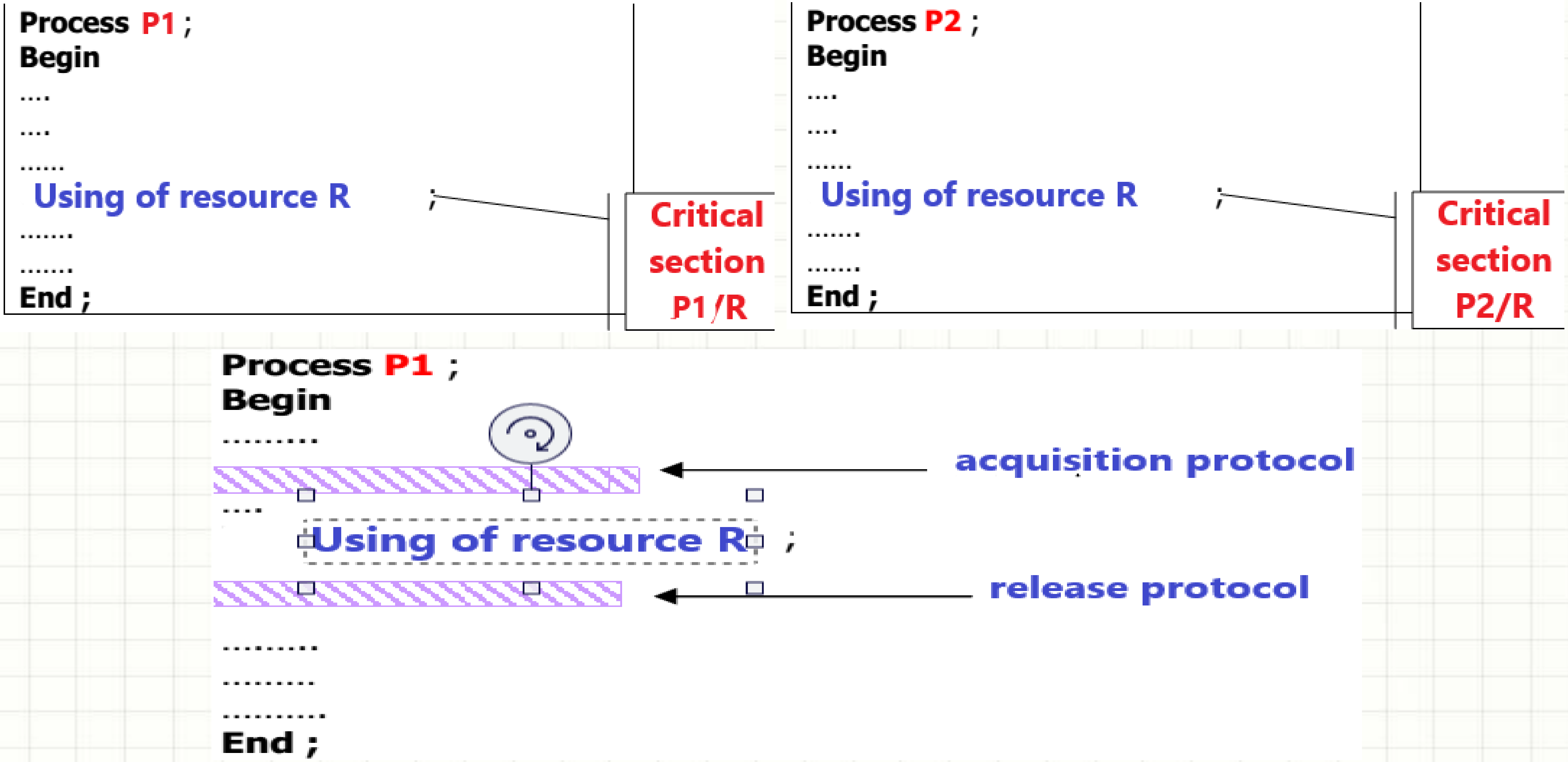
CHAPTER 2

Mutual exclusion between processes

PROBLEM STATEMENT

- **PROBLEM:** To avoid any incorrect use of a **critical resource**, the sequences of instructions that manipulate it (**critical section**) in the different processes must never be executed simultaneously. The critical sections of each process must be executed in **mutual exclusion**.
- **SOLUTION:** A control must take place during the use of this type of resource. This control consists of a protocol that frames the critical sections with special instruction sequences.

PROBLEM STATEMENT



Problem statement

- The sequence of instructions that precedes the critical section is called the **acquisition protocol**. It aims to verify that the process's access to its critical section is possible and also to deny other processes access to their critical sections.
- The sequence of instructions that follows the critical section is called the **release protocol**. It makes access to the critical resource possible.
- The insertion of control protocols is left to the programmer. For this purpose, concurrent programming languages offer adequate concepts such as (Semaphores, monitors).
- Mutual exclusion of processes in their critical sections is only guaranteed if the acquisition and release protocols are used correctly.

Problem statement

- The construction of mutual exclusion protocols is a complex task. In fact, poorly constructed acquisition and release protocols can lead to the following problems:
- **Deadlock:** Deadlock is a situation in which processes cannot progress any further due to a lack of resources.
  - **Each process holds resources that the other process needs.**
- **Starvation:** Starvation is a situation in which some processes are indefinitely blocked while other processes access their critical sections according to their needs.
  - **Some processes never access their critical sections.**

PROBLEM STATEMENT

- Controlling competition between processes is equivalent to finding a solution to the mutual exclusion problem. Any solution to the mutual exclusion problem can be decomposed into three steps:

Entrance

Critical Section

Exit

- There are many different approaches to solving the mutual exclusion problem. However, any solution must guarantee the following four properties, as stated by Di ***DIJKSTRA.***

Dijkstra's Four Properties

- **1. Mutual exclusion:** Only one process can be in its critical section at a time.
- **2.Progress:** If no process is in its critical section, a process waiting to enter its critical section must be able to do so after a finite amount of time. In other words, the critical section is always reachable.
- **3.Bounded waiting:** If a process is blocked outside of a critical section, this blocking must not prevent the entry of another process into its critical section.
- **4.Deadlock freedom:** There will never be a situation in which two or more processes are waiting for each other to release a resource.

ALGORITHMIC SOLUTION TO THE MUTUAL EXCLUSION PROBLEM

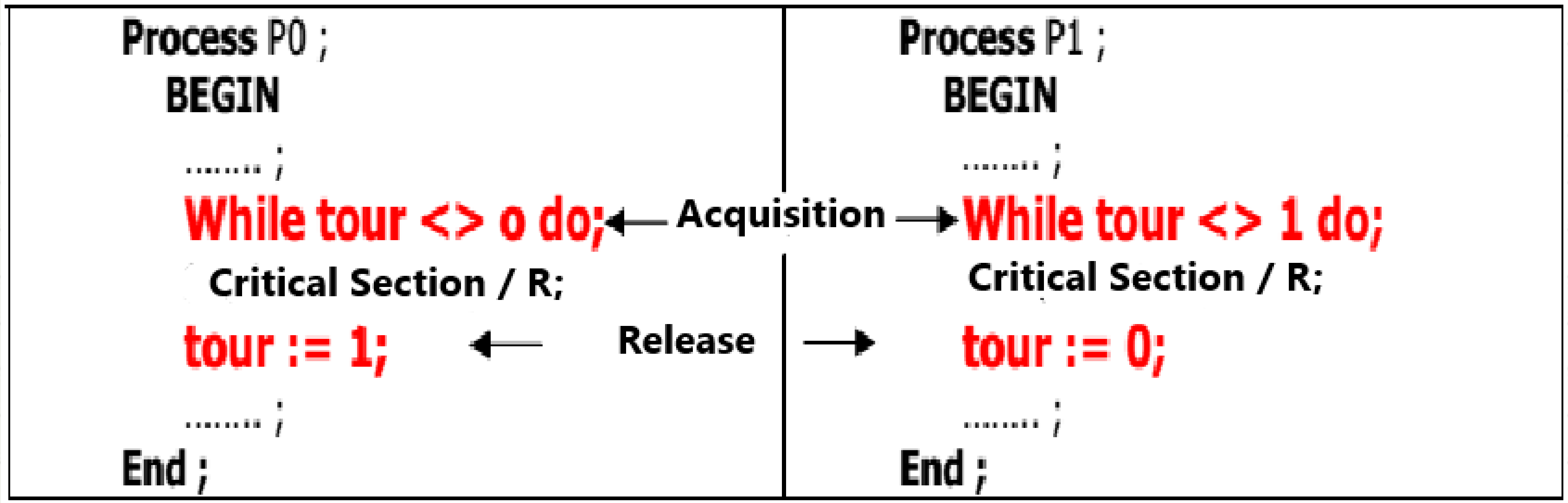
- HYPOTHÈSE. The hypothesis states that we are considering two processes, **P0** and **P1**, which compete for a critical resource **R**. The two processes are defined by the following program:

- The program uses the **Parbegin** and **Parend** instructions to execute the two processes in parallel.

```
1.  Program gestprocess;
    1.  {Declaration of common variables to PO and P1}

    1.  Process PO;
        1.  {Declaration of local variables to PO}
    2.  BEGIN
    3.      // Code executed before entering the critical section
    4.      ...
    5.      Section critique/R;
    6.      ...
    7.      // Code executed after exiting the critical section
    8.  End;

    9.  Process P1;
        1.  {Declaration of local variables to P1}
    10. BEGIN
    11.     // Code executed before entering the critical section
    12.     ...
    13.     Section critique/R;
    14.     ...
    15.     // Code executed after exiting the critical section
    16. End;

2.  BEGIN
    1.  {Initialization of common variables}
    2.  PARBEGIN
    3.      PO;     P1;
    4.  PAREND;
3.  End;
```

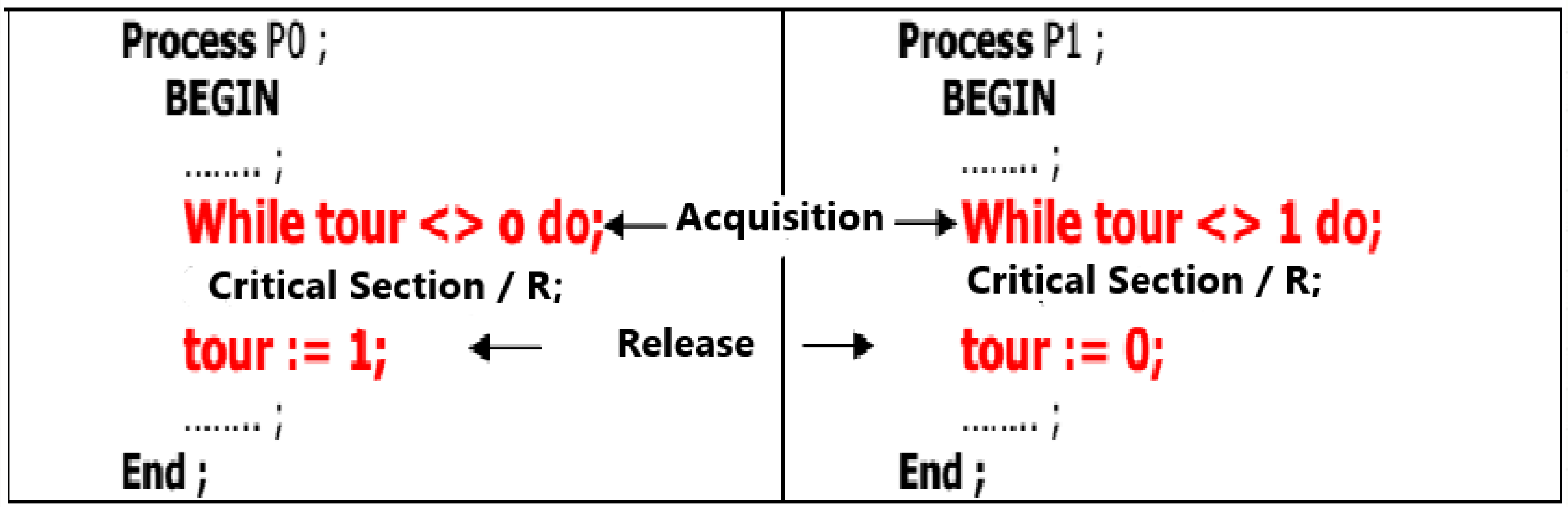
ALGORITHMIC SOLUTION TO THE MUTUAL EXCLUSION PROBLEM

- FIRST SOLUTION: ALTERNATION
- This solution uses a shared variable called Tour to determine which process is allowed to enter the critical section. The value of Tour is either 0 or 1, and the process with the corresponding number is allowed to enter the critical section.
  - **Var** Tour: (0,1)
  - Tour is initialized to 0 or 1.



Algorithmic solution to the mutual exclusion problem

- First Solution: Alternation
- T



- **Note:** There is an **active wait** in the while loop, where the process repeats the same actions without results.
- **Critique:** The second property of Dijkstra is not satisfied.
- **Conclusion:** False solution.

ALGORITHMIC SOLUTION TO THE MUTUAL EXCLUSION PROBLEM

- SECOND SOLUTION: PROCESS REQUESTS
- The first solution does not take into account the wishes of the processes, but only the permission granted to one or the other. To avoid a process remaining waiting for another that does not want to access its critical section, the shared variable Tour can be replaced by a table of indicators called drapeau. The process **Pi** that wants to access its critical section sets its flag to the value **True**.

  – **Var** Drapeau : array[0..1] of boolean ;
  – drapeau is initialized to false for all processes.(*for i:=0 to 1 do drapeau [i]:= false;*)

Solution algorithmique au problème d`exclusion mutuelle

- **Second solution: Process requests**

Process P0	Process P1
Process P0 ;	Process P1 ;
BEGIN	BEGIN
…….. ;	…….. ;
While Drapeau[1] do;	**While Drapeau[0] do;**
drapeau [0]:= true;	**drapeau [1]:= true;**
Critical Section /R ;	Critical Section /R ;
drapeau [o]:= false;	**drapeau [1]:= false;**
…….. ;	…….. ;
End ;	End ;

- **CRITIQUE**. The first property of Dijkstra is not satisfied.
- **Conclusion**. False solution

Algorithmic solution to the mutual exclusion problem

- <u>Third solution:</u>
- In the second solution, processes **P0** and **P1** can simultaneously access the table of indicators. To remedy this, the two instructions (test and assignment) are swapped. In this case, the table of indicators no longer has the same meaning.

ALGORITHMIC SOLUTION TO THE MUTUAL EXCLUSION PROBLEM

- <u>THIRD SOLUTION:</u>

Process P0 ;
BEGIN
.... ;
drapeau [0]:= true;
While Drapeau[1] do;
Critical Section /R ;
drapeau [o]:= false;
.... ;
End ;
	Process P1 ;
	BEGIN
	.... ;
	drapeau [1]:= true;
	While Drapeau[0] do;
	Critical Section /R ;
	drapeau [1]:= false;
	.... ;
	End ;

- **Critique:** The second property of Dijkstra is not satisfied.
- **If both processes execute the first assignment simultaneously, they will then mutually block each other, making the critical section unreachable.
- **Conclusion:** False solution.

ALGORITHMIC SOLUTION TO THE MUTUAL EXCLUSION PROBLEM

- **FOURTH SOLUTION: PETERSON'S ALGORITHM**
- In this solution, proposed by Peterson, the first and third propositions are combined. The shared variables are drapeau and Tour.
  - **Vars:**
    - drapeau[0] and drapeau[1] are initialized to false.
    - Tour is initialized to a value of 0 or 1.

Algorithmic solution to the mutual exclusion problem

- **Fourth solution: Peterson's algorithm**

Process P0 ;	Process P1 ;
BEGIN	BEGIN
.... ;	.... ;
drapeau [0]:= true;	drapeau [1]:= true;
tour:= 1;	tour:= 0;
While (Drapeau[1]) and (tour=1) do;	While (Drapeau[0]) and (tour=0) do;
Critical Section /R;	Critical Section /R;
drapeau [o]:= false;	drapeau [1]:= false;
.... ;	.... ;
End ;	End ;

- **CRITIQUE**. Correct solution because the four Dijkstra properties are satisfied.

HARDWARE SOLUTION TO THE MUTUAL EXCLUSION PROBLEM

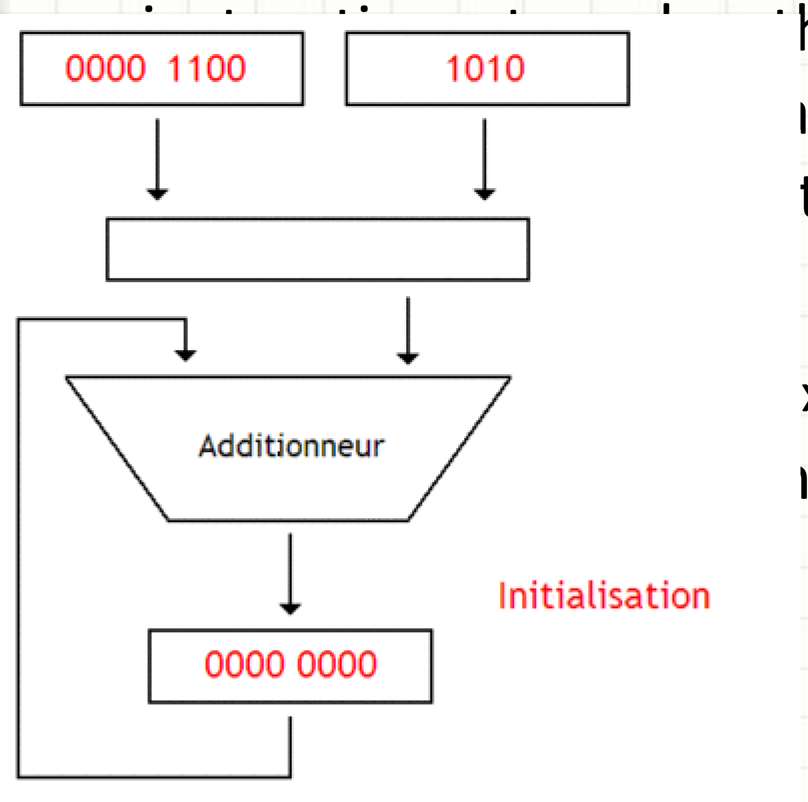
- **A. CASE OF A SINGLE-PROCESSOR MACHINE.** Critical sections must be made indivisible by masking interrupts during their execution.

```
Process Pi
    BEGIN
        ...
         Disable interrupts
          Critical section
         Enable interrupts
        ...
    END
```

- **NB**: *Masking and unmasking interrupts can quickly become penalizing for the operating system.*

4. Solution matérielle au problème d`exclusion mutuelle

- **B. Case of a multiprocessor machine.** Machines offer special atomic … he mutual exclusion problem for a given variable. … rantee the test and update of a variable or the … ts of two variables in a single memory cycle.



… » instruction, also known as TAS for the English … d Set, can be assimilated to the following function:

```
Function TAS (Var cible : boolean) : Boolean;
Begin
    Tas:=cible;
     Cible:=true;
End;
```

4. Solution matérielle au problème d`exclusion mutuelle

- B. Case of a multiprocessor machine.
  - The **SWAP** instruction
  - It can be assimilated to the following procedure:

```
Procedure SWAP (Var R,M :boolean);
Var temp: Boolean;
Begin
Temp:=R;
R:=M;
M:=temp;
End;
```

- The TAS and Swap instructions are executed atomically. Therefore, if two processes execute these instructions simultaneously, one of them will be blocked.

- **4.1 Solution to the mutual exclusion problem using TAS.**.
  - Let ***lock*** be a common variable to the processes and it is concerned by the critical section.
    - **Var** lock : boolean ;
    - lock := false ;

```
Process Pi
    BEGIN
        ...
         While TAS(lock)
          Critical section
         lock := False;
        ...
    END
```

4.2 Solution to the mutual exclusion problem using SWAP.

- **Var** verrou : boolean ;
- Verrou :=false ;

```
Process Pi;
Var key: Boolean;
Begin
....
key := true;
Repeat SWAP (lock, key); until key = false;
Critical section;
lock := false;
...
End;
```