

# Synchronisation

## Module 4

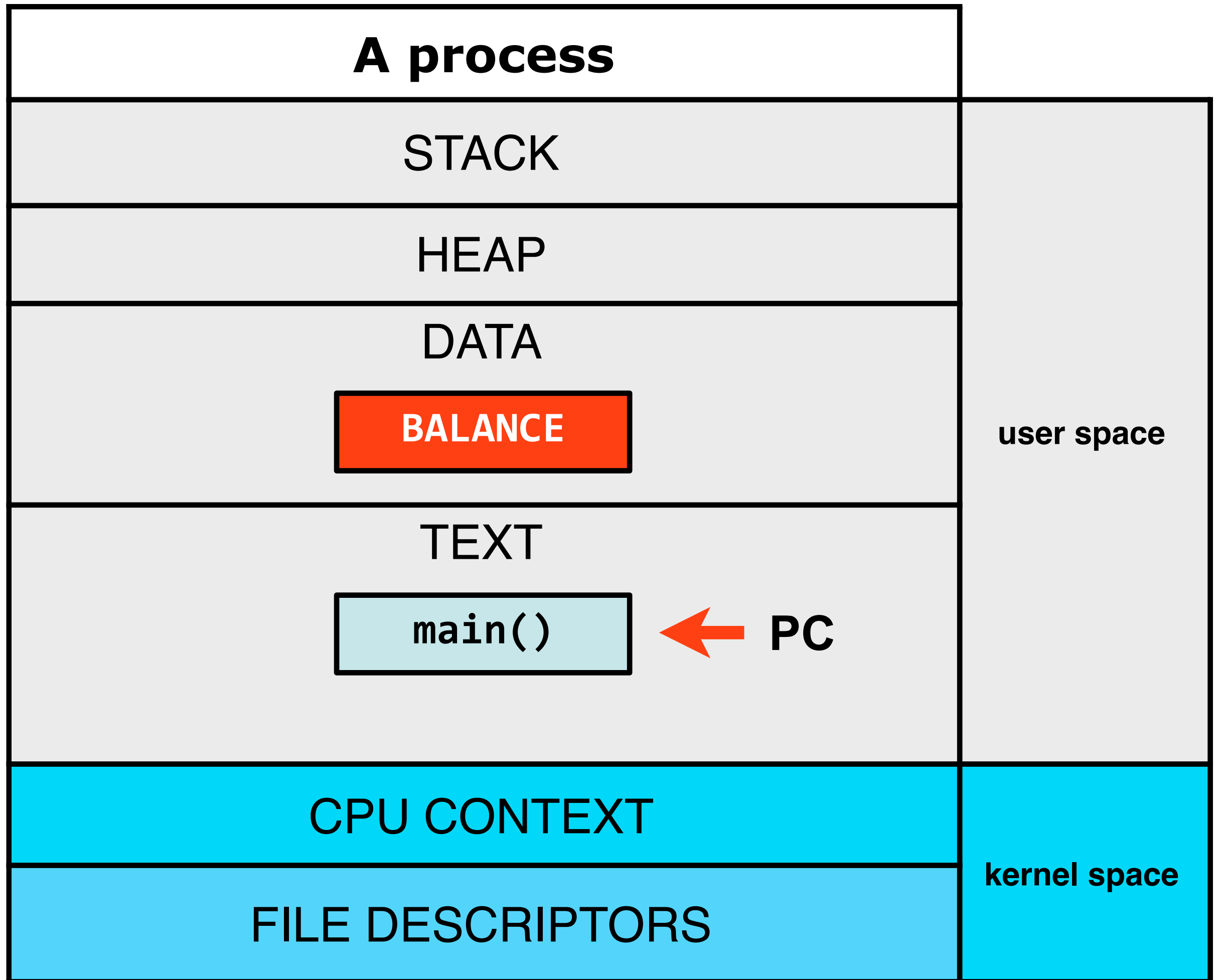
### Lecture

---

**Operating systems 2019**

**1DT044, 1DT096 and 1DT003**

# Background



# main()

```
#define N 10000
```

```
int BALANCE 0;
```

Thread A

Thread B

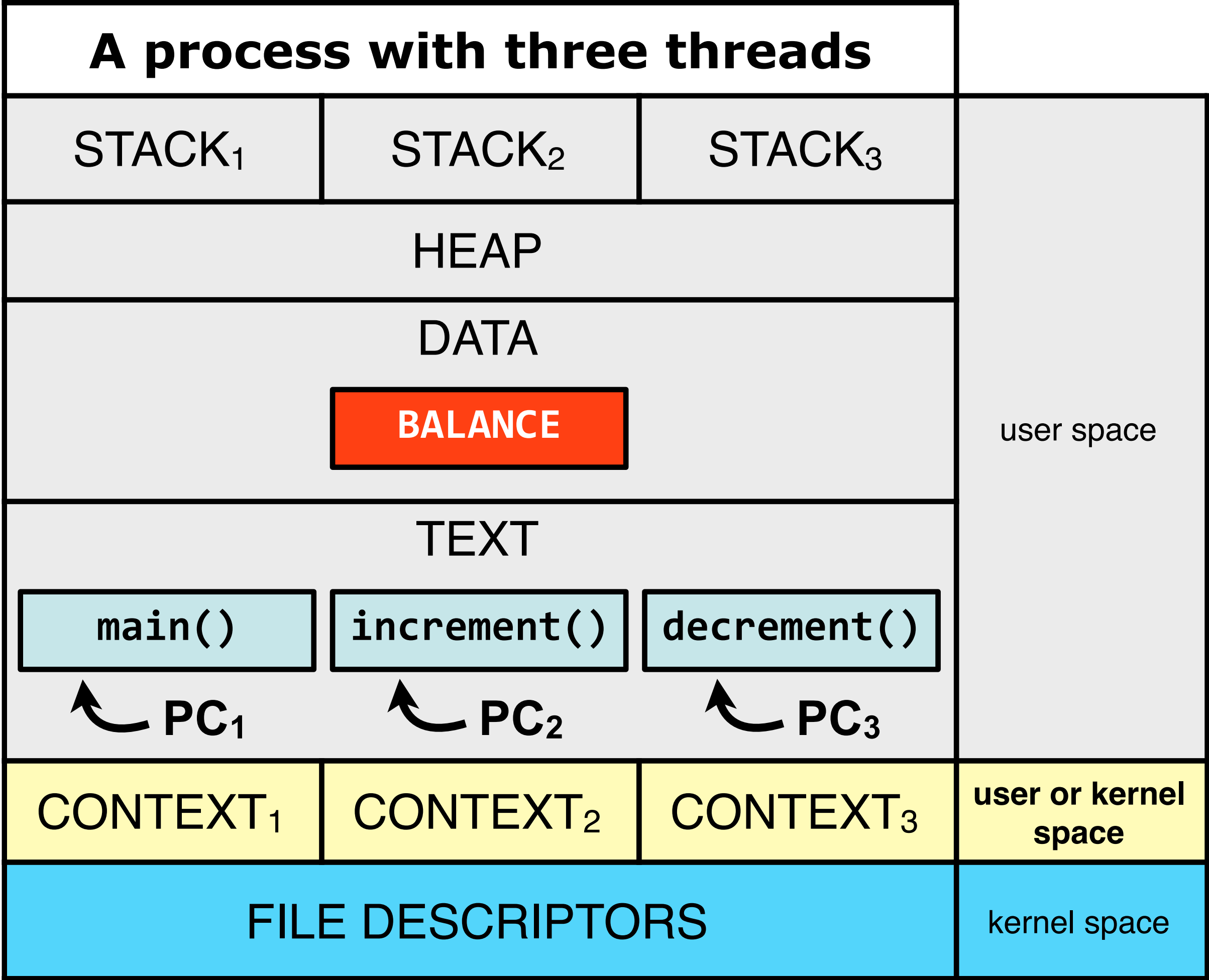
## increment()

```
for (int i = 0; i < N; i++) {  
    BALANCE++;  
}
```

## decrement()

```
for (int i = 0; i < N; i++) {  
    BALANCE--;  
}
```

**BALANCE == ?**



# Atomic operations

- ★ In concurrent programming, an operation (or set of operations) is **atomic** if it appears to the rest of the system to occur at once without being interrupted.
- ★ Other words used synonymously with atomic are: linearizable, indivisible or uninterruptible.
- ★ Additionally, atomic operations commonly have a succeed-or-fail definition—they either successfully change the state of the system, or have no apparent effect.

# Non-atomic operations

When the compiler translates the **shared++** and **shared--** statements, these will be translated to a series of machine instructions (depending on the CPU architecture). On a load/store architecture (for example MIPS):

## **shared++**

- 1) **load** shared from memory into CPU register
- 2) **increment** shared and save result in register
- 3) **store** result back to memory

## **shared--**

- 1) **load** shared from memory into CPU register
- 2) **decrement** shared and save result in register
- 3) **store** result back to memory

# Interleaving of executing threads

Thread A (increment)			BALANCE	Thread B (decrement)		
OP	Operands	\$t0		OP	Operands	\$t0
			0			
lw	\$t0,	0	0			
addi	\$t0, \$t0, 1	1	0			
sw	\$t0, BALANCE	1	1			
			1	lw	\$t0, BALANCE	1
			1	addi	\$t0, \$t0, -1	0
			0	sw	\$t0, BALANCE	0

If the instructions are executed in this order, the increment and decrement cancel each other and the resulting **BALANCE** is **0**.



# Interleaving of executing threads

Thread A (increment)			BALANCE	Thread B (decrement)		
OP	Operands	\$t0		OP	Operands	\$t0
			0			
lw	\$t0, BALANCE	0	0			
			0	lw	\$t0, BALANCE	0
addi	\$t0, \$t0, 1	1	0			
			0	addi	\$t0, \$t0, -1	-1
			-1	sw	\$t0, BALANCE	-1
sw	\$t0, BALANCE	1	+1			

Both threads tries to access and update the shared memory location **BALANCE** concurrently. Updates are not atomic and the result depends on the particular order in which the data accesses take place. In this example the resulting **BALANCE** is **+1**.

# Interleaving of executing threads

Thread A (increment)			BALANCE	Thread B (decrement)		
OP	Operands	\$t0		OP	Operands	\$t0
			0			
lw	\$t0, BALANCE	0	0			
			0	lw	\$t0, BALANCE	0
addi	\$t0, \$t0, 1	1	0			
sw	\$t0, BALANCE	1	1			
			1	addi	\$t0, \$t0, -1	-1
			-1	sw	\$t0, BALANCE	-1

Both threads tries to access and update the shared memory location **BALANCE** concurrently. Updates are not atomic and the result depends on the particular order in which the data accesses take place. In this example the resulting **BALANCE** is **-1**.

# Race condition

A race condition or race hazard is the behaviour of an electronic, software or other system where the output is dependent on the sequence or timing of other uncontrollable events.

It becomes a bug when events do not happen in the intended order.

The term originates with the idea of two signals racing each other to influence the output first.

# Data race

A data race occurs when two instructions from different threads access the same memory location and:

- ★ at least one of these accesses is a write
- ★ and there is no synchronization that is mandating any particular order among these accesses.

# main()

```
#define N 10000
```

```
int BALANCE 0;
```

Thread A

Thread B

## increment()

```
for (int i = 0; i < N; i++) {  
    BALANCE++;  
}
```

## decrement()

```
for (int i = 0; i < N; i++) {  
    BALANCE--;  
}
```

**BALANCE == ?**

# main()

```
#define N 10000
```

```
int BALANCE 0;
```

Thread A

Thread B

## increment()

```
for (int i = 0; i < N; i++) {  
    BALANCE++;  
}
```

## decrement()

```
for (int i = 0; i < N; i++) {  
    BALANCE--;  
}
```

$-N \leq \text{BALANCE} \leq N$

shared state

```
#define N 10000  
  
int BALANCE 0;
```

Thread A

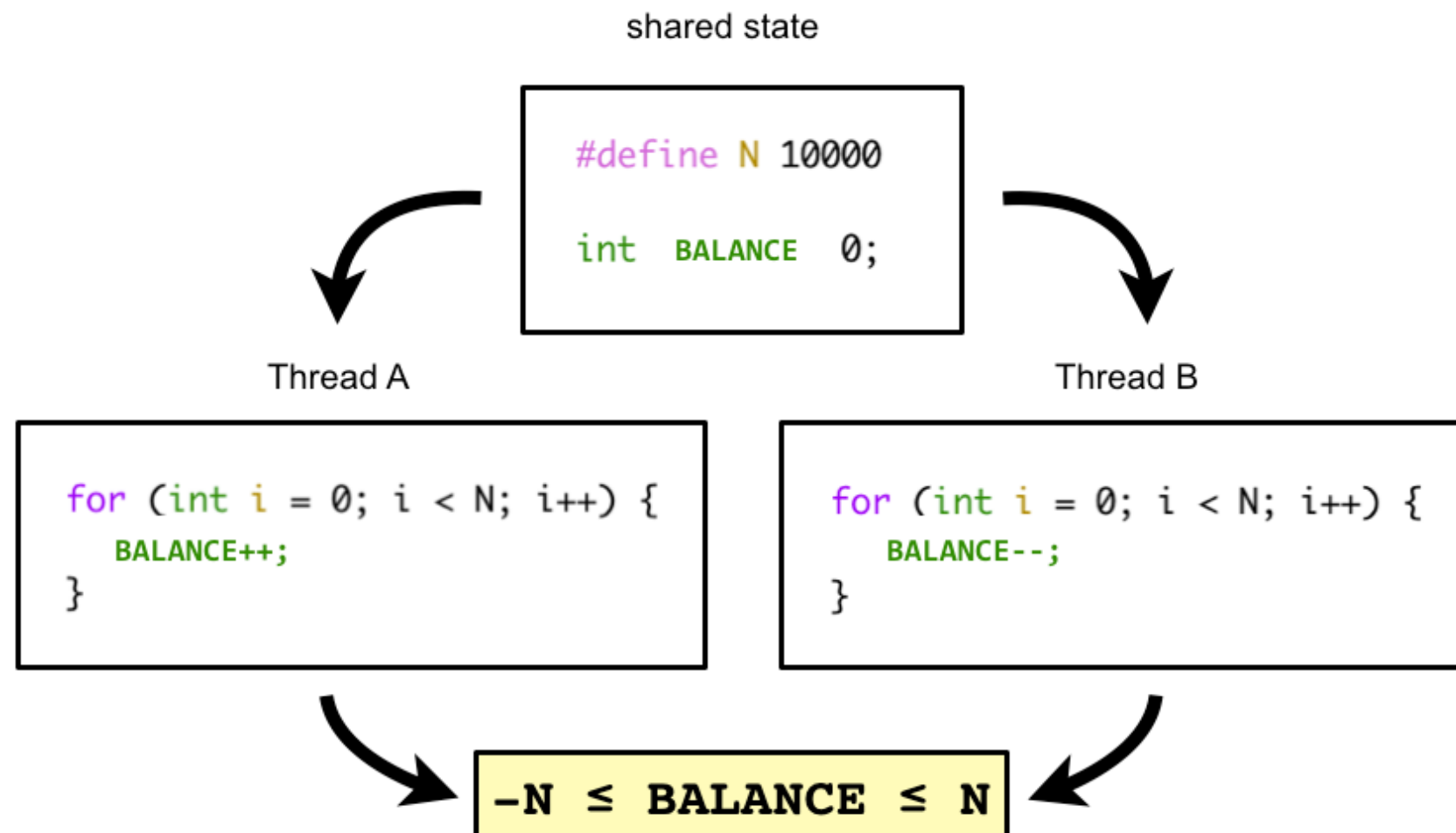
```
for (int i = 0; i < N; i++) {  
    BALANCE++;  
}
```

Thread B

```
for (int i = 0; i < N; i++) {  
    BALANCE--;  
}
```

**$-N \leq \text{BALANCE} \leq N$**

Generally, the OS gives no guarantees regarding the interleaving of the threads. Depending on the non deterministic interleaving of the threads, the program may give different results if executed several times.



Generally, the OS gives no guarantees regarding the interleaving of the threads. Depending on the non deterministic interleaving of the threads, the program may give different results if executed several times.

## Testing and debugging

When a program is executing concurrently, there are many different execution paths. Testing and debugging concurrent programs is inherently more difficult than testing and debugging single-threaded applications.

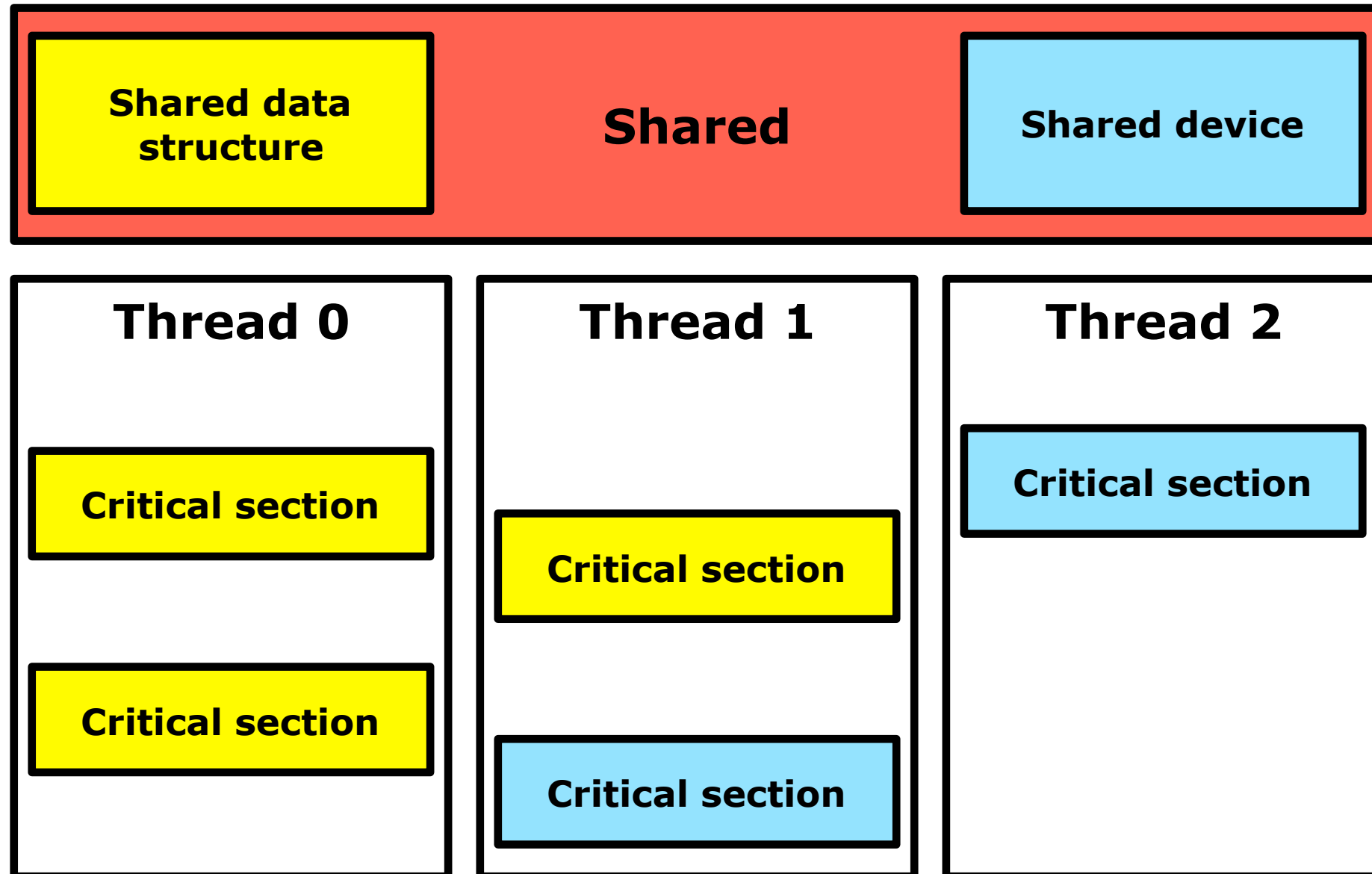


# Critical section

Part of a program that should not be concurrently executed by more than one of the program's concurrent processes or threads at a time.

# Critical section

- ★ Typically, the critical section **accesses a shared resource**, such as a data structure, a peripheral device, or a network connection, that does not allow multiple concurrent accesses.
- ★ By carefully controlling which variables are modified inside and outside the critical section, concurrent access to that state is prevented.



Three threads accessing a shared data structure (yellow) and a shared device (blue).

# Mutual exclusion

The requirement of ensuring that no two concurrent processes or threads are in their critical section at the same time.

- ★ A basic requirement in concurrency control, to prevent race conditions.
- ★ First identified and solved by **Edsger W. Dijkstra** in his seminal 1965 paper titled *Solution of a problem in concurrent programming control*, and is credited as the first topic in the study of concurrent algorithms.

# Conclusions

- ★ Concurrent access to shared data may result in **data races** (data inconsistency) or race conditions.
- ★ Concurrent access to shared data should be done in a **critical section**.
- ★ Must ensure **mutual exclusion**, i.e., ensure that one thread of execution never enter its critical section at the same time that another concurrent thread of execution enters its own critical section.

# **Solution to the critical-section problem**

How can we ensure atomic access to the various critical sections by the various threads?

# Properties of critical sections

Assume that each process/thread executes at a nonzero speed. No assumption concerning relative speed of the  $N$  processes. In the following discussion the term **task** is used to mean a concurrent unit of execution such as a process or thread.

## Mutual Exclusion

If task  $T_i$  is executing in its critical section, then no other tasks can be executing in their critical sections.

## Bounded Waiting

A bound must exist on the number of times that other tasks are allowed to enter their critical sections after a task has made a request to enter its critical section and before that request is granted.

## Progress

If no task is executing in its critical section and there exist some task that wish to enter their critical section, then the selection of the task that will enter the critical section next cannot be postponed indefinitely.

More?

**Deadlock**

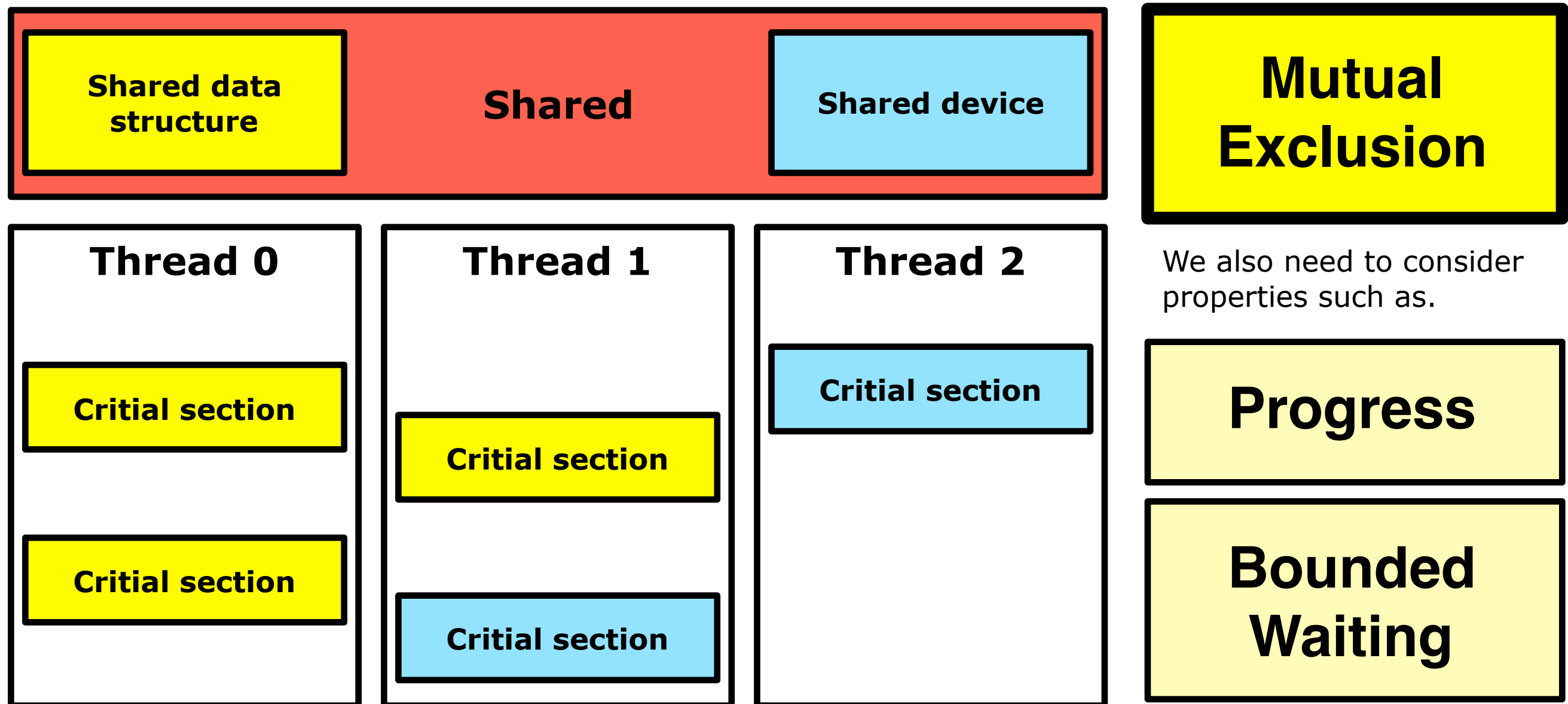
**Fairness**

**Starvation**

We will come back to this ...

# Solution to the critical-section problem

How can we ensure atomic access to the various critical sections by the various threads?

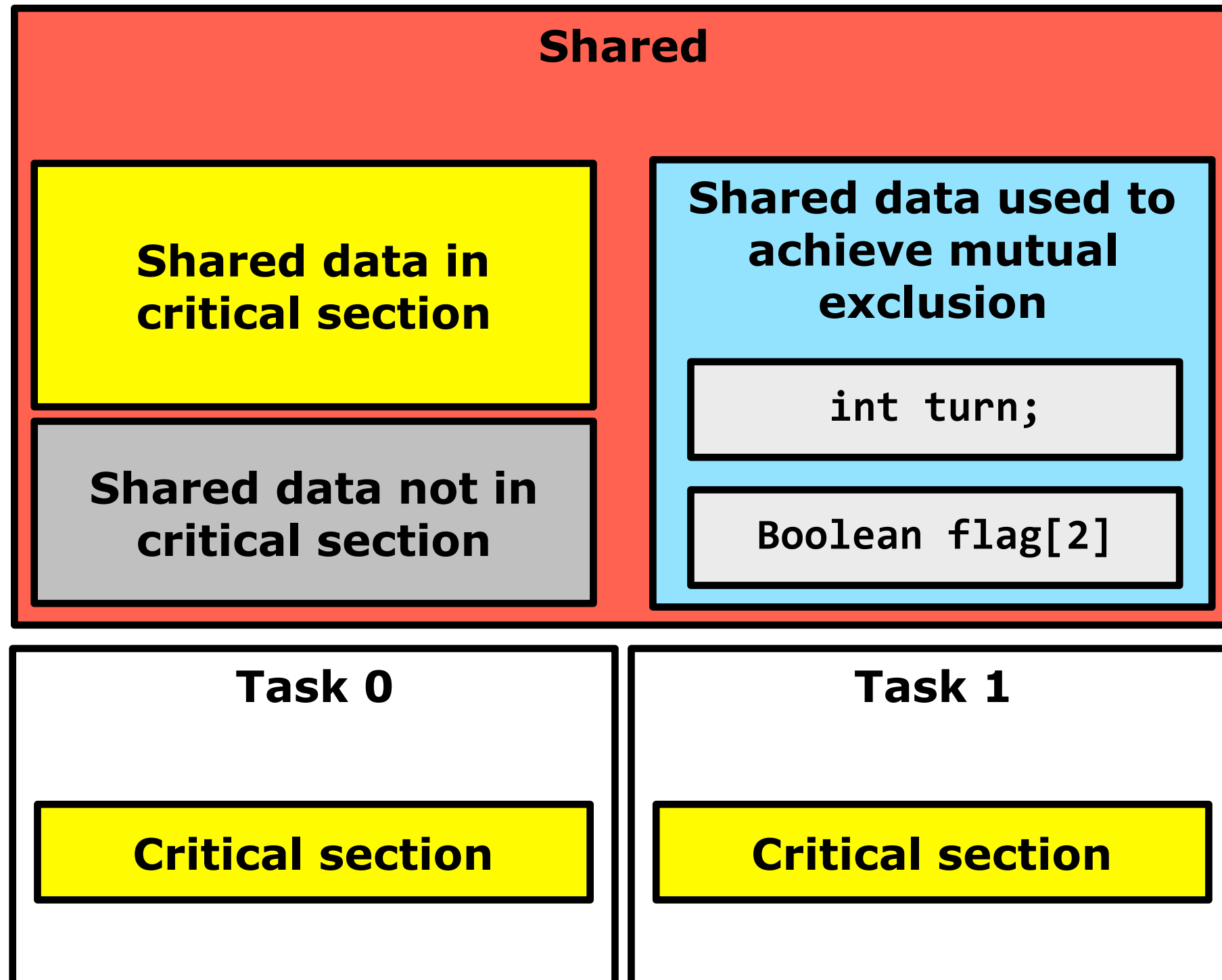




# Peterson's solution

Peterson's algorithm (aka Peterson's solution) is a concurrent programming algorithm for **mutual exclusion** that allows **two tasks** to **share** a single-use **resource** without conflict, **using only shared memory** for communication. It was formulated by Gary L. Peterson in 1981.

The variable **turn** indicates whose turn it is to enter the critical section, **turn** = 0 if task 0 and **turn** = 1 if task 1.

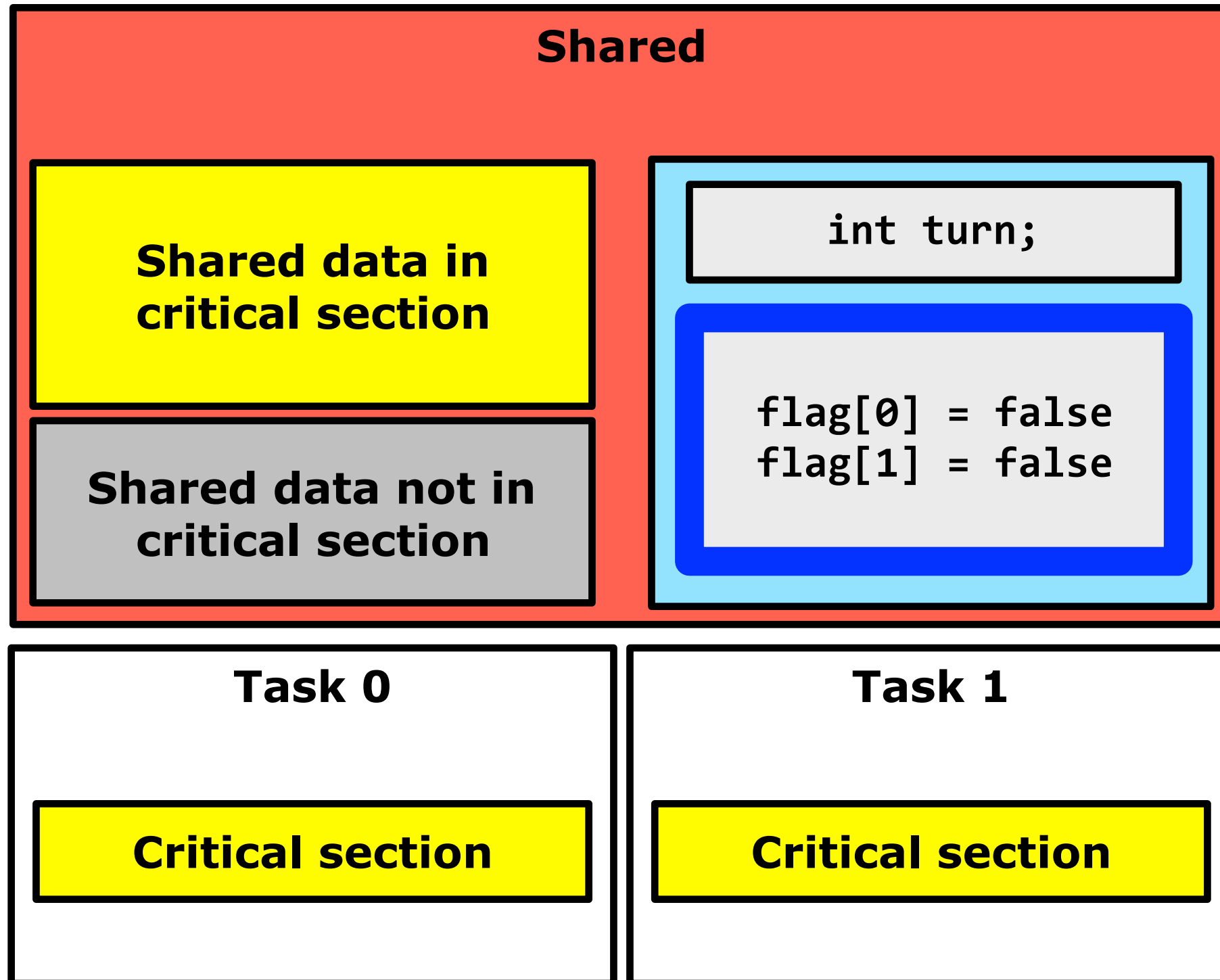


The **flag** array is used to indicate if a process is ready to enter the critical section, i.e.,

**flag[i] = true**

implies that task  $T_i$  is ready!

Initially, neither of  $T_0$  or  $T_1$  is ready to enter the critical section.



# Competing for the critical section

## Task 0

```
flag[0] = true;
```

```
turn = 1;
```

```
while (flag[1] && turn == 1){  
    // busy wait  
}
```

**Critical section**

```
flag[0] = false;
```

## Task 1

```
1 flag[1] = true;
```

```
2 turn = 0;
```

```
3 while (flag[0] && turn == 0){  
    // busy wait  
}
```

**Critical section**

```
4 flag[1] = false;
```

1  $T_i$  wants to enter critical section

3 For  $T_i$  to be able to enter the critical section, the while condition must fail.

2  $T_i$  is polite, after you  $T_{(i+1 \% 2)}$

4  $T_i$  signals - I've left the critical section.

- 3  $T_i$  will block as long as the other task raised its **flag** and its the other tasks **turn** to enter the critical section.
- For  $T_i$  to be allowed enter the critical section, either the other task has **A** left the critical section (put down its **flag**), or **B** the other task says "after you" (setting the value of **turn**).

### Task 0

```
flag[0] = true;
```

```
turn = 1; B
```

```
while (flag[1] && turn == 1){  
    // busy wait  
}
```

**Critical section**

```
flag[0] = false; A
```

### Task 1

```
flag[1] = true;
```

```
turn = 0; B
```

```
while (flag[0] && turn == 0){  
    // busy wait  
}
```

**Critical section**

```
flag[1] = false; A
```

3

# Properties of critical sections

Assume that each process/thread executes at a nonzero speed. No assumption concerning relative speed of the  $N$  processes. In the following discussion the term **task** is used to mean a concurrent unit of execution such as a process or thread.

## Mutual Exclusion

If task  $T_i$  is executing in its critical section, then no other tasks can be executing in their critical sections.

## Bounded Waiting

A bound must exist on the number of times that other tasks are allowed to enter their critical sections after a task has made a request to enter its critical section and before that request is granted.

## Progress

If no task is executing in its critical section and there exist some task that wish to enter their critical section, then the selection of the task that will enter the critical section next cannot be postponed indefinitely.

## Deadlock

## Fairness

## Starvation

We will come back to this ...

# Mutual exclusion (mutex)

$T_0$  and  $T_1$  can never be in the critical section at the same time: If  $T_0$  is in its critical section, then `flag[0]` is `true` and either:

- `flag[1]` is `false` (meaning  $T_1$  has left its critical section) or
- `turn` is `0` (meaning  $T_1$  is just now trying to enter the critical section, but graciously waiting). In both cases,  $T_1$  cannot be in its critical section.

## Task 0

```
flag[0] = true;
```

```
turn = 1;
```

```
while (flag[1] && turn == 1){  
    // busy wait  
}
```

**Critical section**

```
flag[0] = false;
```

## Task 1

```
flag[1] = true;
```

```
turn = 0;
```

```
while (flag[0] && turn == 0){  
    // busy wait  
}
```

**Critical section**

```
flag[1] = false;
```

# Properties of critical sections

Assume that each process/thread executes at a nonzero speed. No assumption concerning relative speed of the  $N$  processes. In the following discussion the term **task** is used to mean a concurrent unit of execution such as a process or thread.

## Mutual Exclusion

If task  $T_i$  is executing in its critical section, then no other tasks can be executing in their critical sections.

## Bounded Waiting

A bound must exist on the number of times that other tasks are allowed to enter their critical sections after a task has made a request to enter its critical section and before that request is granted.

## Progress

If no task is executing in its critical section and there exist some task that wish to enter their critical section, then the selection of the task that will enter the critical section next cannot be postponed indefinitely.

## Deadlock

## Fairness

## Starvation

We will come back to this ...



# Bounded waiting

A process cannot immediately re-enter the critical section if the other process has set its flag to say that it too would like entry to the section.

## Task 0

```
flag[0] = true;  
  
turn = 1;  
  
while (flag[1] && turn == 1){  
    // busy wait  
}
```

**Critical section**

```
flag[0] = false;
```

## Task 1

```
flag[1] = true;  
  
turn = 0;  
  
while (flag[0] && turn == 0){  
    // busy wait  
}
```

**Critical section**

```
flag[1] = false;
```

# Properties of critical sections

Assume that each process/thread executes at a nonzero speed. No assumption concerning relative speed of the  $N$  processes. In the following discussion the term **task** is used to mean a concurrent unit of execution such as a process or thread.

## Mutual Exclusion

If task  $T_i$  is executing in its critical section, then no other tasks can be executing in their critical sections.

## Bounded Waiting

A bound must exist on the number of times that other tasks are allowed to enter their critical sections after a task has made a request to enter its critical section and before that request is granted.

## Progress

If no task is executing in its critical section and there exist some task that wish to enter their critical section, then the selection of the task that will enter the critical section next cannot be postponed indefinitely.

## Deadlock

## Fairness

## Starvation

We will come back to this ...

# Progress

In Peterson's Algorithm, a process will not wait longer than one turn for entrance to the critical section: After giving priority to the other process, this process will run to completion and set its flag to 0, thereby allowing the other process to enter the critical section.

## Task 0

```
flag[0] = true;

turn = 1;

while (flag[1] && turn == 1){
    // busy wait
}
```

**Critical section**

```
flag[0] = false;
```

## Task 1

```
flag[1] = true;

turn = 0;

while (flag[0] && turn == 0){
    // busy wait
}
```

**Critical section**

```
flag[1] = false;
```

# Memory ordering

Memory ordering is a group of properties of the modern microprocessors, characterizing their possibilities in memory operations reordering. It is a type of **out-of-order execution**. Memory reordering can be used to fully utilize different cache and memory banks.

On most modern uniprocessors memory operations are not executed in the order specified by the program code.

But in single-threaded programs from the programmer's point of view, all operations appear to have been executed in the order specified, with all inconsistencies hidden by hardware.

Such processors invariably give some way to force ordering in a stream of memory accesses, typically through a **memory barrier** instruction. This typically means that certain operations are guaranteed to be performed before the barrier, and others after.

Implementation of Peterson's and related algorithms on processors which reorder memory accesses generally requires use of such barrier operations to work correctly to keep sequential operations from happening in an incorrect order.

# Limitations in Peterson's Solution

Peterson's solution is a software based solution to the critical section problem.

- ★ Software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures.
- ★ Peterson's solution only works for two concurrent tasks.
- ★ While Peterson's original formulation worked with only two processes, the algorithm can be generalized for more than two ...

# Locks

In general, any solution to the critical section problem requires a simple tool/abstraction - a **lock**.

```
do {  
    acquire lock  
    // critical section  
    release lock  
    // remainder section  
} while (TRUE);
```

Race conditions are prevented by requiring that critical sections be protected by locks.

- ▶ A process must **acquire** the lock before entering a critical section.
- ▶ A process must **release** the lock when it exits the critical section.

# **Synchronization hardware**

Many systems provide hardware support for critical section code.



# Uniprocessors – could disable interrupts.

- ★ Currently running code would execute without preemption.
- ★ Generally too inefficient on multiprocessor systems.
- ★ Operating systems using this not broadly scalable.

Modern machines provide special **atomic hardware instructions**.

- ★ Atomic = non-interruptable
- ★ Either test memory word and set value, often called a **TAS** (Test And Set) instruction.
- ★ Or **SWAP** contents of two memory words.

# TestAndSet

The **TestAndSet** instruction  
**atomically** first **sets** the **value** at  
the target address **to True** and  
**return** the **old value** stored at the  
target address.

# TestAndSet

Here we use C to define the semantics of the TestAndSet (TAS) instruction. Optimally, the TAS instruction is supported directly by the CPU.

```
boolean TestAndSet (boolean *target) {  
    boolean rv = *target;  
    *target = TRUE;  
    return rv;  
}
```

**TestAndSet** set the value at the target address to True and return the old value at address target. The **TestAndSet** instruction must be **atomic**. If two such instructions are executed simultaneously on the same target address (each on a different CPU or core), they will be executed sequentially in some arbitrary order.

# Swap

The **Swap** instruction **atomically swaps** the content of two memory locations **a** and **b**.

# Swap

Here we use C to define the semantics of the SWAP instruction. Optimally, the TAS instruction is supported directly by the CPU.

```
void Swap (boolean *a, boolean *b) {  
    boolean temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

**Swap** swaps the content of two memory locations **a** and **b**. The **Swap** instruction must be **atomic**. If two such instructions are executed simultaneously on the same target address (each on a different CPU or core), they will be executed sequentially in some arbitrary order.

# Spinlock using TestAndSet

The **TestAndSet** instruction atomically sets the value at the target address to True and return the old value stored at the target address.

► Shared boolean variable **lock** initialized to **false**.

```
do {
```

```
    grab lock
```

```
    /* critical section */
```

```
    release lock
```

```
    /* remainder section */
```

```
} while (TRUE);
```

	lock
Before entering the critical section	<b>FALSE</b>
Inside the critical section	<b>TRUE</b>
After the critical section	<b>FALSE</b>



► Shared boolean variable **lock** initialized to **false**.

```
do {
```

**grab lock**

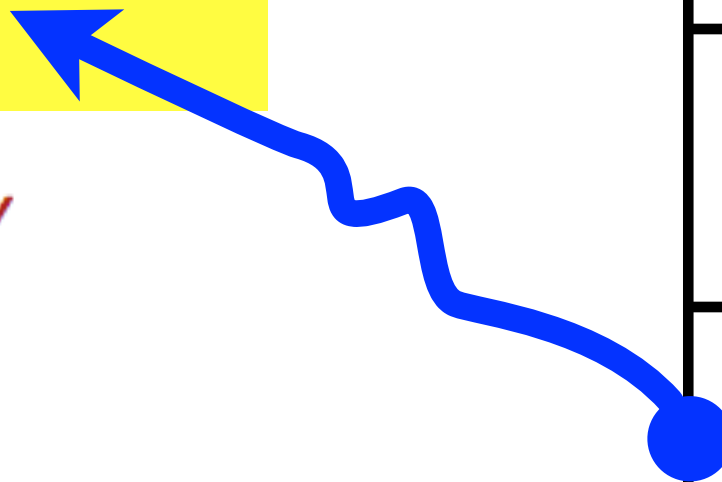
```
/* critical section */
```

**release lock**

```
/* remainder section */
```

```
} while (TRUE);
```

<b>lock</b>	
Before entering the critical section	<b>FALSE</b>
Inside the critical section	<b>TRUE</b>
After the critical section	<b>FALSE</b>



► Shared boolean variable **lock** initialized to **false**.

do {

**grab lock**

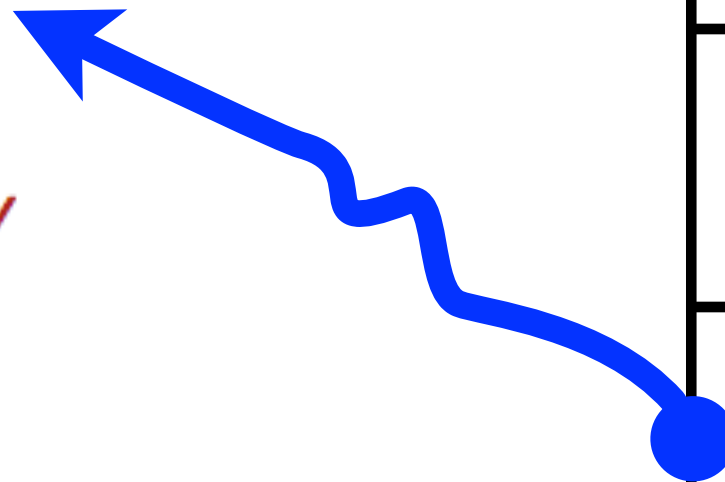
*/\* critical section \*/*

lock = FALSE;

*/\* remainder section \*/*

} while (TRUE);

lock	
Before entering the critical section	<b>FALSE</b>
Inside the critical section	<b>TRUE</b>
After the critical section	<b>FALSE</b>



► Shared boolean variable **lock** initialized to **false**.

```
do {
```

**grab lock**

```
/* critical section */
```

```
lock = FALSE;
```

```
/* remainder section */
```

```
} while (TRUE);
```

**lock**

Before  
entering the  
critical  
section

**FALSE**

Inside the  
critical  
section

**TRUE**

After the  
critical  
section

**FALSE**

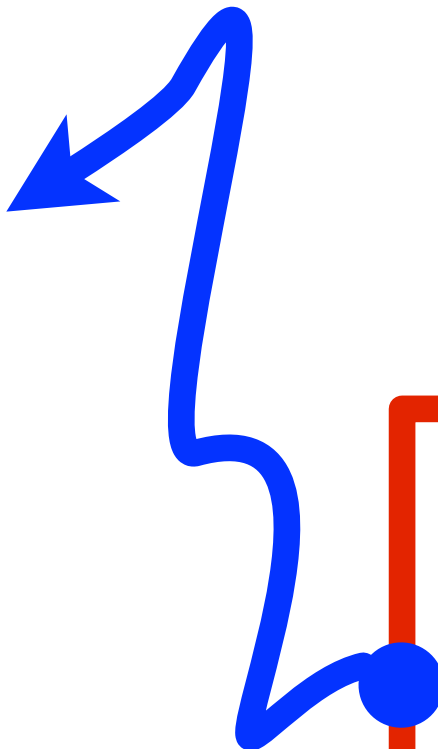
- ▶ Shared boolean variable **lock** initialized to **false**.

```
do {  
  while (           ?           )  
    ; /* do nothing* /  
  
  /* critical section */  
  
  lock = FALSE;  
  
  /* remainder section */  
  
} while (TRUE);
```

<b>lock</b>	
Before entering the critical section	<b>FALSE</b>
Inside the critical section	<b>TRUE</b>
After the critical section	<b>FALSE</b>

► Shared boolean variable **lock** initialized to **false**.

```
do {  
    while ( TestAndSet (&lock ))  
        ;    /* do nothing* /  
  
    /* critical section */  
  
    lock = FALSE;  
  
    /* remainder section */  
  
} while (TRUE);
```



<b>lock</b>	
Before entering the critical section	<b>FALSE</b>
Inside the critical section	<b>TRUE</b>
After the critical section	<b>FALSE</b>

Using **TestAndSet**, once we're allowed to enter the CS, the lock is **atomically** set to true blocking others from entering.

- ▶ Shared boolean variable **lock** initialized to **false**.

```
do {  
    while ( TestAndSet (&lock ))  
        ;    /* do nothing* /  
  
    /* critical section */  
  
    lock = FALSE;  
  
    /* remainder section */  
} while (TRUE);
```

	lock
Before entering the critical section	<b>FALSE</b>
Inside the critical section	<b>TRUE</b>
After the critical section	<b>FALSE</b>

T0	Global shared data	T1
Statement	lock	Statement
	FALSE	
TestAndSet(&lock)	TRUE	
<i>/* critical section */</i>	TRUE	
	TRUE	TestAndSet(&lock)
<i>/* critical section */</i>	TRUE	
lock = FALSE	FALSE	
	FALSE	TestAndSet(&lock)
	TRUE	<i>/* critical section */</i>
TestAndSet(&lock)	TRUE	
	TRUE	<i>/* critical section */</i>

```

do {
    while ( TestAndSet (&lock ))
        ;    /* do nothing* /

    /* critical section */

    lock = FALSE;

    /* remainder section */
} while (TRUE);

```

**Mutual exclusion (mutex):** only one task may execute in the critical section at the time.

# Volatile

In pseudo C it would be like:

```
volatile int lock = 0;

void Critical() {
    while (TestAndSet(&lock) == 1);
    critical section //only one process can be in this section at a time
    lock = 0 //release lock when finished with the critical section
}
```

Note the *volatile* keyword. In absence of volatile, the compiler and/or the CPU(s) will quite certainly optimize access to lock and/or use cached values, thus rendering the above code erroneous.



# Lock using Swap

**Swap** swaps the content of two memory locations a and b.

- ▶ Shared boolean variable **lock** initialized to **FALSE**.
- ▶ Each process has a local **boolean** variable **key**.

```
do {
```

```
    grab lock
```

```
    /* critical section */
```

```
    release lock
```

```
    /* remainder section */
```

```
} while (TRUE);
```

	lock
Before entering the critical section	<b>FALSE</b>
Inside the critical section	<b>TRUE</b>
After the critical section	<b>FALSE</b>

- ▶ Shared boolean variable **lock** initialized to **FALSE**.
- ▶ Each process has a local **boolean** variable **key**.

```
do {
```

**grab lock**

```
/* critical section */
```

**release lock**

```
/* remainder section */
```

```
} while (TRUE);
```

lock	
Before entering the critical section	<b>FALSE</b>
Inside the critical section	<b>TRUE</b>
After the critical section	<b>FALSE</b>

- ▶ Shared boolean variable **lock** initialized to **FALSE**.
- ▶ Each process has a local **boolean** variable **key**.

do {

**grab lock**

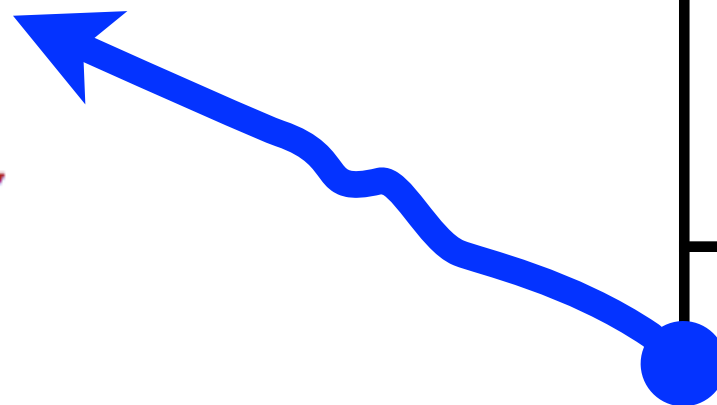
*/\* critical section \*/*

lock = FALSE;

*/\* remainder section \*/*

} while (TRUE);

lock	
Before entering the critical section	<b>FALSE</b>
Inside the critical section	<b>TRUE</b>
After the critical section	<b>FALSE</b>



- ▶ Shared boolean variable **lock** initialized to **FALSE**.
- ▶ Each process has a local **boolean** variable **key**.

```
do {
```

**grab lock**

```
/* critical section */
```

```
lock = FALSE;
```

```
/* remainder section */
```

```
} while (TRUE);
```

**lock**

Before  
entering the  
critical  
section

**FALSE**

Inside the  
critical  
section

**TRUE**

After the  
critical  
section

**FALSE**

- ▶ Shared boolean variable **lock** initialized to **FALSE**.
- ▶ Each process has a local **boolean** variable **key**.

```
do {  
    key = TRUE;  
    while ( key == TRUE)  
        ?  
  
    /* critical section */  
  
    lock = FALSE;  
  
    /* remainder section */  
  
} while (TRUE);
```

<b>lock</b>	
Before entering the critical section	<b>FALSE</b>
Inside the critical section	<b>TRUE</b>
After the critical section	<b>FALSE</b>

- ▶ Shared boolean variable **lock** initialized to **FALSE**.
- ▶ Each process has a local **boolean** variable **key**.

```
do {  
    key = TRUE;  
    while ( key == TRUE)  
        Swap (&lock, &key );  
  
    /* critical section */  
  
    lock = FALSE;  
  
    /* remainder section */  
} while (TRUE);
```

<b>lock</b>	
Before entering the critical section	<b>FALSE</b>
Inside the critical section	<b>TRUE</b>
After the critical section	<b>FALSE</b>

Using **Swap**, once we're allowed to enter the CS, the lock is **atomically** set to true blocking others from entering.

- ▶ Shared boolean variable **lock** initialized to **FALSE**.
- ▶ Each process has a local **boolean** variable **key**.

```
do {  
    key = TRUE;  
    while ( key == TRUE)  
        Swap (&lock, &key );  
  
    /* critical section */  
  
    lock = FALSE;  
  
    /* remainder section */  
} while (TRUE);
```

	lock
Before entering the critical section	<b>FALSE</b>
Inside the critical section	<b>TRUE</b>
After the critical section	<b>FALSE</b>



Local data		Global shared data	Local data	
T0			T1	
key	Statement	lock	Statement	key
?		FALSE		?
True	key = TRUE	FALSE		?
TRUE → FALSE	Swap(&lock, &key)	FALSE → TRUE		?
FALSE	<i>/* critical section */</i>	TRUE		?
FALSE	<i>/* critical section */</i>	TRUE	key = TRUE	TRUE
FALSE	<i>/* critical section */</i>	TRUE	Swap(&lock, &key)	TRUE
FALSE	<i>/* critical section */</i>	TRUE	Swap(&lock, &key)	TRUE
FALSE	lock = FALSE	TRUE → FALSE		TRUE
		FALSE → TRUE	Swap(&lock, &key)	TRUE → FALSE
		TRUE	<i>/* critical section */</i>	FALSE

```

do {
    key = TRUE;
    while ( key == TRUE)
        Swap (&lock, &key );

    /* critical section */

    lock = FALSE;

    /* remainder section */

} while (TRUE);

```

Local data		Global shared data	Local data	
T0			T1	
key	Statement	lock	Statement	key
?		FALSE		?
True	key = TRUE	FALSE		?
TRUE → FALSE	Swap(&lock, &key)	FALSE → TRUE		?
FALSE	<i>/* critical section */</i>	TRUE		?
FALSE	<i>/* critical section */</i>	TRUE	key = TRUE	TRUE
FALSE	<i>/* critical section */</i>	TRUE	Swap(&lock, &key)	TRUE
FALSE	<i>/* critical section */</i>	TRUE	Swap(&lock, &key)	TRUE
FALSE	lock = FALSE	TRUE → FALSE		
		FALSE → TRUE	Swap(&lock, &key)	TRUE → FALSE
		TRUE	<i>/* critical section */</i>	FALSE

**Mutual exclusion (mutex):** only one task may execute in the critical section at the time.

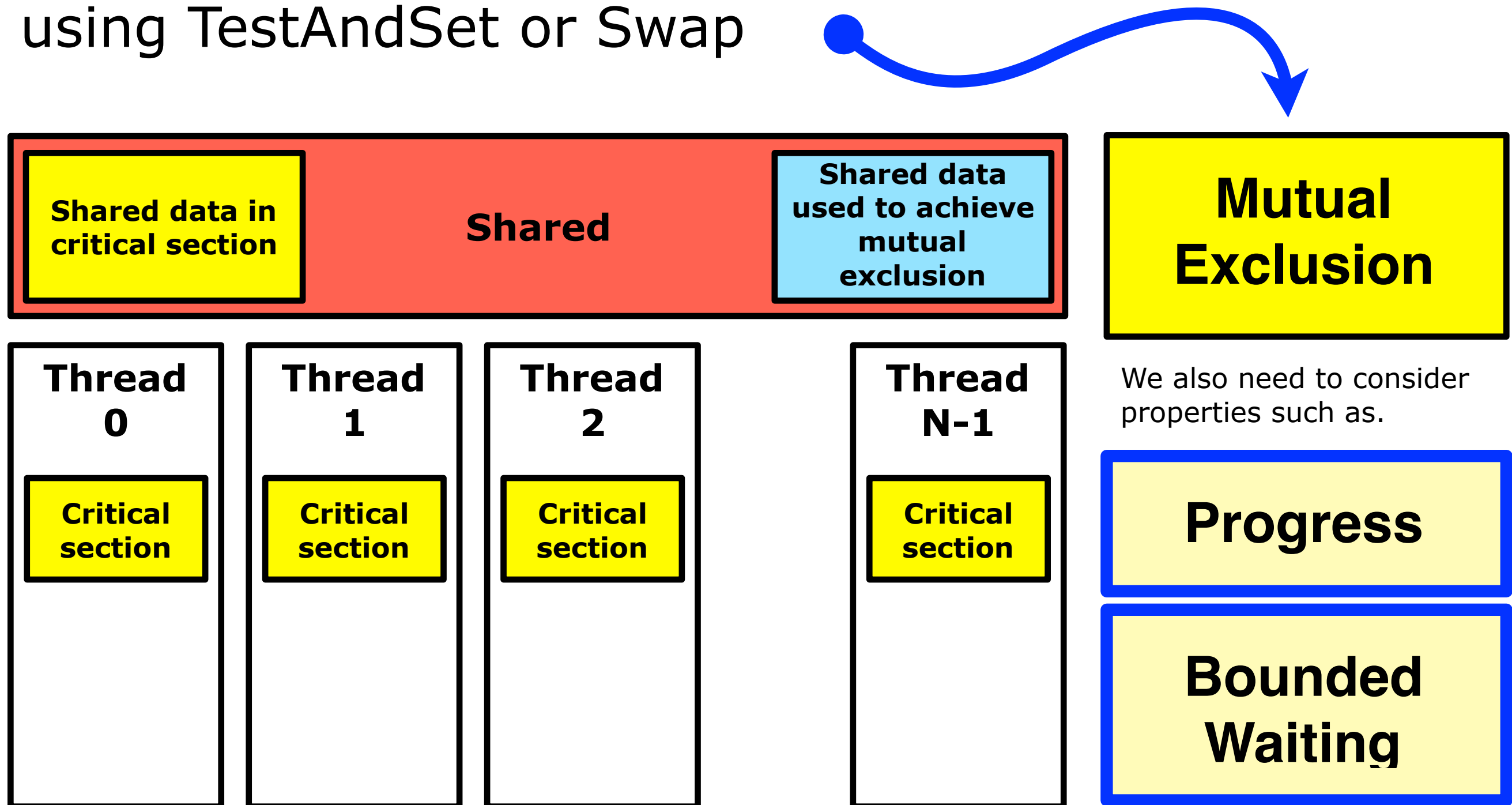
# Spinlock

A **spinlock** is a lock where a task simply waits in a loop ("spins") repeatedly checking until the lock becomes available.

In this sense, both Peterson's solution and the solutions using TestAndSet and Swap can be regarded as spinlock algorithms.

# Solution to Critical-Section Problem

Mutual exclusion can be achieved using TestAndSet or Swap



How can progress and bounded waiting be achieved?

# Implementing bounded waiting and progress using a shared flag array

Similar to the flag array used in Peterson's Solution, we introduce a shared array where each thread indicate with True if they want to enter the critical section.

bool waiting[N]							
Thread	0	1	2	3	...	N-2	N-1
Value	False	False	True	False	...	False	True

In the above example only threads 2 and N-1 wants to enter the critical section.

# Circular scan

When leaving the critical section, a thread scans the array full circle until another thread wanting to enter the critical section is found. Each thread  $i$  start the circular scan at index  $i+1 \% N$ .

bool waiting[N]							
Thread	0	1	2	3	...	N-2	N-1
Value	False	False	True	False	...	False	True

## Bounded waiting

Worst case, all threads wants to enter the critical section. Any process waiting to enter its critical section will thus do so within  $N-1$  turns.

# More details in the following self study material

**Bounded waiting, mutex  
and progress with  
spinlock using TestAndSet**

**Module 4**

**Self study**

---

**Operating systems 2019**

**1DT044, 1DT096 and 1DT003**

# Semaphores

The hardware solutions to the critical section problem using Swap and TestAndSet are complicated for programmers to use.

To overcome this difficulty, we can use a synchronization tool/abstraction called a semaphore.



# Abstractions can be tricky ...



We can think of this abstraction as keeping exactly one cookie in a jar ...



Tasks can look in the jar and **grab** the cookie if available.

If the cookie is absent – **wait** until another task **puts** the cookie back.

shared data

**Account**

```
int balance = 0;
```

**Thread A**

```
deposit(100);
```



**Thread B**

```
withdraw(50);
```

shared data

## Account

```
int balance = 0;
```

### Thread A

```
Grab(cookie);  
    deposit(100);  
Put(cookie);
```



### Thread B

```
Grab(cookie);  
    withdraw(50);  
Put(cookie);
```

Both Grab and Put must be **atomic**!

The cookie jar is called a **binary semaphore**.

Grab is called **wait**.

Put is called **signal**.

shared data

**Account**

```
int balance = 0;
```

**Thread A**

```
wait(S);  
    deposit(100);  
signal(S);
```

semaphore

**S**

**Thread B**

```
wait(S);  
    withdraw(50);  
signal(S);
```

**Both wait and signal must  
be atomic!**





**binary** semaphore



**counting** semaphore

A counting semaphore can be seen as a jar with one or more cookies.



**wait** and  
**signal**  
must be  
**atomic**

Cookie jar analogy		Counting semaphore	
<b>Put</b>	Put a cookie in the jar.	<b>Signal</b>	Increments the semaphore counter.
<b>Grab</b>	Take a cookie from the jar. If the jar is empty, block until a new cookie appears.	<b>Wait</b>	if counter $> 0$ , decrement counter, otherwise <b>wait until</b> counter becomes $> 0$ , then decrement counter.

# Semaphore implementation

How can semaphores be implemented?





A semaphore can be implemented using a single integer variable together with the four operations: `init()`, `destroy()`, `signal()` and `wait()`. Here we use a C like pseudo language to implement an **abstract semaphore datatype**.

```
typedef int sem_t;
```

```
sem_t init(int n) {  
    return n;  
}
```

```
void destroy(sem_t s) {  
    /* no-op */  
}
```

```
void wait(sem_t s) {  
    while(s <= 0) {  
        /* no-op */  
    }  
    s--;  
}
```

```
void signal(sem_t s){  
    s++;  
}
```

Oops ... parameters passed by value (on the stack)!

Must allocate memory for the semaphore dynamically and pass the semaphore by reference to the **destroy()**, **signal()** and **wait()**.

```
typedef int sem_t;
```

```
sem_t *init(int n) {  
    sem_t *s = malloc(sizeof(int));  
    *s = n;  
    return s;  
}
```

```
void destroy(sem_t *s) {  
    free(s);  
}
```

```
void wait(sem_t *s) {  
    while(*s <= 0) {  
        /* no-op */  
    }  
    (*s)--;  
}
```

```
void signal(sem_t *s){  
    (*s)++;  
}
```

Must ensure that both **wait()** and **signal()** are made atomic.

# Origin of the semaphore

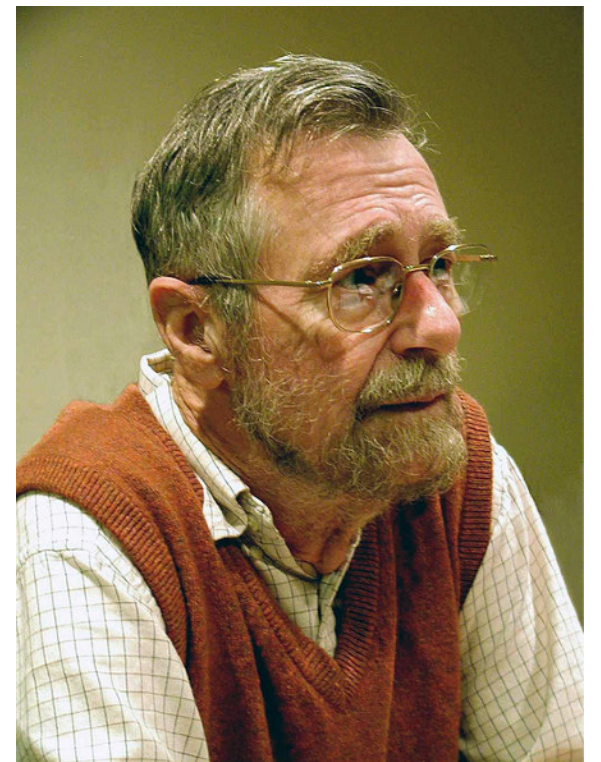
The semaphore concept was invented by Dutch computer scientist Edsger Dijkstra in 1962 or 1963. Originally `wait()` was called `p()` and `signal()` was called `v()` from Dutch "proberen", to test, and "verhogen", to increment.

`init()`

`destroy()`

`wait(S) ⇔ p(S)`

`signal(S) ⇔ v(S)`



Edsger Dijkstra  
1930 - 2002



# Semaphore terminology

Semaphores can be used to implement various types of synchronization.

- ★ **Counting semaphore** – integer value can range over an unrestricted domain.
- ★ **Binary semaphore** – integer value can range only between 0 and 1; can be simpler to implement.
  - ▶ Also known as **mutex locks** (may include priority inheritance).

# Mutual exclusion (mutex)

A binary semaphore can act as a mutex lock, providing mutual exclusive access to critical sections.

Shared

```
int balance = 350;
```

```
mutex = Semaphore(1)
```

Thread A

```
deposit(100);
```

Thread B

```
withdraw(50);
```

# Mutual exclusion (mutex)

A binary semaphore can act as a mutex lock, providing mutual exclusive access to critical sections.

Shared

```
int balance = 350;
```

```
mutex = Semaphore(1)
```

Thread A

```
wait(mutex);  
    deposit(100);  
signal(mutex);
```

Thread B

```
wait(mutex);  
    withdraw(50);  
signal(mutex);
```

# Busy waiting

While a task is in its critical section, any other task that tries to enter its critical section must loop continuously in the entry code.

```
void wait(sem_t s) {  
    while(*s <= 0) {  
        /* no-op */  
    }  
    (*s)--;  
}
```

**Busy waiting** wastes CPU cycles. This type of semaphore is also called a **spinlock**.

**Advantage:** no context switch is required when waiting on a lock.

# **Semaphores** **with no busy** **waiting**

Associate a waiting queue with each semaphore.



Each semaphore has an integer value (number of cookies) and a list of blocked tasks.

```
typedef struct {  
    int value;           // Counter  
    struct task_t *list; // Wait list  
} sem_t;
```

When a task must **wait** on a semaphore, it is **suspended** and **added** to the **wait list**.

A **signal** operation **removes** one task from the wait list and **awakens** that task.

These two operations must be provided by the operating system kernel.

```
block(sem_t *s) {  
    // Suspend caller and add caller to  
    // s -> list (the wait list).  
}  
  
wakeup(sem_t *s) {  
    // Remove a task from the s -> list  
    // (wait list) and resume this task.  
}
```

```

wait(sem_t *s) {
    s -> value--;
    if(s -> value < 0) {
        // Block caller and add caller
        // to semaphore wait list.
        block(s)
    }
}

signal(sem_t *s) {
    s -> value++;
    if(s -> value <= 0) {
        // Remove one task from the
        // semaphore wait list and resume
        // execution of this task.
        wakeup(s)
    }
}

```

In this implementation, the value of the semaphore can be **negative**.

**A negative value counts the number of blocked processes.**

The **signal()** operation first increments the semaphore counter, then, if there is at least one task blocked on the semaphore, wake up one of the blocked tasks.

```

wait(sem_t *s) {
    s -> value--;
    if(s -> value < 0) {
        // Block caller and add caller
        // to semaphore wait list.
        block(s)
    }
}

```

```

signal(sem_t *s) {
    s -> value++;
    if(s -> value <= 0) {
        // Remove one task from the
        // semaphore wait list and resume
        // execution of this task.
        wakeup(s)
    }
}

```

Task	Operation	s.value	s.owner	s.queue
T0	s = init(1)	1		
T0	Create tasks T1, T2 and T3	1		
T1	wait(s)	0	T1	
T2	wait(s)	-1	T1	T2
T3	wait(s)	-2	T1	T2, T3
T1	signal(s)	-1	T2	T3

# Semaphores and mutex (1)

A binary semaphore can act as a mutex lock, providing mutual exclusive access to critical sections.

Shared

```
int balance = 350;
```

```
mutex = Semaphore(1)
```

Thread A

```
wait(mutex);  
    deposit(100);  
signal(mutex);
```

Thread B

```
wait(mutex);  
    withdraw(50);  
signal(mutex);
```

The programmer must make sure to always do wait and signal in the right order.

# Semaphores and mutex (2)

What if the programmer makes a mistake?

Shared

```
int balance = 350;
```

```
mutex = Semaphore(1)
```

Thread A

```
wait(mutex);  
    deposit(100);  
signal(mutex);
```

Thread B

```
signal(mutex);  
    withdraw(50);  
signal(mutex);
```

Ops! Thread B never waits on the semaphore and mutex is not guaranteed.

# Mutex lock

Many locking libraries provides a special mutex lock which can only be used to provide mutual exclusion to critical sections.

- ★ A mutex lock is similar to a binary semaphore initialised to 1.
- ★ A mutex is meant to be taken and released, always in that order, by each task that uses the shared resource it protects.
- ★ A task that successfully takes the lock becomes the **owner** of the mutex lock.
- ★ **Only** the **owner** of the lock **can unlock** the mutex lock.

# Semaphores vs mutex locks

In general, semaphores should not be used to enforce mutual exclusion—use mutex locks for this.

The correct use of a semaphore is to signaling from one task to another.

A mutex is meant to be taken and released, always in that order, by each task that uses the shared resource it protects.

By contrast, tasks that use semaphores either signal or wait— not both. For example a consumer waits for a producer to produce data, and a producer signals the consumer when data is available.