Optimal Memory-Anonymous Symmetric Deadlock-Free Mutual Exclusion

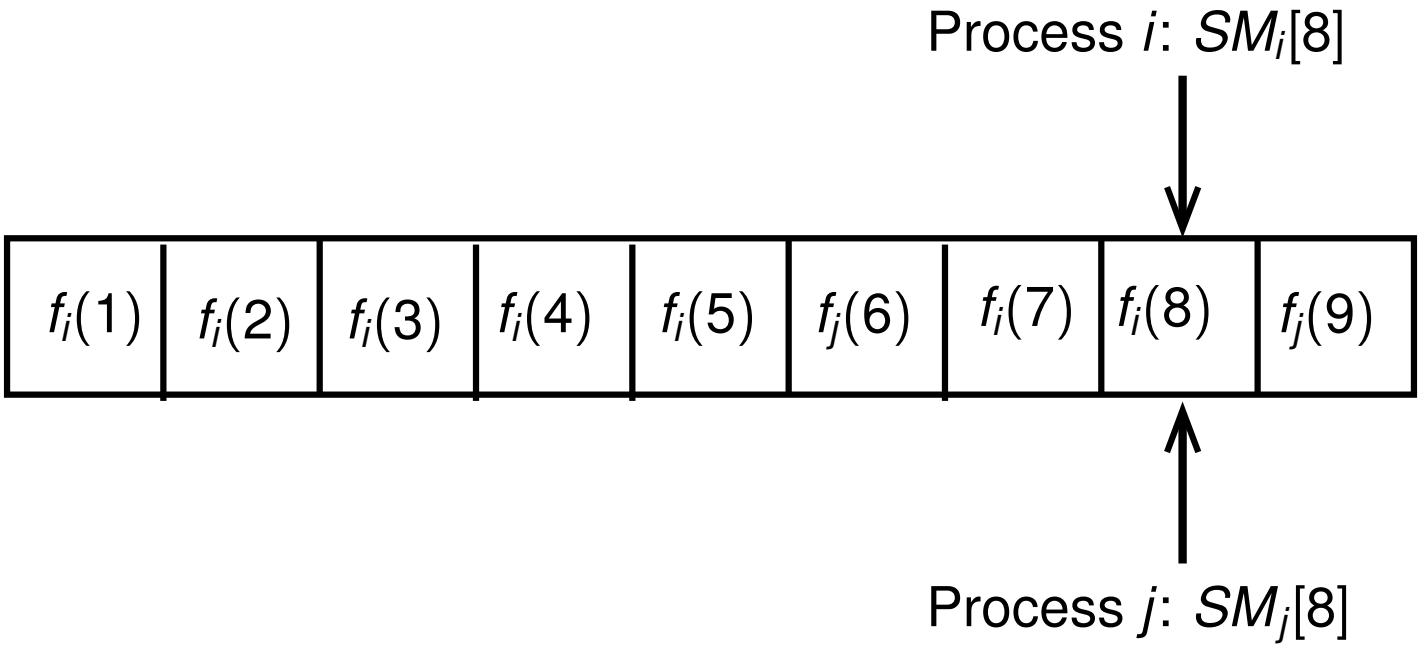
ZAHRA AGHAZADEH DAMIEN IMBS MICHEL RAYNAL
GADI TAUBENFELD PHILIPP WOELFEL

Highlights

- Memory anonymity
- Process symmetry
- An interesting computability condition for Mutual Exclusion

Memory anonymity

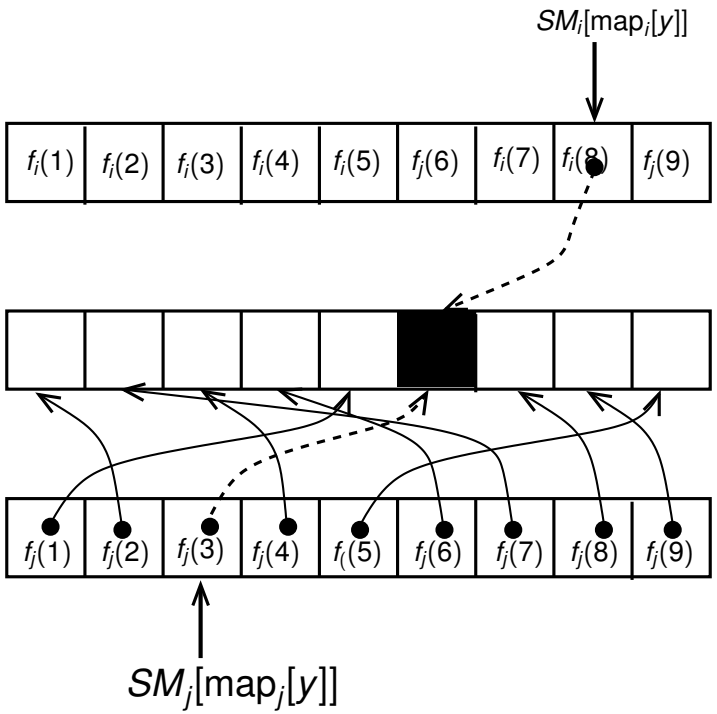
In classical shared memory models,
a priori agreement on the indexes of shared registers:



Memory anonymity

Anonymous memory:

Each process has its own map to access registers:



Other processes have other maps

- Used implicitly in the early 80s (Rabin 82)
- Conceptualized and formalized recently by G. Taubenfeld (PODC'17)

Memory anonymity

An adversarial view

The adversary:

- shuffles the memory map of each process

identifiers for an external observer	identifiers for process p	identifiers for process q
$R[1]$	$R[2]$	$R[3]$
$R[2]$	$R[3]$	$R[1]$
$R[3]$	$R[1]$	$R[2]$
permutation	$2, 3, 1$	$3, 1, 2$

- prohibits agreement on register indexes
  (not true for all combinations of numbers of registers and processes, Godard-I.-Raynal-Taubenfeld, SIROCCO 2019)

Process identities

For n processes:

- The "classical" model
  - Unique ids from 1 to n
- The model for comparison-based algorithms
  - Unique ids from a huge namespace of size $M >> n$
- Process anonymity
  - No ids, no way to identify individual processes
- Process symmetry

Process symmetry

- Processes have unique identities...
  ... but they can only be compared by equality

  No a priori agreement on a total order on identities

  $\Rightarrow$ Only test allowed on ids: equality
  (no $<, \leq, >$ or $\geq$)

- Processes also have the same code
  (otherwise the id could be embedded in the code)

Deadlock-free mutual exclusion

- Safety: mutual exclusion
  At any given time, at most one process
  is in the Critical Section

- Liveness: Deadlock-freedom
  At any time, if a process wants to enter the Critical Section,
  at least one process (not necessarily the same) will enter

Model

- *n* asynchronous symmetric processes
  (ids cannot be compared)
- *m* anonymous registers
  (no a priori agreement on the indexes)
- A process
  - knows its identity
  - knows *n*
  - knows all identities are different
  - does not know the other identities

Operations on registers: either

- Read and Write, or
- Read-Modify-Write (RMW), e.g. Compare&Swap

Symmetric deadlock-free mutual exclusion with an anonymous memory

Previous results: G. Taubenfeld, PODC'17

- Read-Write, for $n = 2$: m odd necessary and sufficient
  Let $M(n) = \{m > n \mid \forall\, \ell : 1 < \ell \leq n : \ \gcd(\ell, m) = 1\}$.
- Read-Write, for $n > 2$: $m \in M(n)$ necessary

In this work:

- Read-Write, for $n \geq 2$: $m \in M(n)$ necessary and sufficient
  - New algorithm
- Read-Modify-Write, for $n \geq 2$:
  $m \in M(n) \cup \{1\}$ necessary, sufficient using C&S
  - New algorithm and extension of previous impossibility proof

The condition

For an anonymous memory consisting of m read/write registers,

$$m \in M(n) = \{m > n \mid \forall\, \ell : 1 < \ell \leq n :\ \gcd(\ell, m) = 1\}$$

is a necessary and sufficient condition.

With Read-Modify-Write, also solvable for $m = 1$ (single reg)

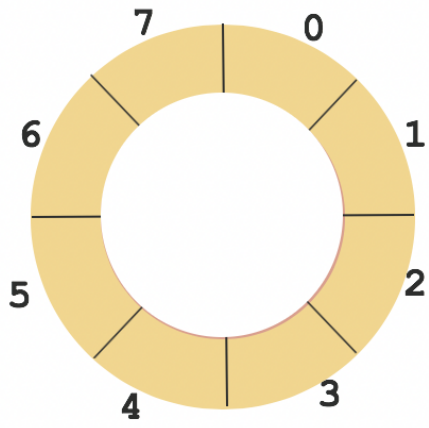
- Not a "threshold" kind of condition
- Another example in the red side of the coin:
  Weak Symmetry Breaking (Do you know any other one?)
- Yet another example, but with mobile agents:
  Leader election in a ring

Impossibility result

Sketch of the proof

Result: with m registers, necessary that m such that
$\forall\, \ell : 1 < \ell \leq n :\ \gcd(\ell, m) = 1$

- Suppose there exists $\ell \in \{2, \ldots, n\}$ that divides m
- Run ℓ processes in lock-step
- Arrange registers $R[0], \ldots, R[m-1]$ on a ring:



- Assign mappings such that
  register x of the k^{th} process is $R[(k \times m/\ell) + x \mod m]$
  (mappings start m/ℓ apart & follow the same cyclic order)

$\Rightarrow$ processes cannot break symmetry

Algorithm

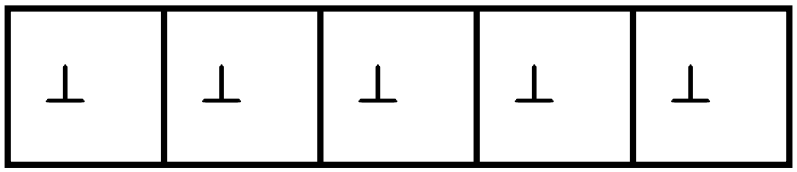
Read-Write

Initially, all registers $= \perp$

To enter the Critical Section:

Take a snapshot. If all registers $= \perp$ or your id appears:

- If there is an empty register ($= \perp$), write your id:



Algorithm

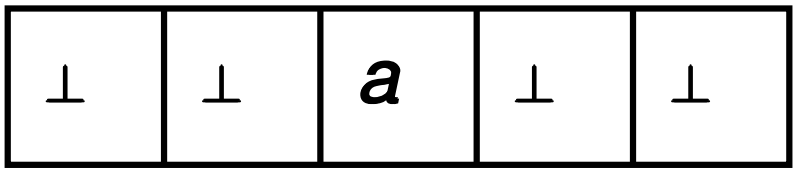
Read-Write

Initially, all registers $= \perp$

To enter the Critical Section:

Take a snapshot. If all registers $= \perp$ or your id appears:

- If there is an empty register ($= \perp$), write your id:



Algorithm

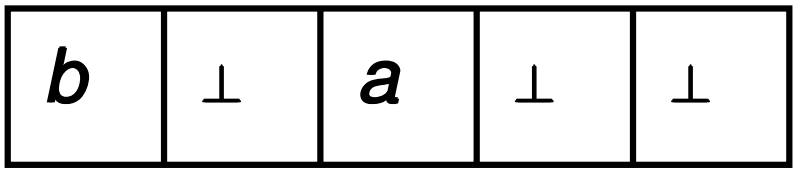
Read-Write

Initially, all registers $= \perp$

To enter the Critical Section:

Take a snapshot. If all registers $= \perp$ or your id appears:

- If there is an empty register ($= \perp$), write your id:



Algorithm

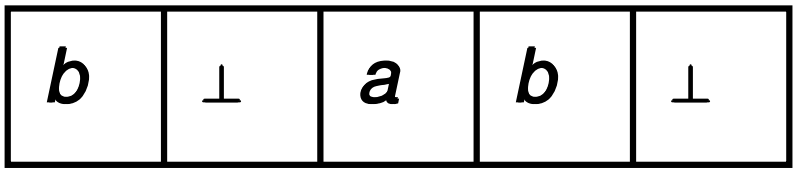
Read-Write

Initially, all registers $= \perp$

To enter the Critical Section:

Take a snapshot. If all registers $= \perp$ or your id appears:

- If there is an empty register ($= \perp$), write your id:



Algorithm

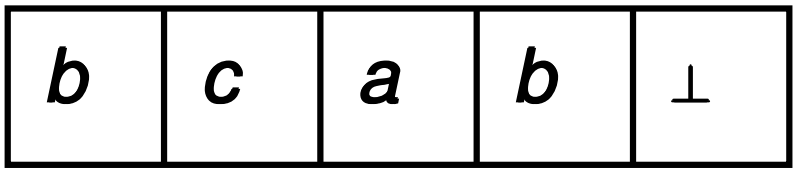
Read-Write

Initially, all registers $= \perp$

To enter the Critical Section:

Take a snapshot. If all registers $= \perp$ or your id appears:

- If there is an empty register ($= \perp$), write your id:



Algorithm

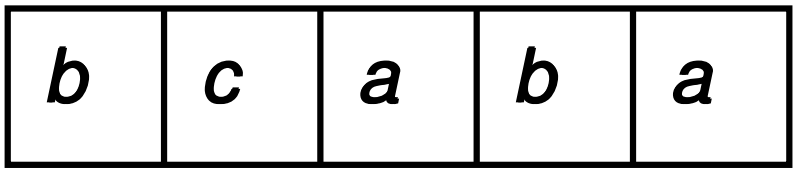
Read-Write

Initially, all registers $= \perp$

To enter the Critical Section:

Take a snapshot. If all registers $= \perp$ or your id appears:

- If there is an empty register ($= \perp$), write your id:



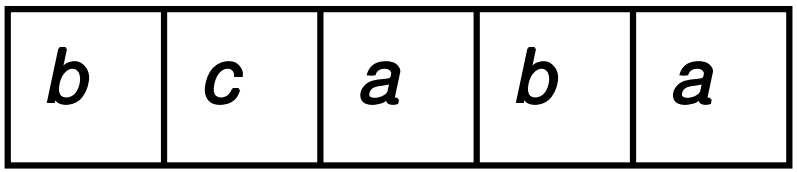
Algorithm

Read-Write

To enter the Critical Section:

Take a snapshot. If all registers $= \perp$ or your id appears:

- If there is an empty register, write your id
- Otherwise, the memory is full:



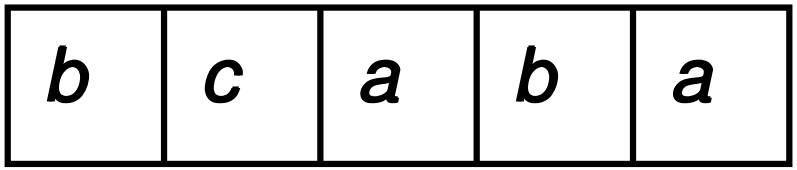
Algorithm

Read-Write

To enter the Critical Section:

Take a snapshot. If all registers $= \perp$ or your id appears:

- If there is an empty register, write your id
- Otherwise, the memory is full:

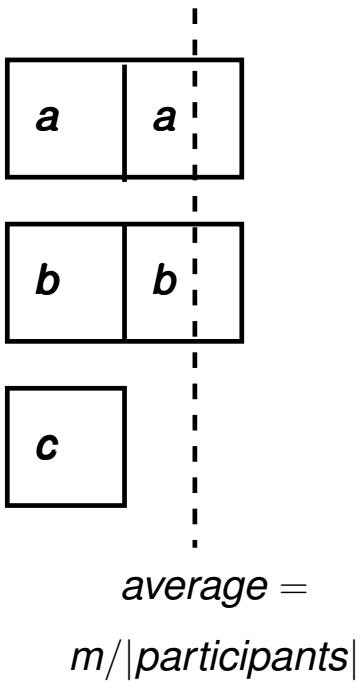


- If you own less than the average (regs that have your id): withdraw from the competition by erasing your id

Algorithm

Read-Write

- Otherwise, the memory is full:
- If you own less than the average (regs that have your id): withdraw from the competition by erasing your id:



At least one process will withdraw:
m not divisible by the current number of participants

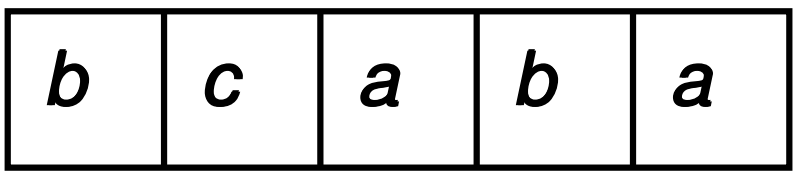
Algorithm

Read-Write

To enter the Critical Section:

Take a snapshot. If all registers $= \perp$ or your id appears:

- If there is an empty register, write your id
- Otherwise, the memory is full:
  - If you own less than the average (regs that have your id): withdraw from the competition by erasing your id



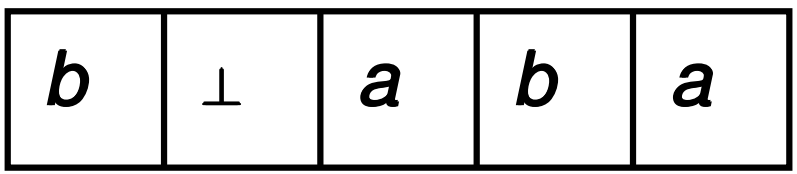
Algorithm

Read-Write

To enter the Critical Section:

Take a snapshot. If all registers $= \perp$ or your id appears:

- If there is an empty register, write your id
- Otherwise, the memory is full:
  - If you own less than the average (regs that have your id): withdraw from the competition by erasing your id



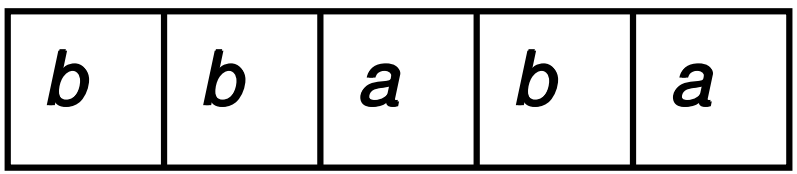
Algorithm

Read-Write

To enter the Critical Section:

Take a snapshot. If all registers $= \perp$ or your id appears:

- If there is an empty register, write your id
- Otherwise, the memory is full:
  - If you own less than the average (regs that have your id): withdraw from the competition by erasing your id



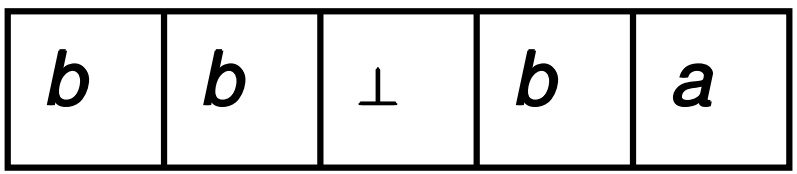
Algorithm

Read-Write

To enter the Critical Section:

Take a snapshot. If all registers $= \perp$ or your id appears:

- If there is an empty register, write your id
- Otherwise, the memory is full:
  - If you own less than the average (regs that have your id): withdraw from the competition by erasing your id



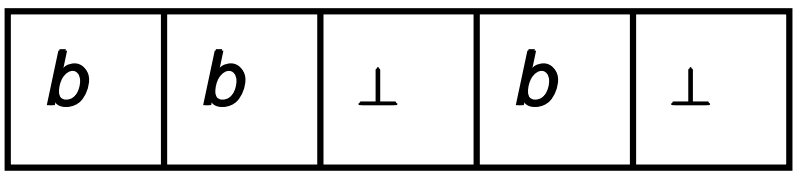
Algorithm

Read-Write

To enter the Critical Section:

Take a snapshot. If all registers $= \perp$ or your id appears:

- If there is an empty register, write your id
- Otherwise, the memory is full:
  - If you own less than the average (regs that have your id): withdraw from the competition by erasing your id



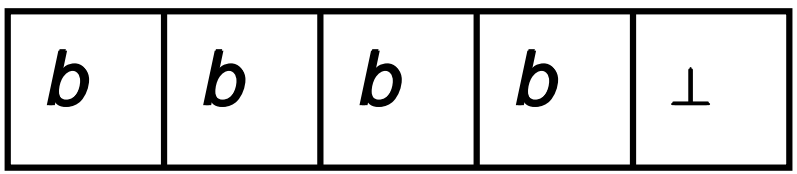
Algorithm

Read-Write

To enter the Critical Section:

Take a snapshot. If all registers $= \perp$ or your id appears:

- If there is an empty register, write your id
- Otherwise, the memory is full:
  - If you own less than the average (regs that have your id): withdraw from the competition by erasing your id



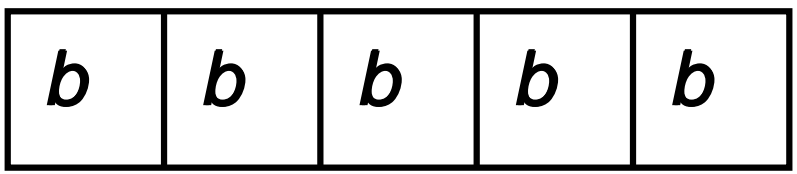
Algorithm

Read-Write

To enter the Critical Section:

Take a snapshot. If all registers $= \perp$ or your id appears:

- If there is an empty register, write your id
- Otherwise, the memory is full:
  - If you own less than the average (regs that have your id): withdraw from the competition by erasing your id



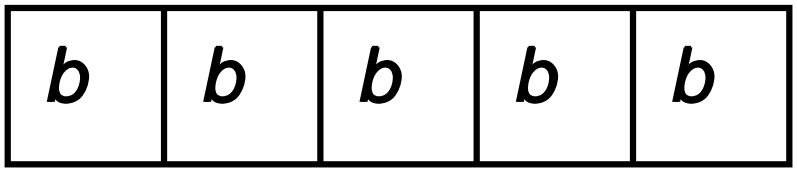
Algorithm

Read-Write

To enter the Critical Section:

Take a snapshot. If all registers $= \perp$ or your id appears:

- If there is an empty register, write your id
- Otherwise, the memory is full:
  If you own less than the average (regs that have your id):
  withdraw from the competition by erasing your id



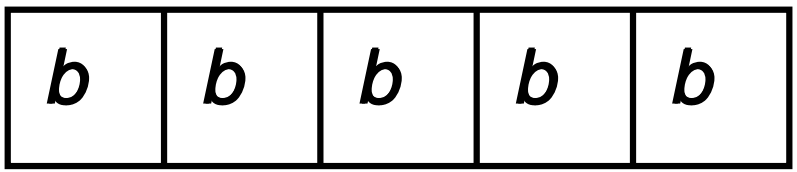
If your id appears in all registers, enter the Critical Section

Algorithm

Read-Write

To exit the Critical Section:

- Erase your id

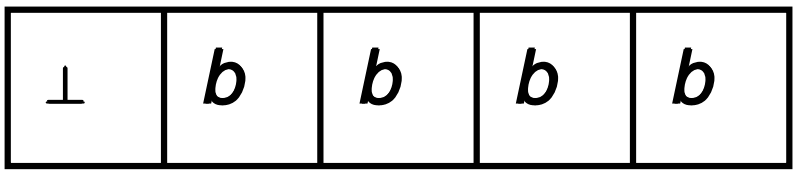


Algorithm

Read-Write

To exit the Critical Section:

- Erase your id

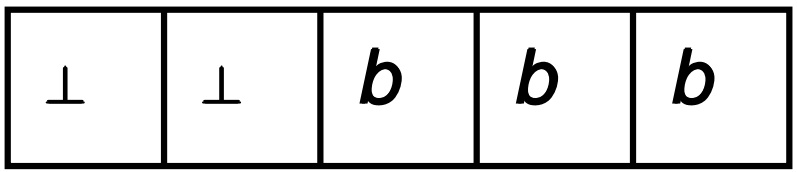


Algorithm

Read-Write

To exit the Critical Section:

- Erase your id

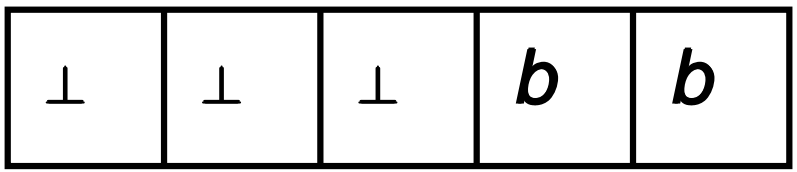


Algorithm

Read-Write

To exit the Critical Section:

- Erase your id

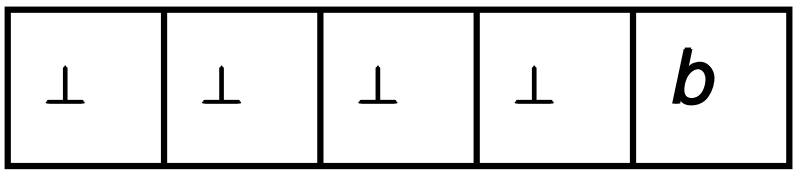


Algorithm

Read-Write

To exit the Critical Section:

- Erase your id

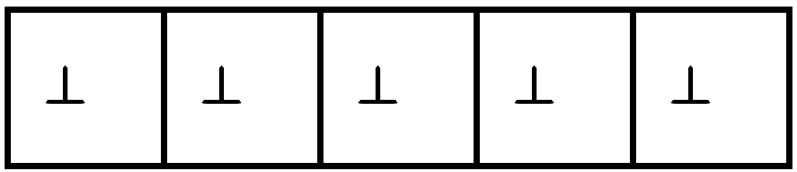


Algorithm

Read-Write

To exit the Critical Section:

- Erase your id



Algorithm

Read-Write

Mutual exclusion:

- A process enters the CS only if it owns all registers
- $m > n$ regs: at most $n - 1$ can be overwritten while in CS

Deadlock-freedom:

- $\forall\, \ell : 1 < \ell \leq n :\ \gcd(\ell, m) = 1$:
- Whatever the current number of participants (≥ 2),
  once the memory is full,
  at least one will withdraw and at least one will continue

Algorithm

Read-Modify-Write (Compare&Swap)

- RMW registers offering Compare&Swap:
  - In addition to Read() and Write(v), a new operation: Compare&Swap(old, new)
- Effect:
  - If the current value is old, replace it with new
  - Otherwise, do nothing

Using Compare&Swap, Mutex also solvable for $m = 1$
$\Rightarrow$ not a big difference; the obstruction is not really linked to the power of operations on individual registers

Algorithm

Read-Modify-Write (Compare&Swap)

The algorithm:

- To enter the CS:
  - At each reg: try to impose your id using Compare&Swap()
  - Scan the memory
    - If a process owns more registers:
      free your registers and
      wait until memory is empty (another proc will enter first)
    - Else: start again
  - Until you own a majority of registers
- To exit the CS: reset all your registers to $\perp$ using C&S

Algorithm

Read-Modify-Write (Compare&Swap)

Mutual exclusion:

- A process enters the CS only if it owns a majority
- Compare&Swap:
  a process can only "capture" a register if it is empty
  (cannot overwrite a process id)

Deadlock-freedom:

- $\forall\, \ell : 1 < \ell \leq n :\ \gcd(\ell, m) = 1$:
- Whatever the current number of participants (≥ 2),
  once the memory is full,
  at least one will withdraw and at least one will continue

Conclusion

- Anonymous memory: a new communication model
- A tight characterization of the solvability of symmetric deadlock-free mutual exclusion
- An interesting condition (not threshold-based)
- A new Read-Write mutex algorithm (also works with a classical shared memory!)
- Another new algorithm based on Compare&Swap()