Synchronization - Part 1
Operating Systems – EDA093/DIT401

Vincenzo Gulisano
vincenzo.gulisano@chalmers.se

UNIVERSITY OF
GOTHENBURG

What to read (Main textbook) - Both for part 1 and part 2

- Chapter 2.3.1-2.3.6, 2.5.1-2.5.2, 6.1-6.2, 6.5-6.6, 6.7.3-6.7.4;
- Quicker reading, for awareness, of sections 2.3.7-2.3.10, 6.3

  (facultative reading: sections 6.1-6.7, 6.9, 7.1-7.5, 7.6-7.8 from OS Concepts by Silberschatz et-al).

Agenda

- Motivation
- The Critical Section problem
  - Check-competition approach
- The check-order / after-you approach
- Peterson’s Algorithm
- Synchronization and hardware support
  - Read-Modify-Write instructions
- Semaphores
- Other techniques

Agenda

- Motivation
- The Critical Section problem
  - Check-competition approach
- The check-order / after-you approach
- Peterson’s Algorithm
- Synchronization and hardware support
  - Read-Modify-Write instructions
- Semaphores
- Other techniques

Previously on Operating Systems

```c
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  if (argc != 2) {
    fprintf(stderr,"usage: a.out <integer value>\n");
    return -1;
  }
  if (atoi(argv[1]) < 0) {
    fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
    return -1;
  }
```

```c
  /* get the default attributes */
  pthread_attr_init(&attr);
  /* create the thread */
  pthread_create(&tid,&attr,runner,argv[1]);
  /* wait for the thread to exit */
  pthread_join(tid,NULL);

  printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
  int i, upper = atoi(param);
  sum = 0;

  for (i = 1; i <= upper; i++)
    sum += i;

  pthread_exit(0);
}
```

On the need for proper synchronization: example

- Setup
  - 2 threads, A and B
  - Each has access to:
    - a **local** variable x
    - a **shared** variable S, initialized with value 20
  - A and B run in parallel, each on its core
  - A and B run 3 instructions to update S (see figure)
- Consider the execution on the right:

Time (in discrete steps)

Thread A	Thread B	Variable S
		20
x ← S		20
x ← x + 50	x ← S	20
S ← x	x ← x - 20	70
	S ← x	0

Agenda

- Motivation
- The Critical Section problem
  - Check-competition approach
- The check-order / after-you approach
- Peterson’s Algorithm
- Synchronization and hardware support
  - Read-Modify-Write instructions
- Semaphores
- Other techniques

The Critical-Section (aka Critical Region) Problem

Structure of process/thread *P*

Repeat

entry section

critical section

exit section

remainder section

Forever

n processes/threads competing

Assumptions:

- atomic read/write
- no process failures
- n=2 in the initial examples

Notice: duration of the [remainder section] is unbounded

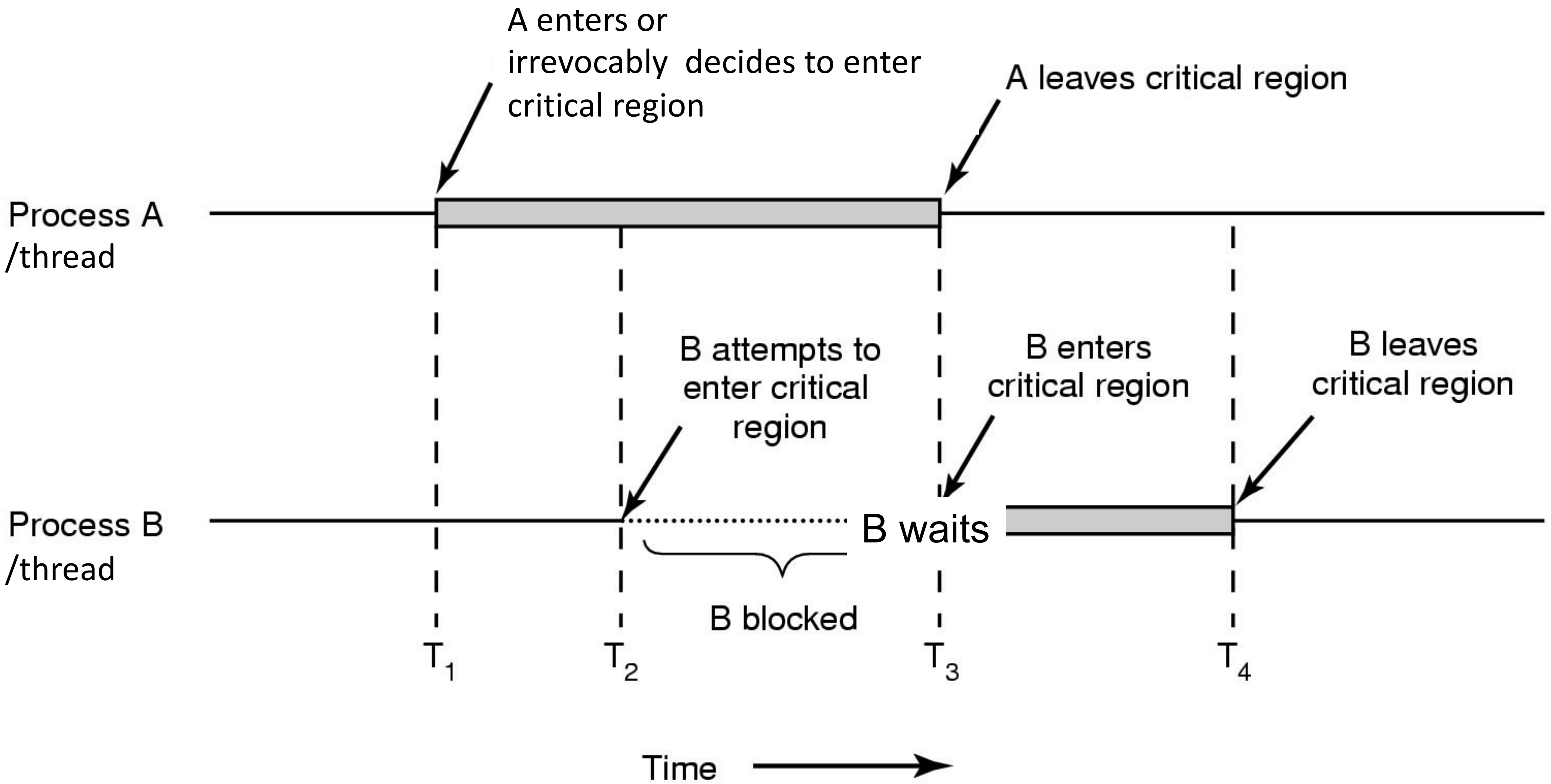
Problem formulation:

A solution must provide the entry and exit sections

It must ensure:

1. **Mutual Exclusion**. Only one process/thread at a time is allowed to execute in its critical section.
2. **Progress (no deadlock/no livelock)**. If no process/thread is in its critical section and some is/are trying, some must enter its critical section in bounded time.
3. **Fairness / Bounded Waiting / no starvation**. Variety of formulations, e.g., FCFS; or bounded #bypasses, …

Critical section: desirable behavior



Beware!

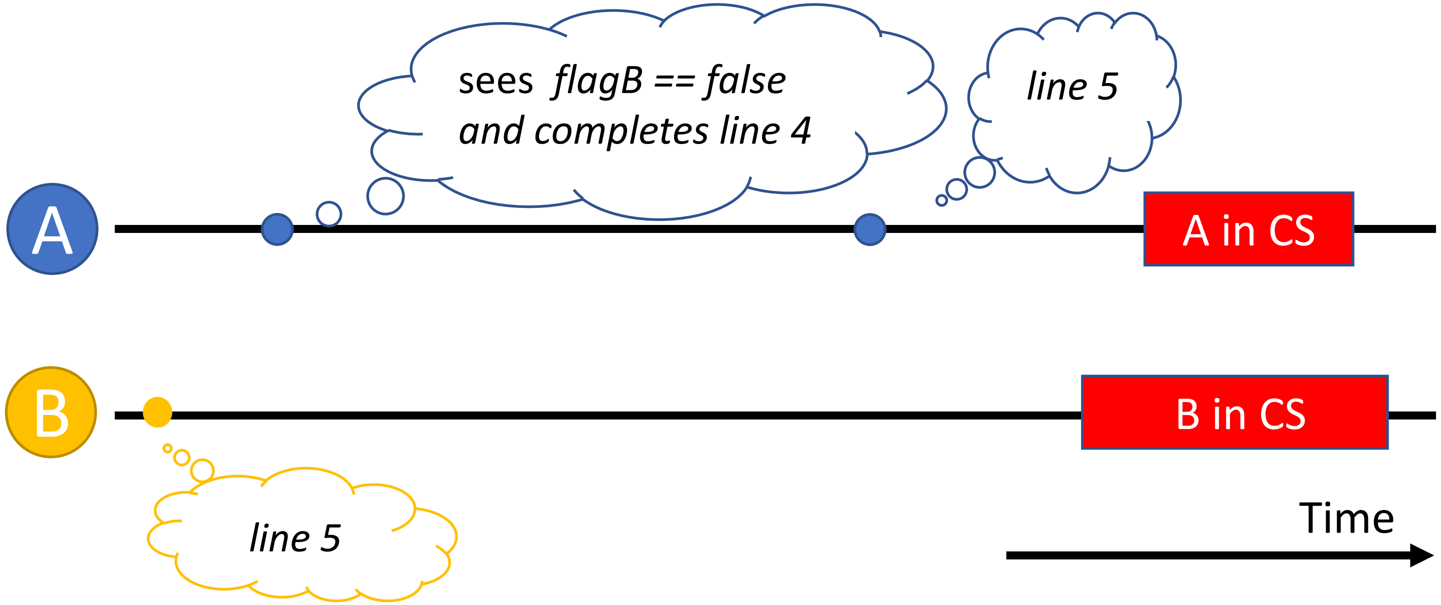
- Is a solution valid?
- We are looking for formal answers
  - Can we find an example showing it is not? then it is NOT valid.
  - Can we show it is impossible to find such an example? Then it is valid.
- **An example showing it works is not sufficient**

check-competition approach

```
Shared var flagA, flagB: boolean; // indicate A's, B's intention to enter; initially both false
Thread A // B is symmetric
  repeat
    while flagB == true do [nothing]  //busywait
      flagA ← true
      [critical section]
      flagA ← false
      [remainder section]
  forever
```

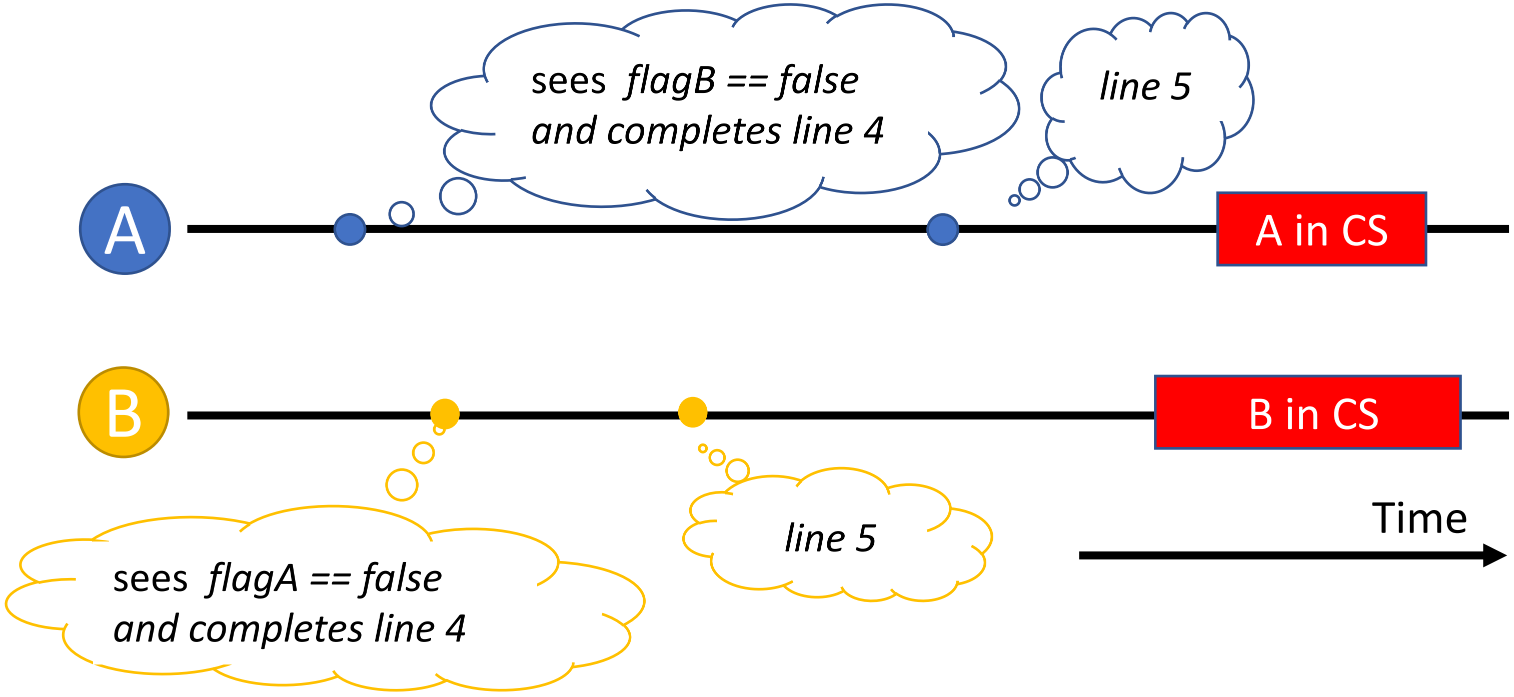
check-competition: mutual exclusion property?

1. Shared **var** *flagA, flagB*: *boolean*; // indicate A's, B's intention to enter; initially both *false*
2. Thread *A* // *B is symmetric*
3.   **repeat**
4.   **while** *flagB* == true **do** [nothing] //busywait
5.   *flagA* ← true
6.   [critical section]
7.   *flagA* ← *false*
8.   [remainder section]
9.   **forever**



check-competition: mutual exclusion property?

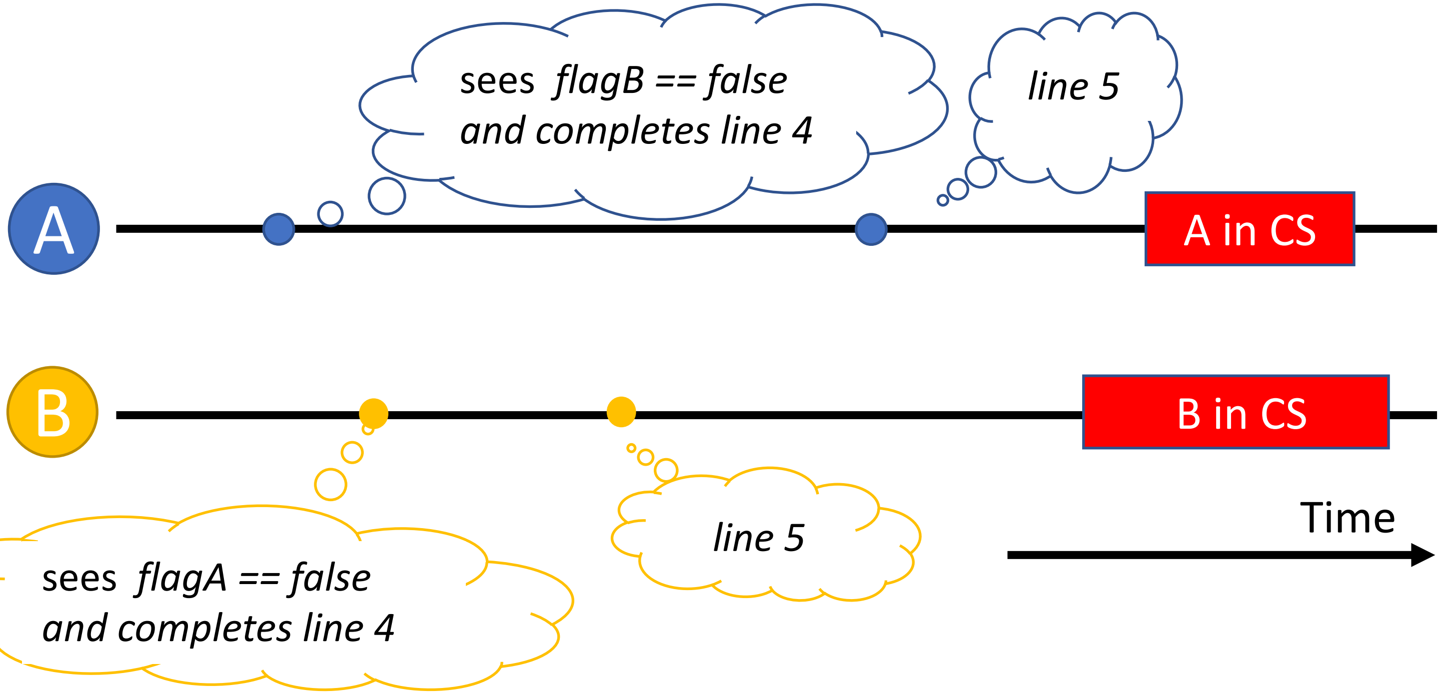
1. Shared **var** *flagA, flagB*: *boolean*; // indicate A's, B's intention to enter; initially both *false*
2. Thread *A* // *B is symmetric*
3. **repeat**
4. **while** *flagB* == true **do** [nothing] //busywait
5. *flagA* ← true
6. [critical section]
7. *flagA* ← *false*
8. [remainder section]
9. **forever**



check-competition: mutual exclusion property?

1. Shared **var** *flagA, flagB*: *boolean*; // indicate A's, B's intention to enter; initially both *false*
2. Thread *A* // *B is symmetric*
3.   **repeat**
4.   **while** *flagB* == true **do** nothing // busywait
5.   *flagA* ← *true*
6.   [critical section]
7.   *flagA* ← *false*
8.   [remainder section]
9.   **forever**

NOT VALID



Agenda

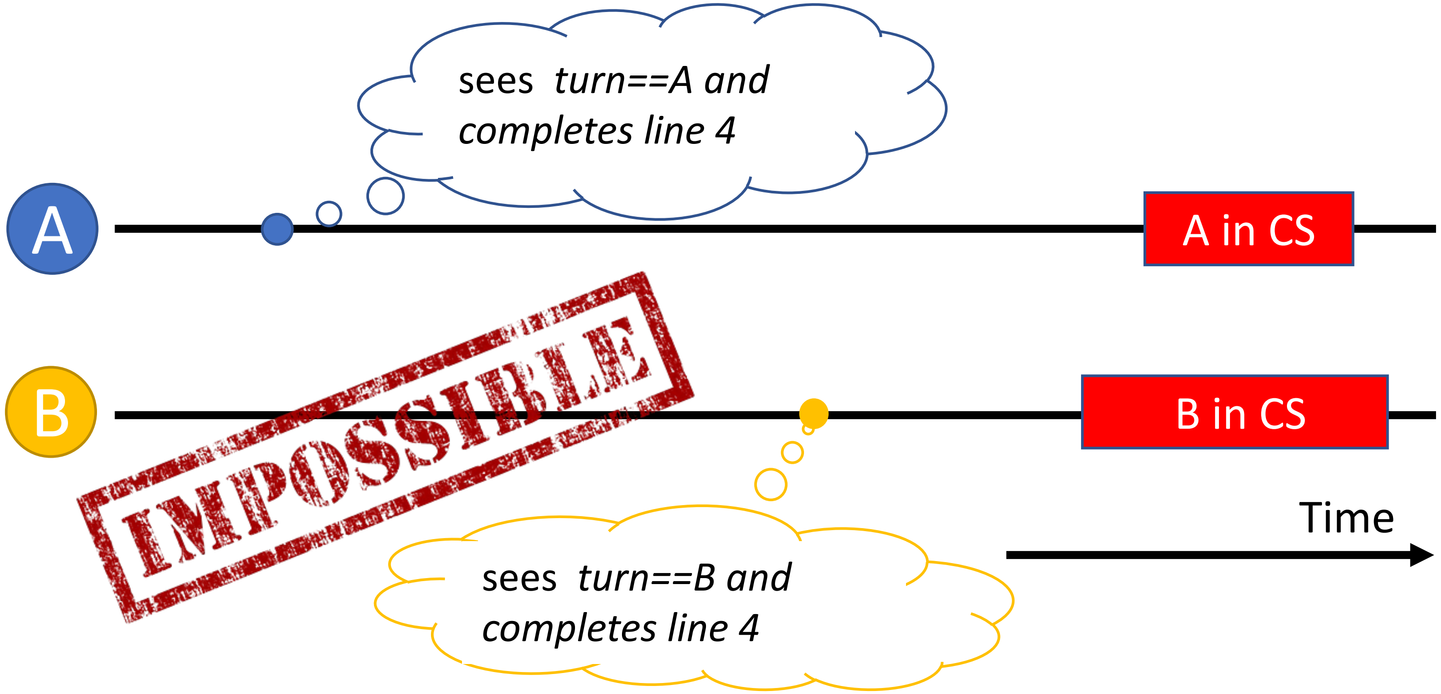
- Motivation
- The Critical Section problem
  - Check-competition approach
- The check-order / after-you approach
- Peterson’s Algorithm
- Synchronization and hardware support
  - Read-Modify-Write instructions
- Semaphores
- Other techniques

check-order (*after-you*) approach

```
Shared var turn: (idA..idB)   // initially e.g. idA
// indicates whose turn it is to enter CS
Thread A  // B is symmetric
  repeat
    while turn ≠ idA do [nothing]  //busywait
    [critical section]
    turn ← idB
    [remainder section]
  forever
```

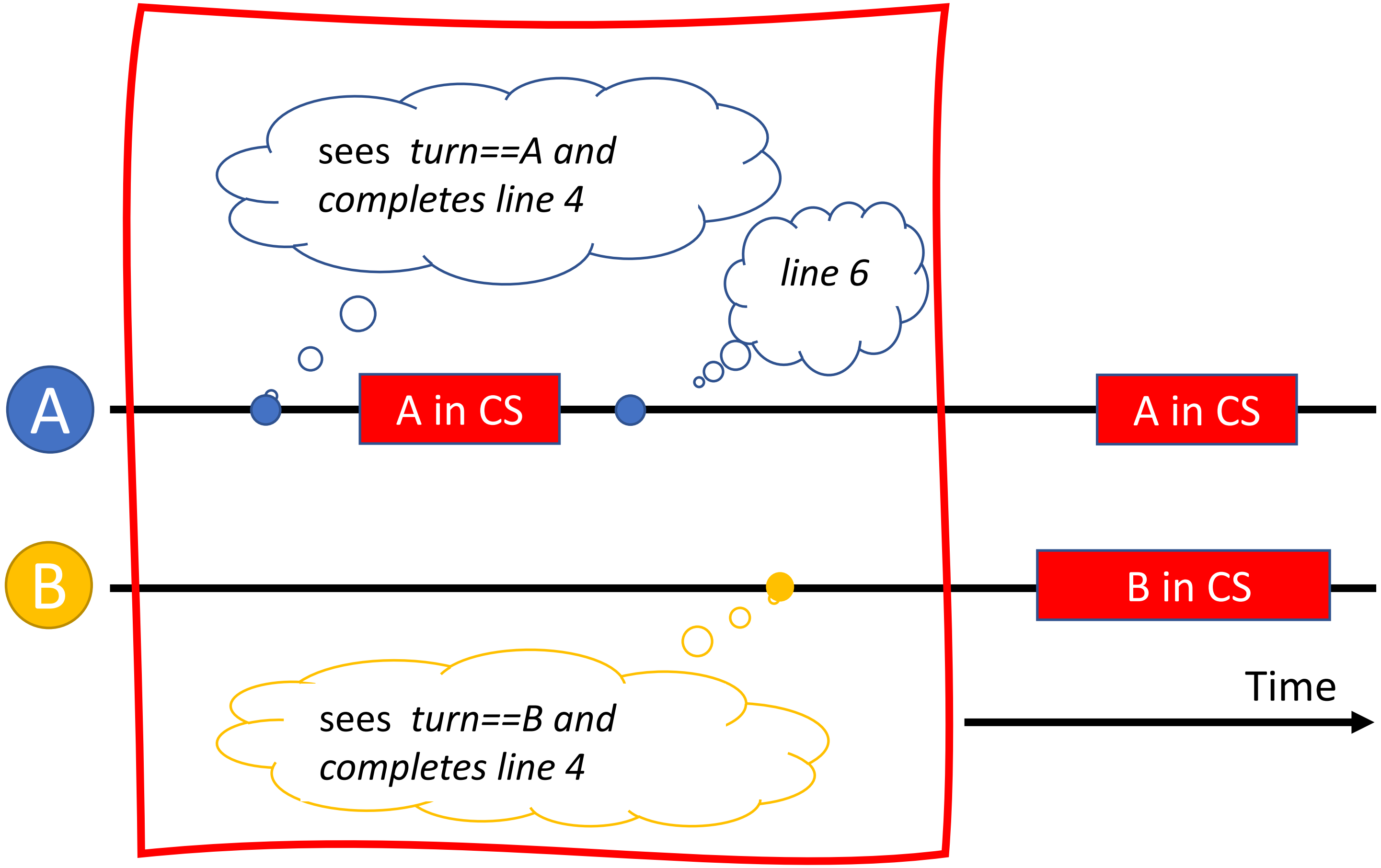
check-order: mutual exclusion property?

```
1.  Shared var turn: (idA..idB)   // initially e.g. idA
    // indicates whose turn it is to enter CS
2.  Thread A  // B is symmetric
3.    repeat
4.      while turn ≠ idA do [nothing]  //busywait
5.      [critical section]
6.      turn ← idB
7.      [remainder section]
8.    forever
```



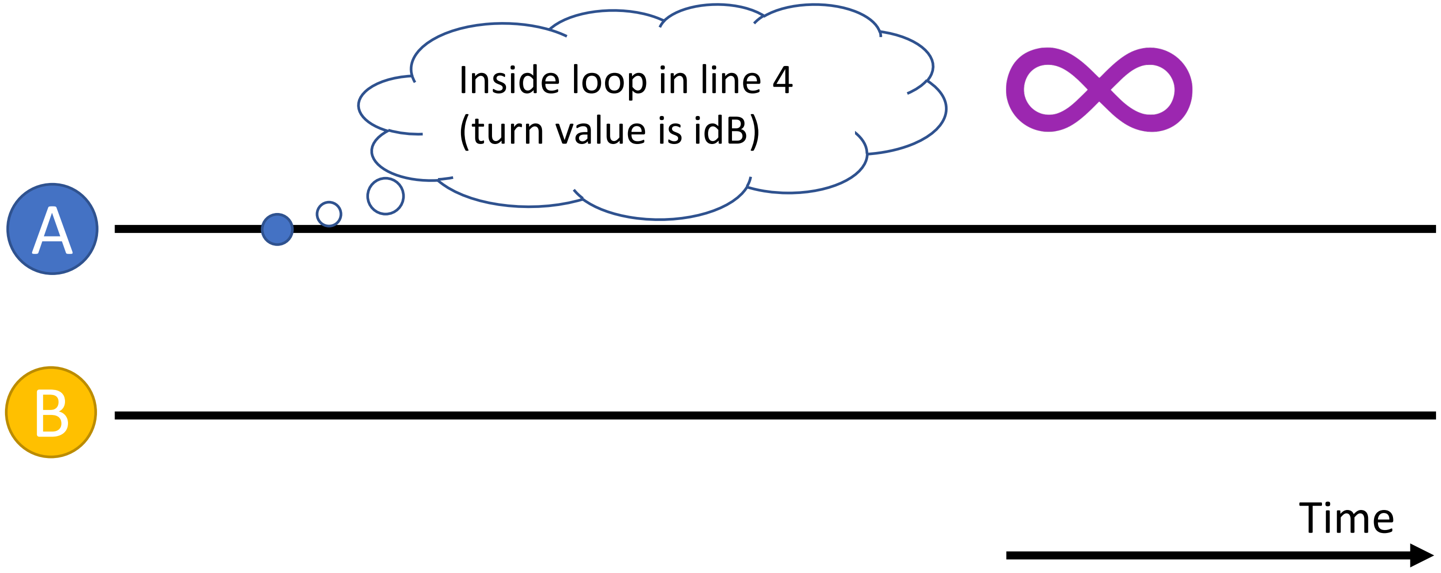
check-order: mutual exclusion property?

```
1. Shared var turn: (idA..idB)   // initially e.g. idA
   // indicates whose turn it is to enter CS
2. Thread A  // B is symmetric
3.   repeat
4.     while turn ≠ idA do [nothing]  //busywait
5.     [critical section]
6.     turn ← idB
7.     [remainder section]
8.   forever
```



check-order: progress property?

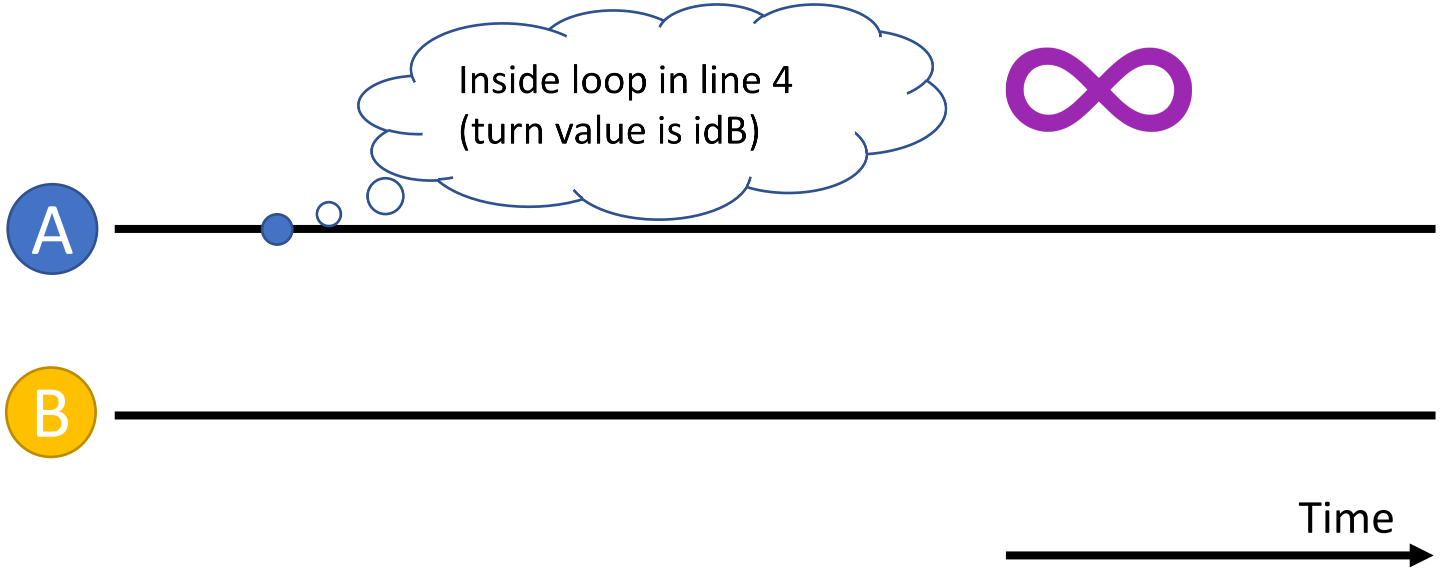
```
1.  Shared var turn: (idA..idB)   // initially e.g. idA
    // indicates whose turn it is to enter CS
2.  Thread A  // B is symmetric
3.    repeat
4.      while turn ≠ idA do [nothing]  //busywait
5.      [critical section]
6.      turn ← idB
7.      [remainder section]
8.    forever
```



check-order: progress property?

```
1.  Shared var turn: (idA..idB)   // initially e.g. idA
    // indicates whose turn it is to enter CS
2.  Thread A  // B is symmetric
3.    repeat
4.      while turn ≠ idA do {nothing}  // busy wait
5.      [critical section]
6.      turn ← idB
7.      [remainder section]
8.    forever
```

NOT VALID



Agenda

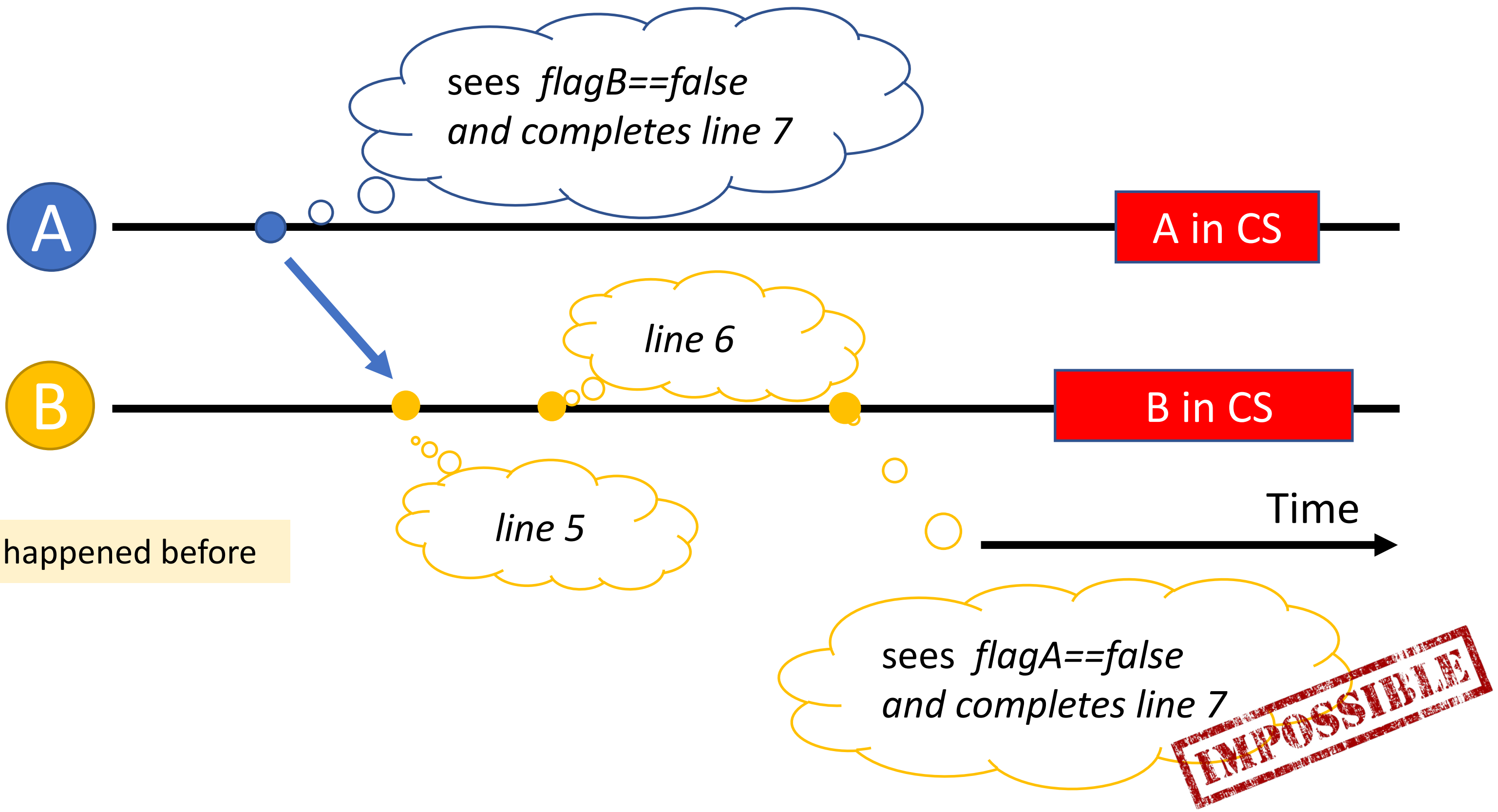
- Motivation
- The Critical Section problem
  - Check-competition approach
- The check-order / after-you approach
- Peterson’s Algorithm
- Synchronization and hardware support
  - Read-Modify-Write instructions
- Semaphores
- Other techniques

Peterson’s algo (*check order&competition*) approach

```
Shared var turn: (idA..idB) // initially e.g., idA
Shared var flagA, flagB: Boolean // initially false

thread A // B is symmetric
  repeat
     flagA ← true
     turn ← idB
     while (flagB and turn == idB) do [nothing]
     [critical section]
     flagA← false
     [remainder section]
  forever
```

Peterson’s algo: mutual exclusion property?

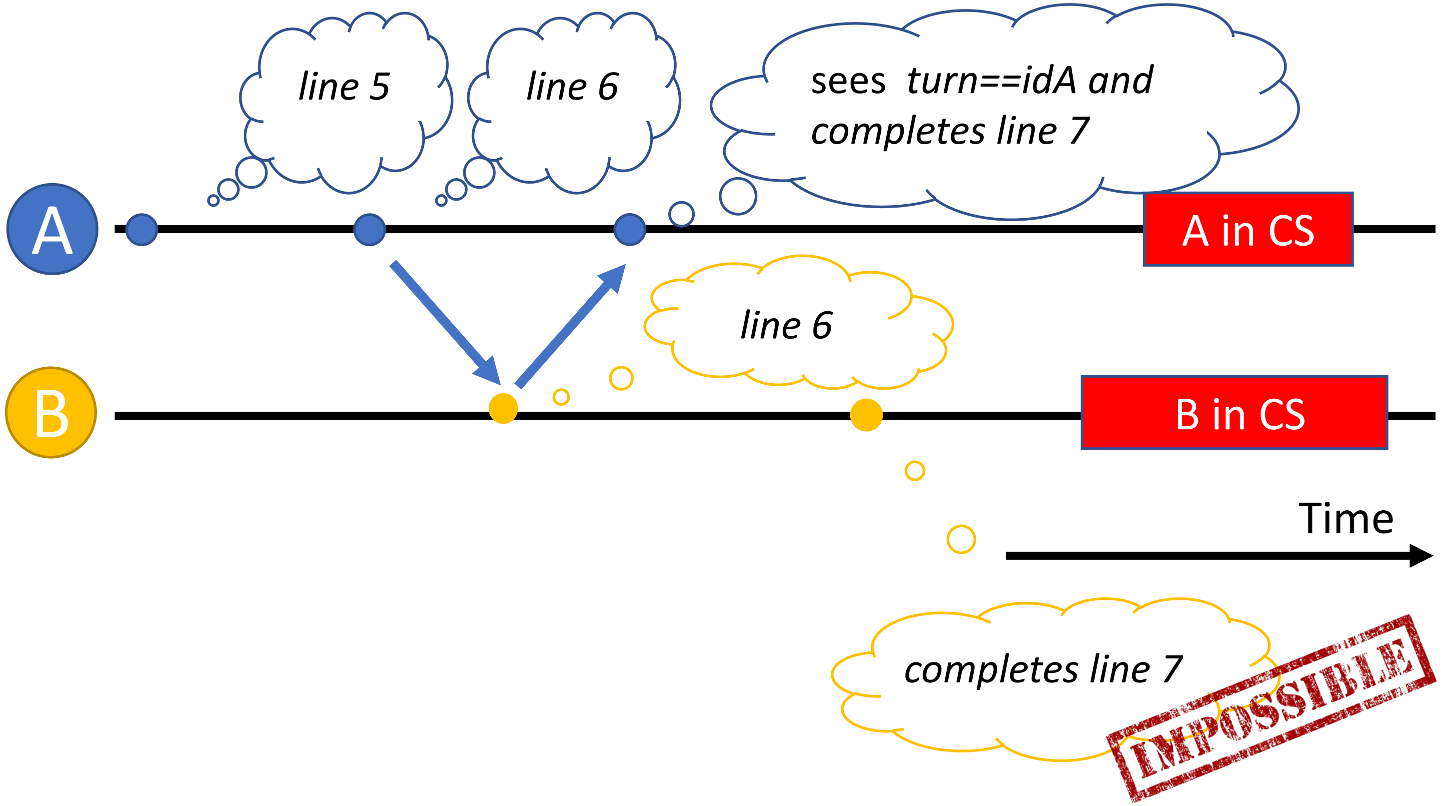
```
1.  Shared var turn: (idA..idB) // initially e.g., idA
2.  Shared var flagA, flagB: Boolean // initially false

3.  thread A // B is symmetric
4.    repeat
5.      flagA ← true
6.      turn ← idB
7.      while (flagB and turn == idB) do [nothing]
8.      [critical section]
9.      flagA← false
10.     [remainder section]
11.   forever
```



-> , ⟶ denote ”happened before

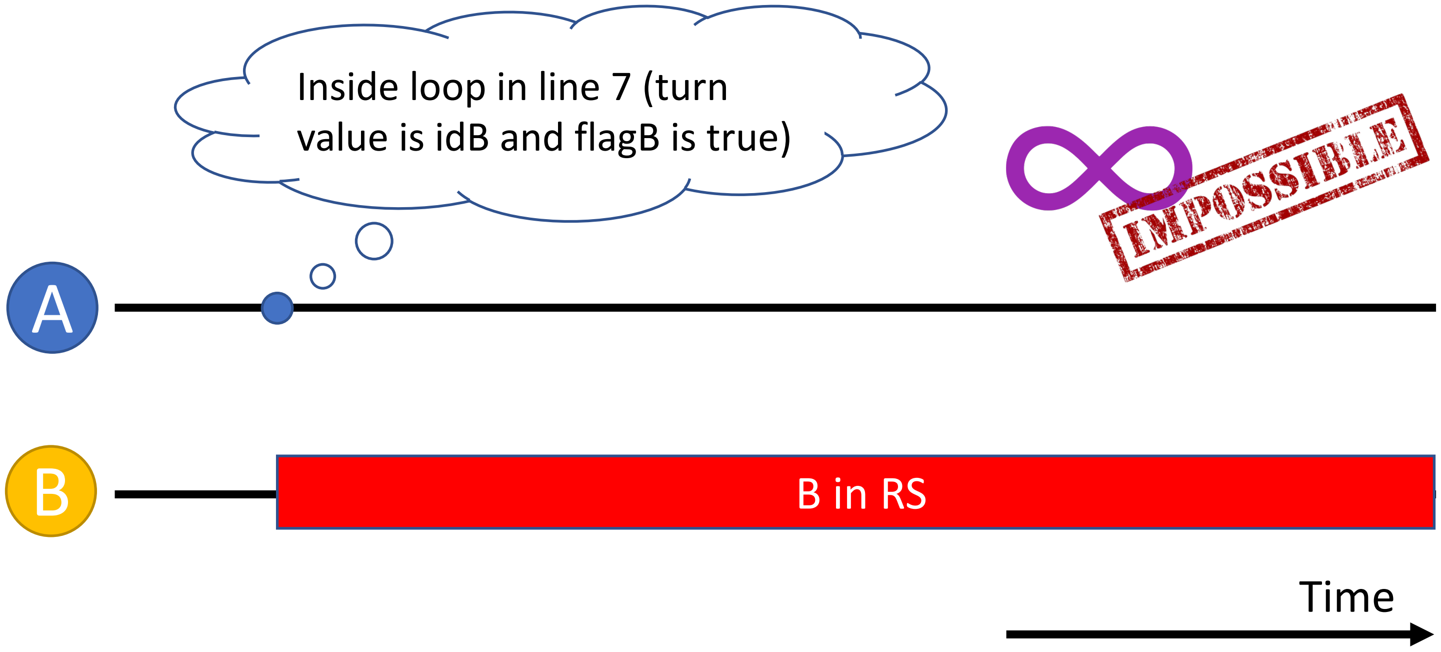
Peterson’s algo: mutual exclusion property?

```
1.  Shared var turn: (idA..idB) // initially e.g., idA
2.  Shared var flagA, flagB: Boolean // initially false

3.  thread A // B is symmetric
4.    repeat
5.      flagA ← true
6.      turn ← idB
7.      while (flagB and turn == idB) do [nothing]
8.      [critical section]
9.      flagA← false
10.     [remainder section]
11.   forever
```



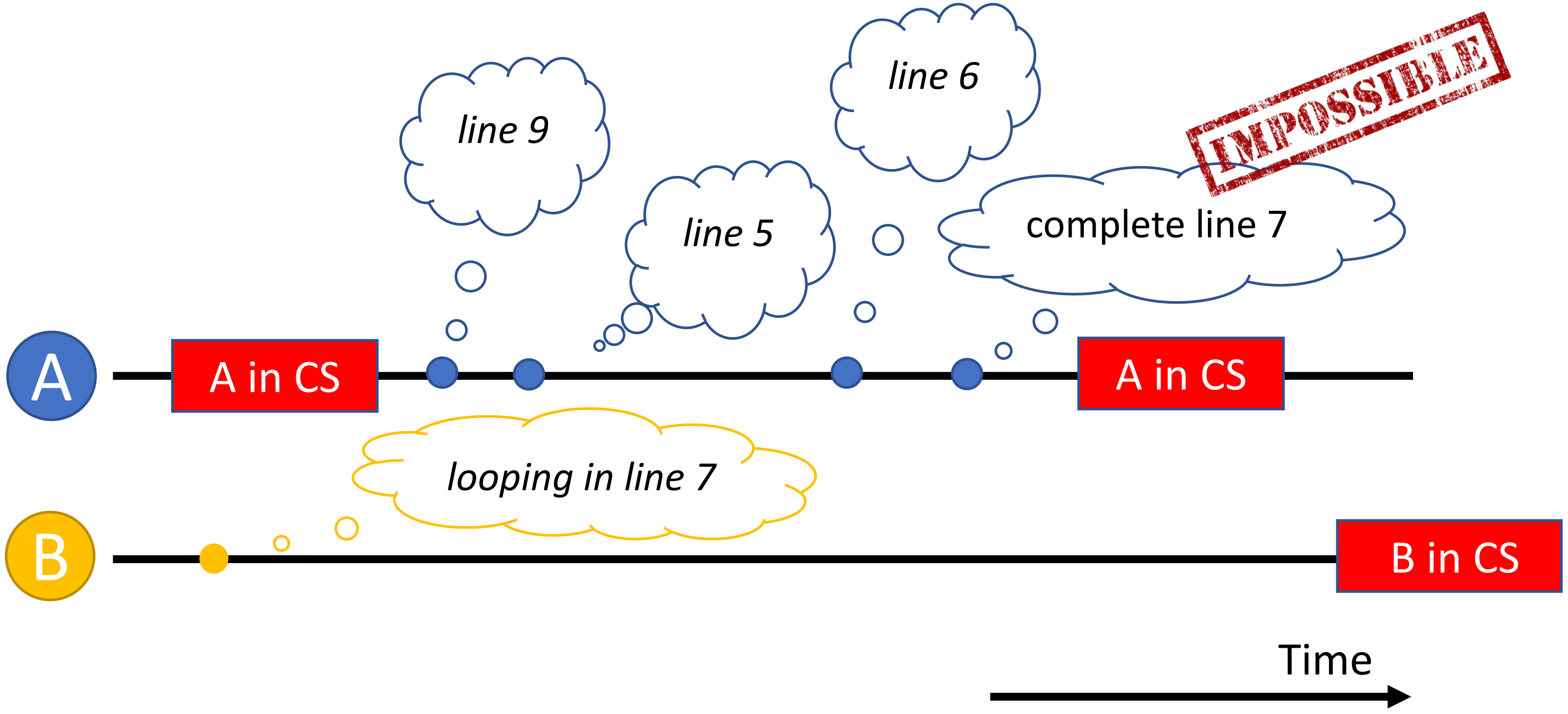
Peterson’s algo: progress property?

```
1.  Shared var turn: (idA..idB) // initially e.g., idA
2.  Shared var flagA, flagB: Boolean // initially false

3.  thread A // B is symmetric
4.    repeat
5.      flagA ← true
6.      turn ← idB
7.      while (flagB and turn == idB) do [nothing]
8.      [critical section]
9.      flagA← false
10.     [remainder section]
11.   forever
```



Peterson's algo: fairness property?

```
1.  Shared var turn: (idA..idB) // initially e.g., idA
2.  Shared var flagA, flagB: Boolean // initially false

3.  thread A // B is symmetric
4.    repeat
5.      flagA ← true
6.      turn ← idB
7.      while (flagB and turn == idB) do [nothing]
8.      [critical section]
9.      flagA← false
10.     [remainder section]
11.   forever
```



On counterexamples & proofs by way of contradiction:

- This way of thinking and arguing, when we are presented with a solution to a synchronization problem/race-condition (i.e., try to "attack" it, try to find if it does not work), is practical and useful (and thus very common)
- It is very frequently used when arguing about correctness/incorrectness of proposed solutions. This way of arguing helps us to:
  - give a proof by contradiction that a proposition in focus is _*correct*_, e.g., that mutual exclusion is preserved;
    - see about proofs by way of contradiction @ e.g., http://cgm.cs.mcgill.ca/~godfried/teaching/dm-reading-assignments/Contradiction-Proofs.pdf p 1-4 or http://web.stanford.edu/class/archive/cs/cs103/cs103.1152/lectures/02/Small02.pdf p 45-47
  - OR identify a counter-example, which implies that the proposition in focus is *incorrect* (i.e., it *disproves* the proposition)
    - see also https://mathforlove.com/lesson/counterexamples/ and/or https://study.com/academy/lesson/counterexample-in-math-definition-examples.html and/or http://web.stanford.edu/class/archive/cs/cs103/cs103.1152/lectures/02/Small02.pdf

Agenda

- Motivation
- The Critical Section problem
  - Check-competition approach
- The check-order / after-you approach
- Peterson’s Algorithm
- Synchronization and hardware support
  - Read-Modify-Write instructions
- Semaphores
- Other techniques

Critical section and hardware support: interrupt disabling

Structure of process/thread *P*

Repeat

interrupt disable

critical section

interrupt enable

remainder section

Forever

Is there a downside?!

- If run in single-processor system: limited ability to interleave programs
- In multiprocessors: disabling interrupts on one processor/core will not guarantee absence of race conditions by threads running on other processors/cores

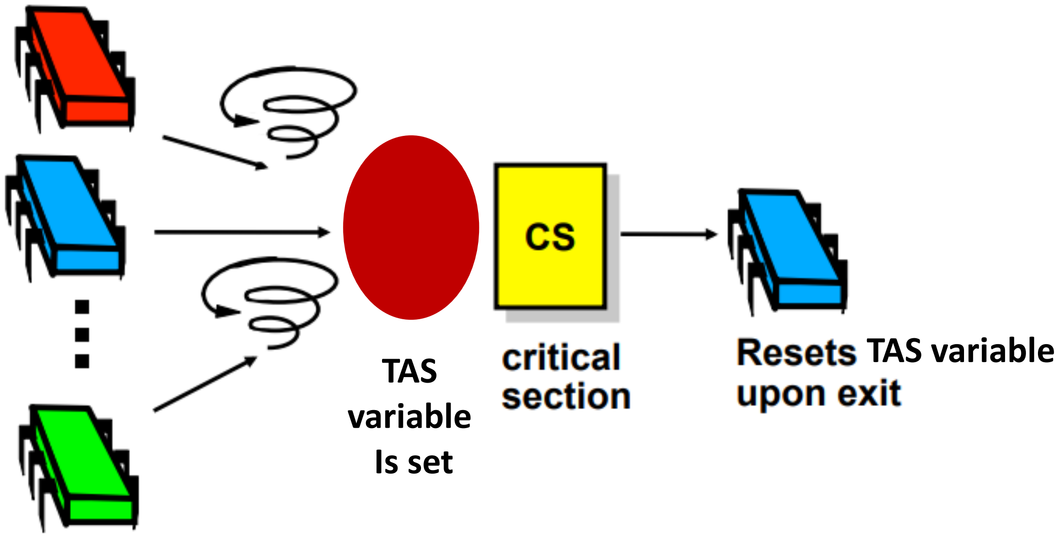
Critical section and hardware support: Read-Modify-Write (RMW) instructions

TestAndSet (aka TAS) Instruction Definition:

```
boolean TestAndSet (boolean *target){
  boolean rv = *target;
  *target := true;
  return rv} // Executed atomically by the HW
```

Shared Boolean **var** *lock;* initialized to *false*

```
repeat
  while(TestAndSet(&lock)) do [nothing];
  [critical section]
  lock := false;
  [remainder section]
forever
```



Homework training: argue as we did for Peterson's algo regarding correctness (mutual exclusion, progress) ; what about fairness?

Critical section and hardware support: Read-Modify-Write (RMW) instructions

TestAndSet (aka TAS) Instruction Definition:

```
boolean TestAndSet (boolean *target){
  boolean rv = *target;
  *target := true;
  return rv} // Executed atomically by the HW
```

Shared Boolean **var** *lock;* initialized to *false*

```
repeat
  while(TestAndSet(&lock)) do [nothing];
  [critical section]
  lock := false;
  [remainder section]
forever
```

CompareAndSwap (aka CAS) Instruction Definition:

```
int CompareAndSwap(int *V, int expected, int
   new_value) {
  int temp := *V;
  if (temp == expected) then *V := new_value;
  return temp} // Executed atomically by the HW
```

Shared int **var** *lock;* initialized to 0;

```
repeat{
  while CompareAndSwap(&lock,0,1)!=0) do nothing;
  [critical section]
  lock := 0 ;
  [remainder section]
forever
```

A reflection about busy-waiting

- Rather simple, convenient; BUT:
  - Busy-waiting consumes processor time unnecessarily
  - Starvation is possible when ***using methods as in the previous slides***
  - Even **Deadlock possible** if spinning in **non-preemptive priority-based scheduling**: example scenario: threads/processes on the same CPU and:
    - low priority thread LP executes in critical section
    - higher priority thread HP needs to access its CS
    - the higher priority thread HP is dispatched (i.e., occupies the processor) and busy-waits for the LP to exit from its critical section…

Critical section + bounded waiting with TaS + busy-waiting for n threads

```
Shared var lock: boolean // init false;
Shared var waiting[0..n-1]: array of boolean // init all false;
do
waiting[i] := TRUE;
while (waiting[i] && TestAndSet(&lock) ) do [nothing]; //busywait
waiting[i] := FALSE;
[critical section ]
j = (i + 1) % n;
while ((j != i) && !waiting[j])  do // find next one waiting, to help it (handover the "lock" )
    j := (j + 1) % n;
    if (j == i)  then
            lock := false;  // completed one round without handing over; release "lock"
    else
            waiting[j] := false; // hand-over "lock"
[remainder section ]
forever;
```

Agenda

- Motivation
- The Critical Section problem
  - Check-competition approach
- The check-order / after-you approach
- Peterson’s Algorithm
- Synchronization and hardware support
  - Read-Modify-Write instructions
- Semaphores
- Other techniques

Synchronization with semaphores

Special variables/data-objects used for signaling

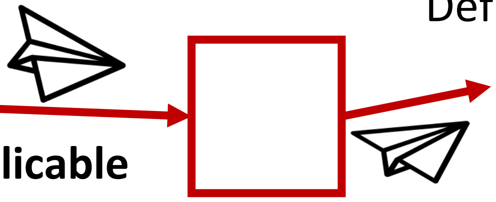
- Accessible via (i.e., API) ***atomic Wait and Signal operations***
- If a process is waiting for a signal, it is blocked until that signal is "sent"

Definition of the **signal() operation**

signal(S) {

S++; // and unblock from queue if applicable

} // Executed atomically



Definition of the **wait() operation**

wait(S) {

while (S == 0) do; // here spin; alt. block in a queue

S--;

} // Executed atomically

- **Binary semaphore** – value can be only 0 or 1
- **Counting semaphore** – value can vary

Critical section of *n* threads using semaphores

```
Shared var mutex_sem : binary semaphore // init = 1


Thread I
    repeat
        wait(mutex_sem);
        [critical section]
        signal(mutex_sem);
        [remainder section]
    forever
```

Semaphores as General Synchronization Tools

Q: how to execute code segment *B* in T_B only after code segment *A* has been executed in T_A:

A: use semaphore *flag,* initialized to 0

T_A	T_B
⋮	⋮
op*A*	*wait*(*flag*)
signal(*flag*)	op*B*

T_B will be able to proceed from wait(flag) only after signal(flag) is executed by T_A

Agenda

- Motivation
- The Critical Section problem
  - Check-competition approach
- The check-order / after-you approach
- Peterson’s Algorithm
- Synchronization and hardware support
  - Read-Modify-Write instructions
- Semaphores
- Other techniques

Other synchronization operations/constructs:

Tools for programmers to solve critical section and synchronization problems (can be implemented using hardware atomic instructions or algorithms such as Peterson's or other such tools

mutex lock x; API :

- x.acquire() wait/block if x is not available; else make x not available
- x.release() unblock a waiting/blocked process; else make x available

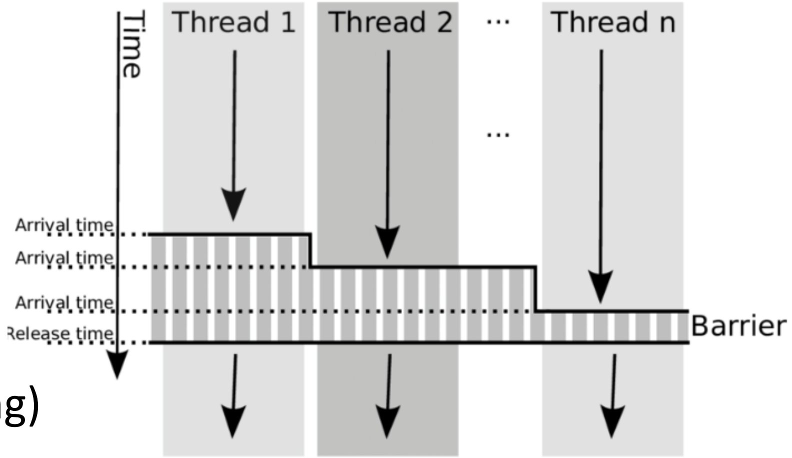
- If implemented with busy waiting, called spinlock

Condition Variables: condition x; API:

- x.wait () – a process that invokes the operation is blocked.
- x.signal () – unblocks one of processes that invoked x.wait () (if any; else, does nothing)

Other high-level synchronization constructs

- Monitors (combination of mutex locks and condition variables)
- Barriers



3.: Global event synchronisation: an example of n threads synchronised at a barrier with corresponding arrival times and release time of all n threads when the barrier condition is fulfilled.

Linux Synchronization provides system calls for:

- semaphores
- spin locks for multiprocessors (for short critical sections): why?
  - [hint: wastes CPU time and can cause deadlocks if used with strict priority scheduling on uniprocessor]
- futex
  - *try_lock* through 1 invocation of RMW instruction*; if already locked, then process enters blocked queue; why?*
    - *[Balance the cost of entering a queue directly with the cost of spinning on the RMW instruction]*
  - https://man7.org/linux/man-pages/man2/futex.2.html

Windows XP Synchronization

- processor-specific interrupt masks to protect access to global resources on uniprocessors
- busy-wait spinlocks on multiprocessors
  - why not on uniprocessors?
  - [hint: can cause deadlocks if used with strict priority scheduling]
- dispatcher objects
  - may act as mutexes, semaphores, or provide events (similar to condition variables)

Pthreads Synchronization

OS-independent; provides:

- mutex locks with blocking queues (also fairness in mind)
- condition variables
- Non-portable extensions include:
  - read-write locks (we will discuss these in upcoming lectures)
  - spin locks

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them