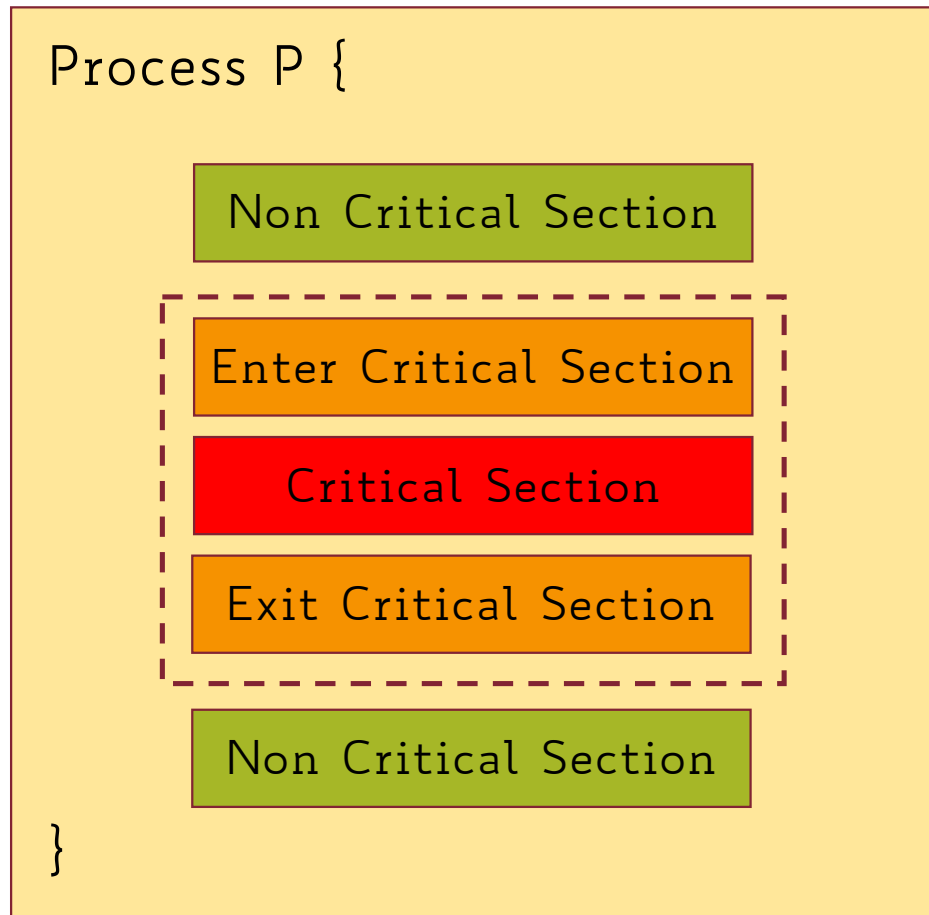
# Locking Critical Section



# Locking Critical Section



# Locking Critical Section



All synchronization  
involves waiting!

# Synchronization: Goals

- Any synchronization solution to the critical section problem must satisfy 3 properties:

# 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 **Carla** 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 **Carla** will buy milk if needed)
  - Ensuring **liveness** means that someone should buy some milk (i.e., the option where both **Bob** and **Carla** 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 **Carla** will buy milk if needed)
  - Ensuring **liveness** means that someone should buy some milk (i.e., the option where both **Bob** and **Carla** do not do anything is surely safe but undesirable)
  - Ensuring **bounding waiting** means that eventually **Bob** and **Carla** 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 Carla
```

```
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 Carla  
  
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 Carla  
  
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 Carla  
  
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(Carla)):
    if (!milk):
        buy_milk()

remove_note()
```

```
# Thread Carla

leave_note(Carla)

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(Carla)):  
    if (!milk):  
        buy_milk()  
  
remove_note()
```

```
# Thread Carla  
  
leave_note(Carla)  
  
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(Carla)):  
    if (!milk):  
        buy_milk()  
  
remove_note()
```

```
# Thread Carla  
  
leave_note(Carla)  
  
if (!note(Bob)):  
    if (!milk):  
        buy_milk()  
  
remove_note()
```

Does this solution work regardless of the scheduling?

No! Liveness property can be violated

# Too Much Milk: Solution 3

Use 2 (labeled) notes... more cleverly

```
# Thread Bob

leave_note(Bob)

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

remove_note()
```

```
# Thread Carla

leave_note(Carla)

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(Carla)):
    do_nothing()
if (!milk):
    buy_milk()

remove_note()
```

```
# Thread Carla

leave_note(Carla)

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(Carla)):  
    do_nothing()  
if (!milk):  
    buy_milk()  
  
remove_note()
```

```
# Thread Carla  
  
leave_note(Carla)  
  
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(Carla)):  
    do_nothing()  
if (!milk):  
    buy_milk()  
  
remove_note()
```

Y: →

```
# Thread Carla  
  
leave_note(Carla)  
  
if (!note(Bob)):  
    if (!milk):  
        buy_milk()  
  
remove_note()
```

# Too Much Milk: Solution 3

```
# Thread Bob  
leave_note(Bob)  
  
while (note(Carla)):  
    do_nothing()  
if (!milk):  
    buy_milk()  
  
remove_note()
```

Y: →

```
# Thread Carla  
leave_note(Carla)  
  
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(Carla)):
    do_nothing()
if (!milk):
    buy_milk()

remove_note()
```

Y: →

```
# Thread Carla
leave_note(Carla)

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(Carla)):
    do_nothing()
if (!milk):
    buy_milk()
remove_note()
```

Y: →

```
# Thread Carla
leave_note(Carla)

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

Case 1: no note from Bob



Thread Bob must be  
executing different  
code



Carla will buy milk  
only if needed



# Too Much Milk: Solution 3

```
# Thread Bob  
leave_note(Bob)  
  
while (note(Carla)):  
    do_nothing()  
if (!milk):  
    buy_milk()  
  
remove_note()
```

Y: →

```
# Thread Carla  
leave_note(Carla)  
  
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(Carla)):  
    do_nothing()  
if (!milk):  
    buy_milk()  
  
remove_note()
```

Y: →

```
# Thread Carla  
leave_note(Carla)  
  
if (!note(Bob)):  
    if (!milk):  
        buy_milk()  
  
remove_note()
```

Case 2: Bob has left a note



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

# Too Much Milk: Solution 3

```
# Thread Bob  
leave_note(Bob)  
  
while (note(Carla)):  
    do_nothing()  
    if (!milk):  
        buy_milk()  
  
remove_note()
```

Y: →

```
# Thread Carla  
leave_note(Carla)  
  
if (!note(Bob)):  
    if (!milk):  
        buy_milk()  
  
remove_note()
```

Case 2: Bob has left a note



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



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

# Too Much Milk: Solution 3

X: →

```
# Thread Bob
leave_note(Bob)
while (note(Carla)):
    do_nothing()
    if (!milk):
        buy_milk()
remove_note()
```

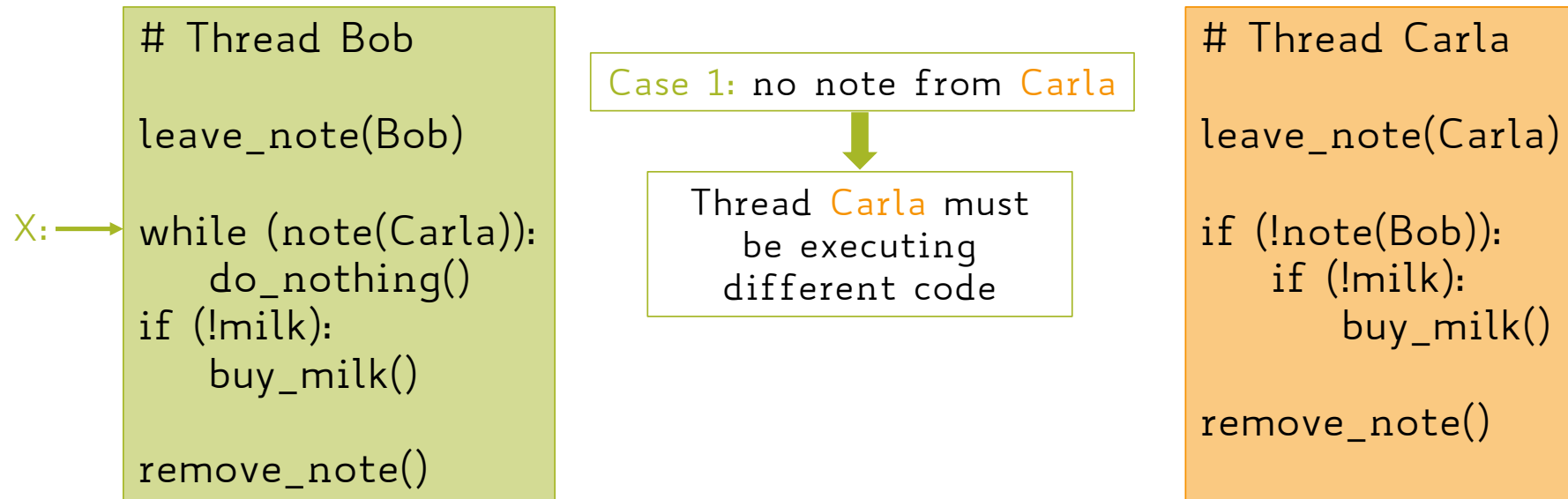
Case 1: no note from Carla

```
# Thread Carla
leave_note(Carla)

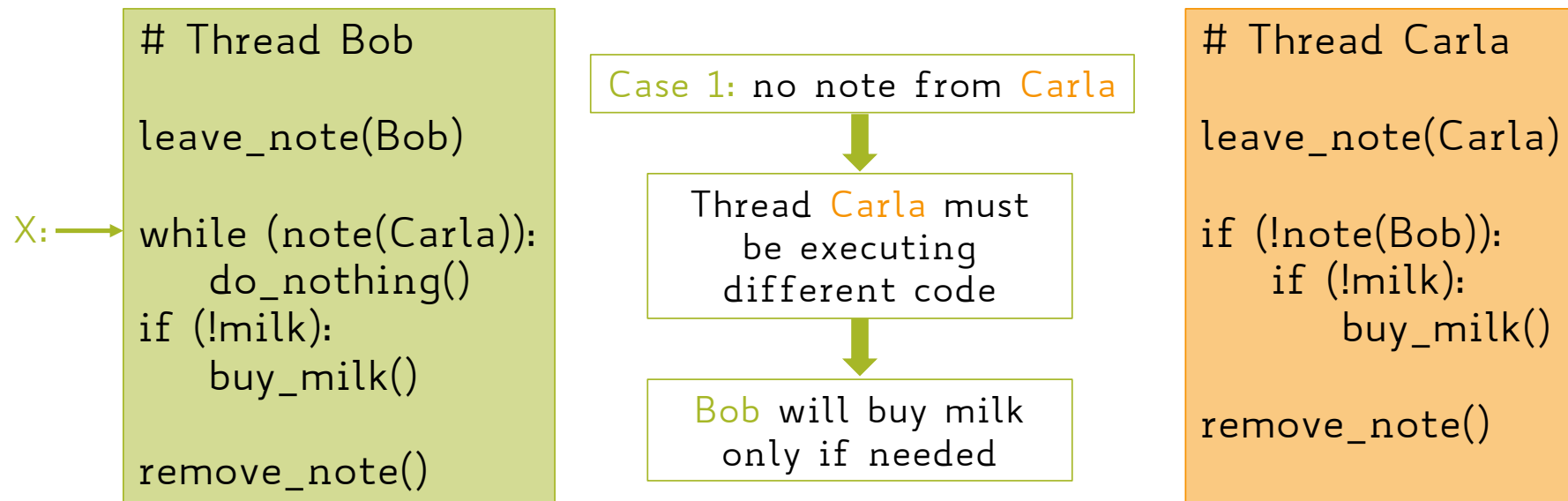
if (!note(Bob)):
    if (!milk):
        buy_milk()

remove_note()
```

# Too Much Milk: Solution 3



# Too Much Milk: Solution 3



# Too Much Milk: Solution 3

X: →

```
# Thread Bob
leave_note(Bob)

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

remove_note()
```

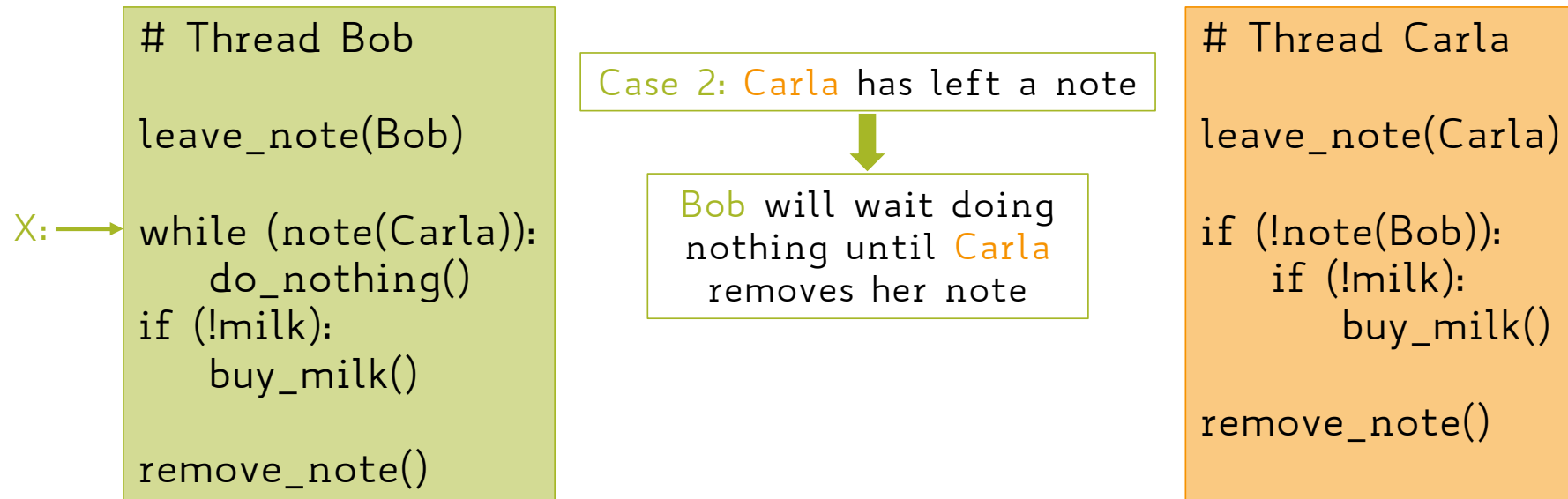
Case 2: Carla has left a note

```
# Thread Carla
leave_note(Carla)

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

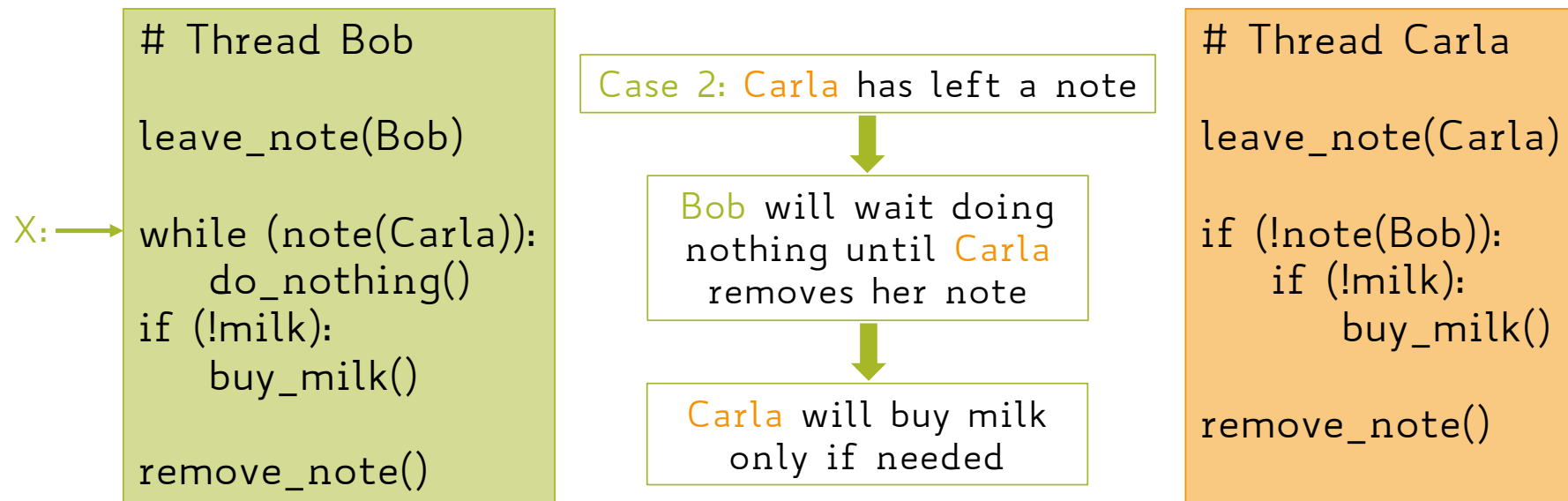
remove_note()
```

# Too Much Milk: Solution 3





# Too Much Milk: Solution 3



# 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 Carla 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 Carla 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 Carla 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 Carla  
  
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 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

# 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 interrupts, atomic instructions (test&set)

# 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** ensure only one thread at a time executes a critical section (**mutual exculsion**), e.g., **locks**