Fundamentos de los Sistemas Operativos (FSO)

Departamento de Informática de Sistemas y Computadoras (DISCA)
Universitat Politècnica de València

Part 2: Process management

Unit 6

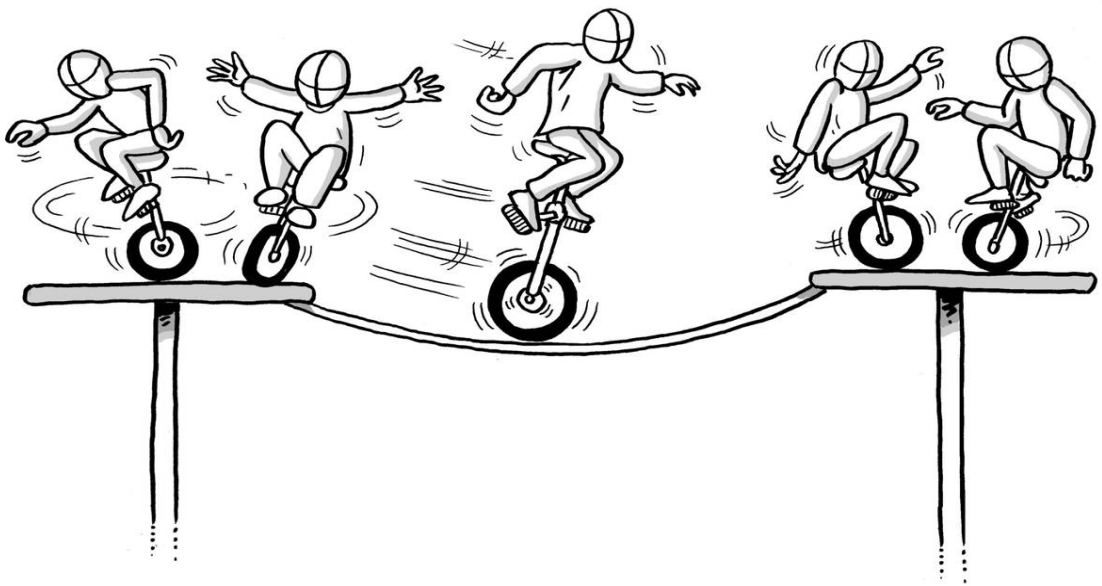
Synchronization: Basic solutions

fSO

- **Goals**
  - To understand the **critical section** concept
  - To know the **mechanisms provided by the computer hardware** to support synchronization
  - To solve **synchronization** problems by means of **software and hardware solutions**

- **Bibliography**
  - Silberschatz, chapter 6

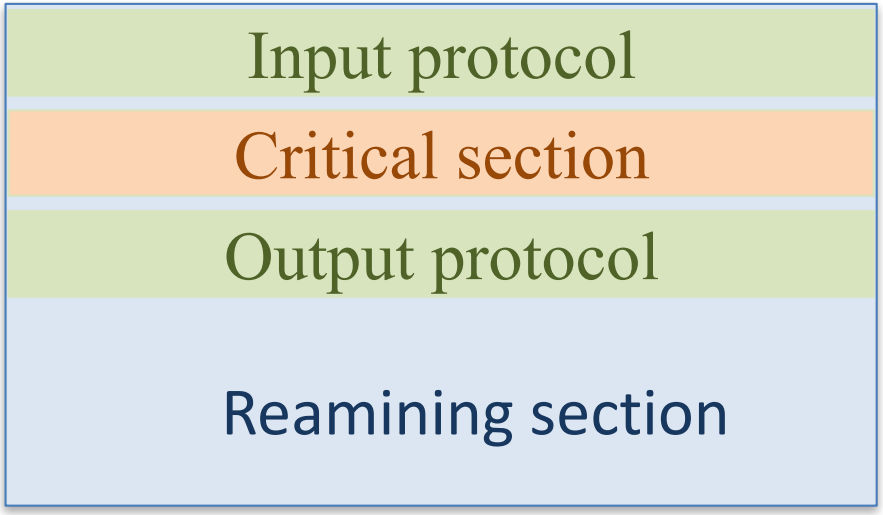
- **The critical section problem**
- Software solutions
- Hardware solutions
- Active waiting
- Exercises



The critical section problem

- **Critical section problem**
  - N>1 activities (processes / threads) execute concurrently accessing (writing/reading) shared data
  - Every activity
    - It has (at least) a code zone where **it accesses to shared data** -> This code zone is named **critical section**
    - It can have other code zones where there are no access to shared data -> These code zones are together named **remaining section**
- **Solution**
  - To find a protocol to **synchronize access to critical sections avoiding race condition**

Thread/process access to the critical section must be **serialized**

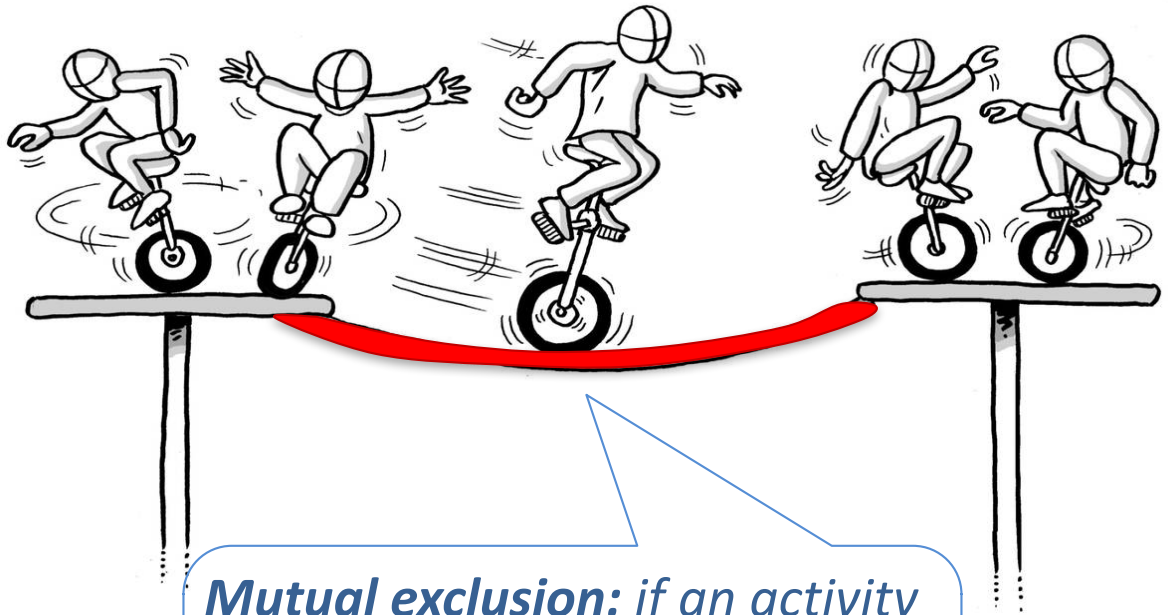


The critical section problem

- The **critical section access protocol** has to verify the following three conditions:
  - ***Mutual exclusion:*** if an activity is inside its critical section then no other activities can be at the same time in their critical sections, the others must wait until the critical section is free
  - ***Progress:*** if the critical section is free and there are more than one activity willing to enter it, the decision about which one enters is taken in finite time and only depends on the waiting activities
  - ***Limited waiting:*** After an activity has asked to enter the critical section there is a limited number of times other activities are allowed entering the critical section

- It is supposed that:
  - Activities are executed at non zero speed
  - Correctness cannot rely on the relative activity execution speed

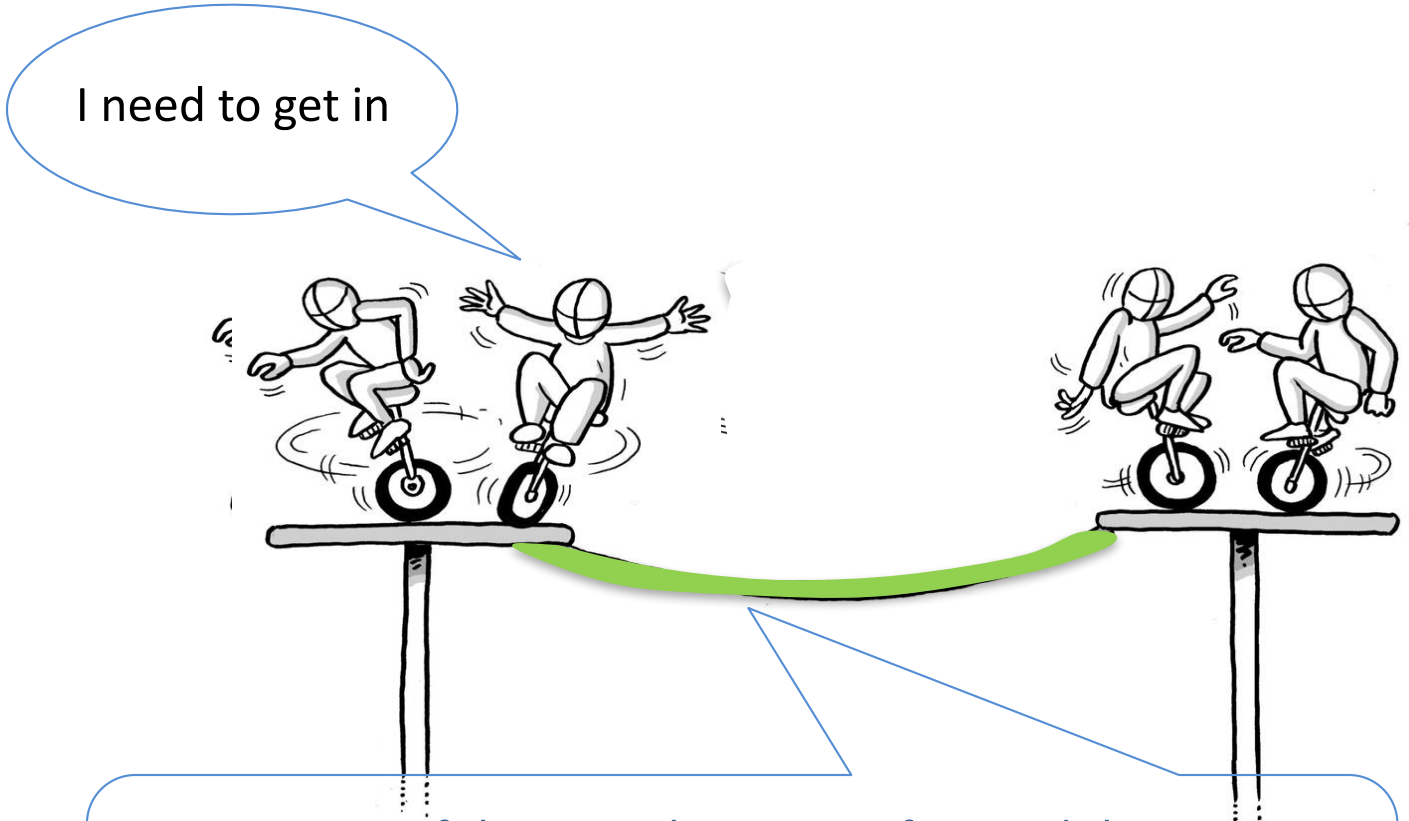
- **Critical section access protocol** has to verify the following three conditions:



Mutual exclusion: *if an activity is inside a critical section then no other activities can be there at the same time*

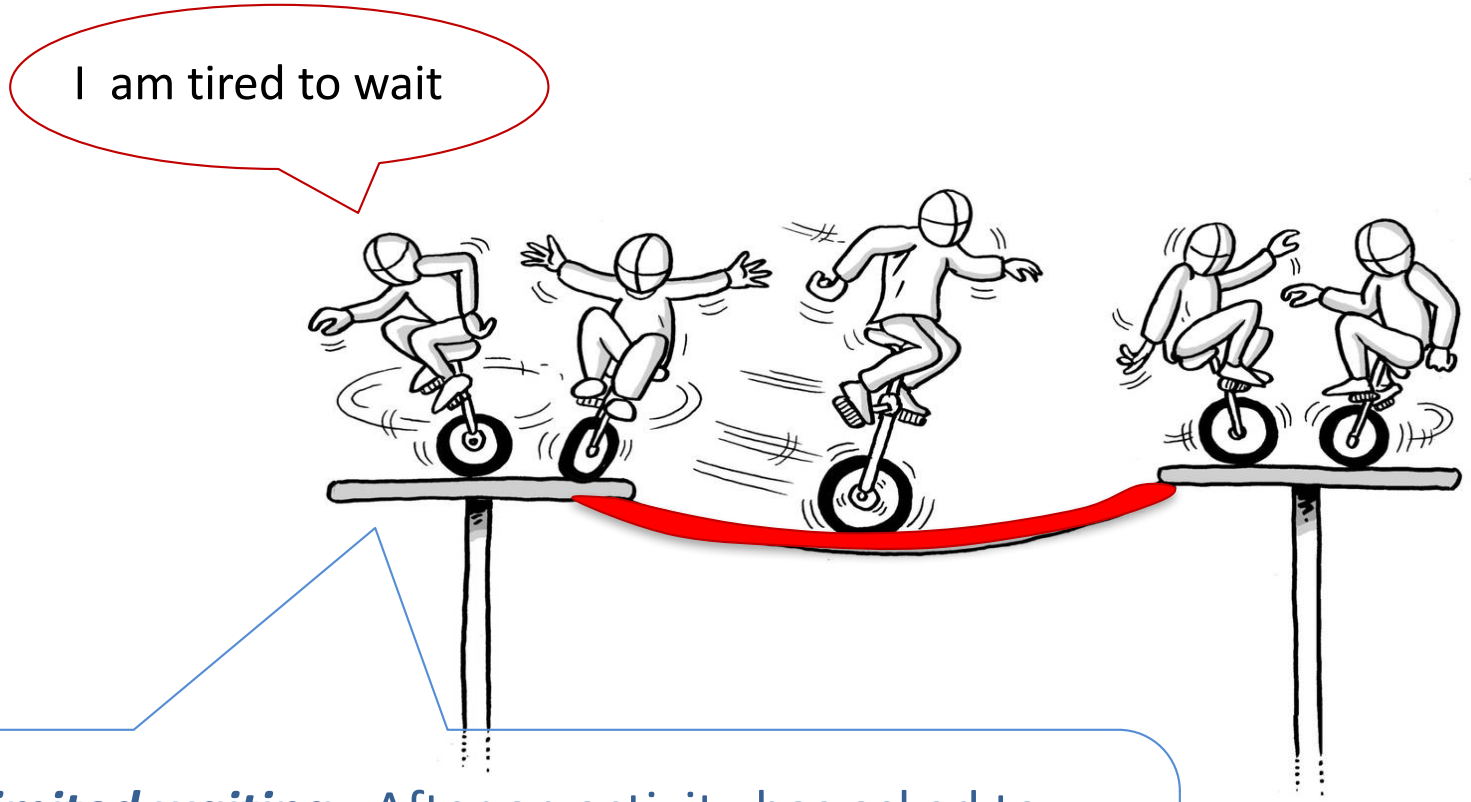
The critical section problem

- **Critical section access protocol** has to verify the following three conditions:



Progress: if the critical section is free and there are more than one activity willing to enter it, the decision about which one enters is taken in finite time and only depends on the waiting activities

- **Critical section access protocol** has to verify the following three conditions:

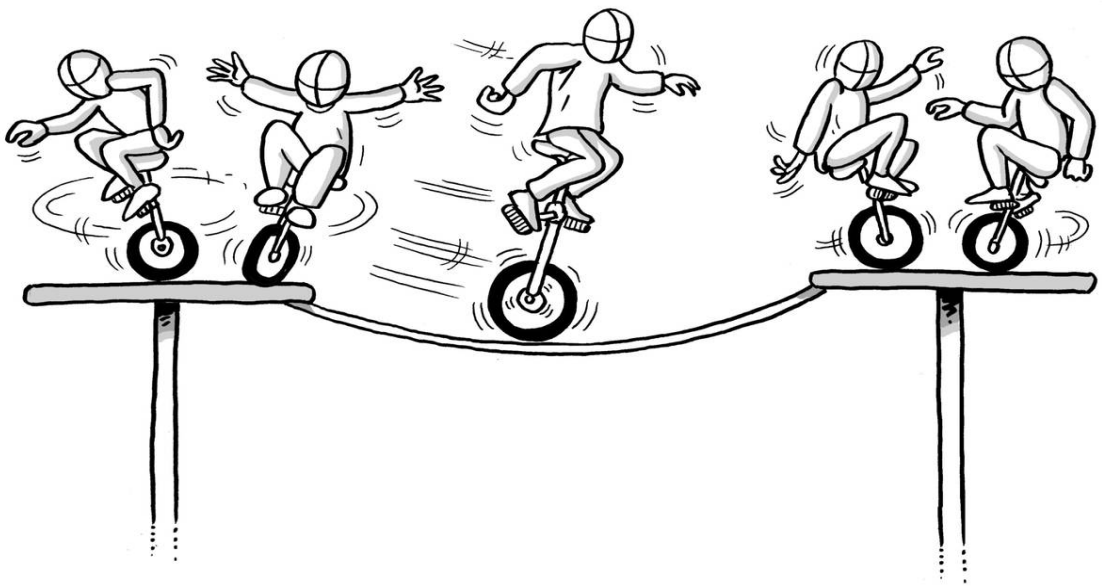


Limited waiting: After an activity has asked to enter the critical section there is a limited number of times other activities are allowed entering the critical section

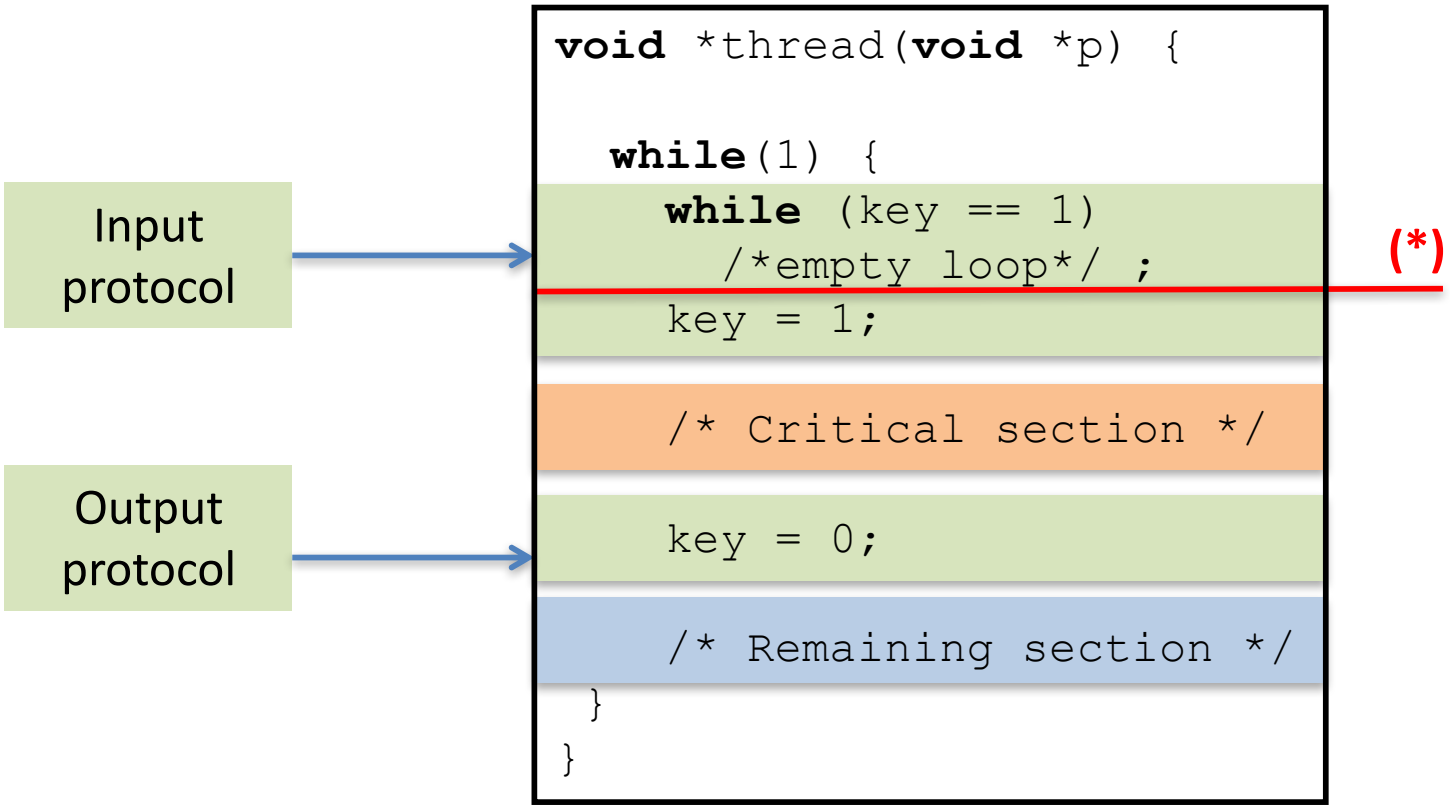
The critical section problem

- Solutions
  - Basic
    - **Software** solutions
      - The protocol is implemented at user level, without added mechanisms (i.e. Decker algorithms)
    - **Hardware** solutions
      - Special machine instructions are used to implement the protocol
  - With OS support
    - The protocol is provided by the OS by means of **system calls**
  - With programming language support
    - Some programming languages (i.e. Ada95, Java) provide data types that assure automatic mutual exclusion

- The critical section problem
- **Software solutions**
- Hardware solutions
- Active waiting
- Exercises



- **Basic algorithm**
  - Several activities use a shared variable "key" that indicates the critical section is busy, **"key" is initialized to 0**



  - **It doesn't verify "mutual exclusion".** If there is a context switch in (*) it is possible that several threads enter the critical section

- **Decker solution nº 1, strict alternation**
  - Two activities synchronize by means of a global variable `turn` that indicates the activity turn to enter the critical section
    - `turn` must be initialized to I or J

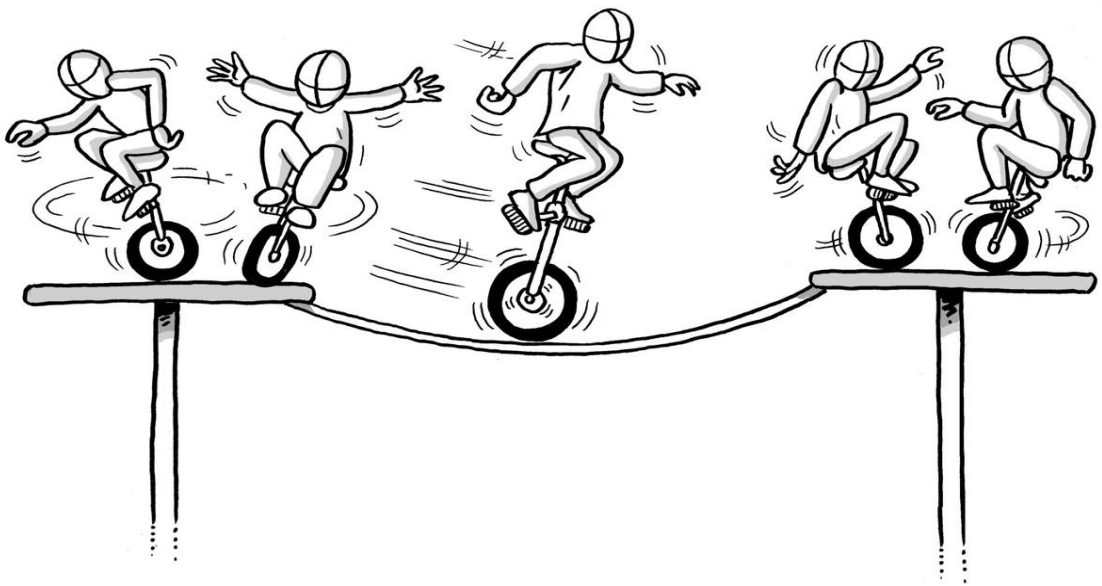
```
void *thread_I(void *p) {

  while(1) {
    while (turn != I)
      /*empty loop*/ ;

    /* Critical section */

    turn = J;

    /* Remaining section */
  }
}
```

```
void *thread_J(void *p) {

  while(1) {
    while (turn != J)
      /*empty loop*/ ;

    /* Critical section */

    turn = I;

    /* Remaining section */
  }
}
```

  - Works only with TWO threads
  - **It doesn't verify the "progress" condition.** It imposes a "relative speed" between threads

- **There exist completely correct software solutions**:
  - Decker, Peterson and Lamport algorithms
  - Software solutions are applied in **distributed systems**

- The critical section problem
- Software solutions
- **Hardware solutions**
- Active waiting
- Exercises



Hardware solutions

- When the activities are executed in the **same processor or in a shared memory multiprocessor** then **hardware solutions** are used, like:
  - Disabling interrupts (**privileged instructions**)
    - Activities must run on the same processor
  - Atomic instructions to access memory (i.e. **test-and-set**)
    - General solution for shared memory

Disable interrupts

- Execution of the following machine instructions
  - **DI**: disable interrupts
  - **EI**: enable interrupts

- Mutual exclusion is achieved by avoiding context switching inside the critical section -> atomic execution

```
...
DI;
/* Critical section*/
EI;
/* Remaining section */
...
```

This is much more than required, we only need to avoid simultaneous linked critical sections

- Only applicable to kernel level: DI and EI can only be executed in **kernel mode**

Atomic instruction "test-and-set"

- It allows atomic evaluation and change of a variable with only one machine instruction
- "test and set" functional specification is the following:

```
int test_and_set (int *target)
{
  int aux;

  aux = *target;
  *target = 1;
  return aux;
}
```

Done by only one machine instruction

- It returns the actual value (1 or 0) of "target", indicating true or false, respectively; and finally it sets target to 1 (true). The operation sequence is done atomically

Atomic = Indivisible, execution cannot be interrupted

Atomic instruction "test-and-set" (cont)

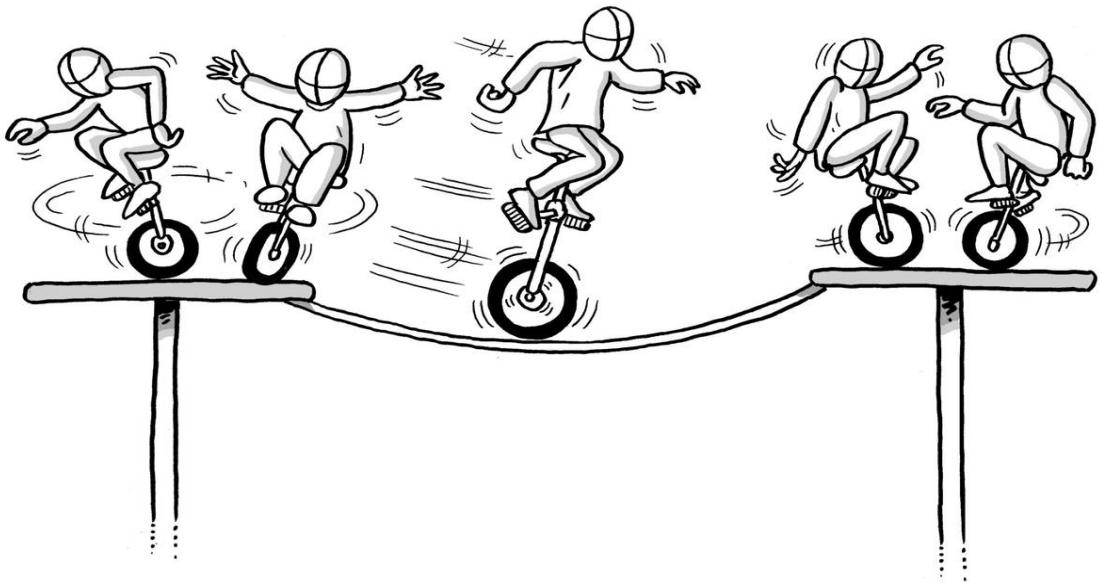
- Protocol to manage the critical section
  - Suppose N>1 threads executing the following code
  - Variable **`key` is shared and initialized to 0** (false)

```
void *thread_i(void *p) {

  while(1) {
    while (test_and_set(&key))
      /*Empty loop*/ ;

    /* Critical section */

    key = 0; /*false*/

    /* Remaining section */
  }
}
```

```
/* Global variable*/
int  key=0;
```

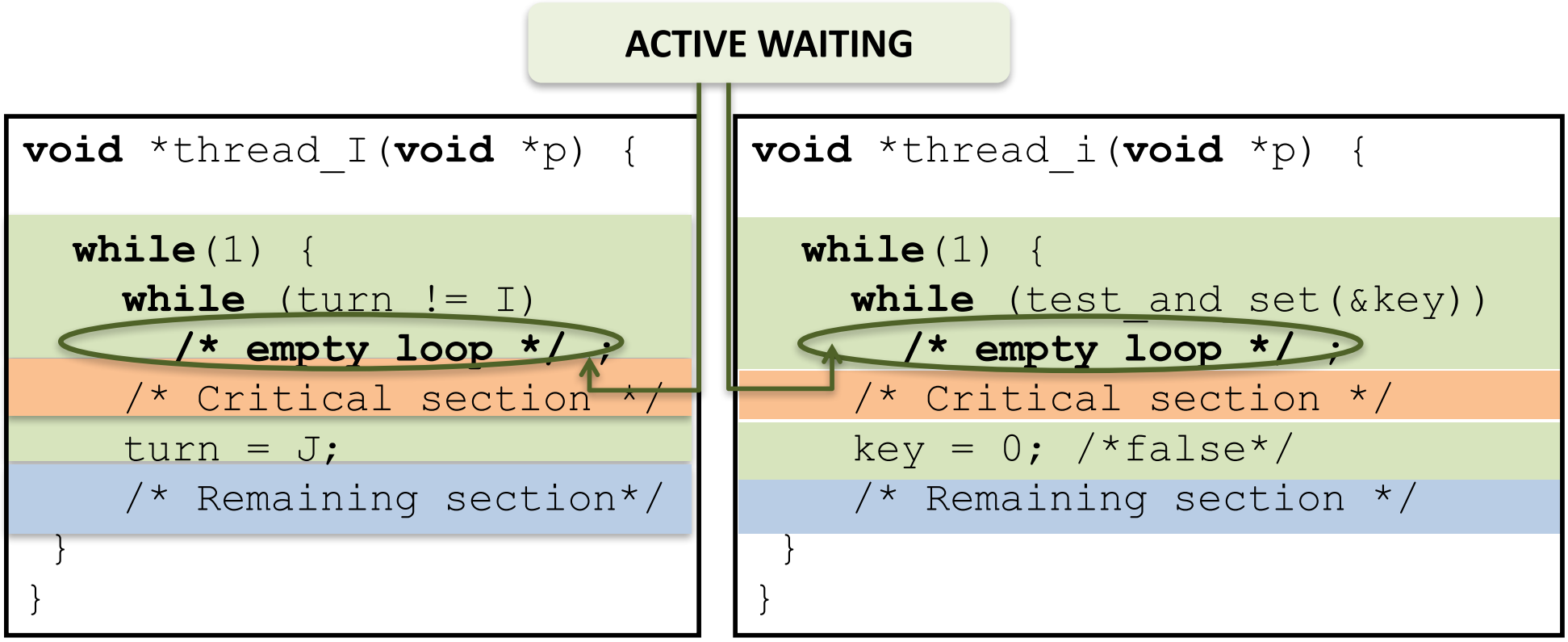
- It complies with mutual exclusion and progress but it doesn't with limited waiting
  - The following thread entering the critical section is unknown

- The critical section problem
- Software solutions
- Hardware solutions
- **Active waiting**
- Exercises



Active waiting

- All the described solutions (software and `test_and_set`) have a common feature: **active waiting**
- The critical section entering protocol prevents entering by **forcing the execution of an empty loop that consumes CPU time**

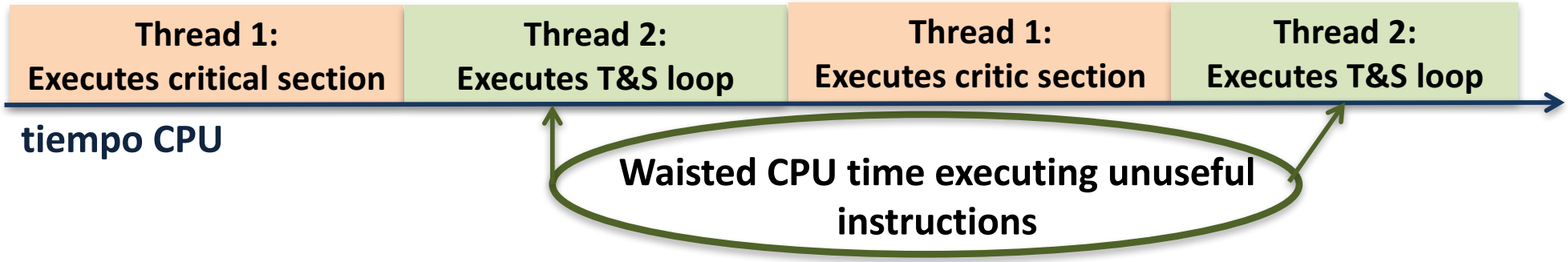


Software solutions
("Decker nº 1", strict alternance)

Protocol **"test-and-set" atomic instruction**

Active waiting

- Advantage:
  - Allows stopping an activity without calling the OS
- Disadvantages:
  - **If scheduling is based on priorities:**
    - If a thread with higher priority gets cached in an active waiting while a thread with lower priority is inside the critical section, all the other threads will keep waiting forever (priority inversion)
  - **Even within less troubling schedule policies like round-robin:**
    - Active waiting wastes processor time



Active waiting

- Active waiting alternatives
  - Rely on the OS to suspend the thread that does `test_and_set` and can't enter the critical section

```
while( test_and_set(&key) ) {usleep(time);}
```

  - Or simply forcing the thread to leave the CPU

```
while( test_and_set(&key) ) {yield();}
```

  - The **OS provides synchronization objects** where activities can keep suspended until another activity will awake them, i.e.

  - **SEMAPHORES AND MUTEXES**
    - **They are accessed through system calls** synchronization primitives are provided by the OS
    - **No active waiting** the OS can keep activities suspended efficiently avoiding CPU waste in active waiting

Event based waiting

- **Example**: Producer/consumer
  - Hardware solution
    - test_and_set , atomic instruction

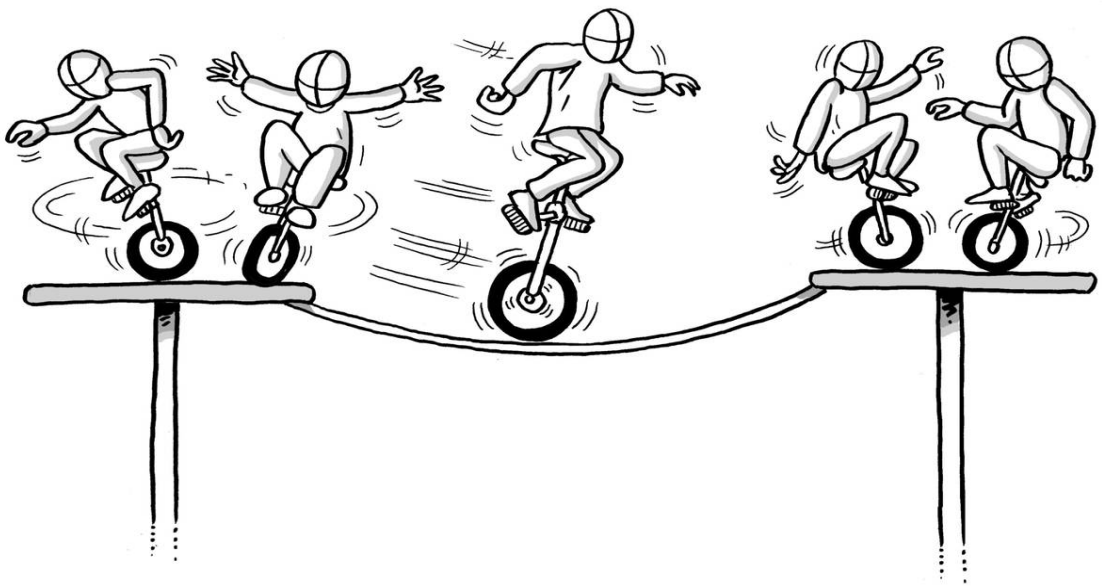
```
#define N  20
int  buffer[N];
int  input, output, counter;
int key = 0;
```

```
void *func_prod(void *p) {
  int item;

  while(1) {
    item = produce();

    while (test_and_set(&key))
      /*empty loop*/ ;

    while (counter == N)
      /*empty loop*/ ;
    buffer[input] = item;
    input = (input + 1) % N;
    counter = counter + 1;

    key = 0;
  }
}
```

```
void *func_cons(void *p) {
  int item;

  while(1) {
    while (test_and_set(&key))
      /*empty loop*/ ;

    while (counter == 0)
      /*empty loop*/ ;
    item = buffer[output];
    output = (output + 1) % N;
    counter = counter - 1;

    key = 0;

    consume(item);
  }
}
```

  - There are problems in this code and it doesn't work
    - ¿Where are the problems?

- The critical section problem
- Software solutions
- Hardware solutions
- Active waiting
- **Exercises**



Exercise 6.1:

Describe if the following code is a good solution to the critical section problem when there are only two threads involved (thread_0 and thread_1)

```
#include <stdio.h>
/** Shared **/
  int flag[2];
```

```
thread_i(void) {
  while ( 1 ) {

   remaining_section;

    flag[i] = 1;
    while(flag[(i+1) % 2]);

    critical_section;

    flag[i] = 0;
  }
}
```

¿Does it comply with the three critical section protocol conditions?

Exercise 6.2:

Compare the three test-and-set based proposed solutions to the critical section problem

```
/* Solution a */
void *thread(void *p) {

  while(1) {
   while (test_and_set(&key));
   /* Critical section */
   key = 0;
   /* Remaining section */
  }
}
```

```
/* Solution b */
void *thread(void *p) {

 while(1) {
 while (test_and_set(&key)) usleep(1000);
   /* Critical section */
   key = 0;
   /* Remaining section */
  }
}
```

```
#include <stdio.h>
/** Shared **/
  int key=0;
```

```
/* Solution c*/
void *thread(void *p) {

 while(1) {
 while (test_and_set(&key)) yield();
   /* Critical section */
   key = 0;
   /* Remaining section */
  }
}
```

Note. yield() call allows leaving the remaining CPU time to others. In this way the thread that calls to yield() goes to READY and frees the CPU, and gives the scheduler the opportunity to select another activity for execution

Exercise 6.3:

The following functions are executed by two concurrent threads inside the same process. Indicate what shared resources appear in the code and what mechanisms are used to avoid race conditions.

```
void *add (void *argument)
{
  int ct,tmp;
  for (ct=0;ct<REPE;ct++)
  {
     while(test_and_set(&key)== 1);

     tmp = V;
     tmp++;
     V = tmp;
     key = 0;
  }
  printf("->ADD (V=%ld)\n",V);
  pthread_exit(0);
}
```

```
void *subtract (void *argument)
{
  int ct,tmp;
  for (ct=0;ct<REPE;ct++)
  {
     while(test_and_set(&key)== 1);

     tmp = V;
     tmp--;
     V = tmp;
     key = 0;
  }

 printf("->SUBSTRACT (V=%ld)\n", V);
  pthread_exit(0);
}
```

Note. The variables and functions not defined inside functions add and subtract are defined as global