
Global Environment Model

MUTUAL EXCLUSION PROBLEM

The operations used
by processes to access
to common resources
(critical sections) must
be mutually exclusive
in time

- No assumptions should be made about relative process speed
- A, B critical sections

Example

→ During their execution P1 and P2 access a common variable count(i.v.=0) and increment it by 1

Example

→ When P1 and P2 terminate their executions count must be incremented by 2
count=2

Note that the increment of count, when P1 execute it, may be implemented in machine language as:

```
reg1 = count
reg1 = reg1 +1
count = reg1
```

Where reg1 is a local CPU register used by P1

Similarly the increment of count, when P2 executes it, may be implemented in machine language as:

```
reg2 = count
reg2 = reg2 +1
count = reg2
```

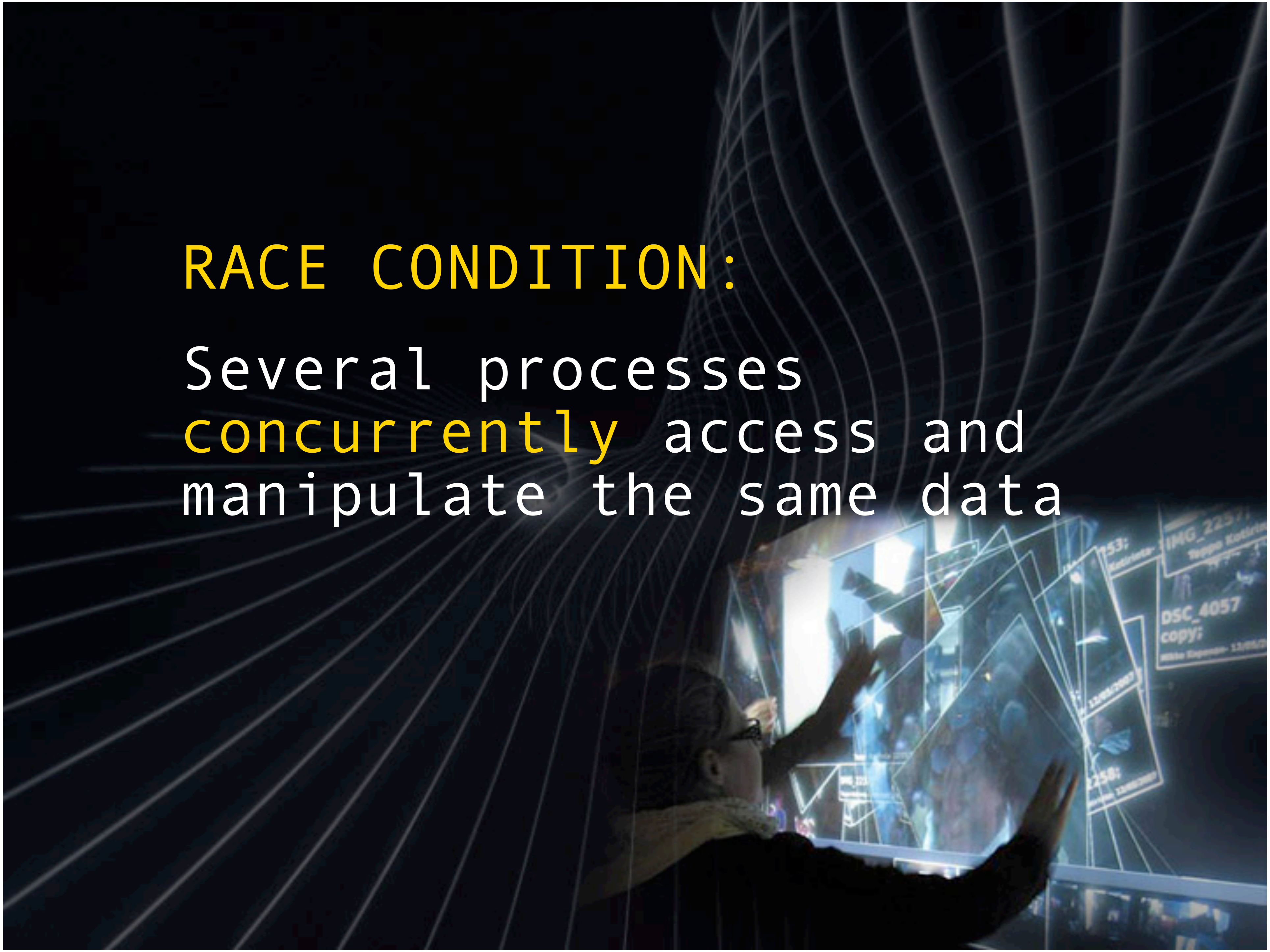
where reg2 is a local CPU register used by P2

The concurrent execution
of the statement

`++count`

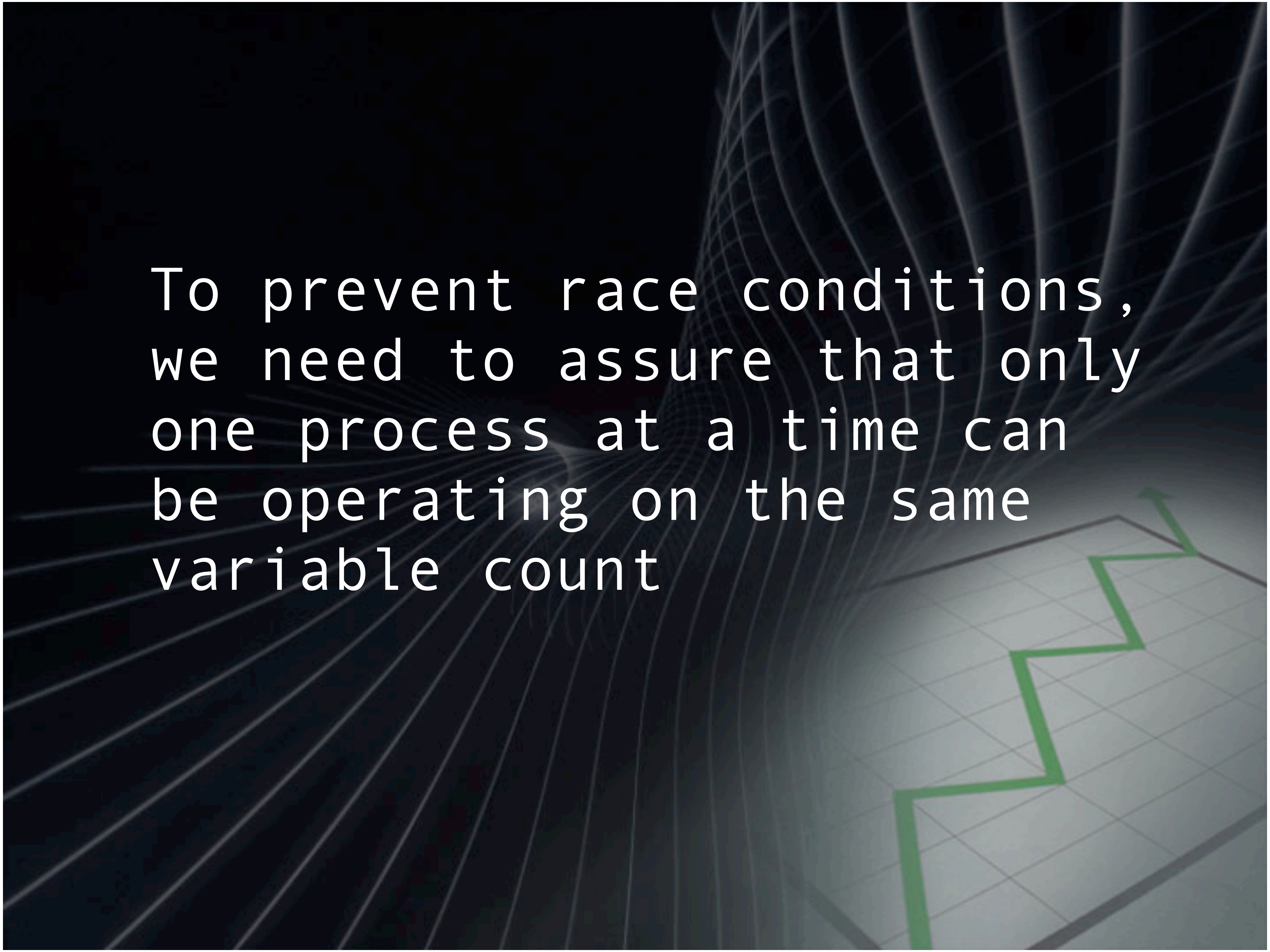
is equivalent to
a sequential execution
where the statements
can be interleaved in any
arbitrary order

```
T0:  reg1=count     (reg1=0)      (P1)
T1:  reg2=count     (reg2=0)      (P2)
T2:  reg2=reg2+1    (reg2=1)      (P2)
T3:  count=reg2     (count=1)     (P2)
T4:  reg1=reg1+1    (reg1=1)      (P1)
T5:  count=reg1     (count=1)     (P1)
```


RACE CONDITION:
Several processes concurrently access and manipulate the same data

RACE CONDITION:

The outcome of the execution depends on the particular order in wich the access takes place


To prevent race conditions, we need to assure that only one process at a time can be operating on the same variable count

To grant that invariant,
we need some form
of process synchronization

SOLUTION TO THE MUTUAL EXCLUSION PROBLEM

busy = 1 the resource is busy
busy = 0 the resource is free

P_1

```
while (busy ==1);
    busy =1
    < critical section A>;
busy =0;
```

P_2

```
while (busy ==1);
      busy =1
      < critical section B>;
busy =0;
```

T_0: P_1 executes while and busy=0
T_1: P_2 executes while and busy=0
T_2: P_1 set busy=1 and accesses to A
T_3: P_2 set busy=1 and accesses to B

Both processes have simultaneous access to their critical section

TSL (Test and Set Lock)

Instruction that reads and modifies the contents of a memory word in an indivisible way

The CPU, while executing the TSL instruction, locks the memory bus to prohibit other CPUs from accessing memory until the instruction is completed

TSL R, x:

It reads the content of x into the register R and then stores a non zero value at that memory address

TSL R, x:

The operations of reading a word and storing into it are garantee to be indivisible by the hardware level

lock(x) , unlock(x):

lock(x):

```
TSL register, x     (copy x to register
                     and set x=1)
CMP register,0      (was x zero?)
JNE lock            (if non zero the
                     cycle is restarted)
RET                 (return to caller;
                     critical region
                     entered)
```

```
unlock(x):

MOVE x, 0  (store a 0 in x)
RET        (return to caller)
```

Soluzione con lock(x) e unlock(x):

P_1

```
lock(x);
<sezione critica A>;
unlock (x);
```

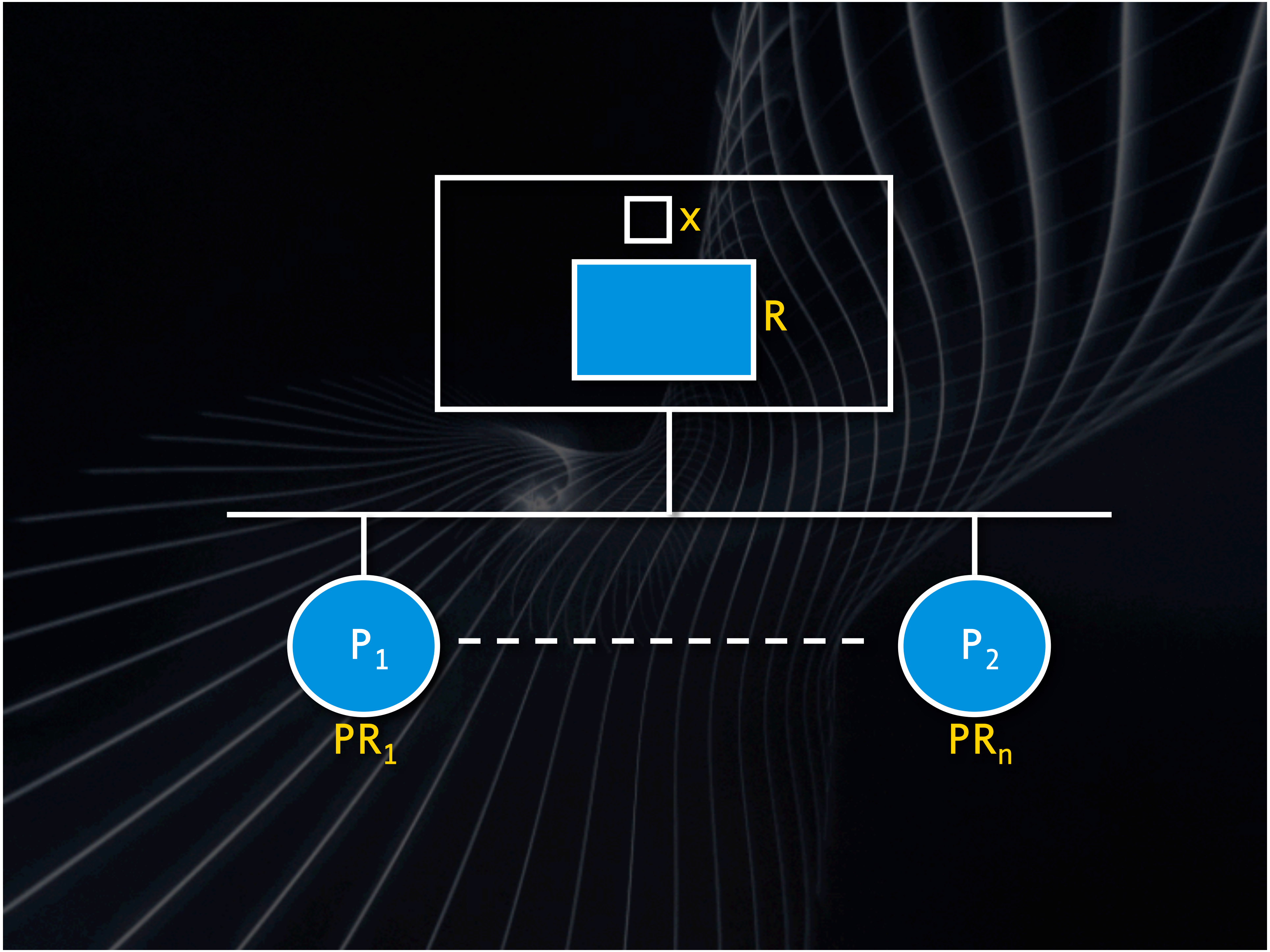
Soluzione con lock(x) e unlock(x):

P_2

```
lock(x);
<sezione critica B>;
unlock (x)  ;
```

SOLUTION PROPERTIES

- busy waiting
- multiprocessor systems
- “very shorts” critical sections



x
R
P_1
P_2
PR_1
PR_n

SEMAPHORES

A semaphor s is a

integer non negative variable, initialized to a nonnegative value

s.value

s is associated with a
waiting list, in wich
are linked the PCBs
of processes blocked on s.

s.queue

A semaphore s is accessed only via two standard operations

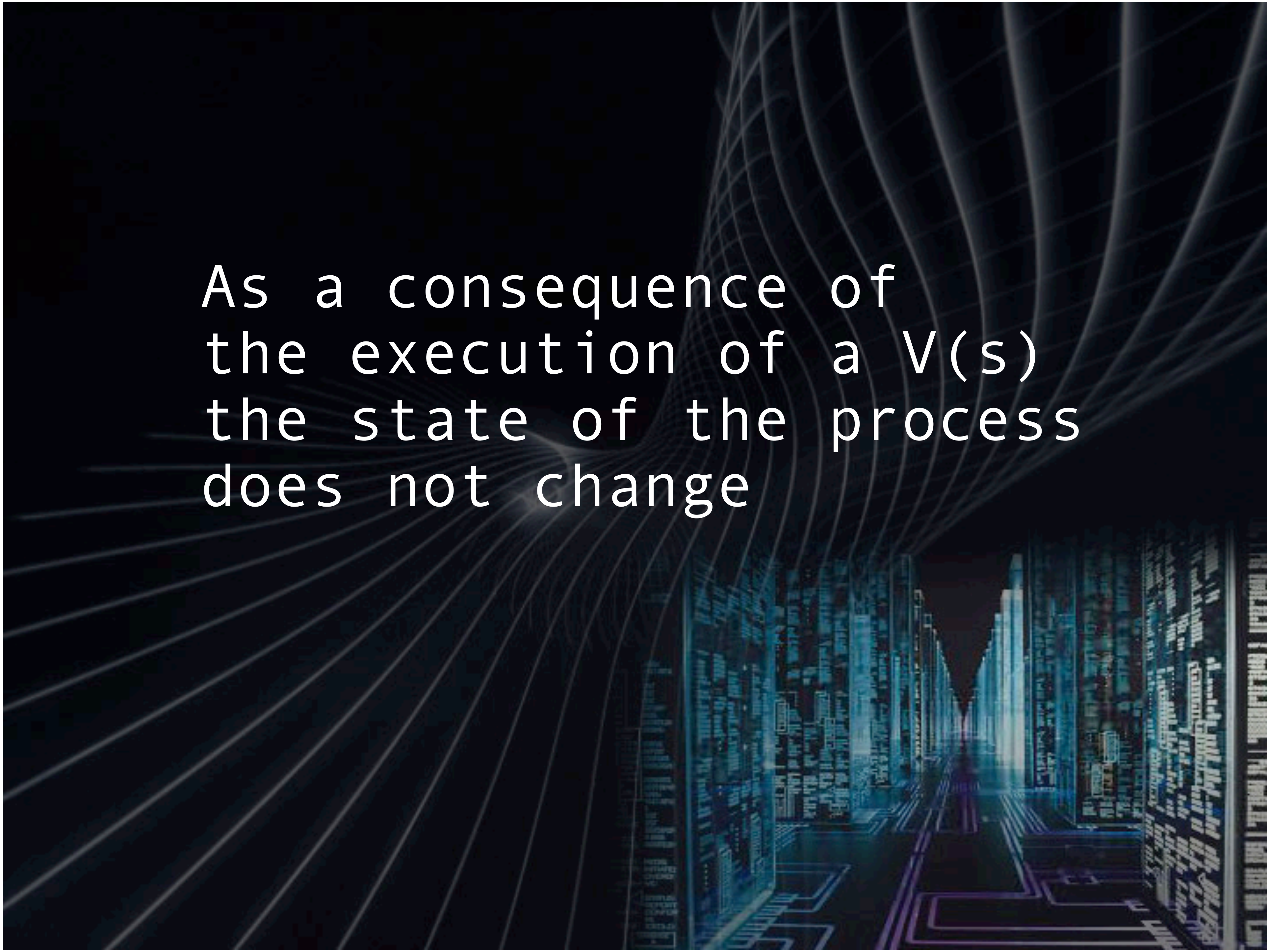
- P (s)
- V (s)

P(s) If s.value>0 the process continues its execution, if s.value=0 the process is blocked in the s.queue

V(s) A process in the s.queue is waked and extracted; its state is modified from blocked to ready

```
void P(s)
{
   if (s.value==0)
      <the process is blocked and its
      PCB is inserted in the s.queue>;
   else s.value= s.value-1;
}
```

```
void V(s)
{
   if ( < there is at least one
   process in s.queue>)
      <the PCB of the first of these
      processes is taken out from
      s.queue and its state is
      modified from blocked to ready>;
   else s.value = s.value + 1;
}
```


As a consequence of
the execution of a V(s)
the state of the process
does not change

The decision of wich process is to be extracted from the s.queue is taken with a FIFO policy

MUTUAL EXCLUSION

mutex: semaphore associated to the shared resource (i. v. mutex=1)

```
P (mutex)
<sezione critica>
V (mutex)
```

P1

P(mutex)

<A>

V (mutex)

P2

P(mutex)

<B>

V (mutex)

P3

P (mutex)

<C>

V (mutex)

P and V must be indivisible operations

The modification of the value of the semaphore and the possible blocking or waking up of a process must be indivisible operations

P,V: critical sections

with reference to the
data structure
represented by mutex

(mutex.value, mutex.queue)

INDIVISIBILITY OF P AND V BY

Disabling interrupts

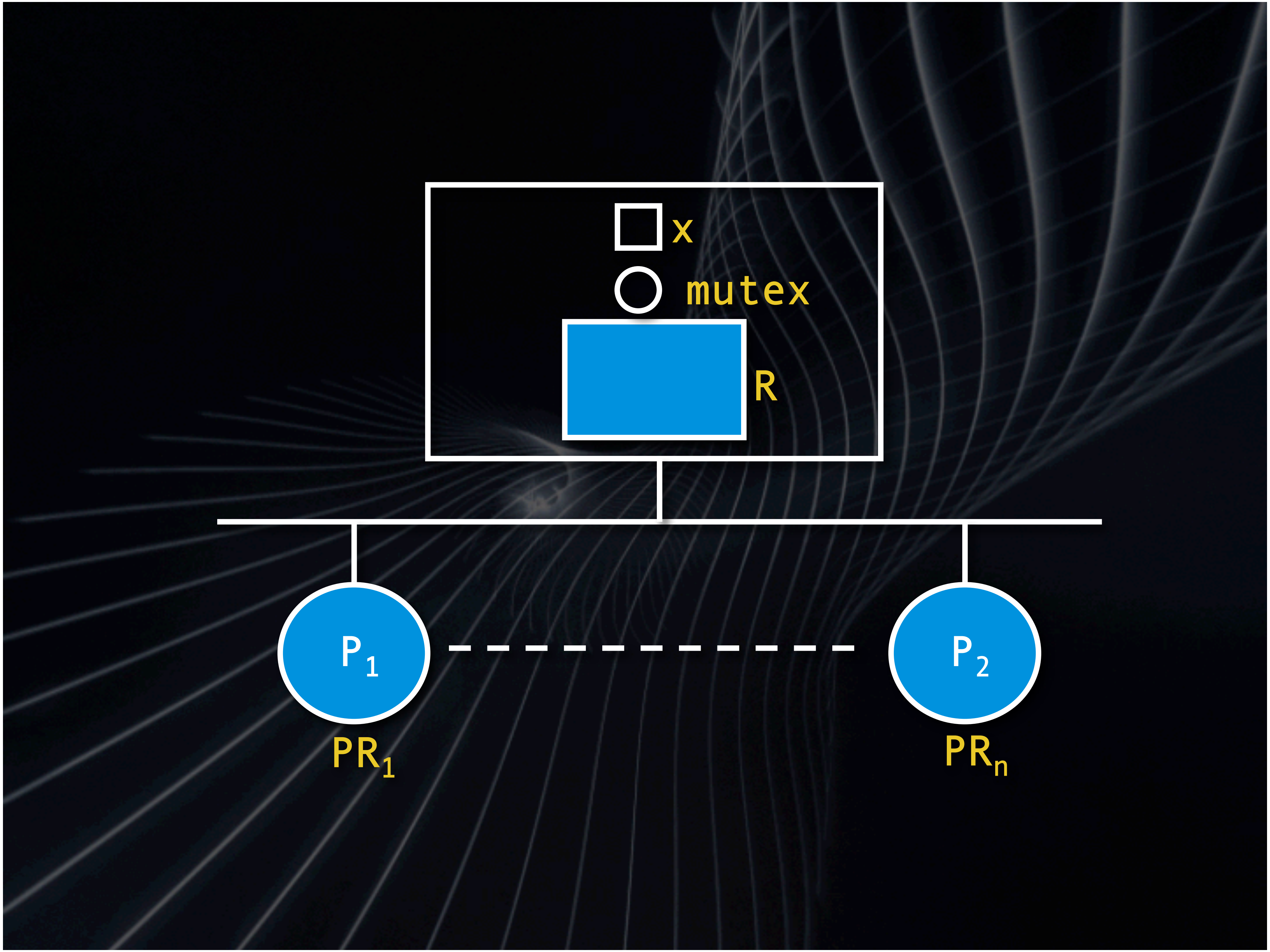
(when P,V are executing on the same processor)

INDIVISIBILITY OF P AND V BY

Using lock(x), unlock(x)(when P,V are executing on different processors)

indivisible P,V

```
lock(x);
P (mutex);
unlock(x);
    <sezione critica>;
lock(x);
V (mutex);
unlock(x);
```


x
mutex
R
P1
P2
PR1
PRn