Synchronisation

Lecture 4b

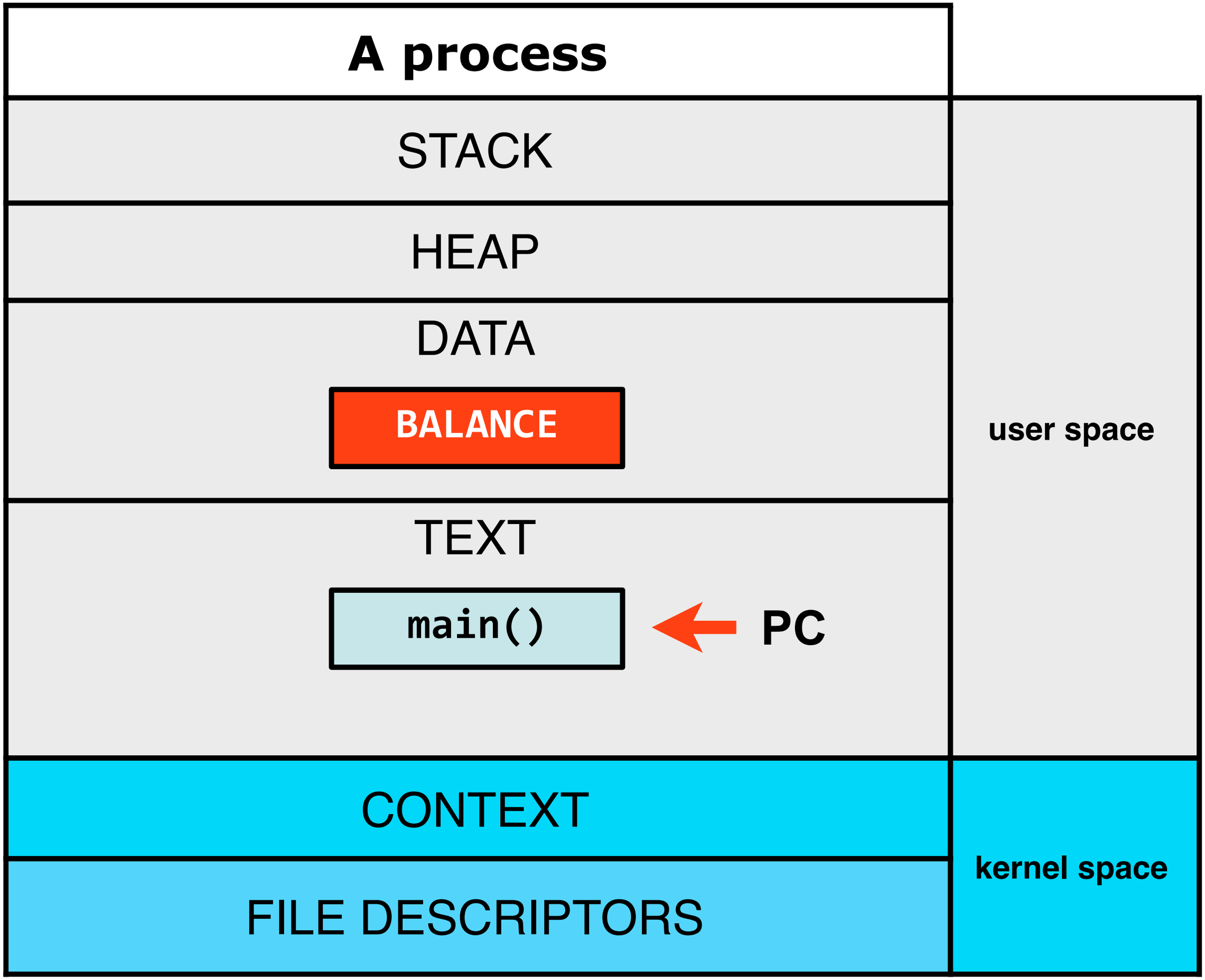
Module 4

Operating systems 2018

1DT044 and 1DT096

2018-02-09 karl.marklund@it.uu.se Uppsala University

Background


A process
STACK
HEAP
DATA
BALANCE
TEXT
main()
PC
CONTEXT
FILE DESCRIPTORS
user space
kernel space

A process with three threads

$STACK_1$	$STACK_2$	$STACK_3$	
HEAP			
DATA **BALANCE**			user space
TEXT `main()` ← PC_1	`increment()` ← PC_2	`decrement()` ← PC_3	
$CONTEXT_1$	$CONTEXT_2$	$CONTEXT_3$	**user or kernel space**
FILE DESCRIPTORS			kernel space

Programming using shared memory with threads

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--;
}
```

BALANCE == ?

Thread A | **int BALANCE = 0;** | **Thread A**

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

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

```
$> make bin/balance
gcc -c -std=gnu99 -Werror -Wall -O2 src/balance.c -o obj/balance.o
gcc obj/balance.o -o bin/balance
$>
```

```
$> ./bin/balance

Number of iterations: 100
     Initial balance: 0
       Final balance: 0
```

```
./bin/balance 77777777

ber of iterations: 77777777
     Initial balance: 0
       Final balance: 0
```

What? The final balance is not always zero!

```
$> /bin/balance 999999999

Number of iterations: 999999999
     Initial balance: 0
       Final balance: -372112
```

```
./bin/balance 999999999

umber of iterations: 999999999
     Initial balance: 0
       Final balance: 0
```

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)				Thread B (decrement)		
OP	Operands	$t0	BALANCE	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)				Thread B (decrement)		
OP	Operands	$t0	BALANCE	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.

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--;
}
```

BALANCE == ?

shared state

```
#define N 10000

int  BALANCE  0;
```

Thread A

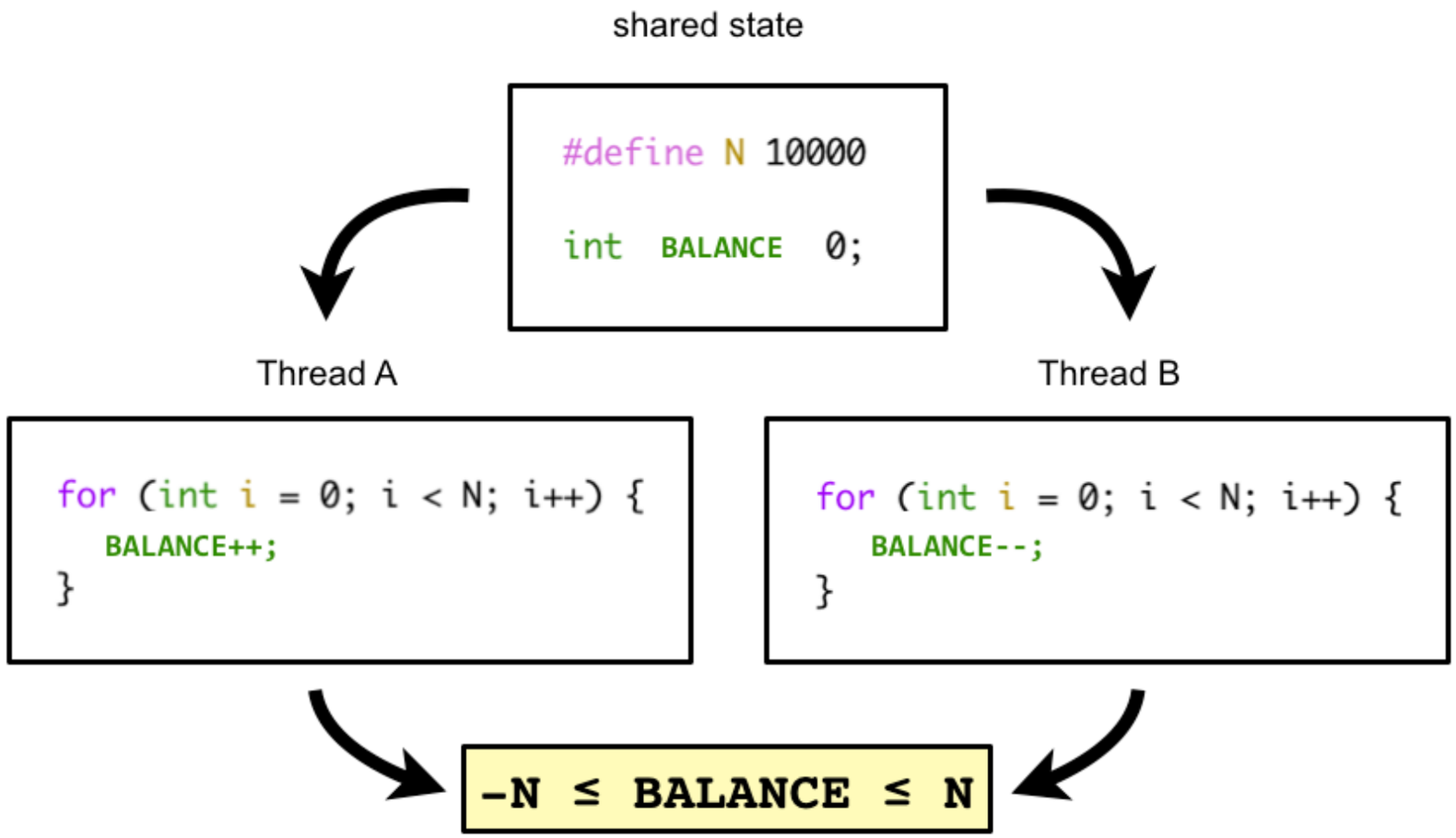
```
for (int i = 0; i < N; i++) {
    BALANCE++;
}
```

Thread B

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

-N ≤ BALANCE ≤ 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.

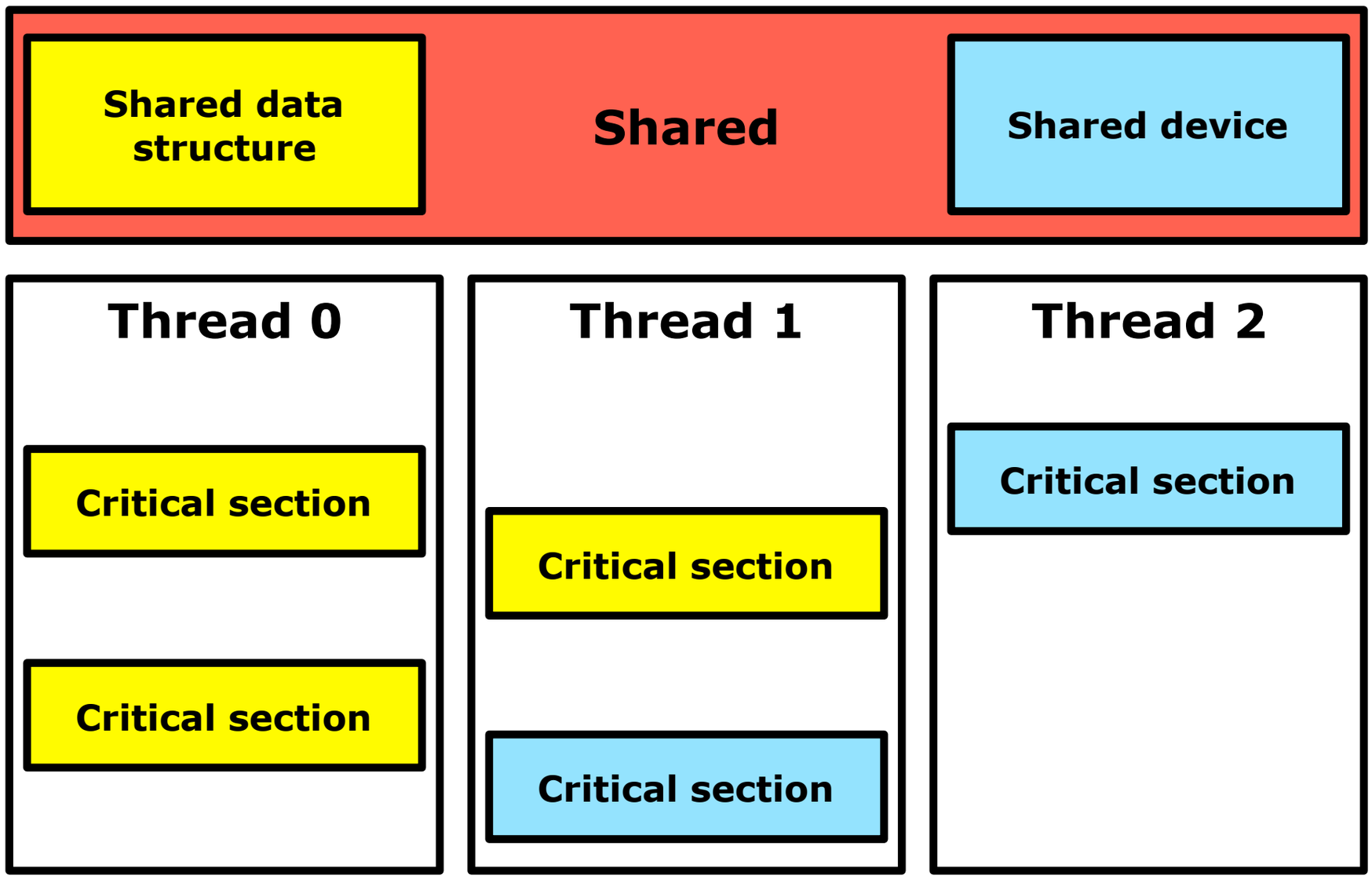
Critical section

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

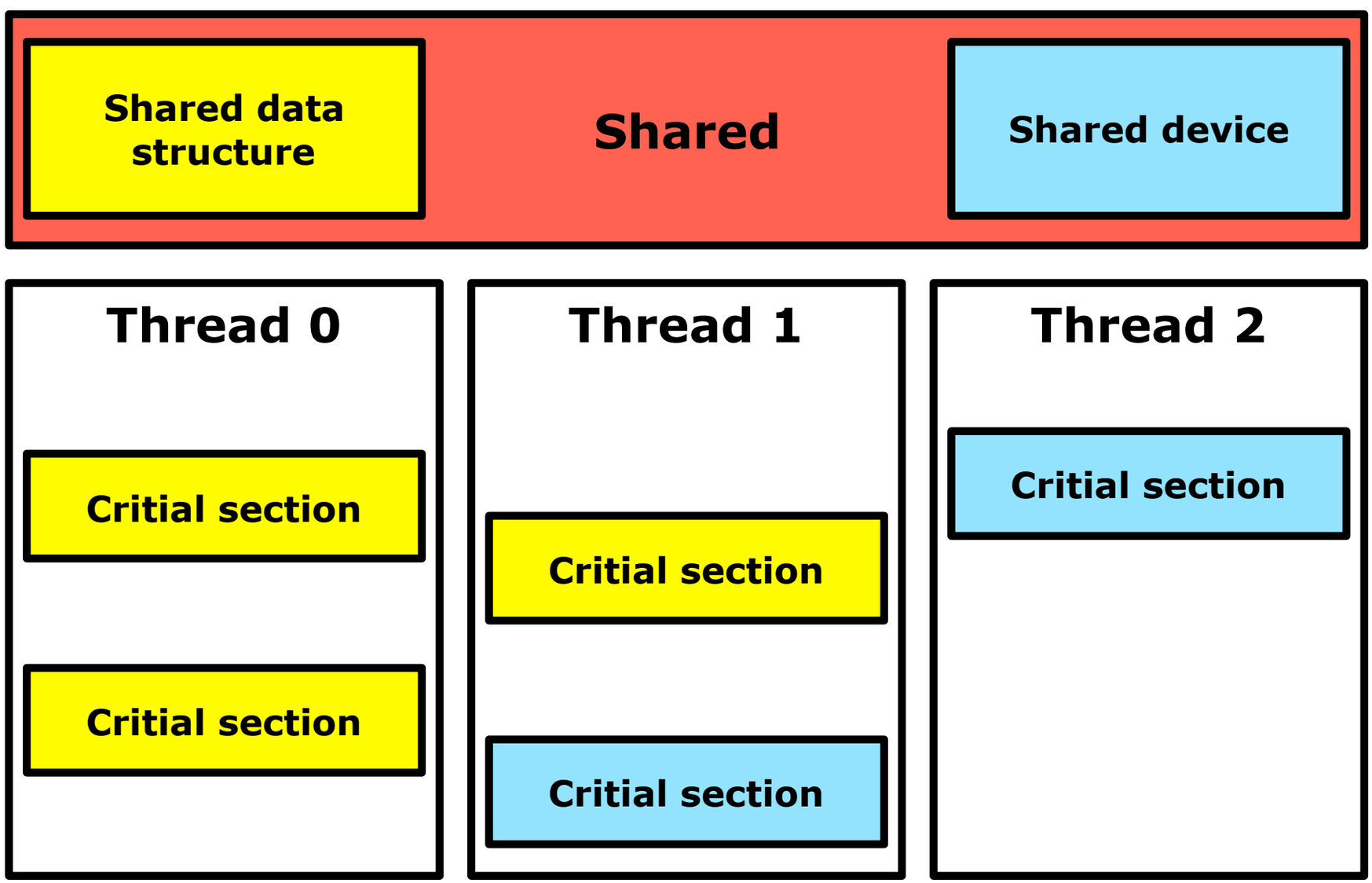
Fairness

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?



Mutual Exclusion

We also need to consider properties such as.

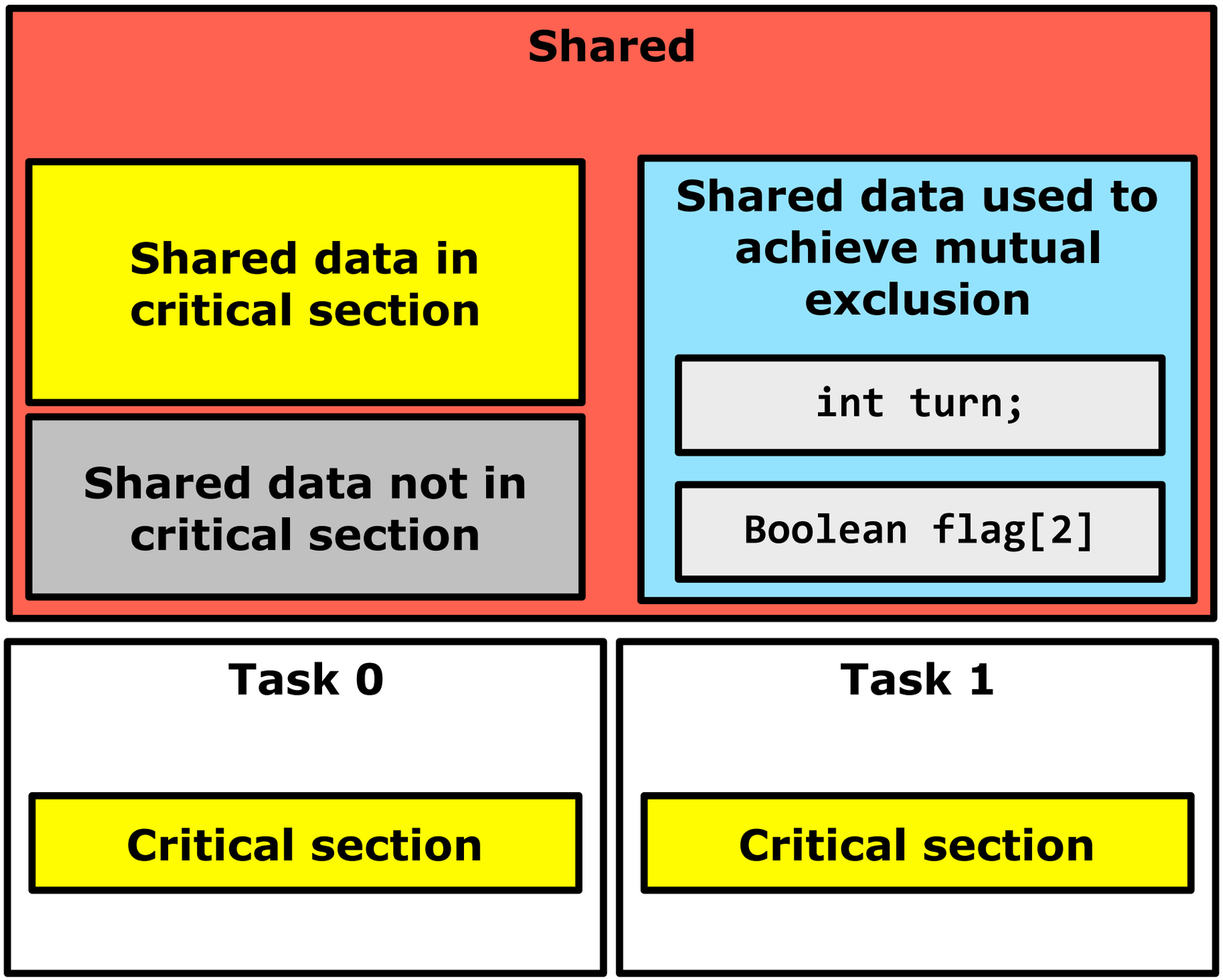
Progress

Bounded Waiting

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.

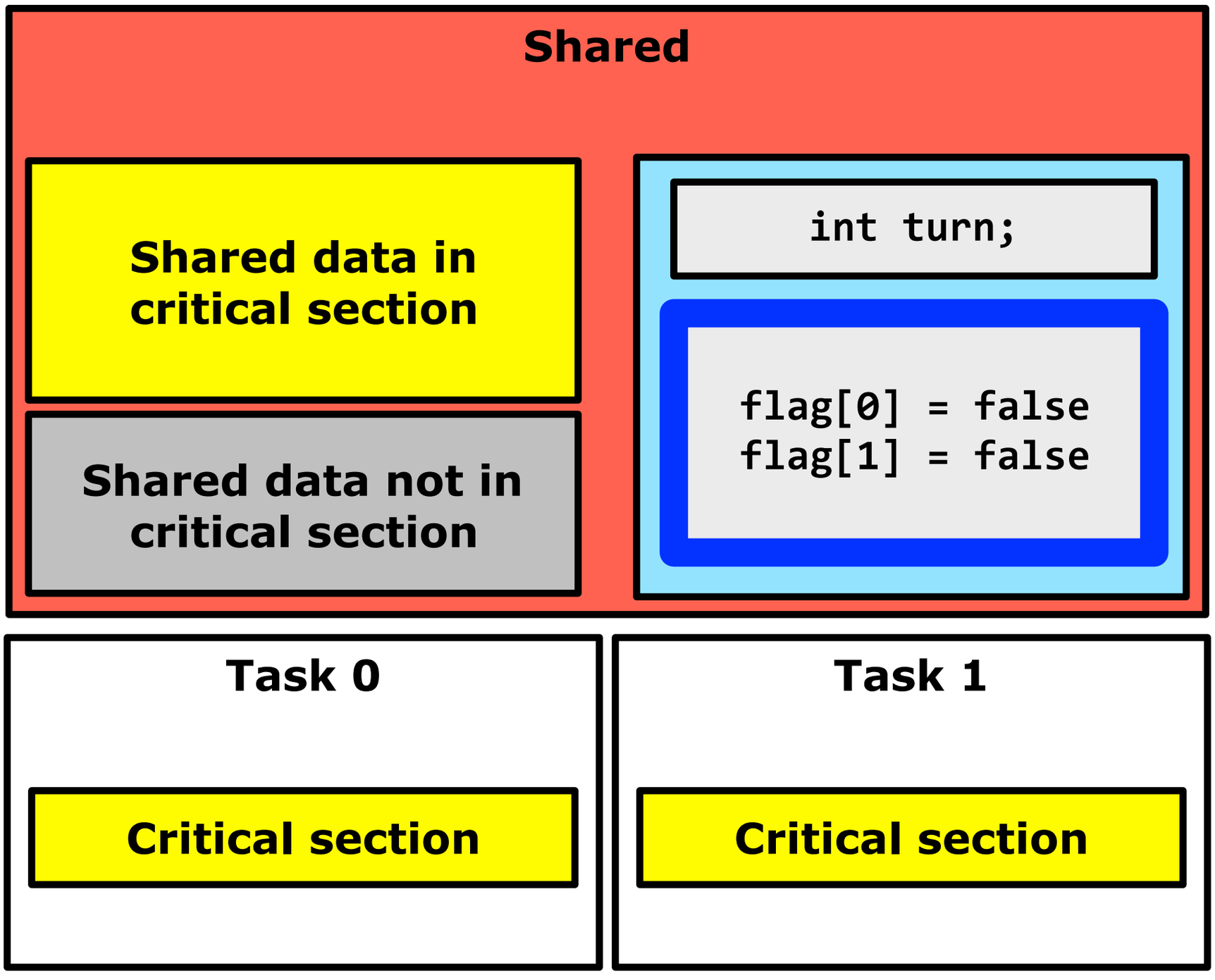


The **`flag`** array is used to indicate if a process is ready to enter the critical section, i.e.,

`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

```
flag[1] = true;

turn = 0;

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

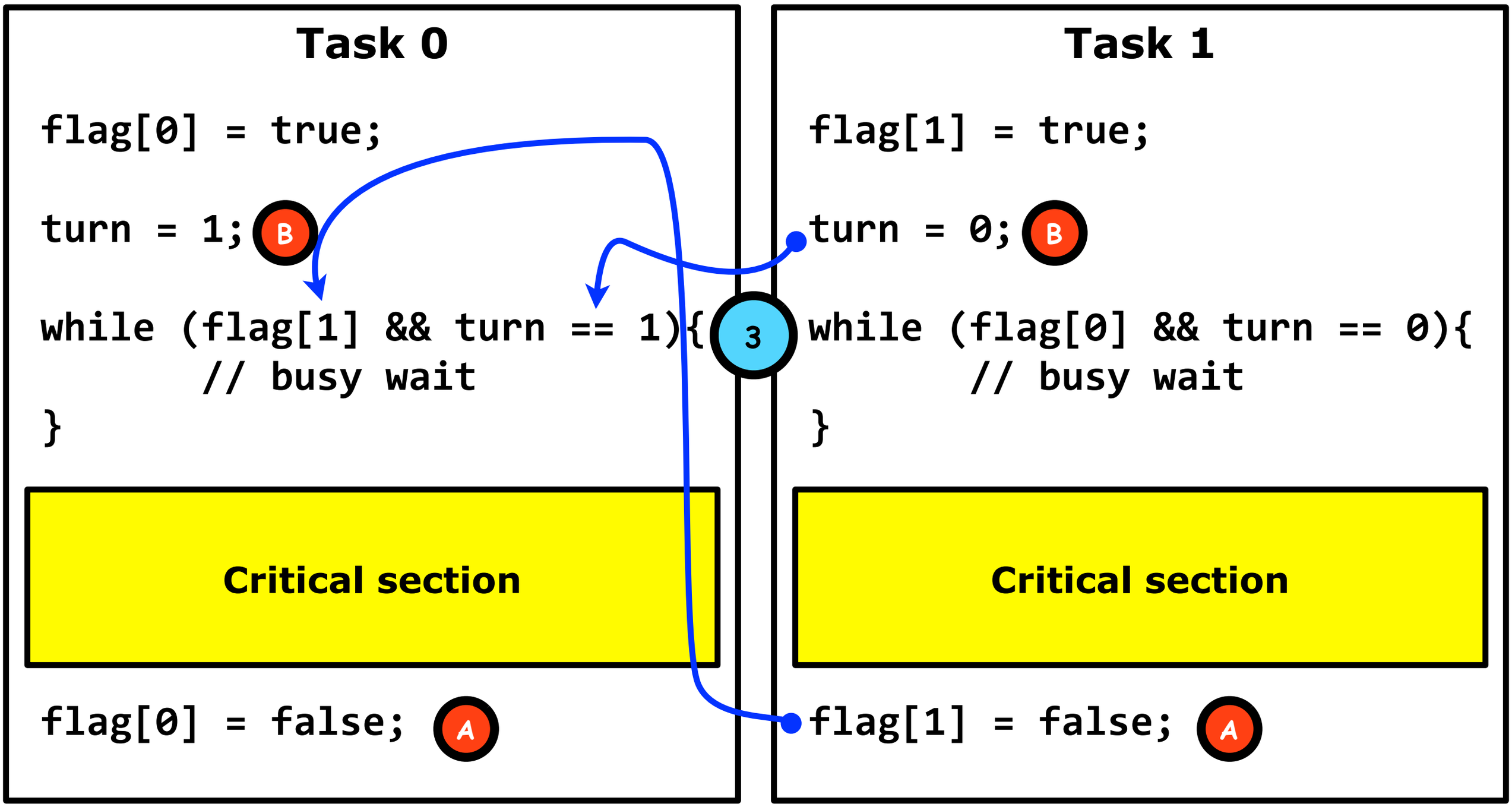
Critical section

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

1. T_i wants to enter critical section
2. T_i is polite, after you $T_{(i+1 \% 2)}$
3. For T_i to be able to enter the critical section, the while condition must fail.
4. T_i signals - I've leaft 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 is A not interested in entering (puts down its **flag**), or B the other task says ”after you” (setting the value of **turn**).



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

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

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.

Lock using

TestAndSet

`TestAndSet` set the value at the target address to True and return the old value stored at the target addres.

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

```
do {
    grab lock

    /* critical section */

    release lock

    /* 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 {
    grab lock

    /* critical section */

    release lock

    /* 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 {
    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`**.

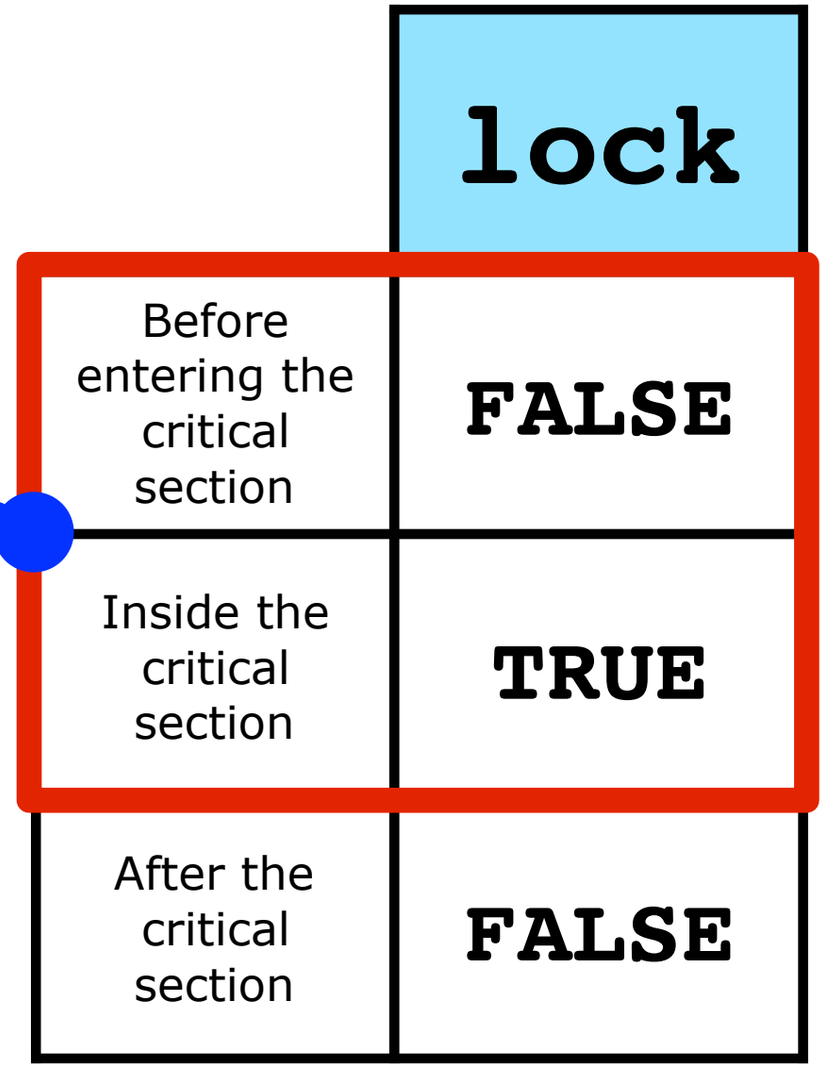
```
do {
    grab lock

    /* critical section */

    lock = FALSE;

    /* remainder section */

} while (TRUE);
```



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

```
do {
  while (                ?                )
    ;    /* do nothing* /

  /* 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`**.

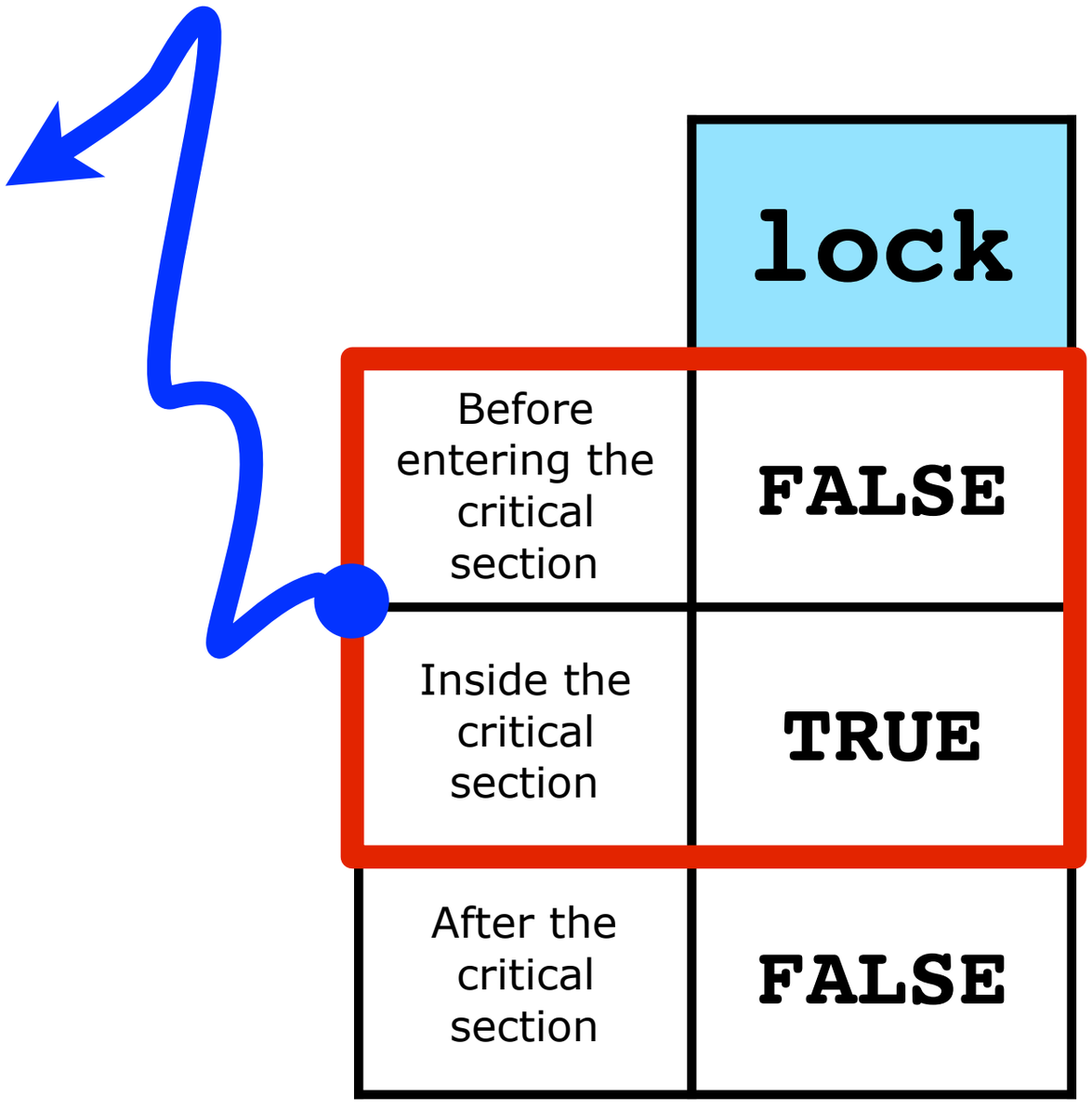
```
do {
  while ( TestAndSet (&lock ))
     ;    /* do nothing* /

  /* critical section */

  lock = FALSE;

  /* remainder section */

} while (TRUE);
```



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	**FALSE**
Inside the critical section	**TRUE**
After the critical section	**FALSE**

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

```
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	**`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 {
        grab lock

        /* critical section */

        release lock

        /* 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 {
    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`**.

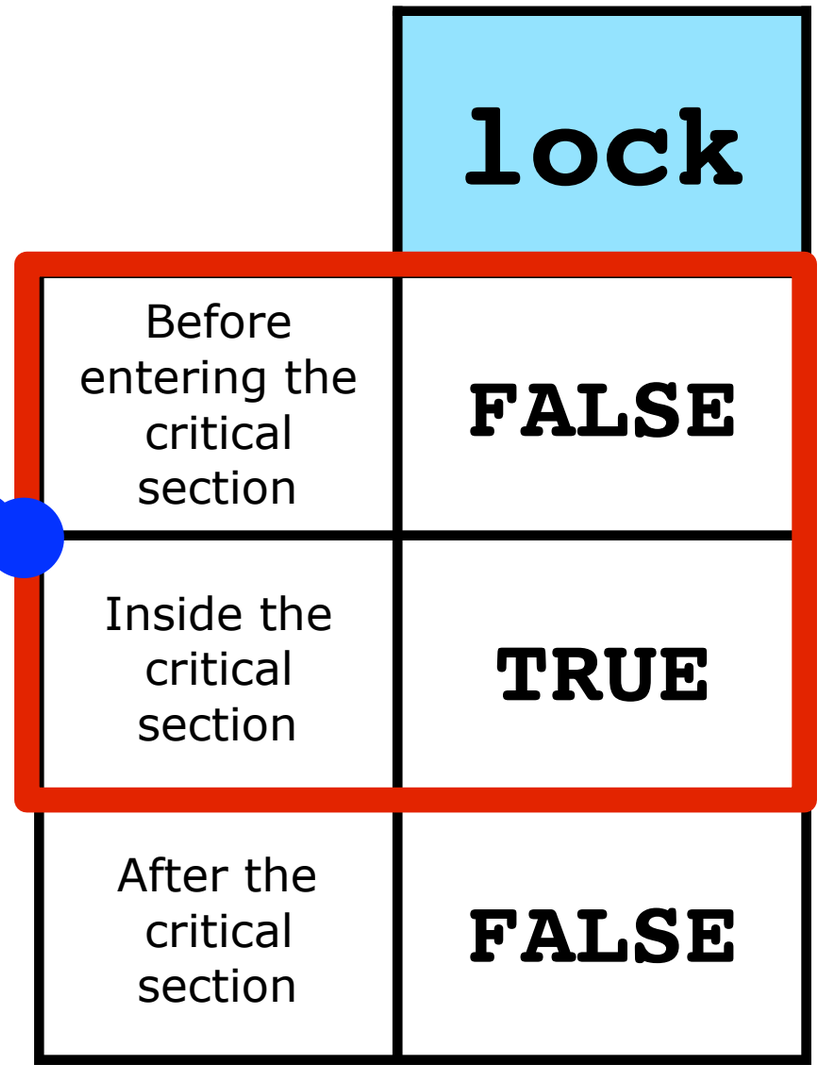
```
do {
    grab lock

    /* critical section */

    lock = FALSE;

    /* remainder section */

} while (TRUE);
```



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

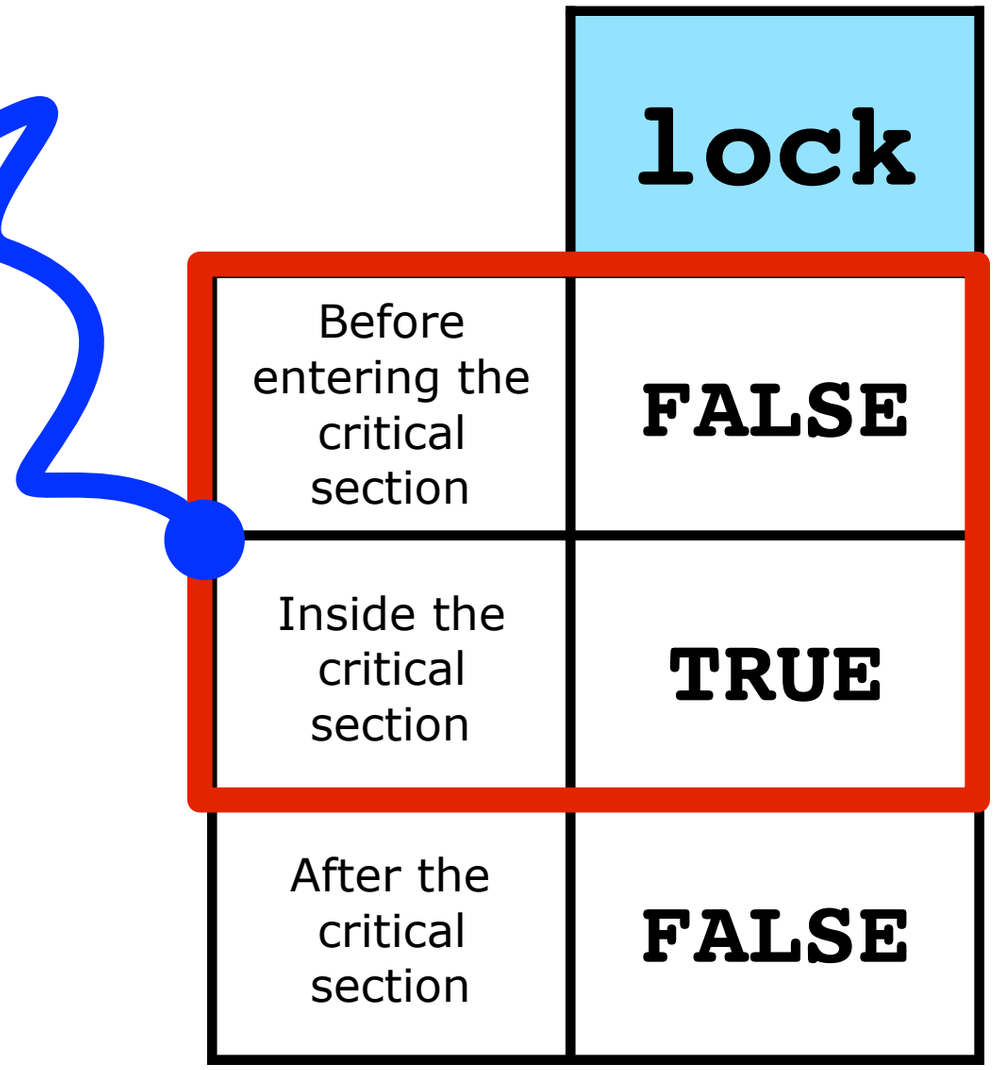
```
do {
   key = TRUE;
   while ( key == TRUE)
                ?

   /* critical section */

   lock = FALSE;

   /* remainder section */

} while (TRUE);
```



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

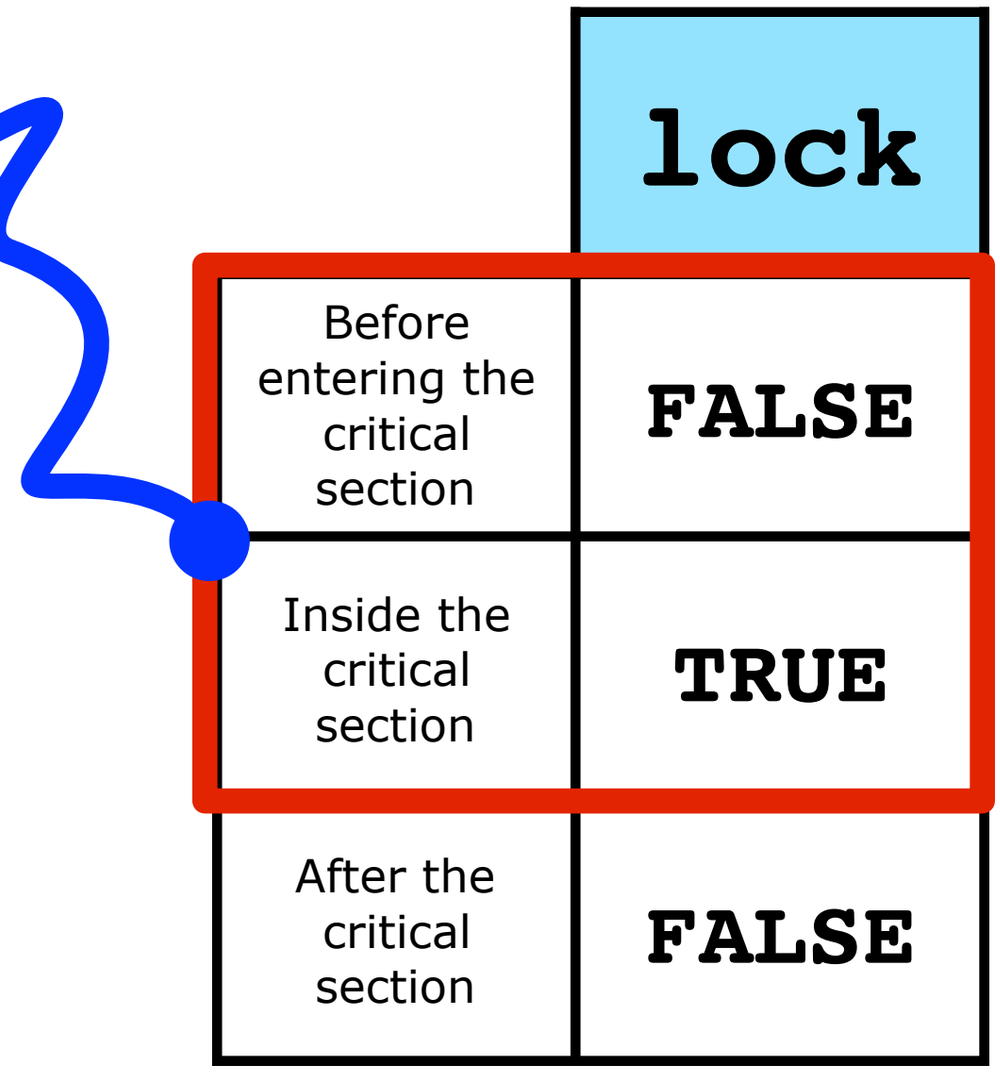
```
do {
   key = TRUE;
   while ( key == TRUE)
      Swap (&lock, &key );

   /* critical section */

   lock = FALSE;

   /* remainder section */

} while (TRUE);
```



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	**FALSE**
Inside the critical section	**TRUE**
After the critical section	**FALSE**

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	/* critical section */	TRUE		?
FALSE	/* critical section */	TRUE	key = TRUE	TRUE
FALSE	/* critical section */	TRUE	Swap(&lock, &key)	TRUE
FALSE	/* critical section */	TRUE	Swap(&lock, &key)	TRUE
FALSE	lock = FALSE	TRUE → FALSE		
		FALSE → TRUE	Swap(&lock, &key)	TRUE → FALSE
		TRUE	/* critical section */	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	/* critical section */	TRUE		?
FALSE	/* critical section */	TRUE	key = TRUE	TRUE
FALSE	/* critical section */	TRUE	Swap(&lock, &key)	TRUE
FALSE	/* critical section */	TRUE	Swap(&lock, &key)	TRUE
FALSE	lock = FALSE	TRUE → FALSE		
		FALSE → TRUE	Swap(&lock, &key)	TRUE → FALSE
		TRUE	/* critical section */	FALSE

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

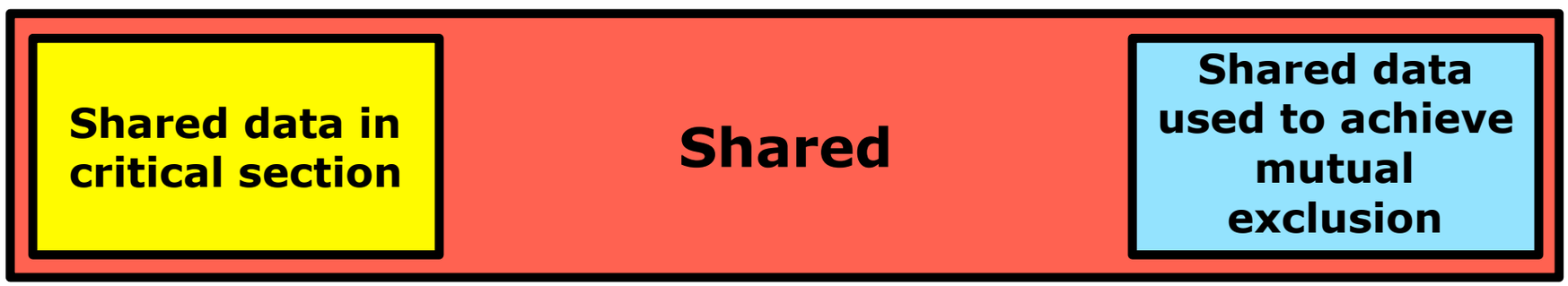
Spinlock

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

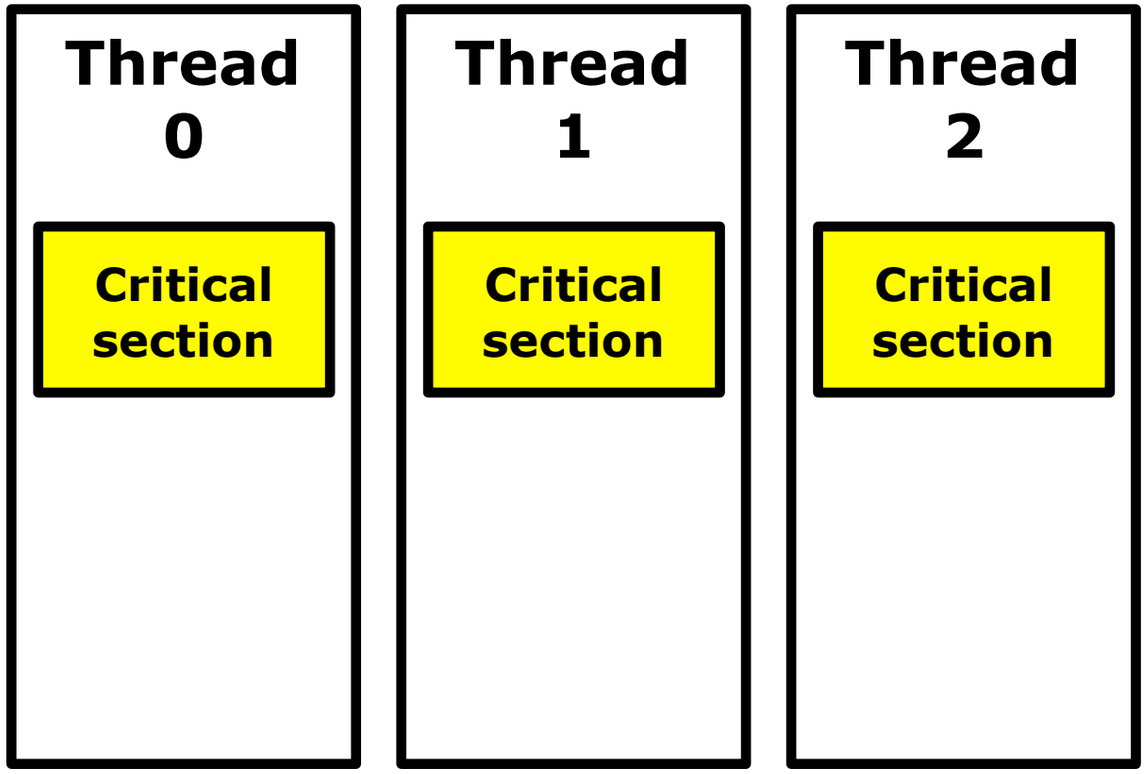
In this sense, both Peterson's solution and the solutions using TestAndSet and Swap can be regarded as spinlock algorithms.

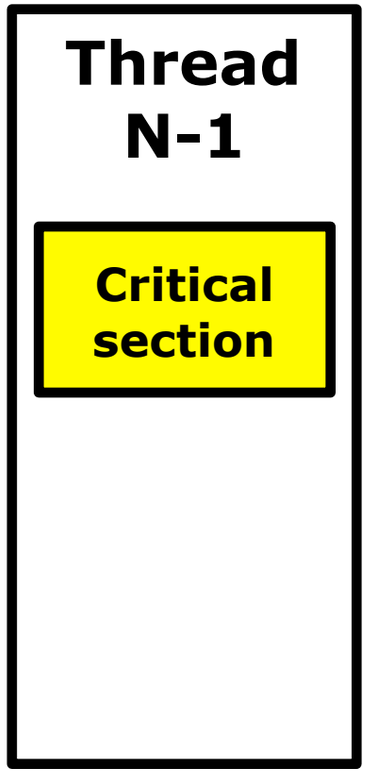
Solution to Critical-Section Problem

Mutual exclusion Can be achieved using TestAndSet or Swap



Mutual Exclusion





We also need to consider properties such as.

Progress

Bounded Waiting

How can bounded progress and bounded waiting be achieved?

Progress and bounded waiting

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

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

The slides showing how bounded waiting is implemented was left as an exercise for students to study by themselves.

Bounded-waiting mutual exclusion with `TestAndSet()`

- Shared **`boolean`** `waiting[n]` initialized to **`FALSE`**.
- Shared **`boolean`** `lock` initialized to **`FALSE`**.

```
do {
```

Grab lock

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

Release lock

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

Bounded-waiting mutual exclusion with TestAndSet()

- Shared **boolean** `waiting[n]` initialized to **FALSE**.
- Shared **boolean** **lock** initialized to **FALSE**.

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

```
waiting[i] = TRUE;
```

```
do {
  waiting[i] = TRUE;
  key = TRUE;
```

... grab lock continues.

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

Release lock

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

Bounded-waiting mutual exclusion with TestAndSet()

- Shared **boolean** **waiting[n]** initialized to **FALSE**.
- Shared **boolean** **lock** initialized to **FALSE**.

Mutual Exclusion

Pi can enter its critical section only if

waiting[i] == FALSE || key == FALSE

The value of **key** can become false only if the **TestAndSet** is executed.

The first process to execute

key = TestAndSet(&lock)

where,

lock == FALSE

will cause:

key ← FALSE

lock ← TRUE

, entering the critical section and forcing all others to wait.

```
do {
  waiting[i] = TRUE;
  key = TRUE;

  while (waiting[i] && key)
    key = TestAndSet(&lock);

    ... grab lock continues.

  /* critical section */

    Release lock

} while (TRUE);
```

Bounded-waiting mutual exclusion with `TestAndSet()`

- Shared **`boolean`** `waiting[n]` initialized to **`FALSE`**.
- Shared **`boolean`** `lock` initialized to **`FALSE`**.

No longer waiting, enter the critical section.

```
do {
  waiting[i] = TRUE;
  key = TRUE;

  while (waiting[i] && key)
    key = TestAndSet(&lock);

  waiting[i] = FALSE;

  /* critical section */
```

Release lock

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

Bounded-waiting mutual exclusion with `TestAndSet()`

- Shared **`boolean`** **`waiting[n]`** initialized to **`FALSE`**.
- Shared **`boolean`** **`lock`** initialized to **`FALSE`**.

Bounded-Waiting

When a process leaves the critical section, it scans the array waiting in the cyclic order

`(i+1, i+2, ..., n-1, 0, ..., i-1)`

and designates the first process in this ordering that is in the entry section (**waiting[j] == true**) as the next one to enter the critical section.

Any process waiting to enter its critical section will thus do so within n - 1 turns.

```
do {
  waiting[i] = TRUE;
  key = TRUE;

  while (waiting[i] && key)
    key = TestAndSet(&lock);

  waiting[i] = FALSE;

  /* critical section */

  j = (i + 1) % n;

  while ((j != i) && !waiting[j])
    j = (j + 1) % n;

  if (j == i)
    lock = FALSE;
  else
    waiting[j] = FALSE;

  /* remainder section */

} while (TRUE);
```

Bounded-waiting mutual exclusion with `TestAndSet()`

- Shared **boolean** `waiting[n]` initialized to **FALSE**.
- Shared **boolean** **lock** initialized to **FALSE**.

Mutual Exclusion

Pi can enter its critical section only if

waiting[i] == FALSE || key == FALSE

The value of **key** can become false only if the **TestAndSet** is executed.

The first process to execute

key = TestAndSet(&lock)

where,

lock == FALSE

will cause:

key ← FALSE

lock ← TRUE

, entering the critical section and forcing all others to wait.

Mutual Exclusion

waiting[i] can become **FALSE** only if another process leaves its critical section, only one **waiting[i]** is set to false, maintaining the mutual-exclusion requirement.

```
do {
  waiting[i] = TRUE;
  key = TRUE;

  while (waiting[i] && key)
    key = TestAndSet(&lock);

  waiting[i] = FALSE;

  /* critical section */

  j = (i + 1) % n;

  while ((j != i) && !waiting[j])
    j = (j + 1) % n;

  if (j == i)
    lock = FALSE;
  else
    waiting[j] = FALSE;

  /* remainder section */

} while (TRUE);
```

Bounded-waiting mutual exclusion with `TestAndSet()`

- Shared **`boolean`** `waiting[n]` initialized to **`FALSE`**.
- Shared **`boolean`** `lock` initialized to **`FALSE`**.

Progress

When a process leaves the critical section, either **`lock`** or **`waiting[j]`** is set to **`FALSE`**. Both allow a process that is waiting to enter its critical section.

```
do {
  waiting[i] = TRUE;
  key = TRUE;

  while (waiting[i] && key)
    key = TestAndSet(&lock);

  waiting[i] = FALSE;

  /* critical section */

  j = (i + 1) % n;

  while ((j != i) && !waiting[j])
    j = (j + 1) % n;

  if (j == i)
    lock = FALSE;
  else
    waiting[j] = FALSE;

  /* remainder section */

} while (TRUE);
```

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 tas **put**s 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

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

semaphore

S

Thread B

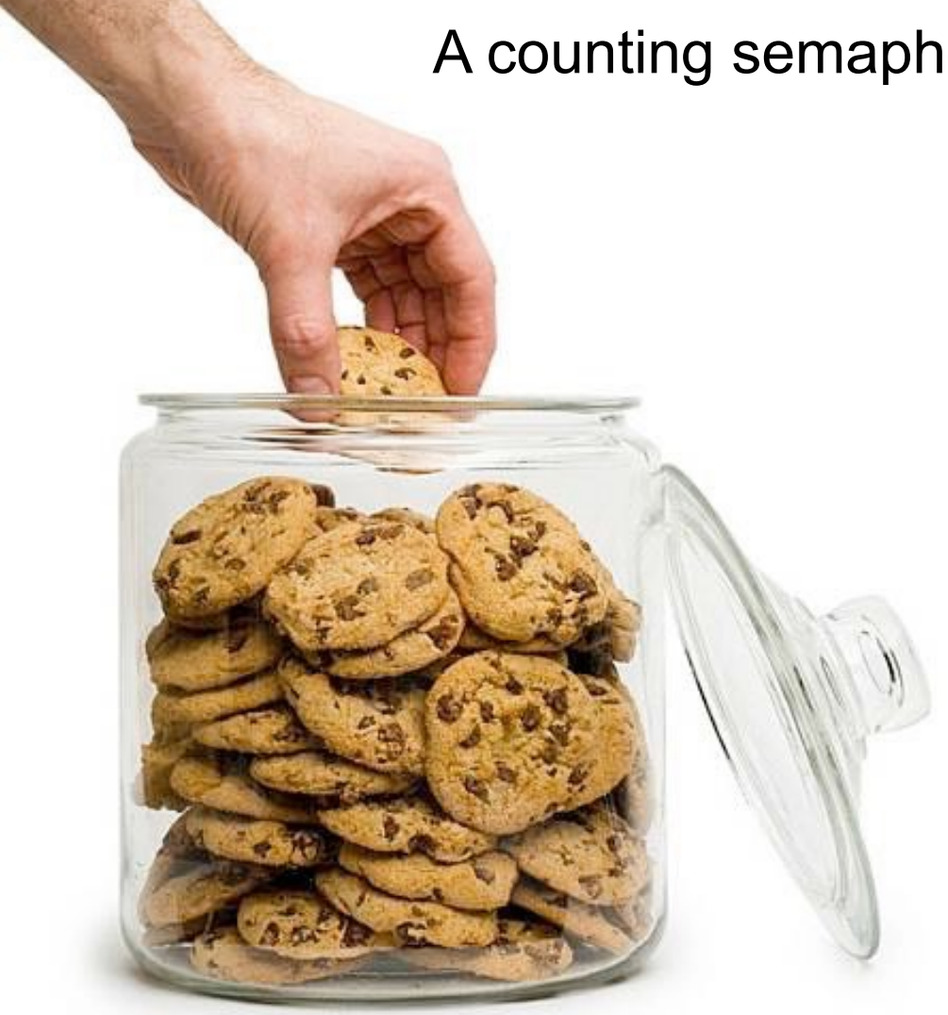
```
Grab(cookie);
    withdraw(50);
Put(cookie);
```

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**	
Put	Put a cookie in the jar.	**Signal**	Increments the semaphore counter.
Grab	Take a cookie from the jar. If the jar is empty, block until a new cookie appears.	**Wait**	if counter > 0, decrement counter, otherwise **wait until** 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.

Edsger Dijkstra
1930 - 2002

`init()`

`destroy()`

`wait(S)` ⇔ **`p(S)`**

`signal(S)` ⇔ **`v(S)`**

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 process is in its critical section, any other process 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.

Spinlocks (1)

A spinlock is a lock which causes a thread trying to acquire it to simply wait in a loop ("spinning") while repeatedly checking if the lock is available.

While spinning on a lock, the thread remains active but is not performing a useful task, the use of such a lock is a kind of **busy waiting**.

Because they avoid overhead from operating system process rescheduling or context switching, **spinlocks are efficient if threads are likely to be blocked for only short periods**.

- For this reason, **spinlocks are often used inside operating system kernels**.
- However, spinlocks become wasteful if held for longer durations.

Spinlocks (2)

The longer a lock is held by a thread, the greater the risk is that the thread will be interrupted by the OS scheduler while holding the lock.

If this happens, other threads will be left "spinning" (repeatedly trying to acquire the lock), while the thread holding the lock is not making progress towards releasing it.

The result is an indefinite postponement until the thread holding the lock can finish and release it.

This is especially true on a single-processor system, where each waiting thread of the same priority is likely to waste its quantum (allocated time where a thread can run) spinning until the thread that holds the lock is finally finished.

Spinlocks (3)

Implementing spin locks correctly is difficult because one must take into account the possibility of simultaneous access to the lock.

Simultaneous access to the lock could cause **race conditions**.

Generally, such implementation is possible only with special assembly language instructions, such as **atomic test-and-set** or **swap** operations.

On architectures without such operations, or if high-level language implementation is required, a non-atomic locking algorithm may be used, e.g. **Peterson's algorithm**. But note that such an implementation may:

- require more memory than a spinlock.
- be slower to allow progress after unlocking.
- not be implementable in a high-level language if out-of-order execution is allowed.

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 processes.

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

When a process/thread must wait on a semaphore, it is added to the list. A **signal()** operation removes one process from the list and awakens that process.

```
// Implementation of wait:

wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      // add this process to S->list;
      block();
   }
}

// Implementation of signal:

signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      // remove a process P from S->list;
      wakeup(P);
   }
}
```

The **`block()`** operation suspends the invoking process.

These two operations are provided by the operating system as basic system calls.

The **`wakeup(P)`** operation resumes the execution of a blocked process P.

```
// Implementation of wait:

wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      // add this process to S->list;
      block();
   }
 }

// Implementation of signal:

signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      // remove a process P from S->list;
      wakeup(P);
   }
}
```

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 processes blocked on the semaphore, wake up one of the blocked processes.

```
// Implementation of wait:

wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
     // add this process to S->list;
     block();
   }
}
```

```
// Implementation of signal:

signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
     // remove a process P from S->list;
     wakeup(P);
   }
}
```

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

Deadlock

When two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes this is called deadlock.

Two tasks **T0** and **T1**, semaphores A and B, both initialized to **1**.

T0 waits for **T1** to release **B**.

A = Semaphore(1) B = Semaphore(1)		
T0	**Time**	**T1**
wait(A)	↓	wait(B)
wait(B)		wait(A)

At the same time as **T1** waits for **T0** to release **A**.

But neither of the processes will ever release the locks - they are blocked waiting for each other. The system has reached a deadlock.