departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Solving Mutual Exclusion (1)

Concurrency and Parallelism — 2017-18

Master in Computer Science

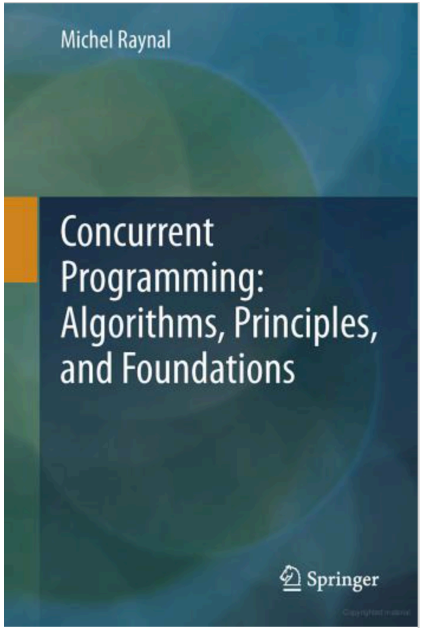
(Mestrado Integrado em Eng. Informática)

Joao Lourenço <joao.lourenco@fct.unl.pt>

Summary

- **Solving Mutual Exclusion**
  - Mutex based on atomic read-write registers
  - Concurrency-abortable operation

- **Reading list:**
  - **Chapter 2** of the book
    Raynal M.;
    Concurrent Programming: Algorithms, Principles, and Foundations;
    Springer-Verlag Berlin Heidelberg (2013);
    ISBN: 978-3-642-32026-2

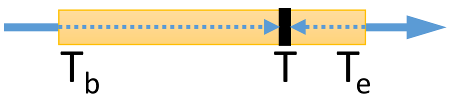


Mutex Based on Atomic Read/Write Registers

- A *register R* can be accessed by two base operations:
- ***R*.read()**, which returns the value of *R* (also denoted **x** ← ***R*** where x is a local variable of the invoking process); and
- ***R*.write(v)**, which writes a new value into *R* (also denoted ***R*** ← **v**, where v is the value to be written into *R*).

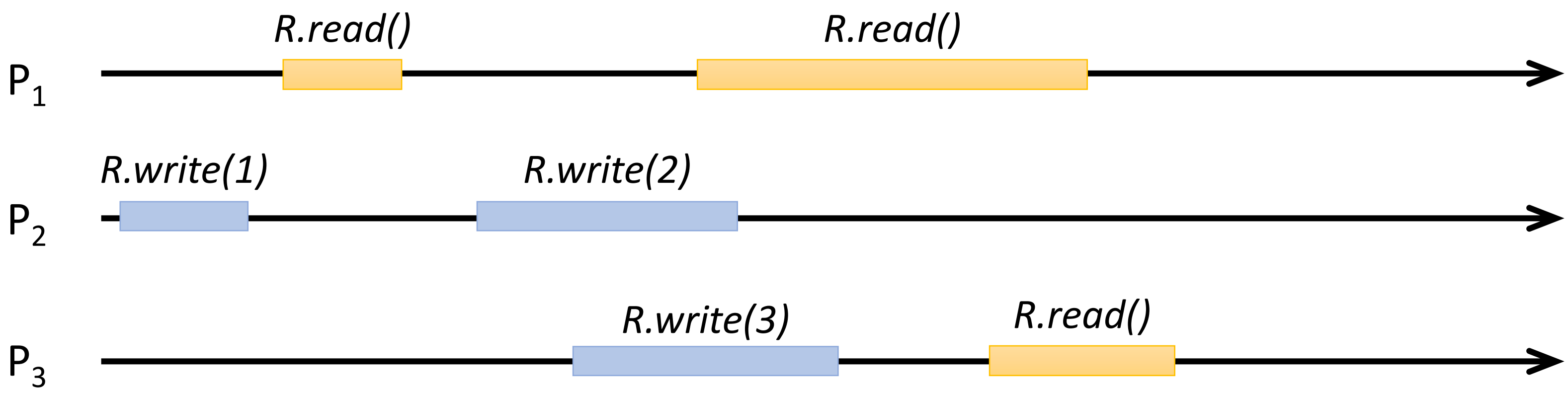
Mutex Based on Atomic Read/Write Registers

- An *atomic* shared register satisfies the following properties:
- Each invocation op of a read or write operation:
  - Appears as if it was executed at a single point *T(op)* of the time line;
  - *T(op)* is such that $T_b(op) \leq T(op) \leq T_e(op)$, where $T_b(op)$ and $T_e(op)$ denote the time at which the operation *op* started and finished, respectively;
  - For any two operation invocations *op1* and *op2*: $(op1 \neq op2) \Rightarrow T(op1) \neq T(op2)$.

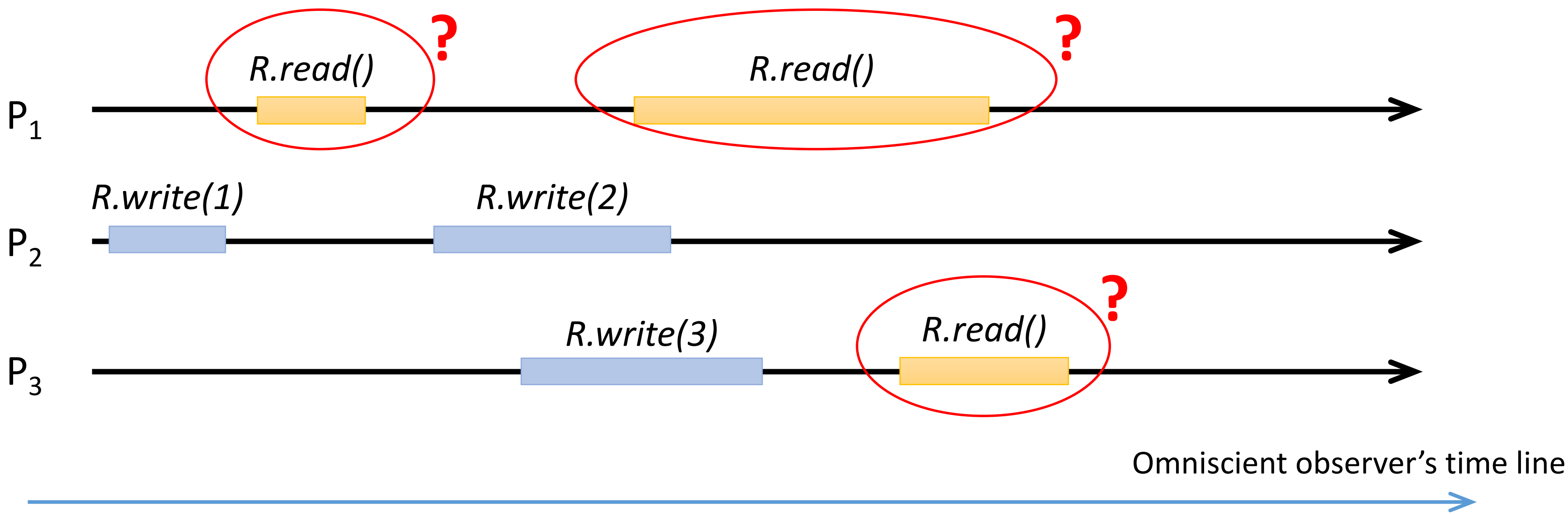


- Each read invocation:
  - Returns the value written by the closest preceding write invocation in the sequence defined by the *T(…)* instants associated with the operation invocations (or the initial value of the register if there is no preceding write operation).

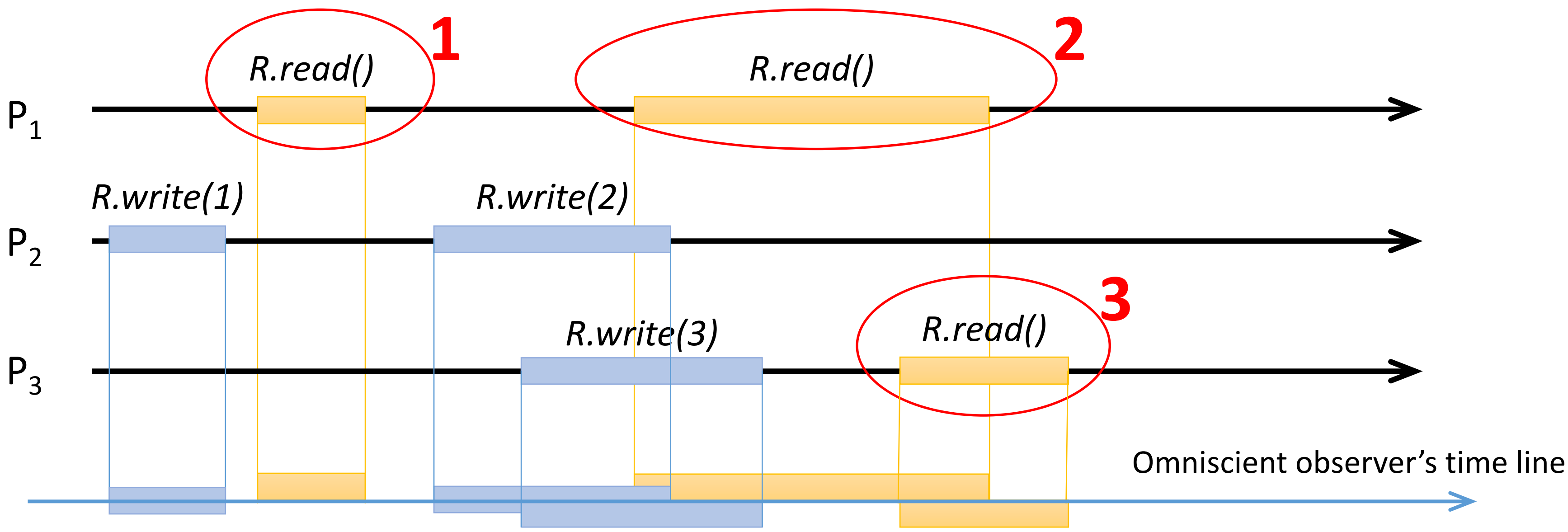
Mutex Based on Atomic Read/Write Registers



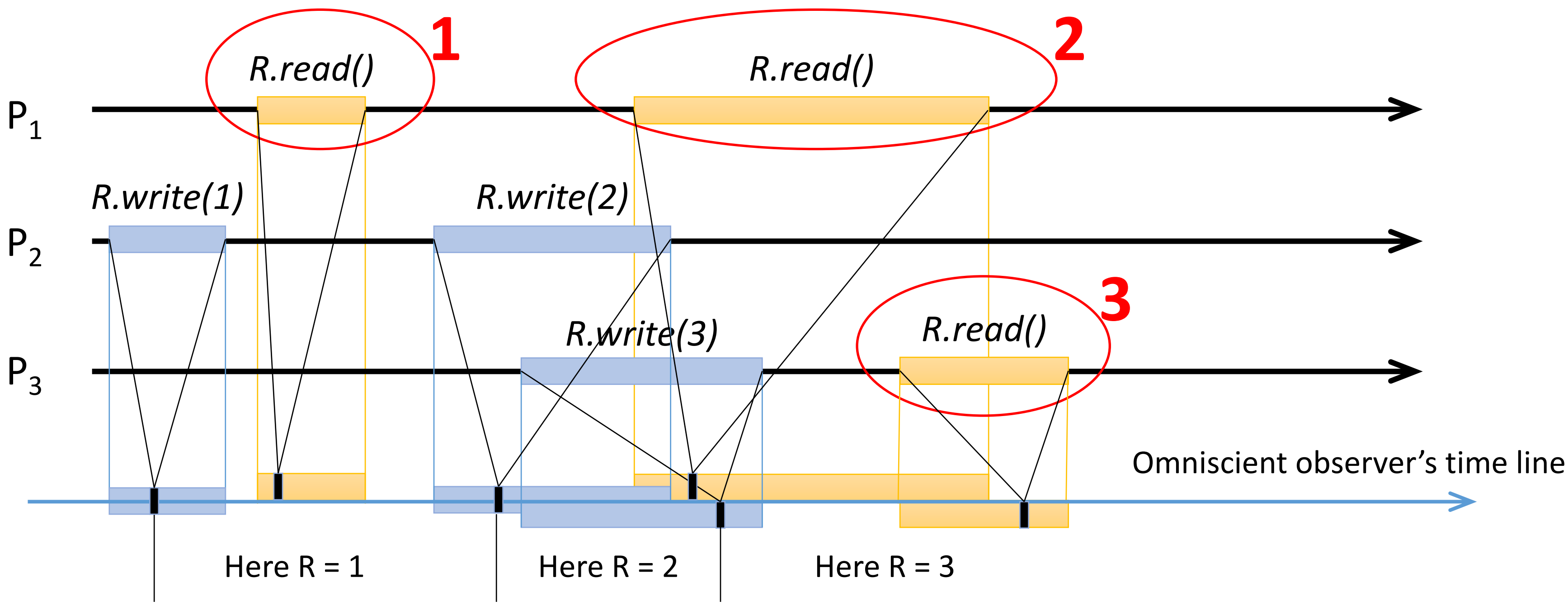
Mutex Based on Atomic Read/Write Registers



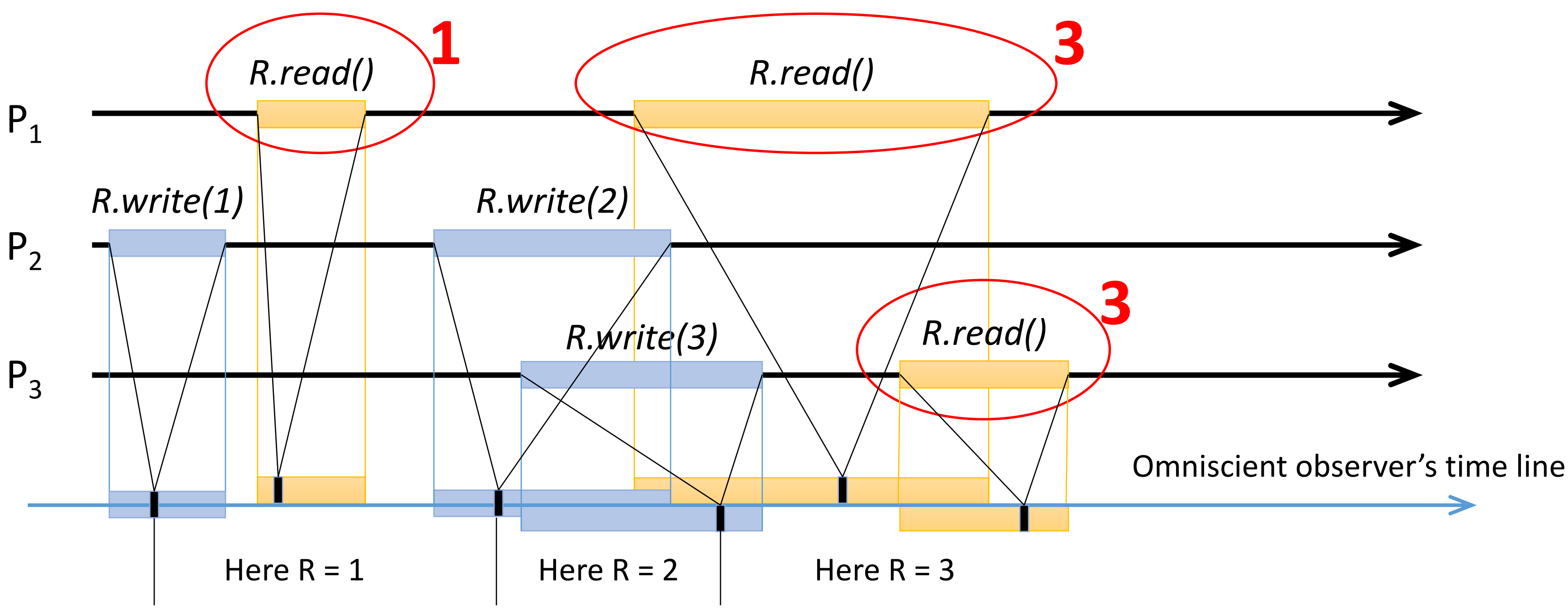
Mutex Based on Atomic Read/Write Registers



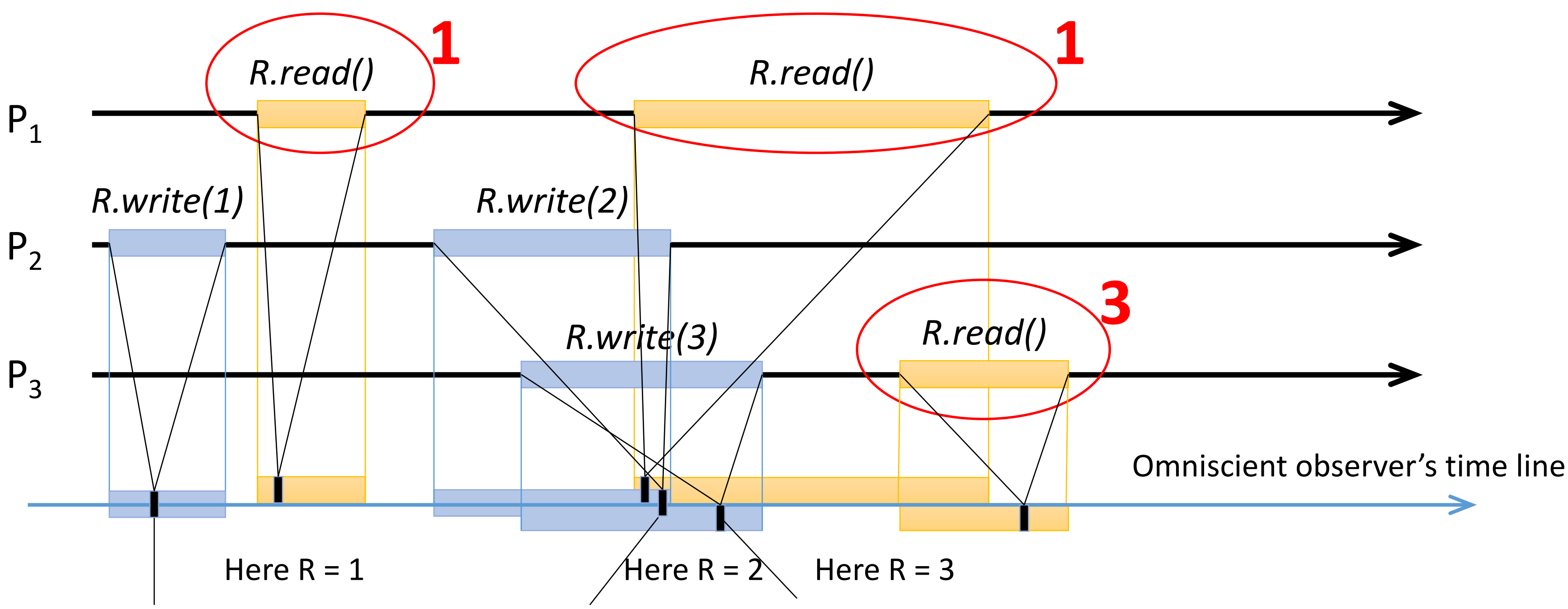
Mutex Based on Atomic Read/Write Registers



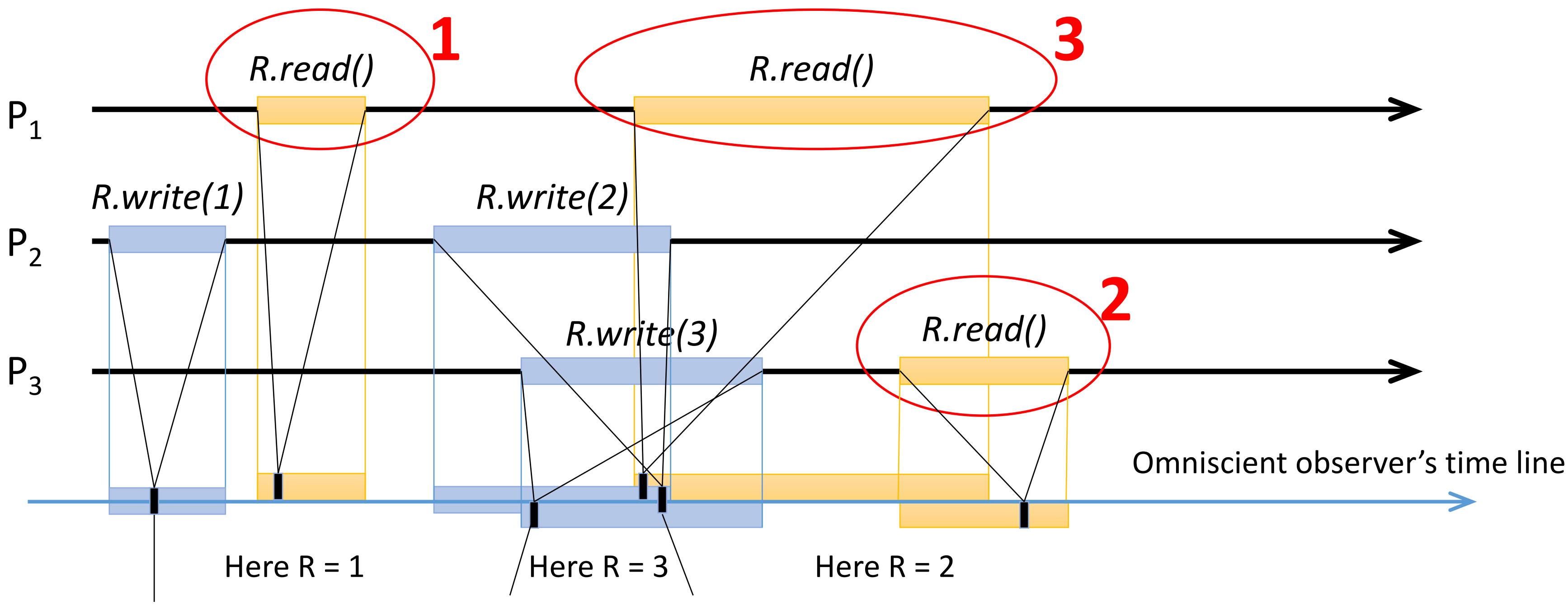
Mutex Based on Atomic Read/Write Registers



Mutex Based on Atomic Read/Write Registers



Mutex Based on Atomic Read/Write Registers



Mutex for Two Processes: An Incremental Construction

operation acquire_mutex$_1(i)$ **is**
$AFTER\_YOU \leftarrow i$; **wait** $(AFTER\_YOU \neq i)$; return()
end operation.

operation release_mutex$_1(i)$ **is** return() **end operation**.

Must have contention to have progress

May cause deadlock (by starvation)

G.L. Peterson (1981)

✓ mutual exclusion
X progress

Mutex for Two Processes: An Incremental Construction

operation acquire_mutex$_2(i)$ **is**
$FLAG[i] \leftarrow up$; **wait** $(FLAG[j] = down)$; return()
end operation.

operation release_mutex$_2(i)$ **is** $FLAG[i] \leftarrow down$; return() **end operation**.

May cause deadlock

G.L. Peterson (1981)

✓ mutual exclusion
X progress

Mutex for Two Processes: An Incremental Construction

?

while $(FLAG[j] = up)$ **do**
    $FLAG[i] \leftarrow down;$
    p_i delays itself for an arbitrary period of time;
    $FLAG[i] \leftarrow up$
end while.

operation acquire_mutex$_2(i)$ **is**
    $FLAG[i] \leftarrow up;$ **wait** $(FLAG[j] = down);$ return()
end operation.

operation release_mutex$_2(i)$ **is** $FLAG[i] \leftarrow down;$ return() **end operation**.

May cause livelock

G.L. Peterson (1981)

✓ mutual exclusion
X progress

Mutex for Two Processes: An Incremental Construction

operation acquire_mutex(i) **is**
$FLAG[i] \leftarrow up;$
$AFTER\_YOU \leftarrow i;$
wait $((FLAG[j] = down) \vee (AFTER\_YOU \neq i));$
return()
end operation.

operation release_mutex(i) **is** $FLAG[i] \leftarrow down;$ return() **end operation**.

Only works for two processes!
Can we make it work for more?

G.L. Peterson (1981)

✓mutual exclusion
✓progress

Mutex for n Processes: Generalizing the Previous Two-Process Algorithm

operation acquire_mutex(i) **is**
(1) **for** ℓ **from** 1 **to** $(n-1)$ **do**
(2) $FLAG\_LEVEL[i] \leftarrow \ell;$
(3) $AFTER\_YOU[\ell] \leftarrow i;$
(4) **wait** $(\forall\, k \neq i:\ FLAG\_LEVEL[k] < \ell) \vee (AFTER\_YOU[\ell] \neq i)$
(5) **end for**;
(6) return()
end operation.

operation release_mutex(i) **is** $FLAG\_LEVEL[i] \leftarrow 0$; return() **end operation**.

✓mutual exclusion
✓progress

G.L. Peterson (1981)

p_i is allowed to progress to level 'l+1' if, from its point of view,

- Either all the other processes are at a lower level (i.e., $\forall\ k \neq i$:*FLAG_LEVEL* [*k*] < l).
- Or it was not the last one entering level 'l' (i.e., *AFTER_YOU*[l] ≠ *i*).

The END