6.852: Distributed Algorithms
Fall, 2009

Class 15

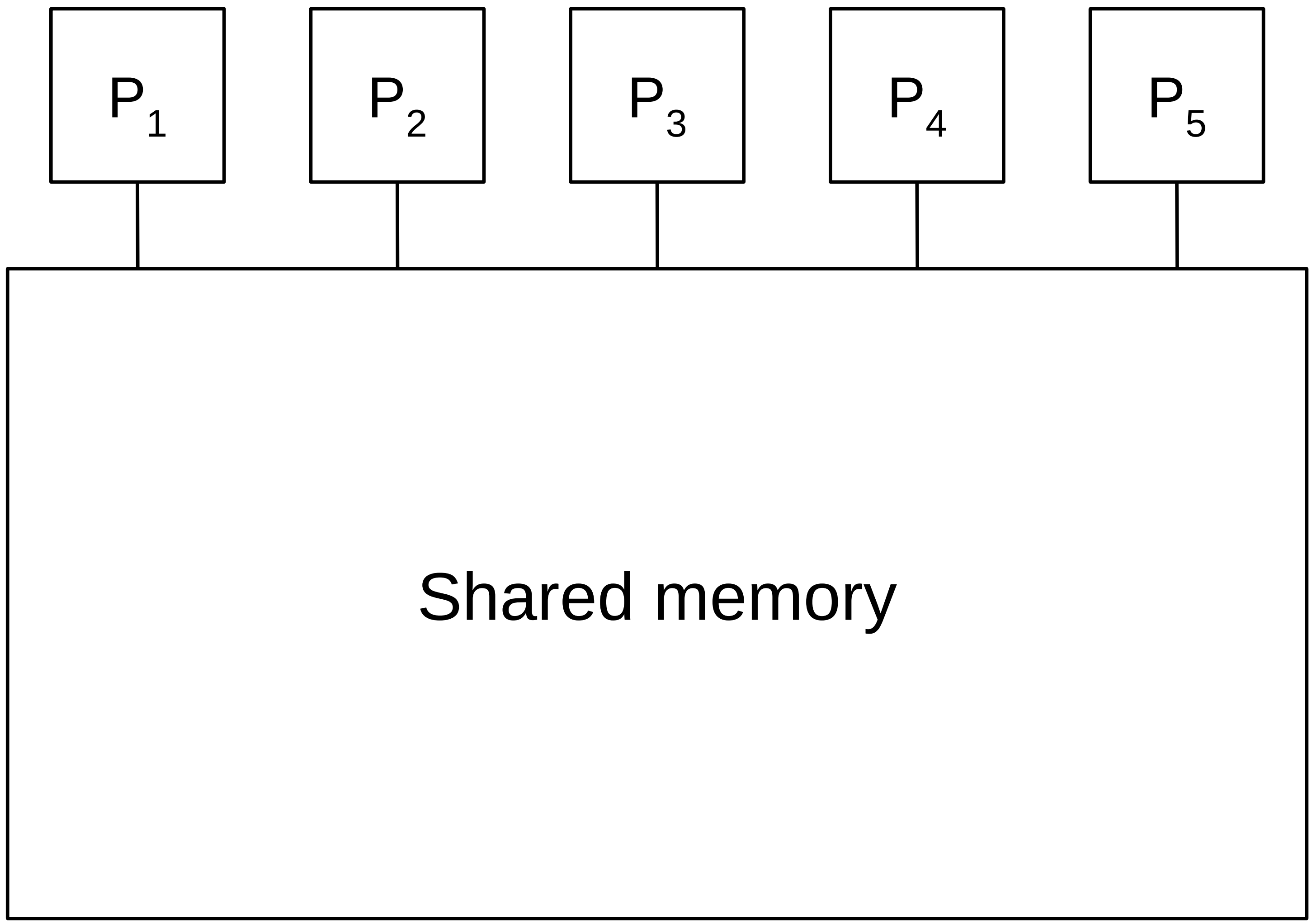
Today's plan

- Pragmatic issues for shared-memory multiprocessors
- Practical mutual exclusion algorithms
  - Test-and-set locks
  - Ticket locks
  - Queue locks
- Generalized exclusion/resource allocation problems
- Reading:
  - Herlihy, Shavit, Chapter 7
  - Mellor-Crummey, Scott paper (Dijkstra prize winner)
  - Magnussen, Landin, Hagersten paper
  - Lynch, Chapter 11
- Next:
  - Consensus
  - Lynch, Chapter 12

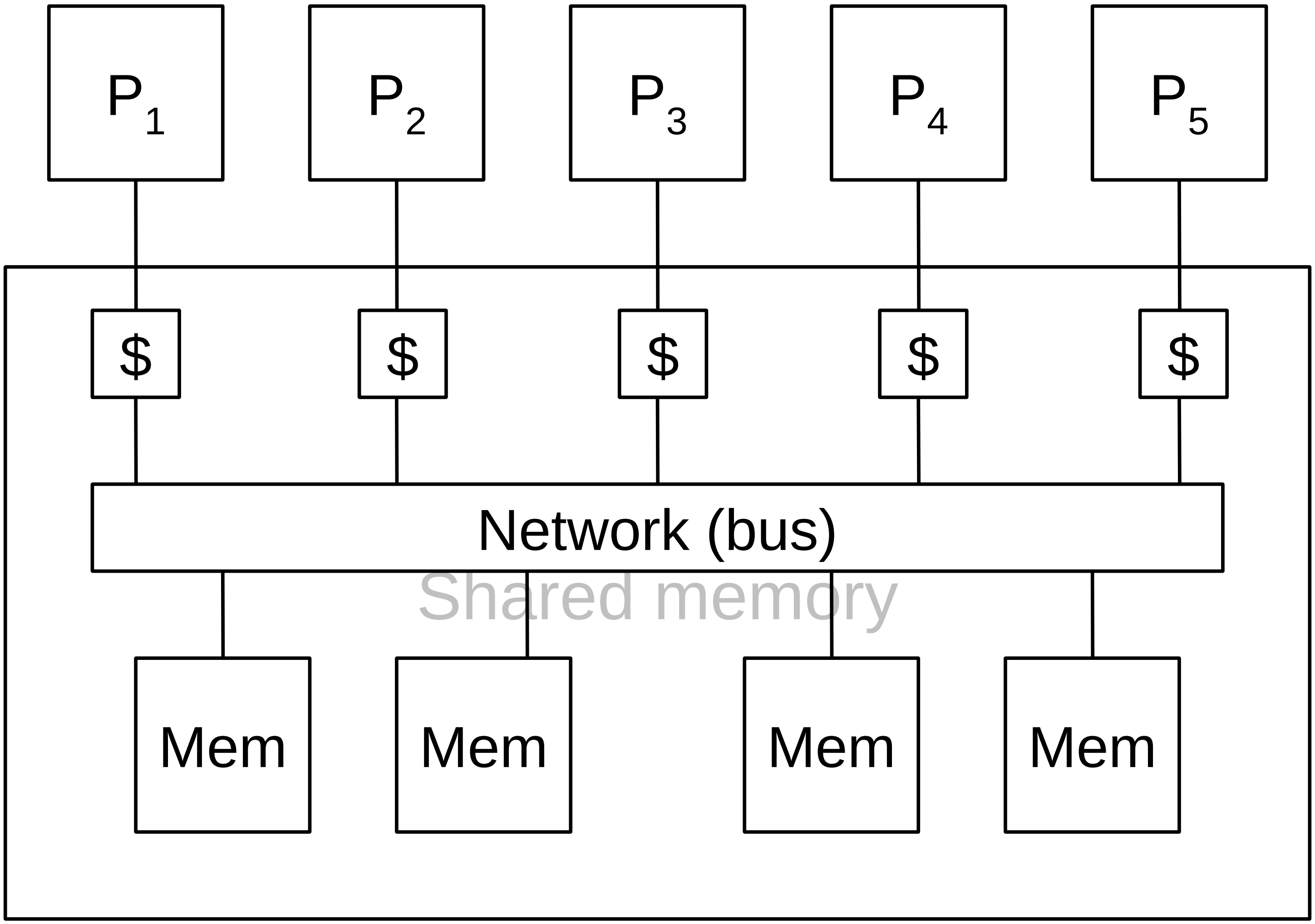
Last time

- Mutual exclusion algorithms using read/write shared memory:
  - Dijkstra, Peterson, Lamport Bakery, Burns
- Mutual exclusion algorithms using read/modify/write (RMW) shared memory:
  - Trivial 1-bit Test-and-Set algorithm, Queue algorithm, Ticket algorithm
- Single-level shared memory
- But modern shared-memory multiprocessors are somewhat different.
- The difference affects the design of practical mutex algorithms.

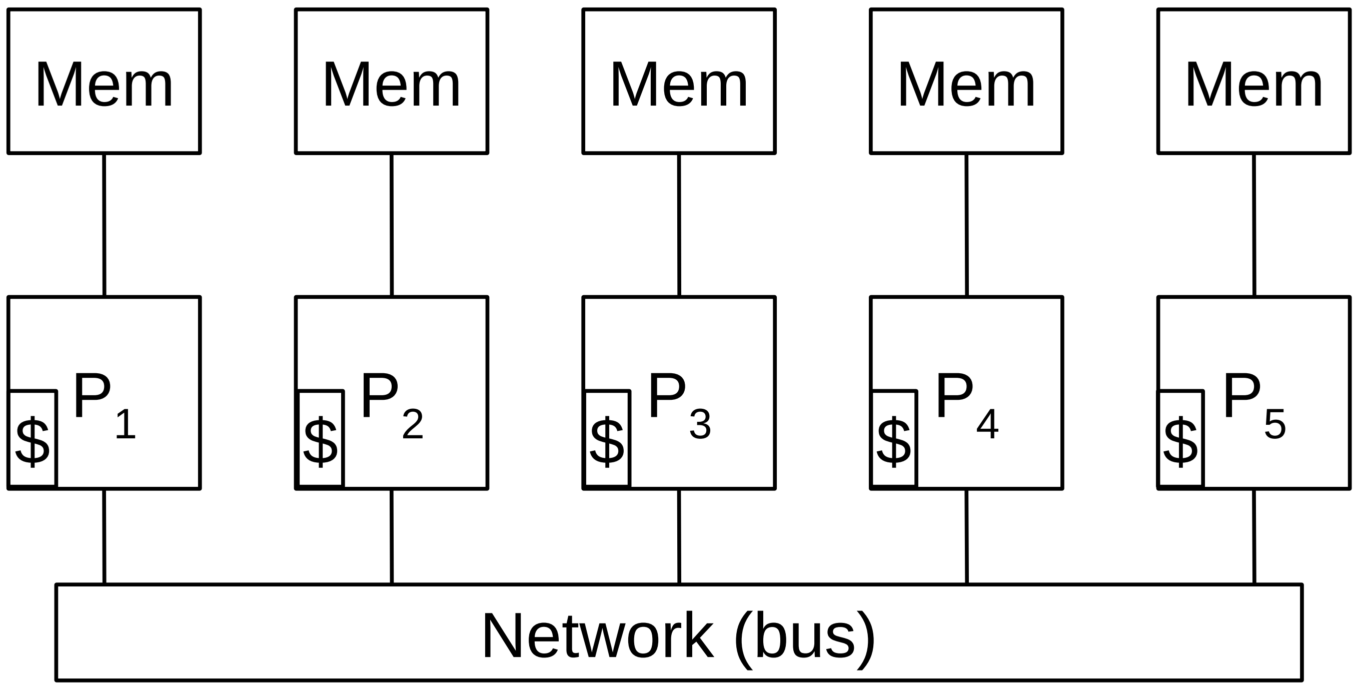
Shared-memory multiprocessors



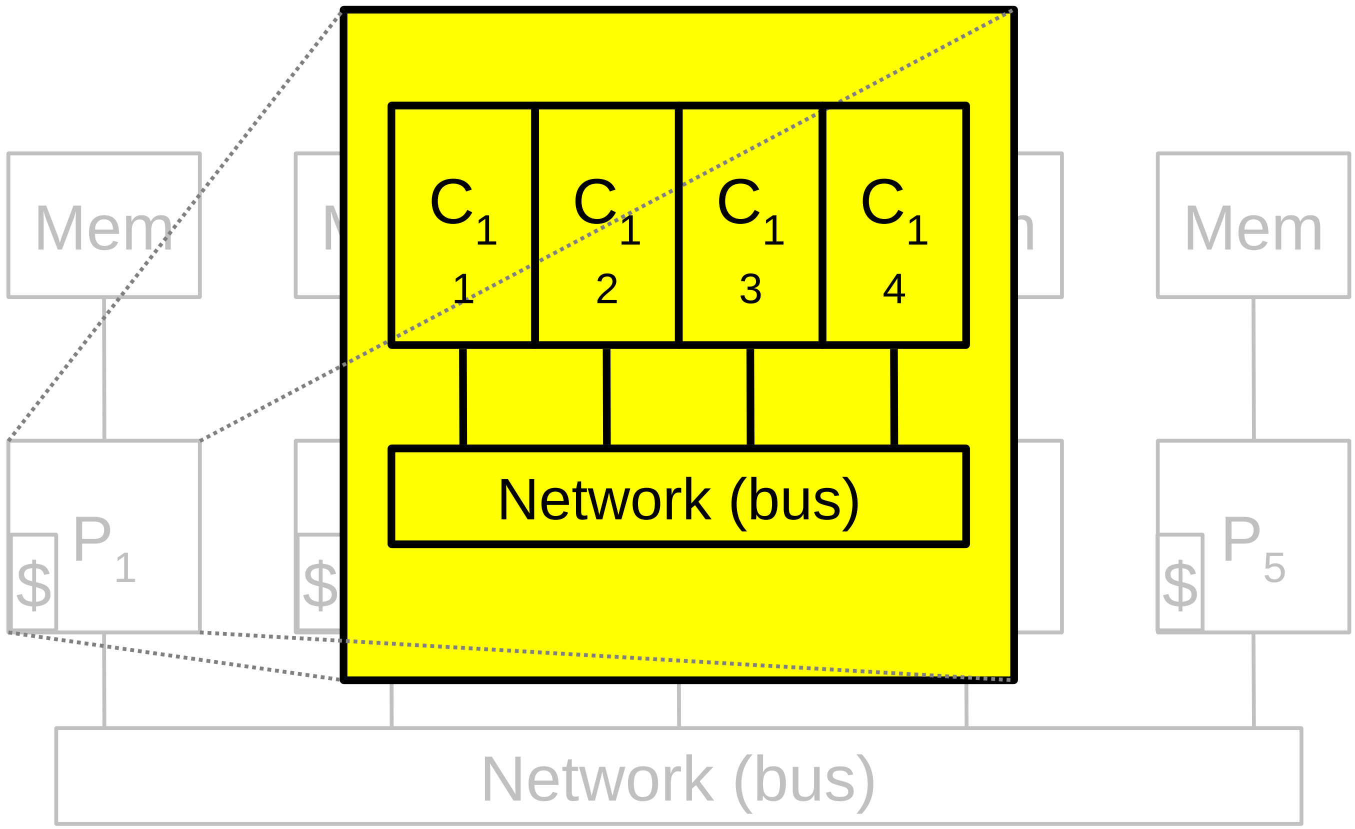
Shared-memory multiprocessors



Shared-memory multiprocessors



Shared-memory multiprocessors



Costs for shared-memory multiprocessors

- Memory access costs are non-uniform:
  - Next-level cache access is ~10x more expensive (time delay).
- Remote memory access produces network traffic.
  - Network bandwidth can be a bottleneck.
- Writes invalidate cache entries.
  - A process that wants to read must request again.
- Reads typically don’t invalidate cache entries.
  - Processes can share read access to an item.
- All memory supports multiple writers, but most is reserved for individual processes.

Memory operations

- Modern shared-memory multiprocessors provide stronger operations than just reads and writes.
- “Atomic” operations:
  - Test&Set: Write 1 to the variable, return the previous value.
  - Fetch & Increment: Increment the variable, return the previous value.
  - Swap: Write the submitted value to the variable, return the previous value.
  - Compare&Swap (CAS): If the variable’s value is equal to the first submitted value, then reset it to the second submitted value; return the previous value. (Alternatively, return T/F indicating whether the swap succeeded.)
  - Load-link (LL) and Store-conditional (SC): LL returns current value; SC stores a new value only iff no updates have occurred since the last LL.

Mutual exclusion in practice

- Uses strong, “atomic” operations, not just reads and writes:
  - Test&Set, Fetch&Increment, Swap, Compare&Swap (CAS), LL/SC
- Examples:
  - One-variable Test&Set algorithm
  - Ticket lock algorithm: Two Fetch&Increment variables.
  - Queue lock algorithms:
    - One queue with enqueue, dequeue and head.
    - Since multiprocessors do not support queues in hardware, implement this using Fetch&Increment, Swap, CAS.
- Terminology: Critical section called a “Lock”.

Spinning vs. blocking

- What happens when a process wants a lock (critical section) that is currently taken? Two possibilities:
- Spinning:
  - The process keeps performing the trying protocol.
  - Our theoretical algorithms do this.
  - In practice, often keep retesting certain variables, waiting for some "condition" to become true.
  - Good if waiting time is expected to be short.
- Blocking:
  - The process deschedules itself (yields the processor)
  - OS reschedules it later, e.g., when some condition is satisfied.
  - Monitors, conditions (See HS, Chapter 8).
  - Better than spinning if waiting time is long.
- Choice of spinning vs. blocking applies to other synchronization constructs besides locks, e.g., producer-consumer synchronization, barrier synchronization.

Our assumptions

- Spinning, not blocking.
  - Spin locks are commonly used, e.g., in OS kernels.
  - Assume critical sections are very short.
  - Processes usually hold only one lock at a time.
- No multiprogramming (one process per processor).
  - Processes are not "swapped out" while in the critical region, or while executing trying/exit code.
- Performance is critical.
  - Must consider caching and contention effects.
  - Unknown set of participants (adaptive).

Spin locks

- Test&Set locks
- Ticket lock
- Queue locks
  - Anderson
  - Graunke/Thakkar
  - Mellor-Crummey/Scott (MCS)
  - Craig-Landin-Hagersten (CLH)
- Adding other features
  - Timeout
  - Hierarchical locks
  - Reader-writer locks
- Note: No formal complexity analysis here!

Test&Set Locks

- Simple T&S lock, widely used in practice.
- Test-and-Test&Set lock, reduces contention.
- T&S with backoff.

Simple Test&Set lock

lock: {0,1}; initially 0

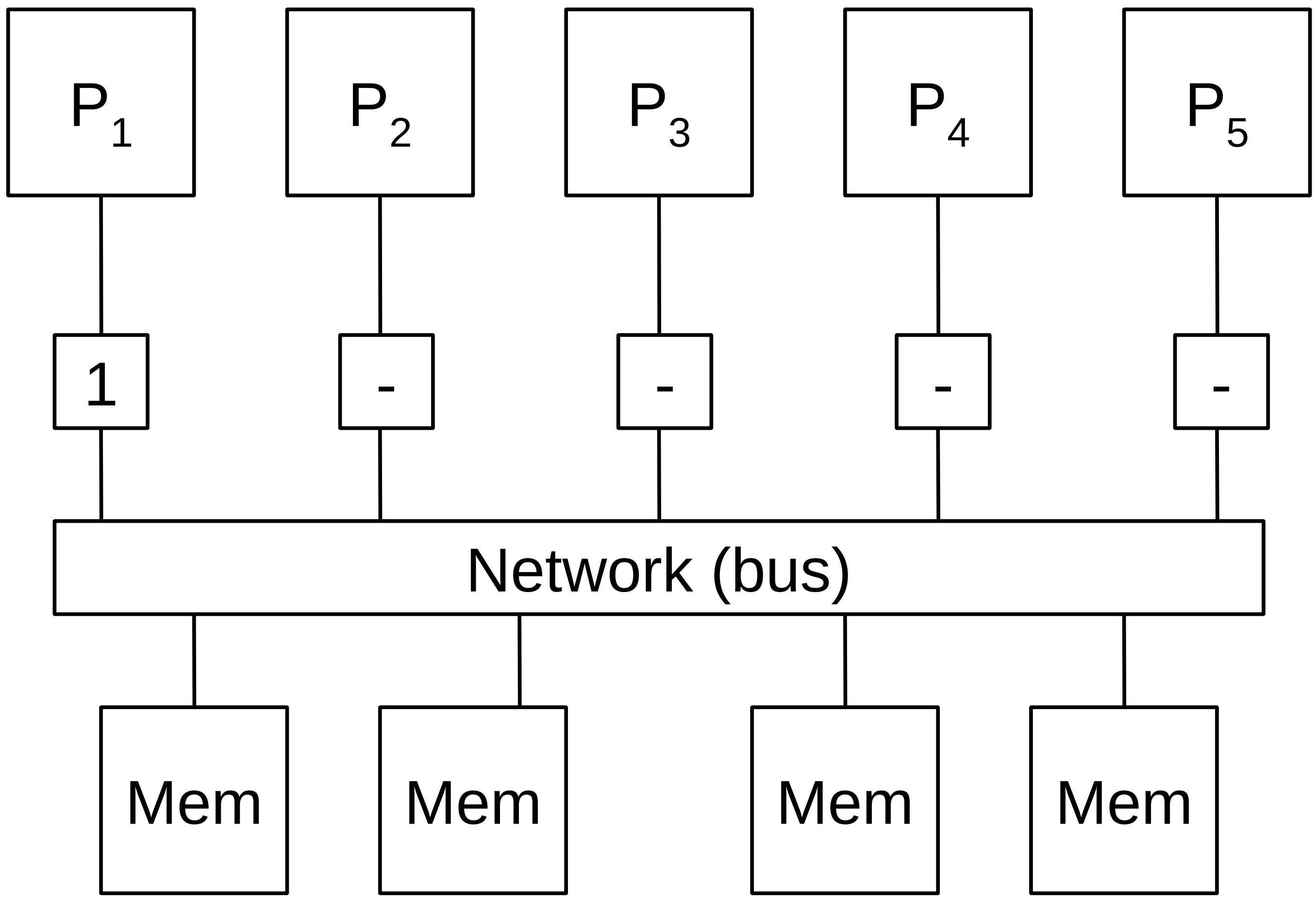
try_i
  waitfor(test&set(**lock)** = 0)
$crit_i$

$exit_i$
  lock := 0
rem_i

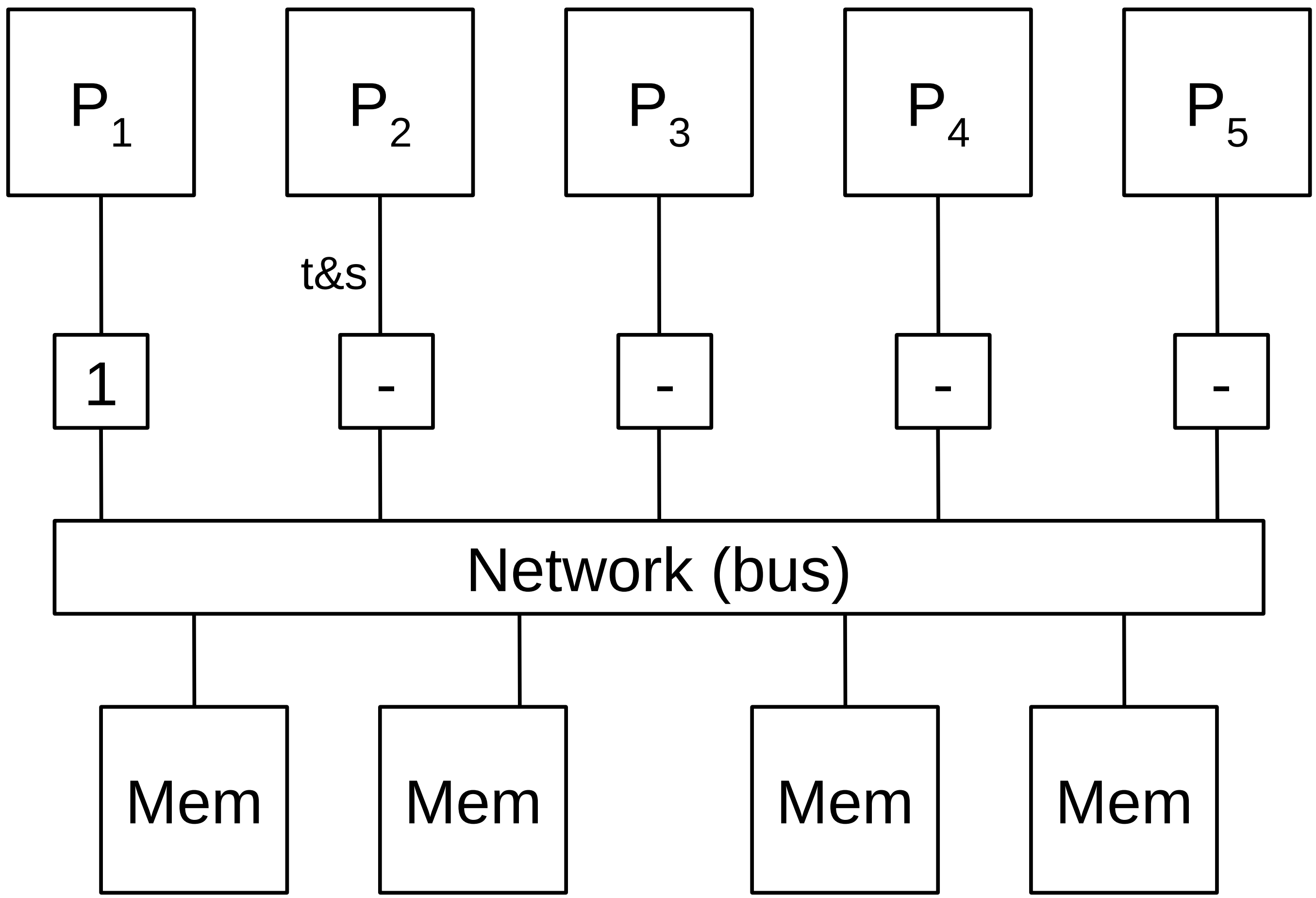
- Simple.
- Low space cost (1 bit).
- But lots of network traffic if highly contended.

Many processes waiting for lock to become free.

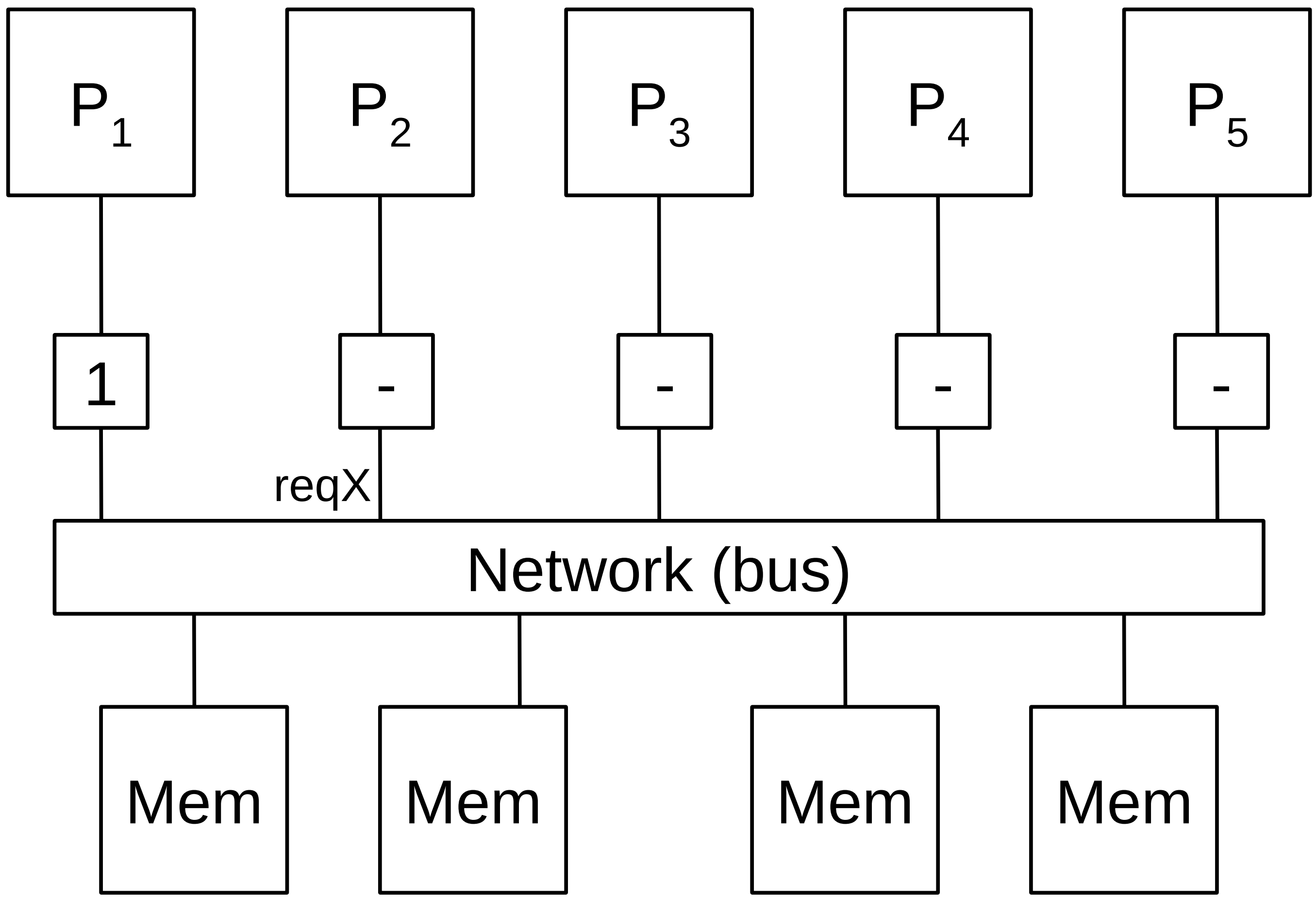
Simple test&set lock



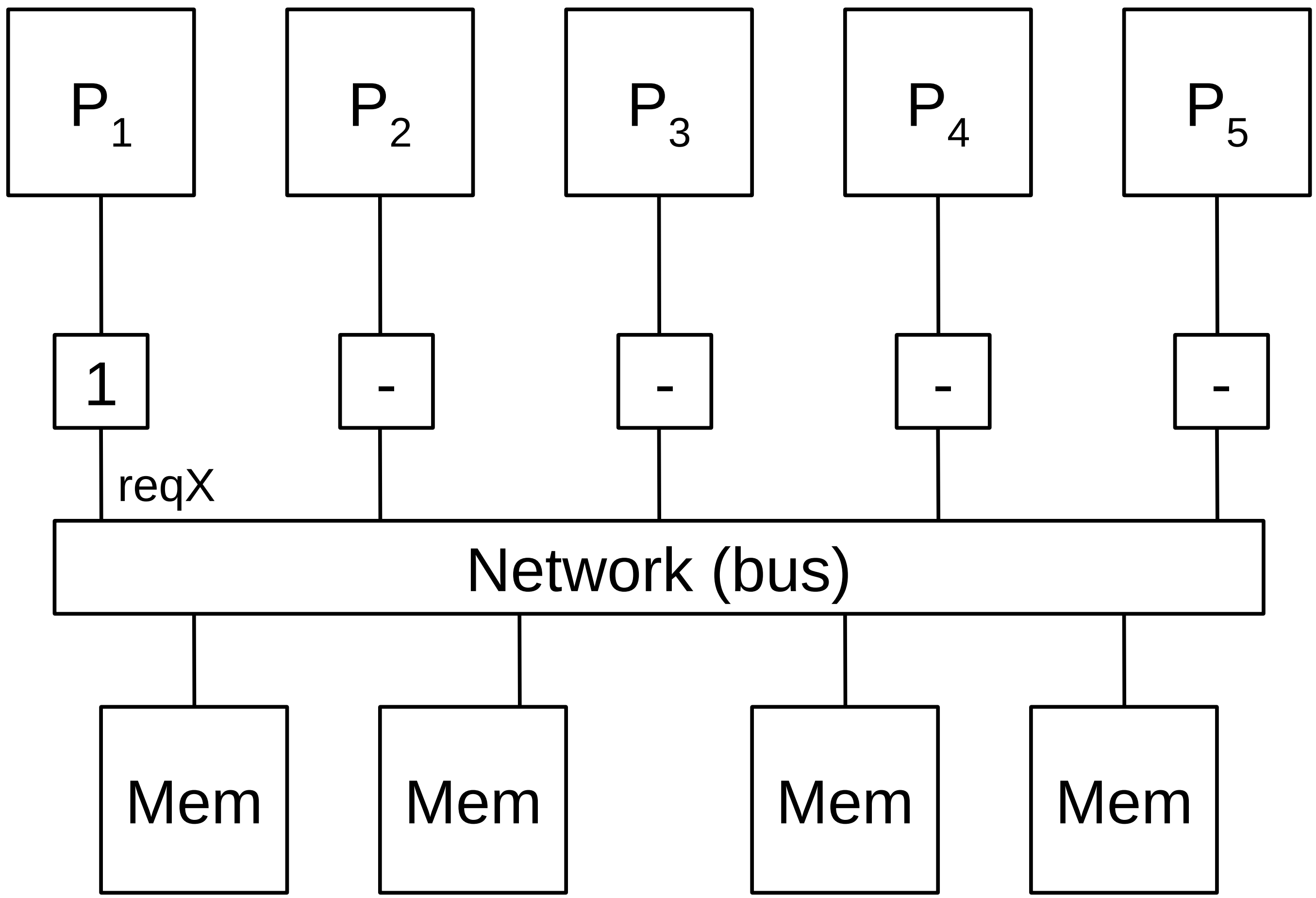
Simple test&set lock



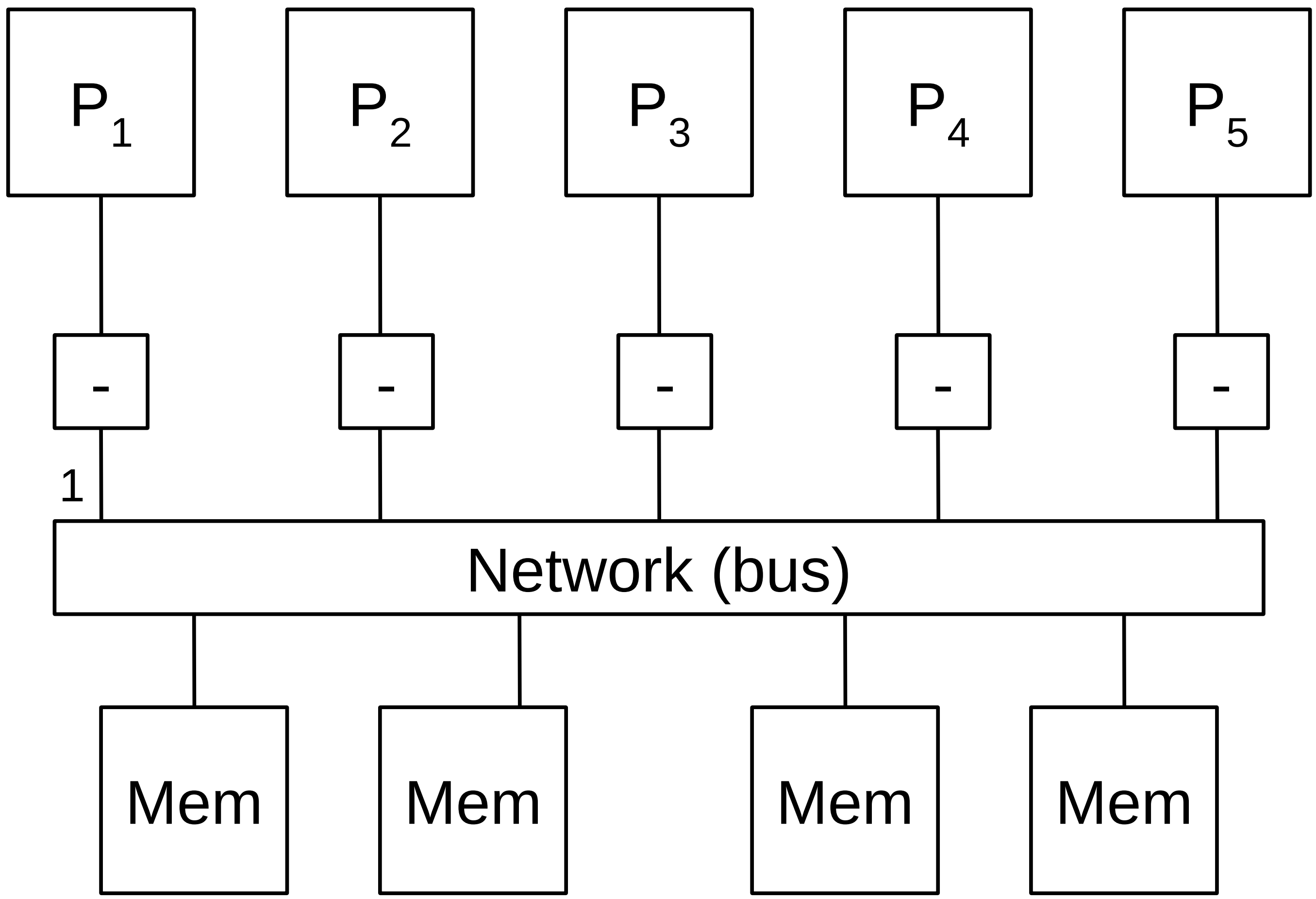
Simple test&set lock



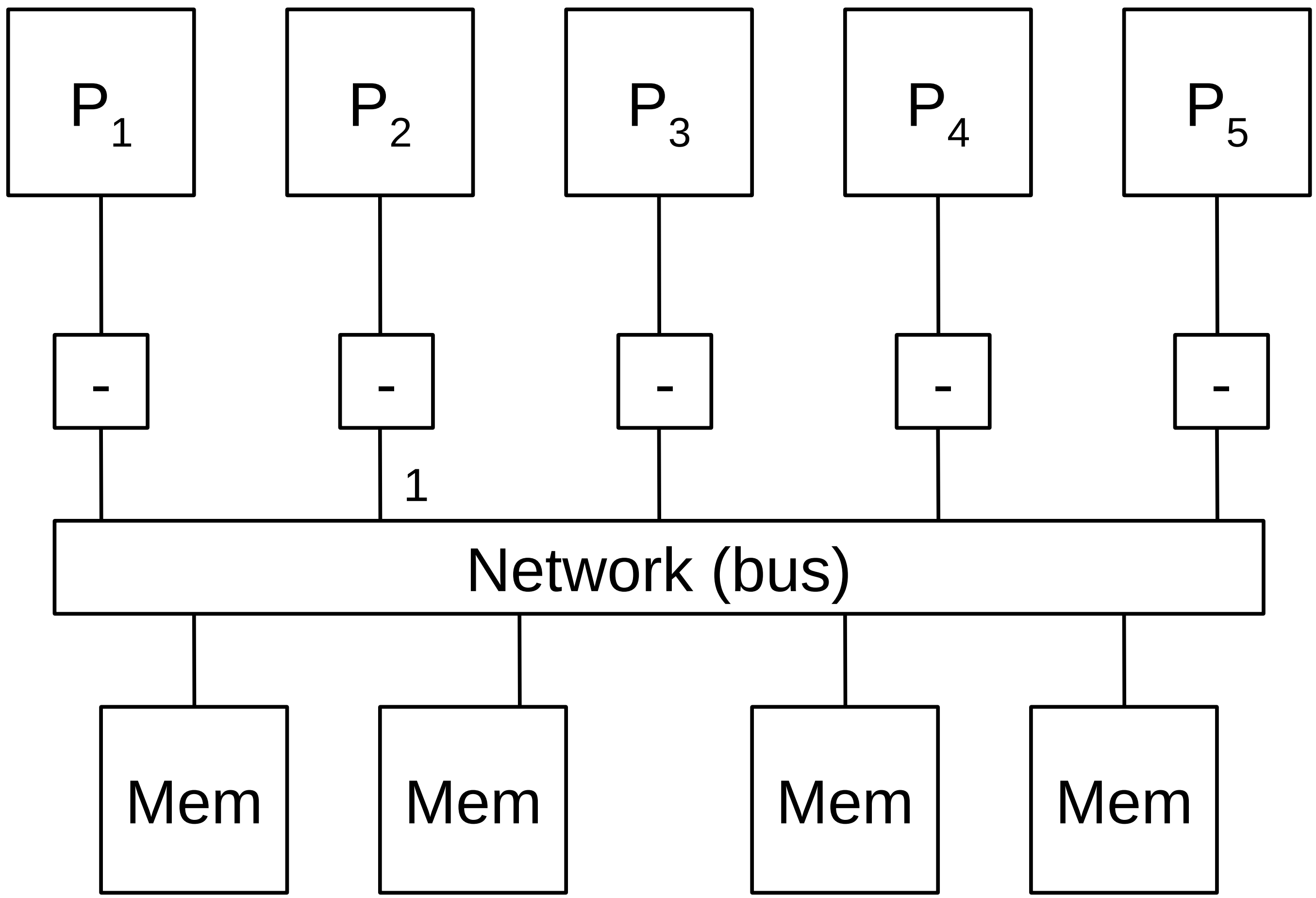
Simple test&set lock



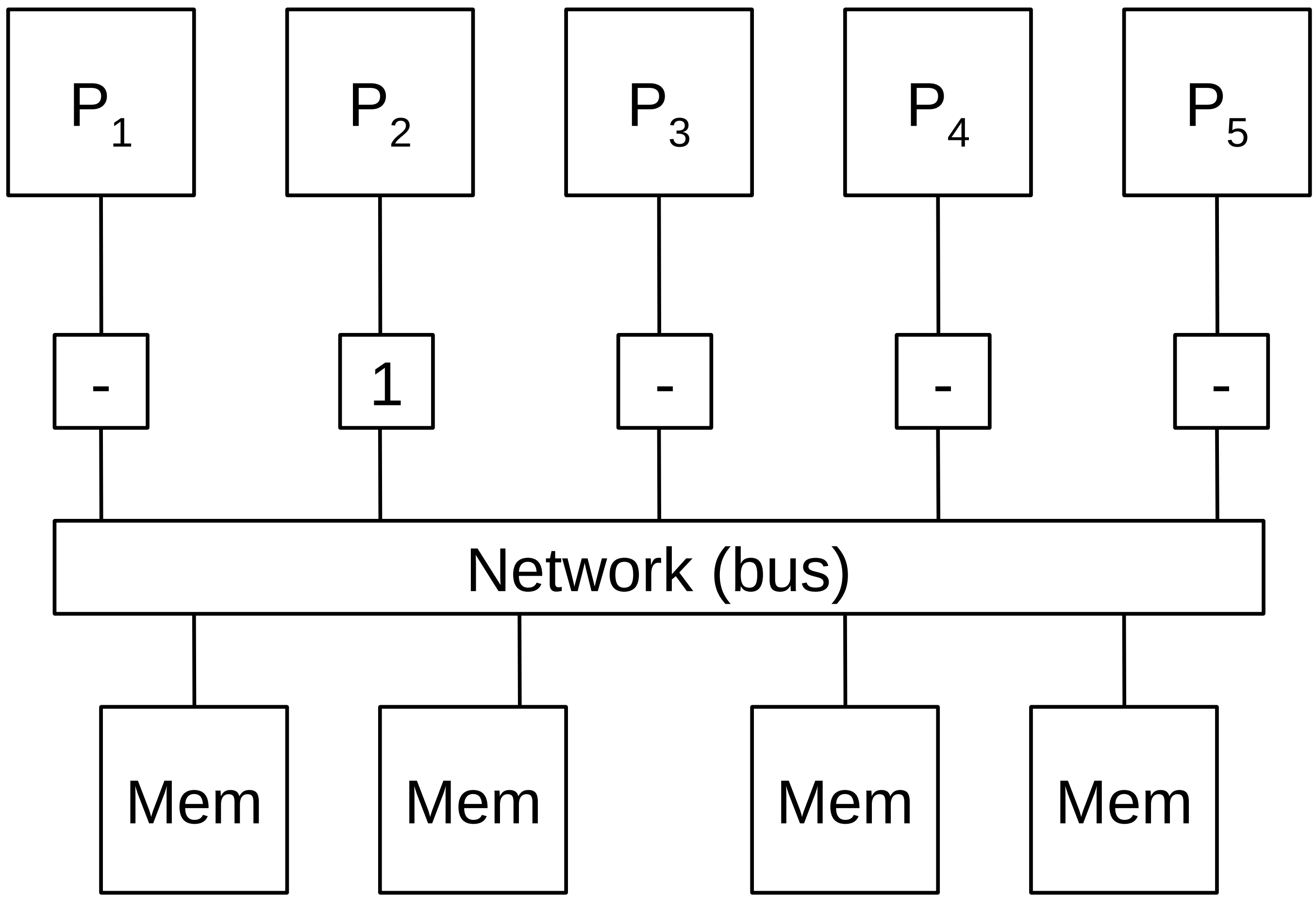
Simple test&set lock



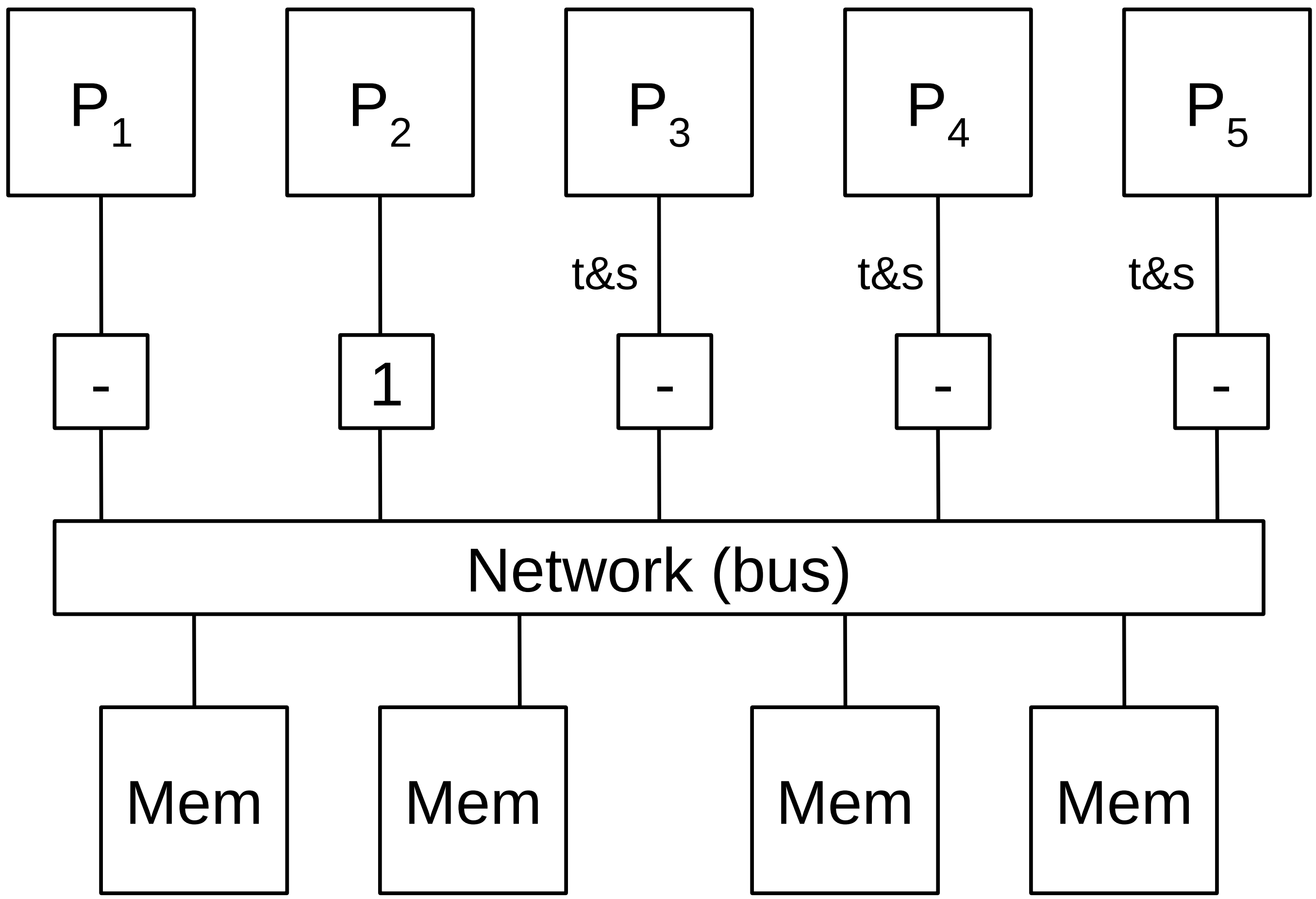
Simple test&set lock



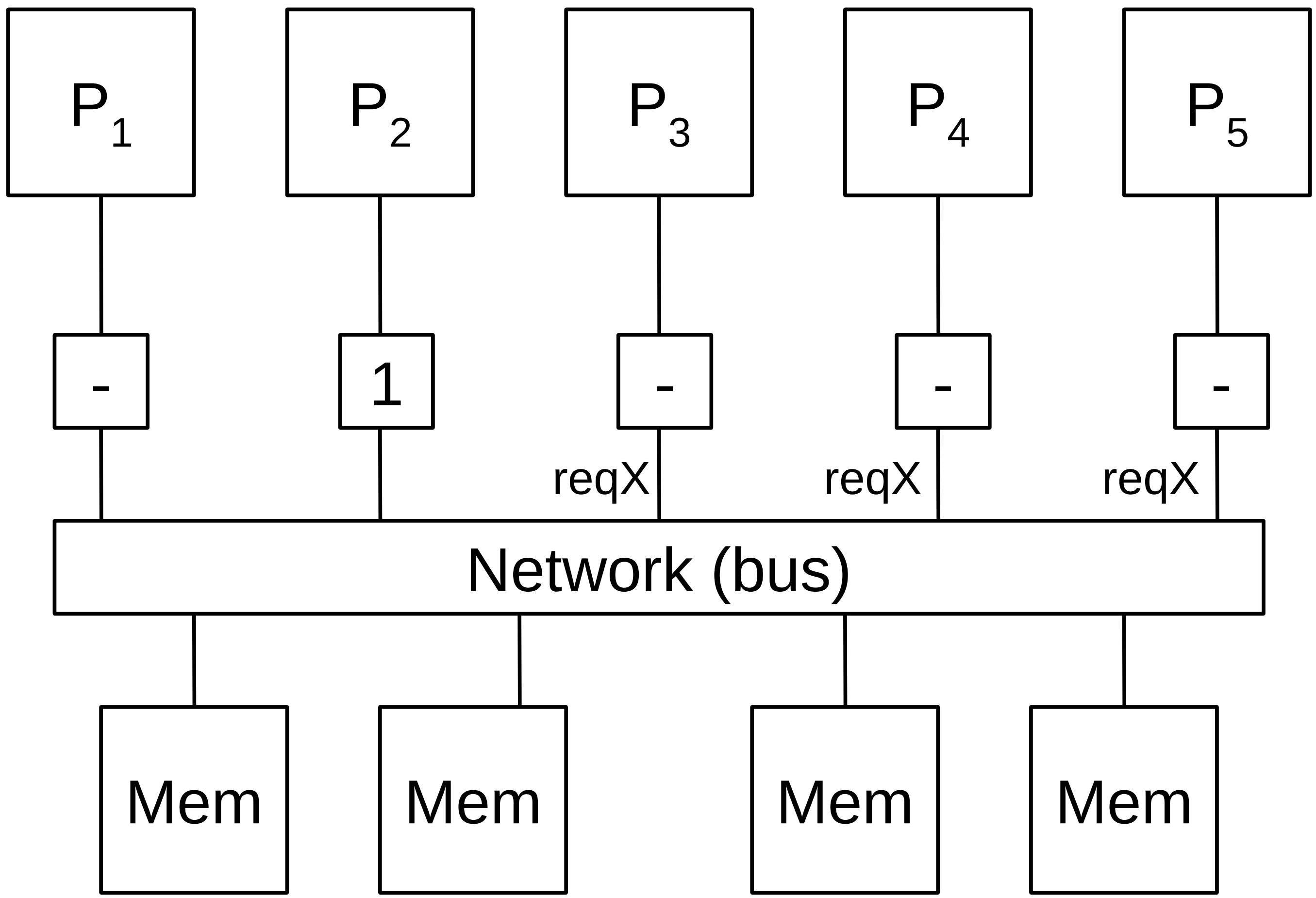
Simple test&set lock



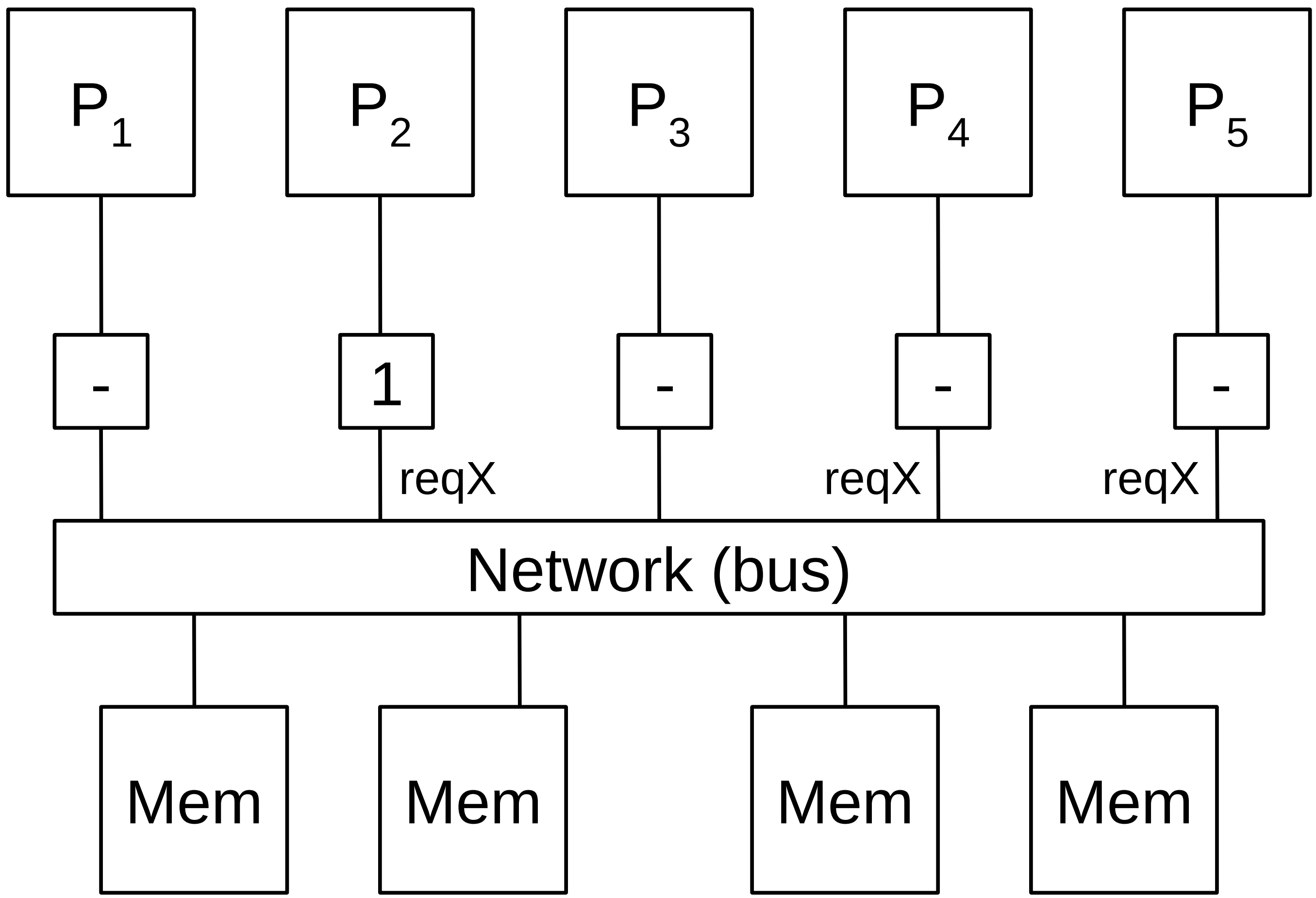
Simple test&set lock



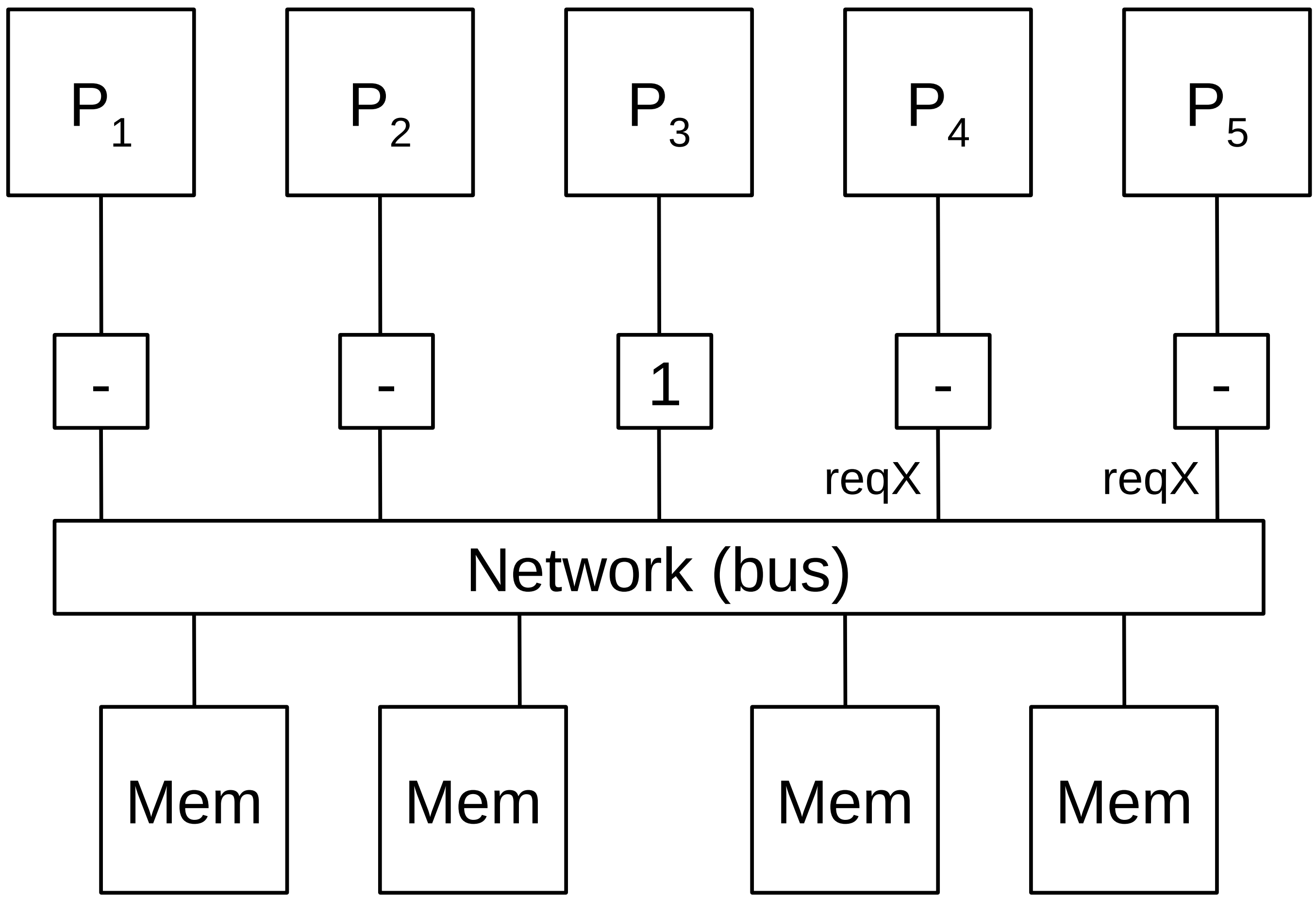
Simple test&set lock



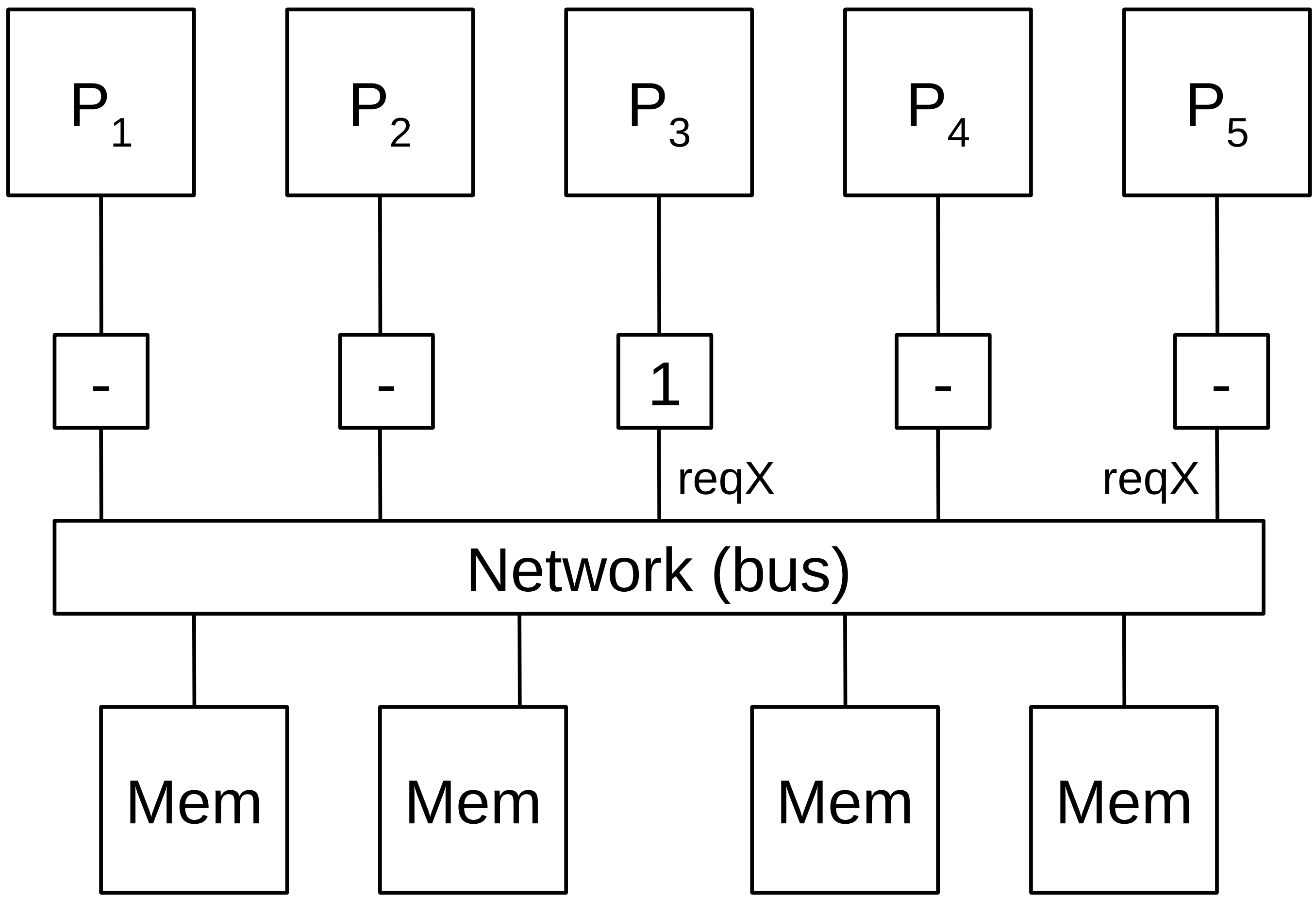
Simple test&set lock



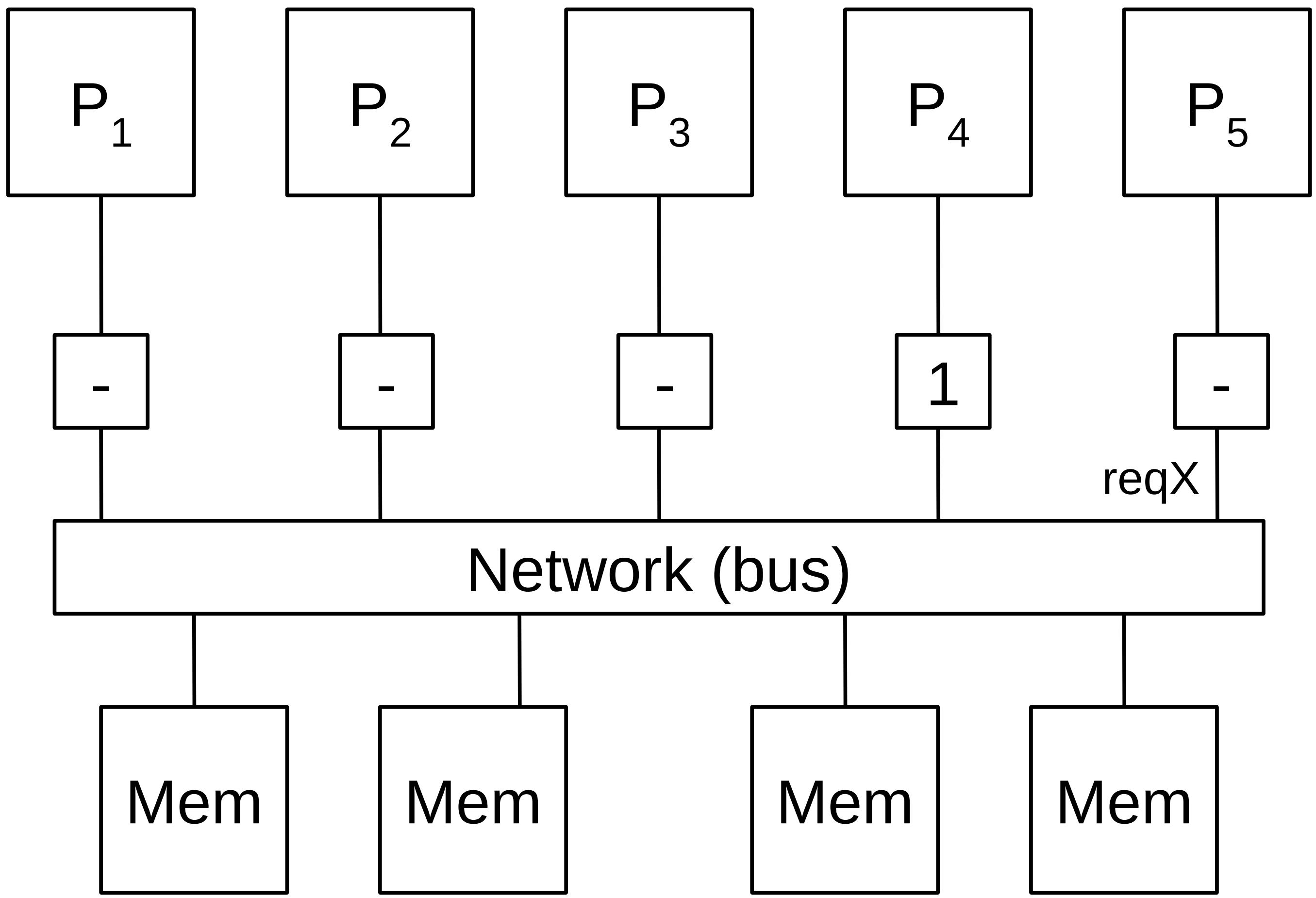
Simple test&set lock



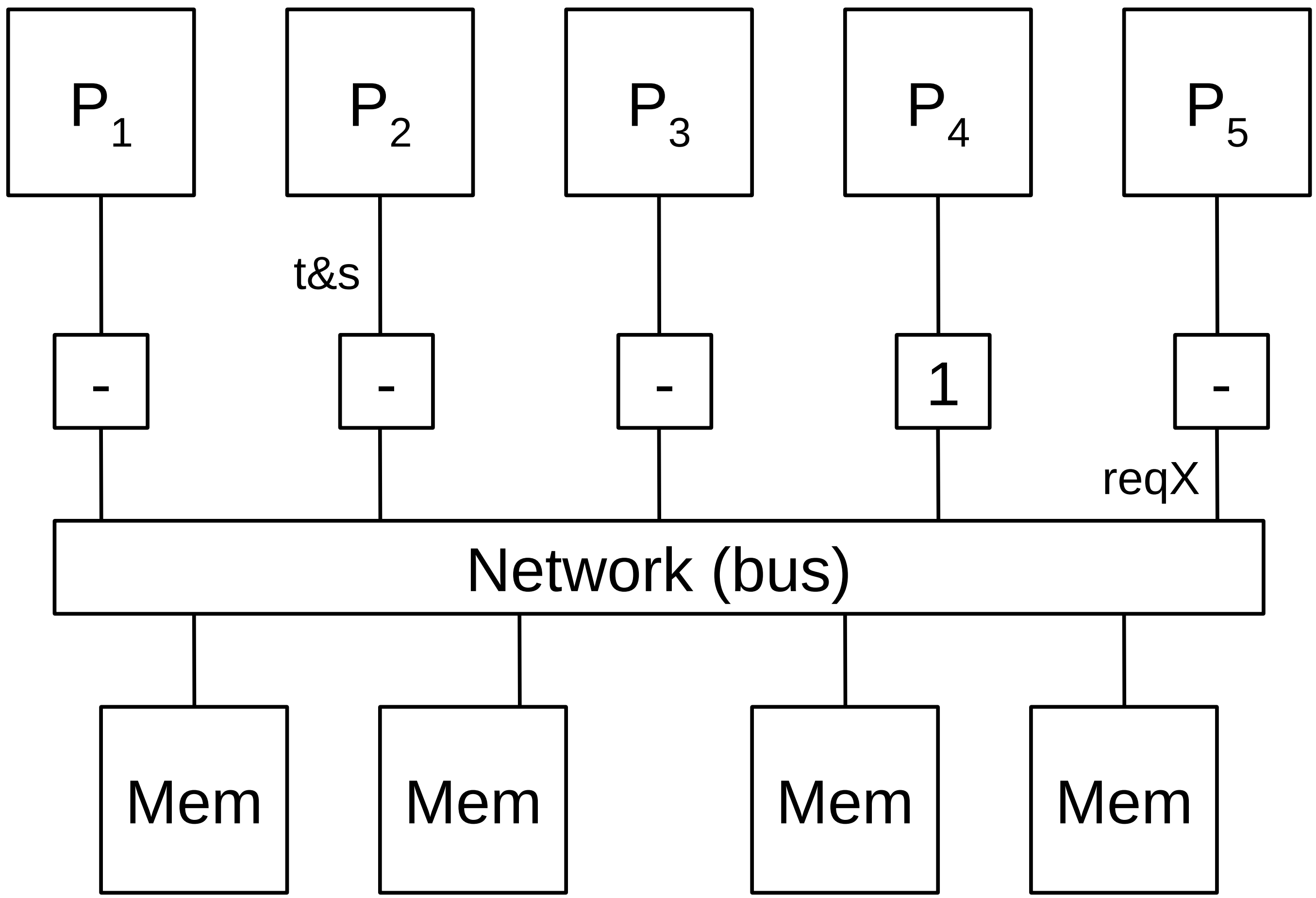
Simple test&set lock



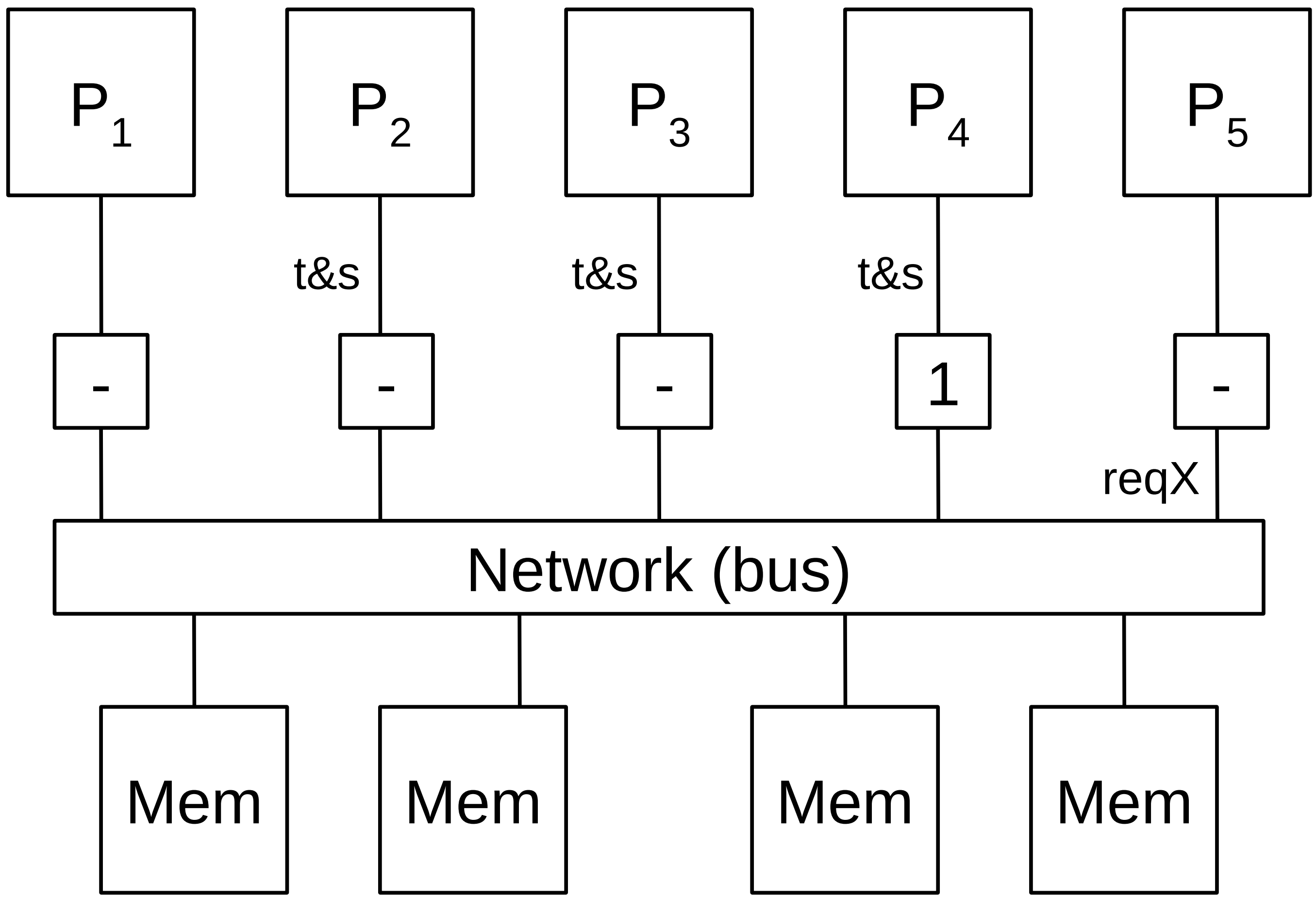
Simple test&set lock



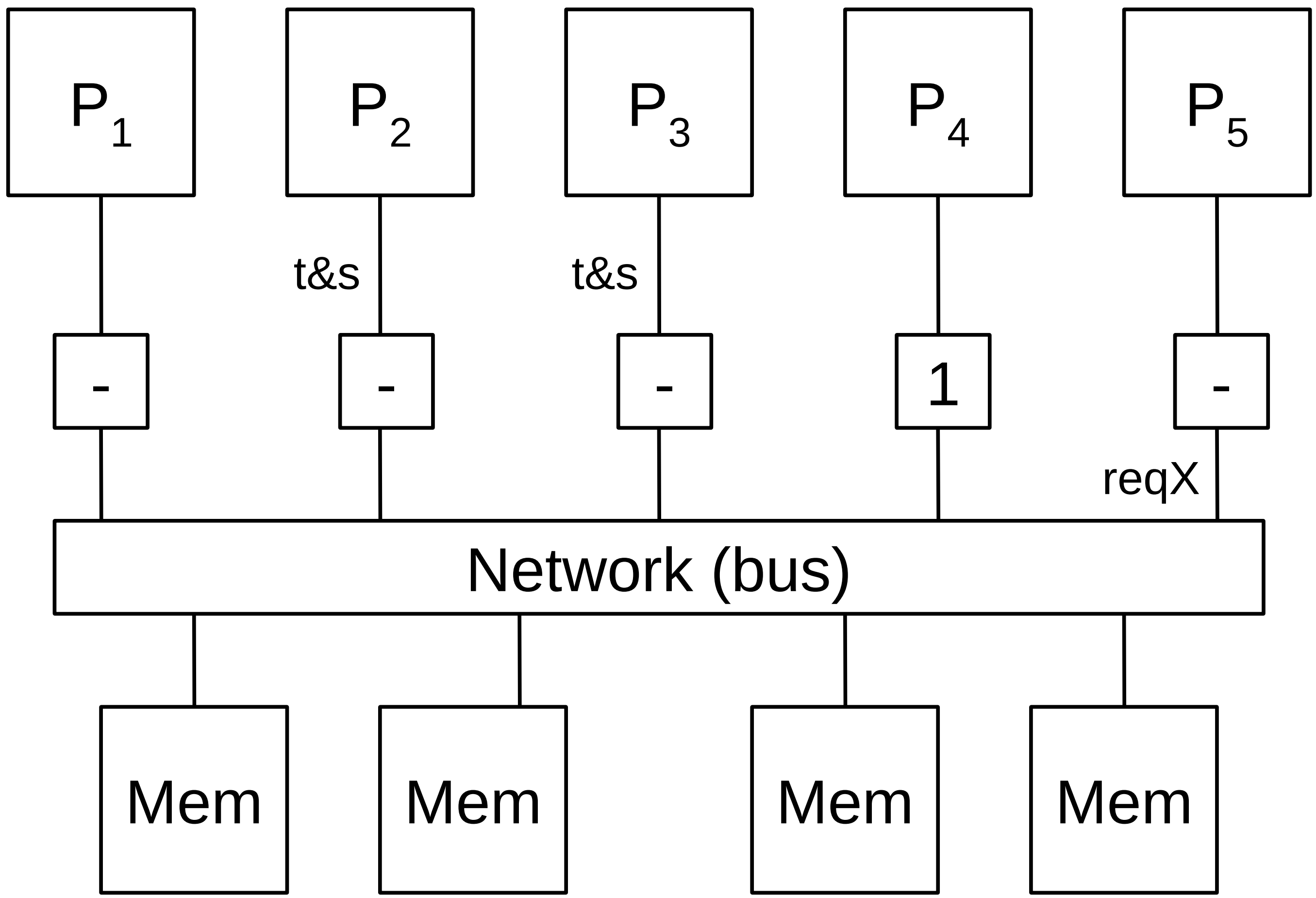
Simple test&set lock



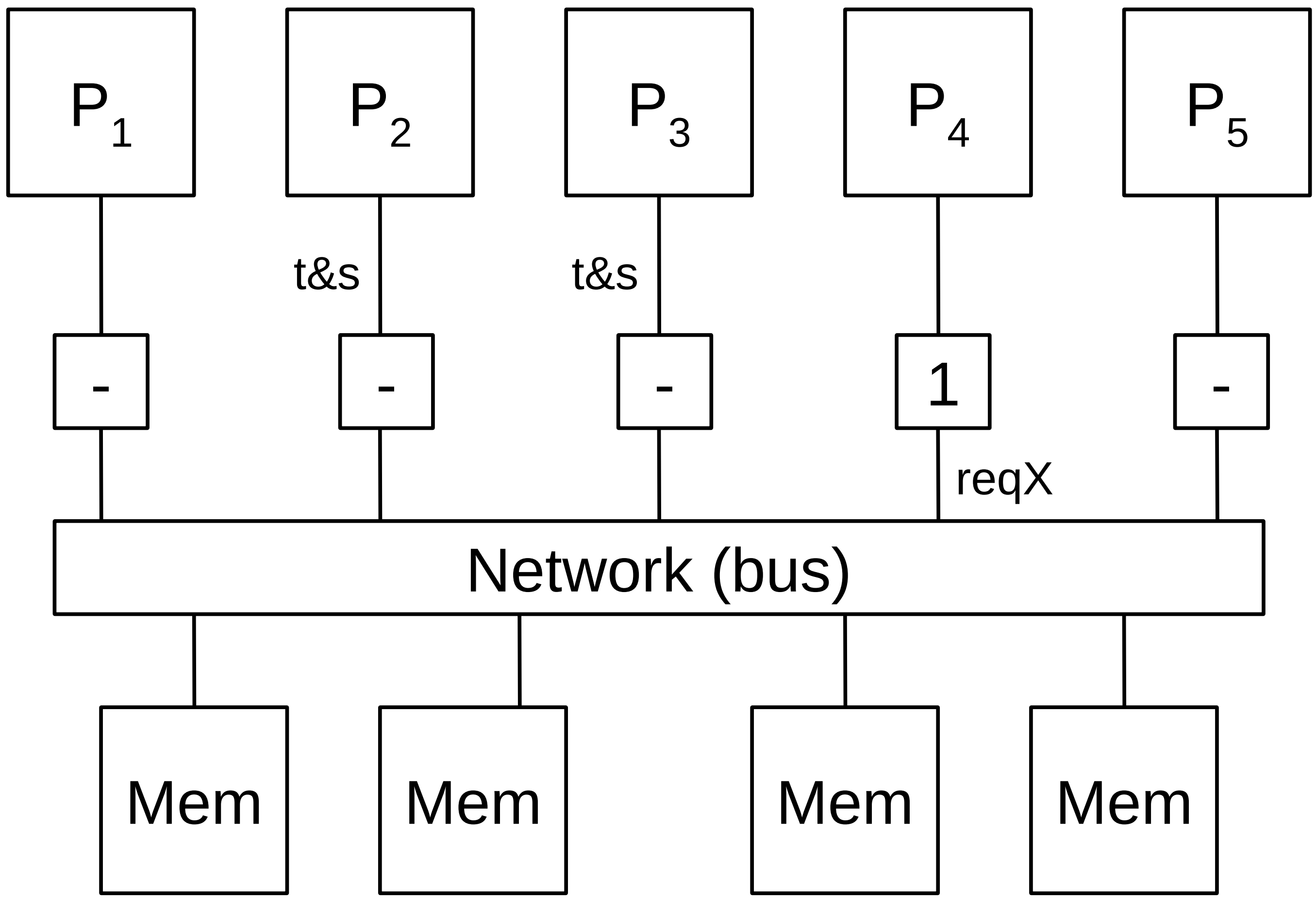
Simple test&set lock



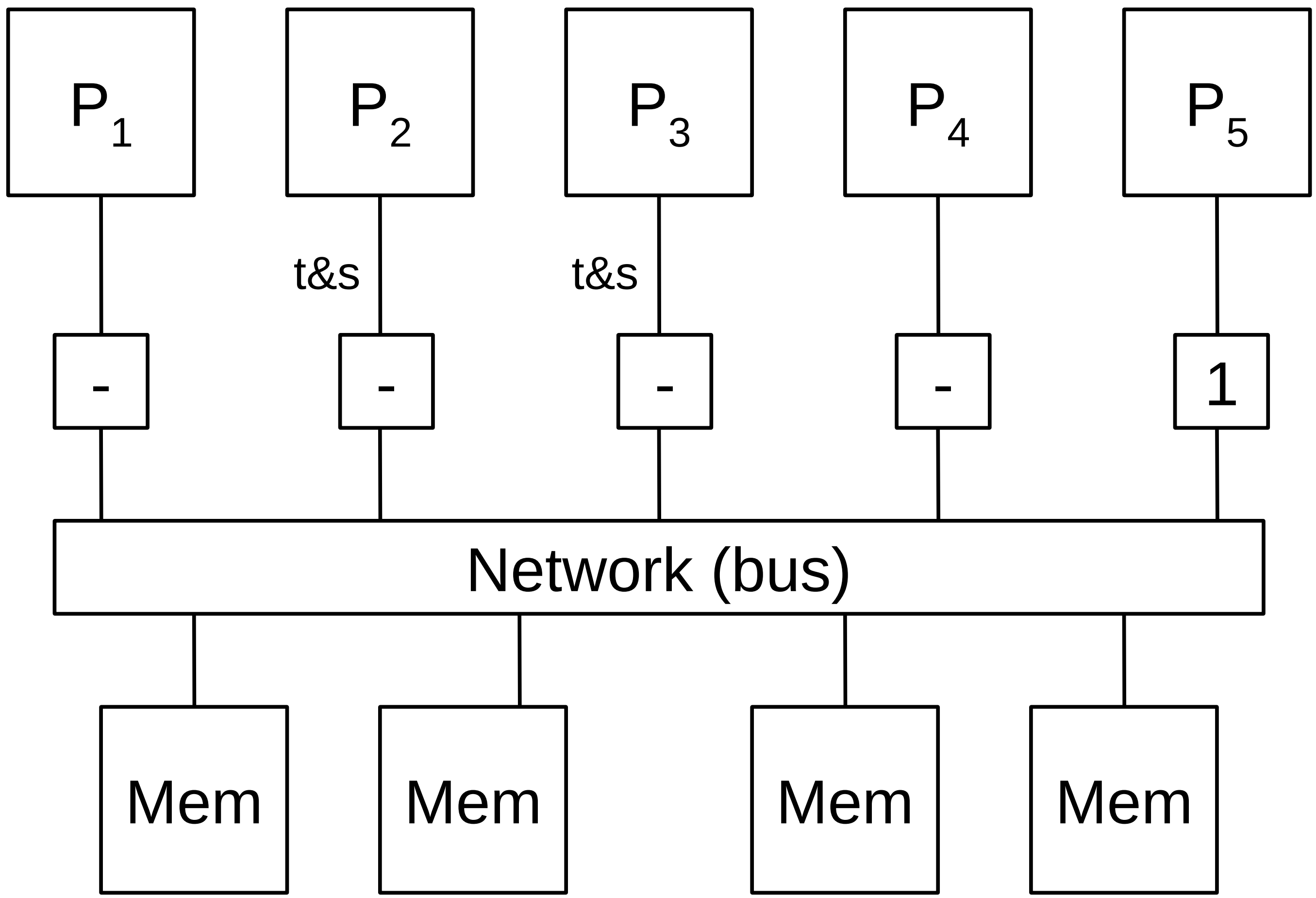
Simple test&set lock



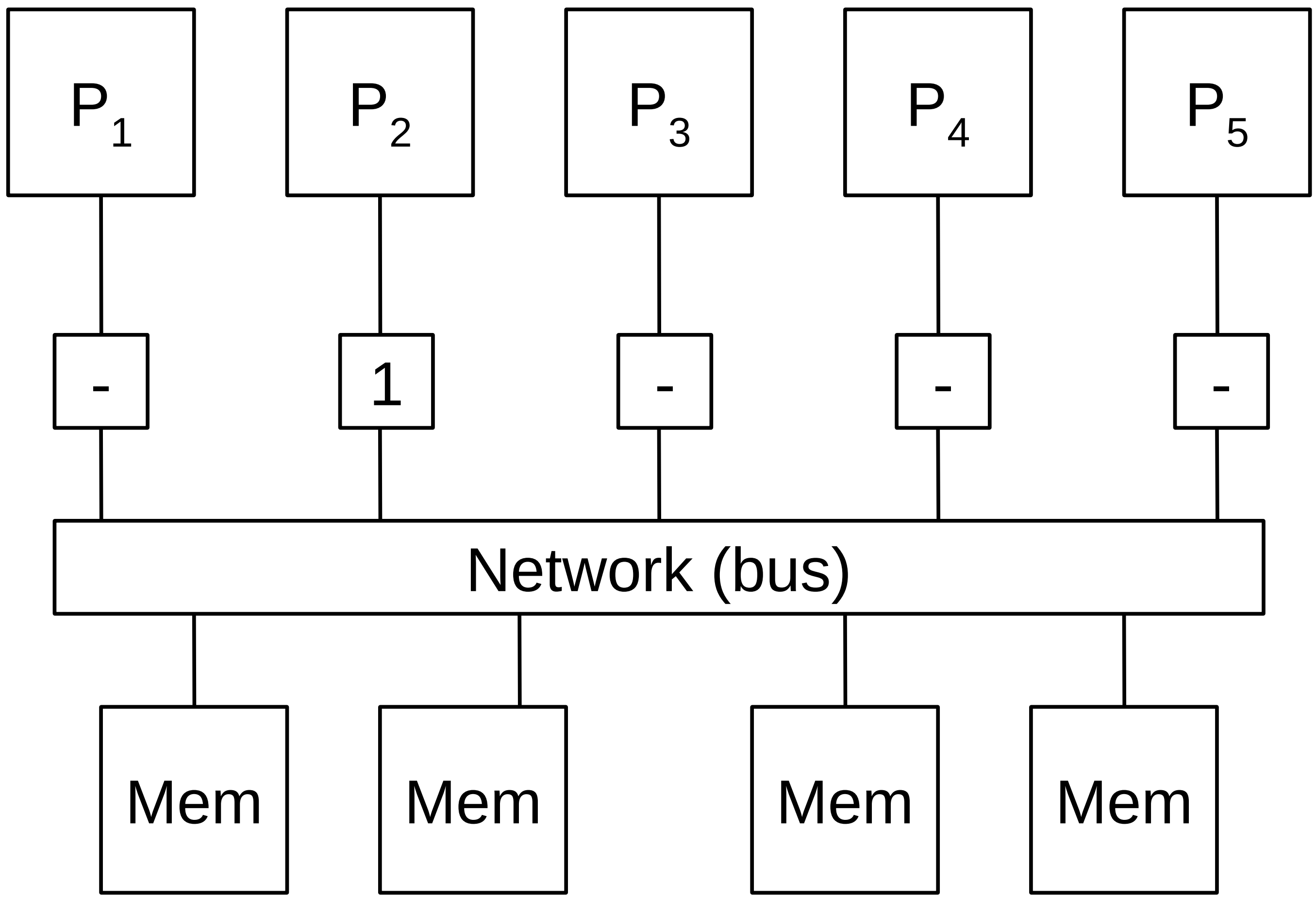
Simple test&set lock



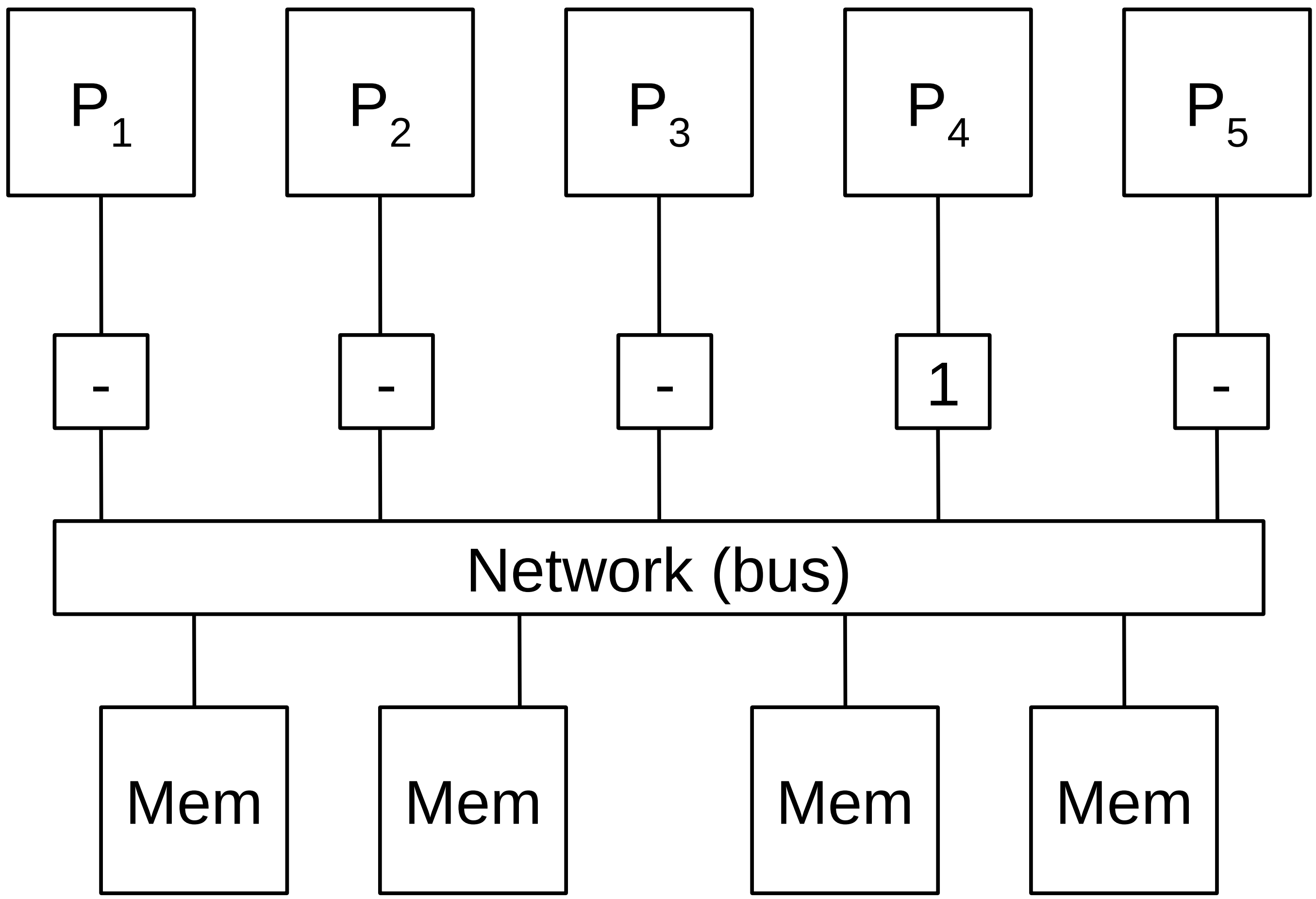
Simple test&set lock



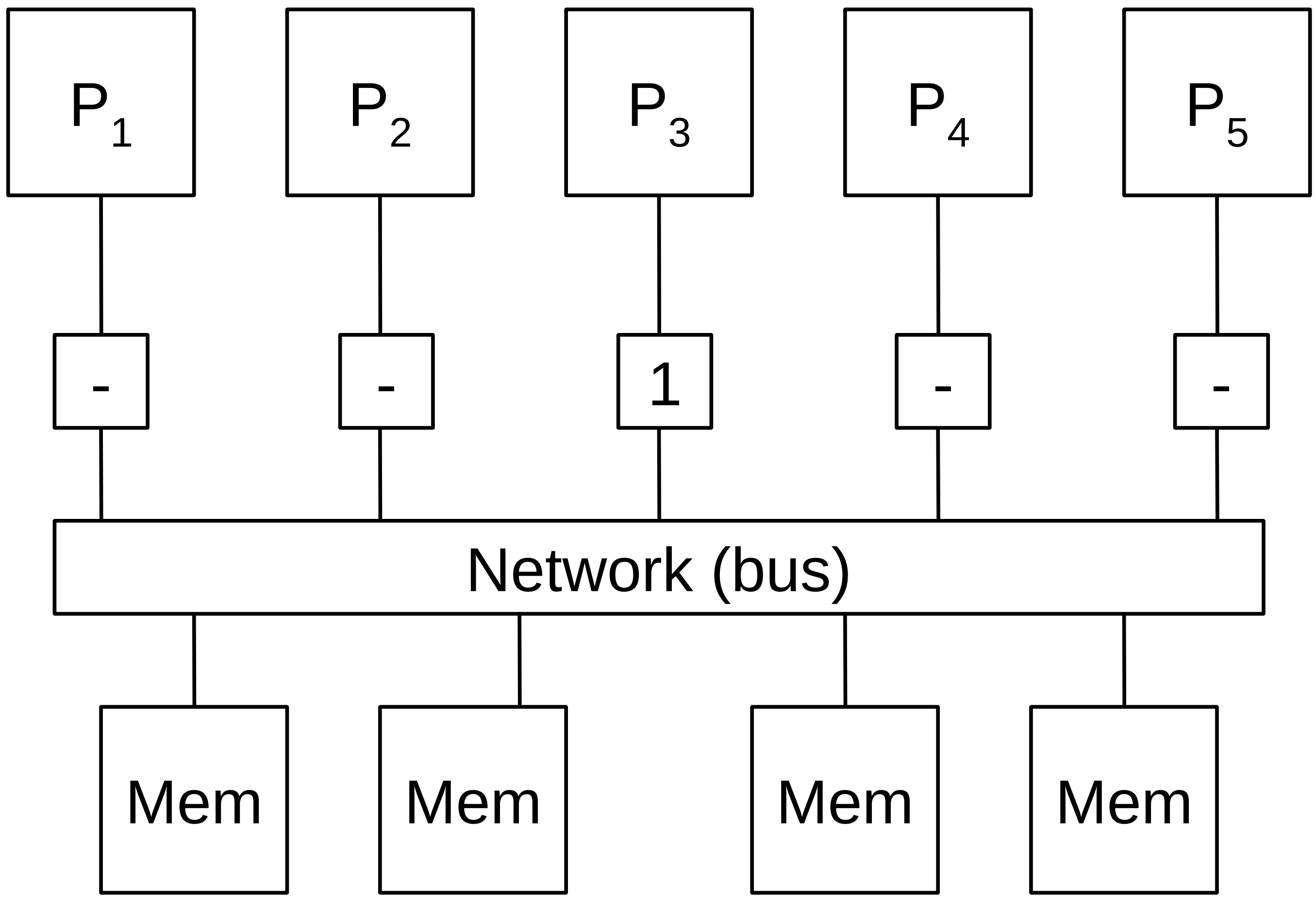
Simple test&set lock



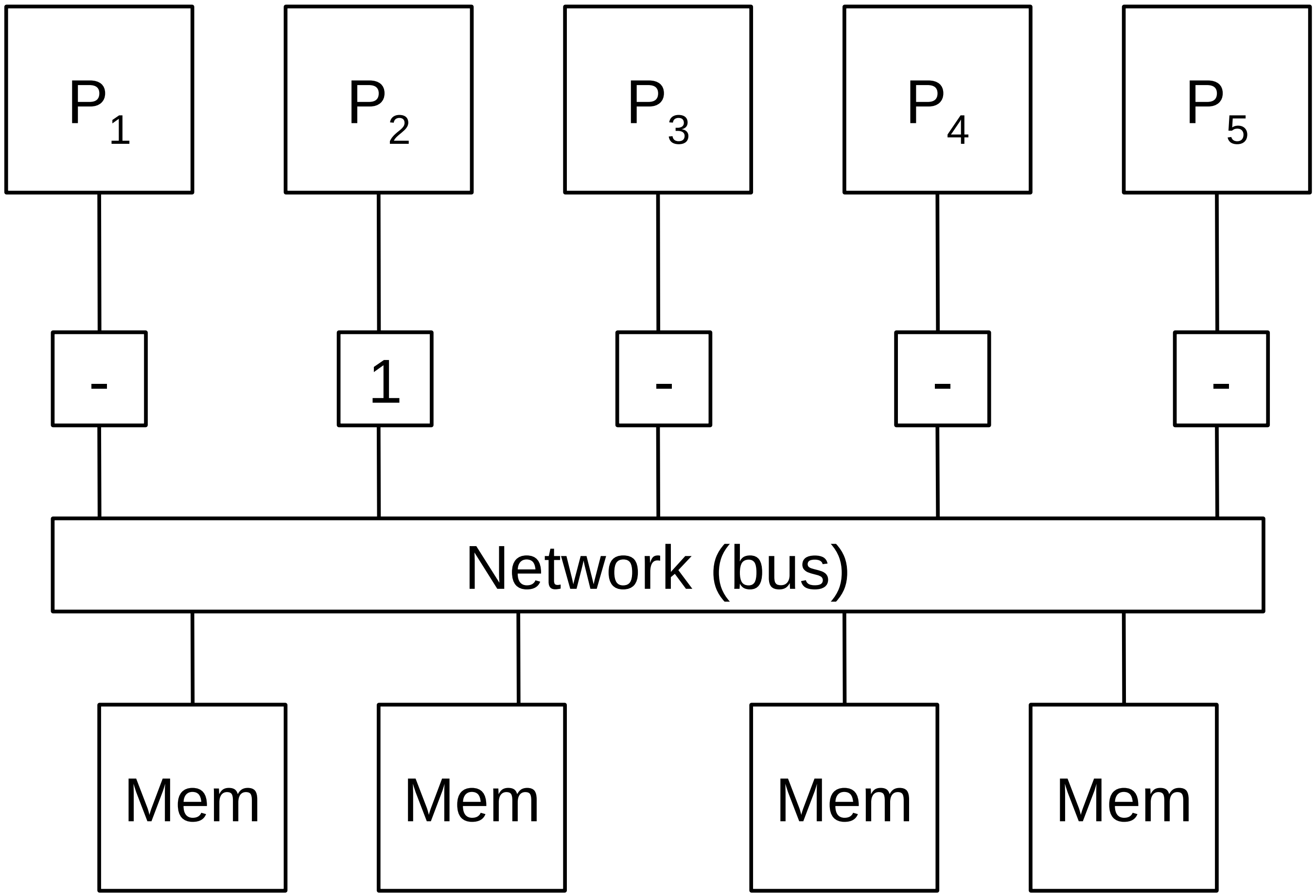
Simple test&set lock



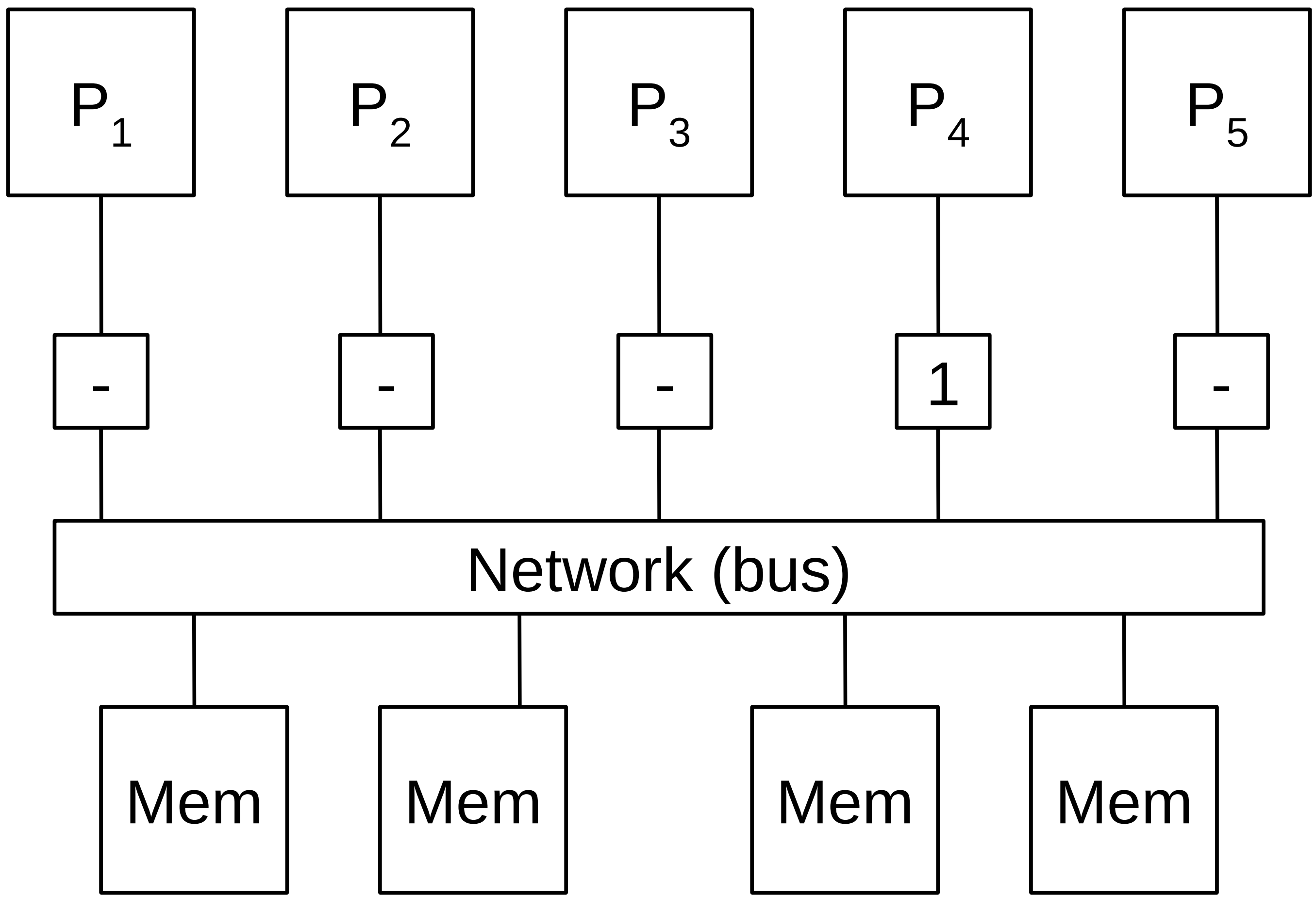
Simple test&set lock



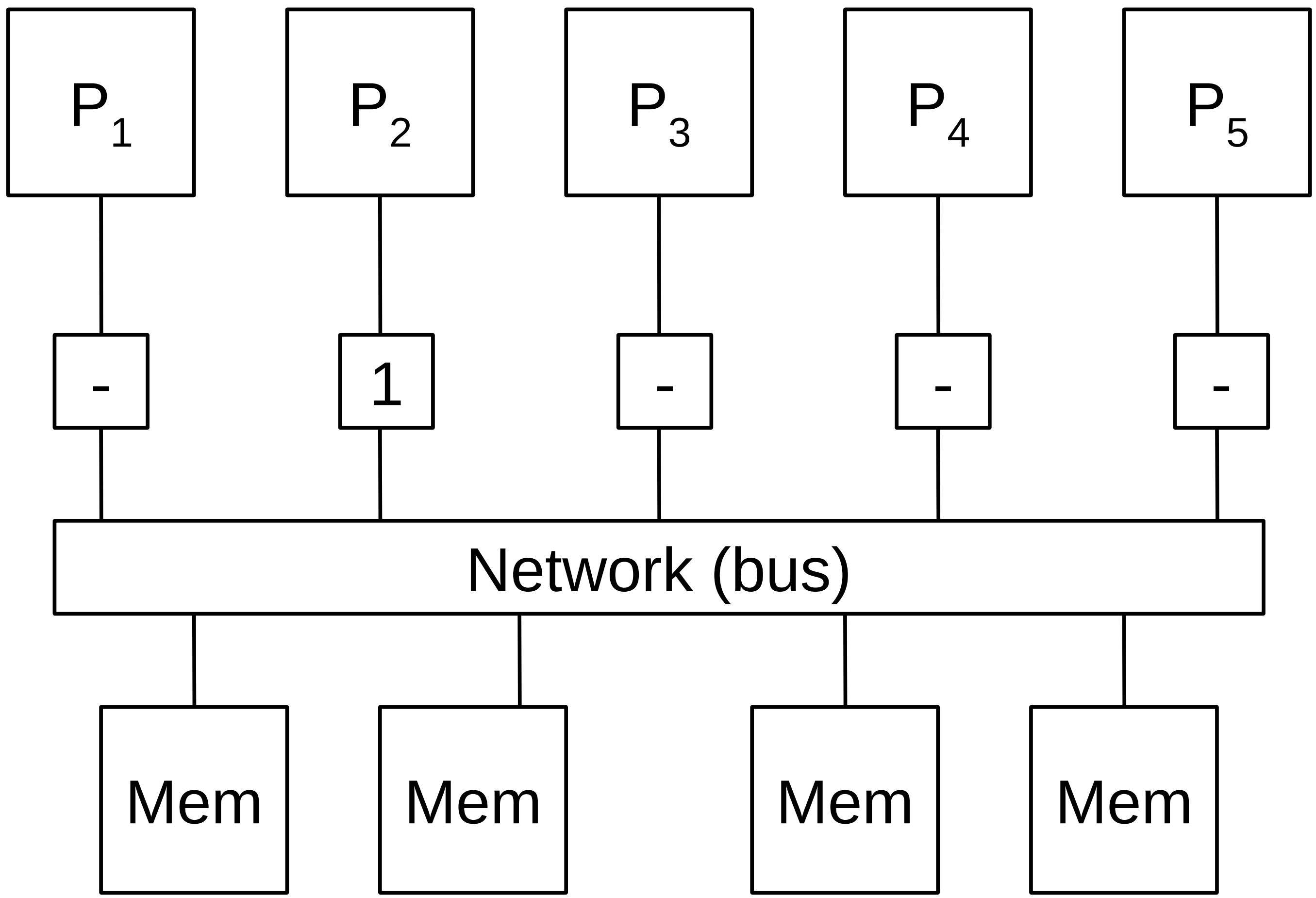
Simple test&set lock



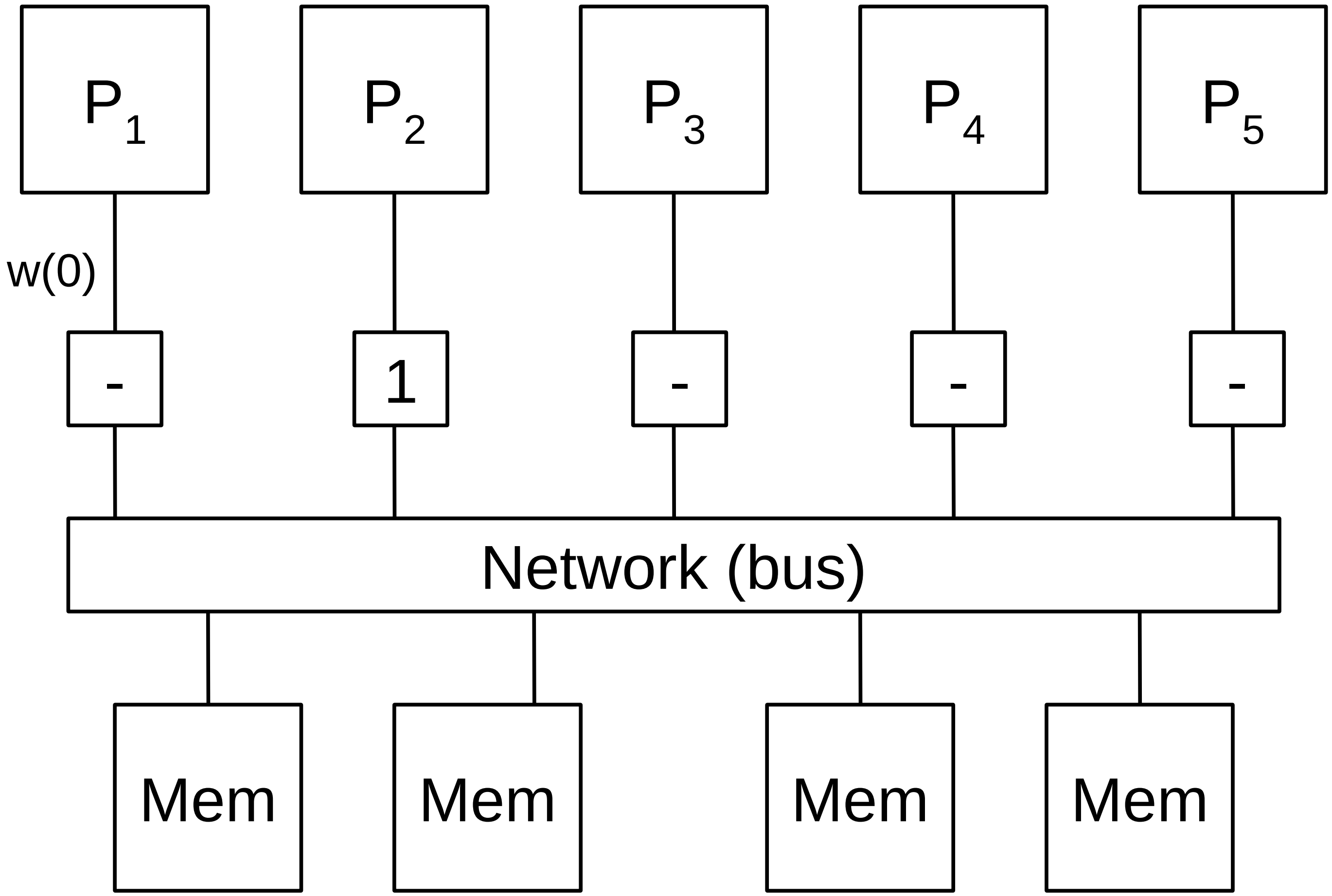
Simple test&set lock



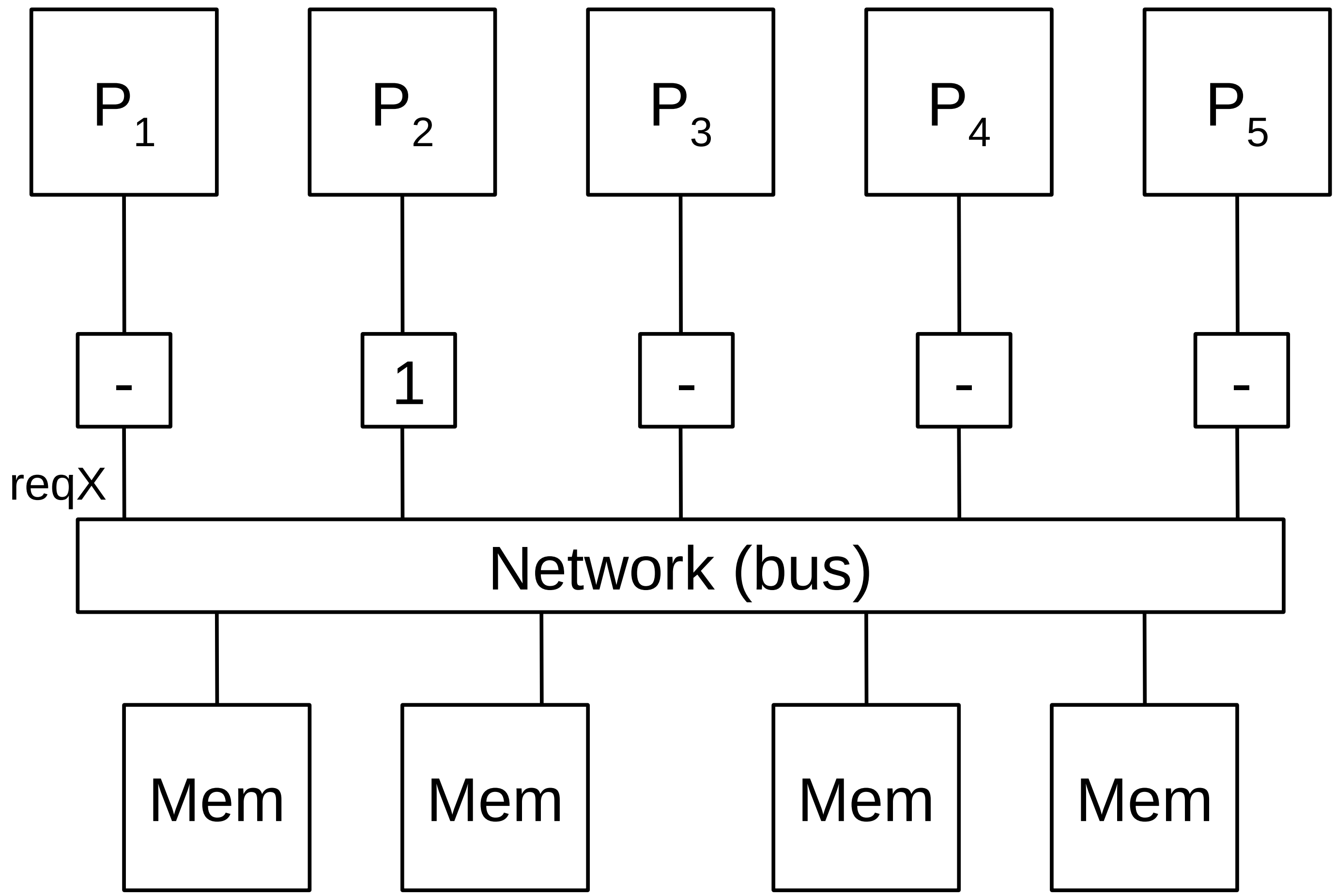
Simple test&set lock



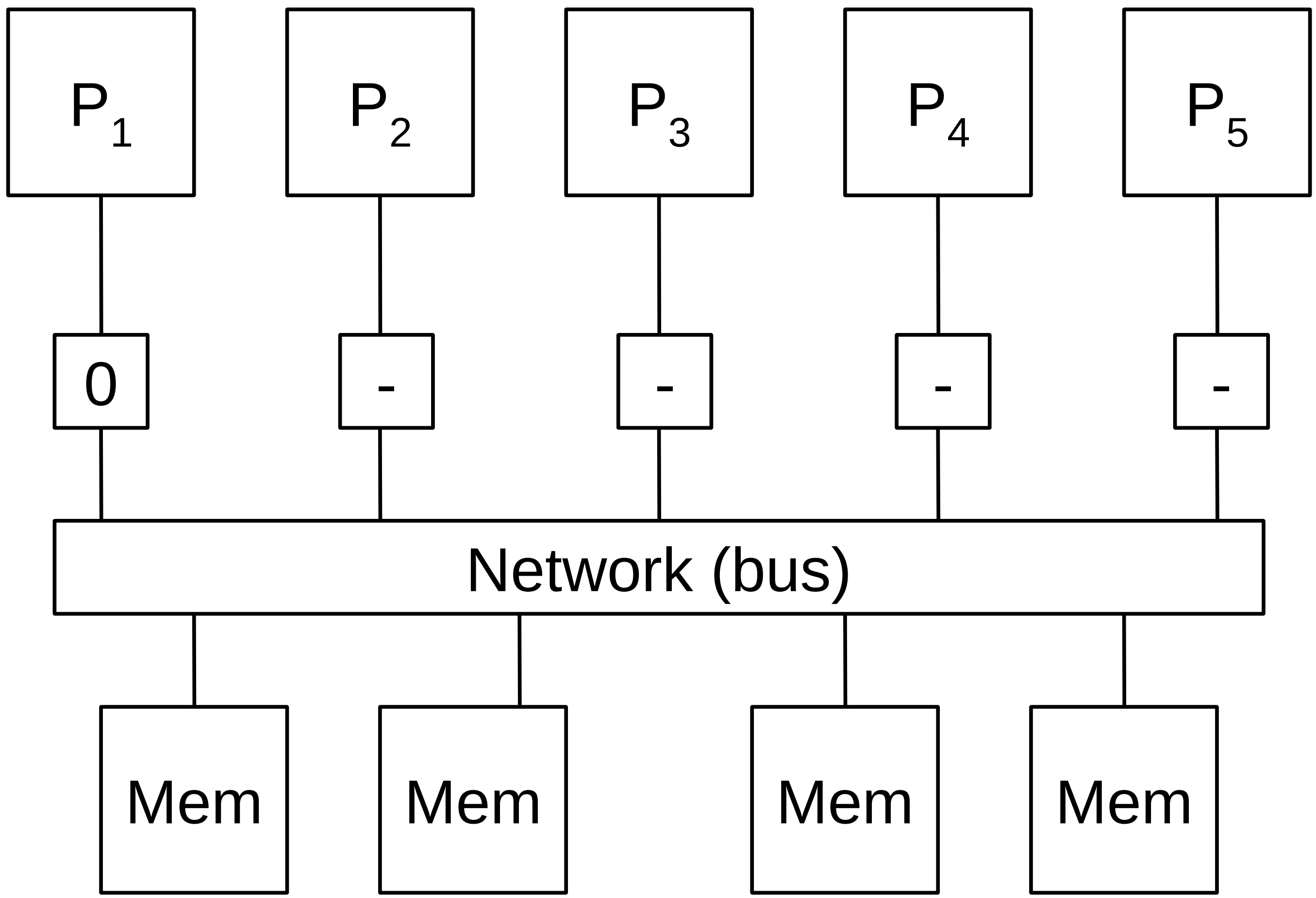
Simple test&set lock



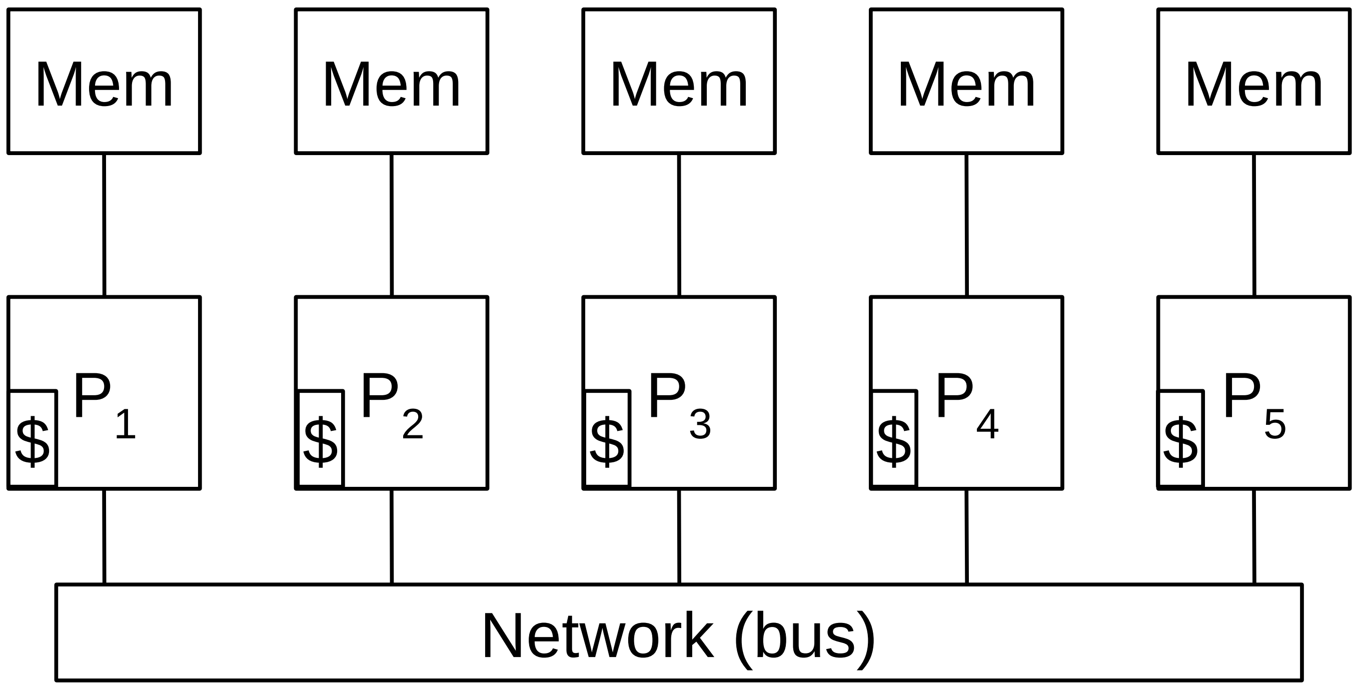
Simple test&set lock



Simple test&set lock



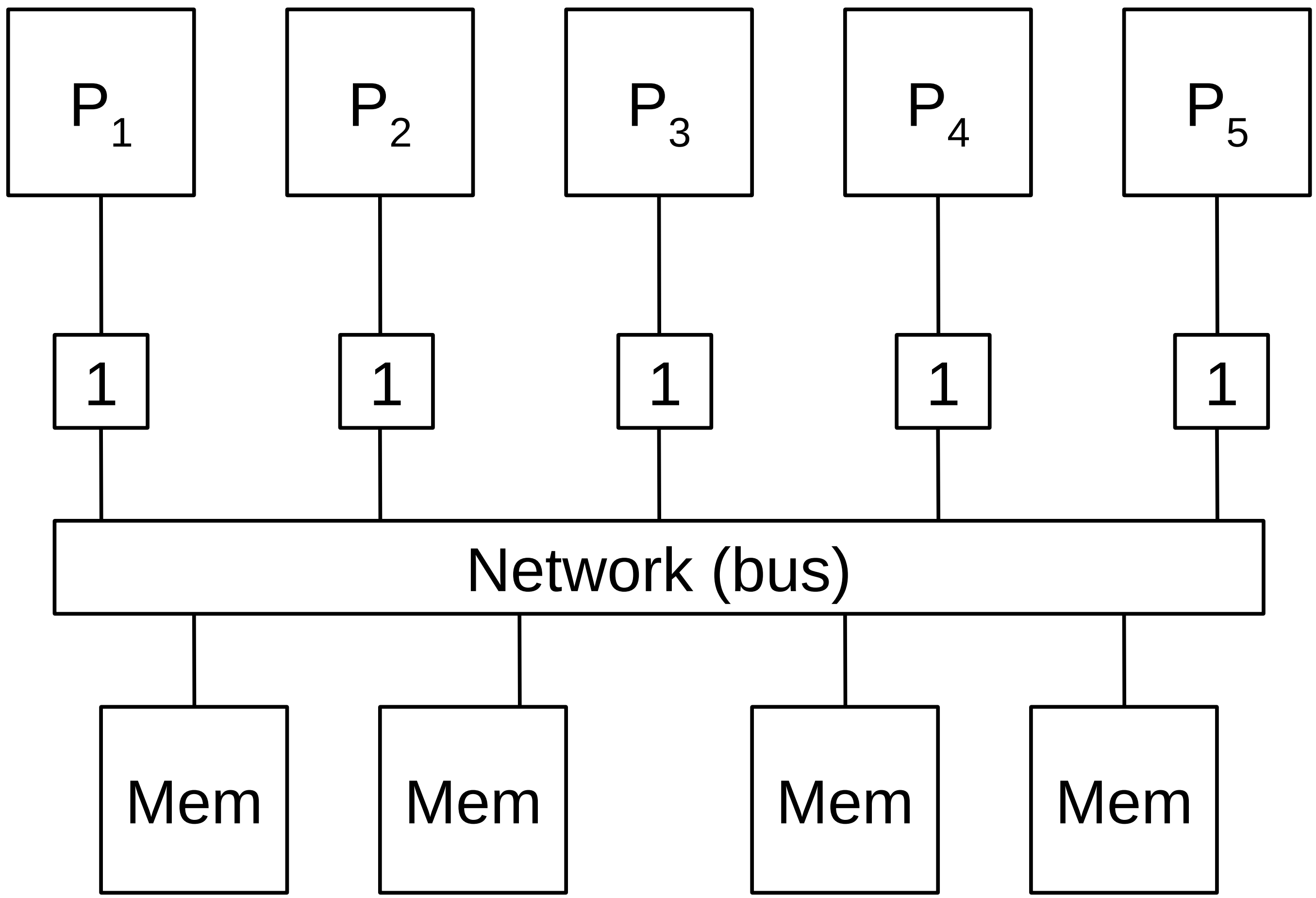
Simple test&set lock



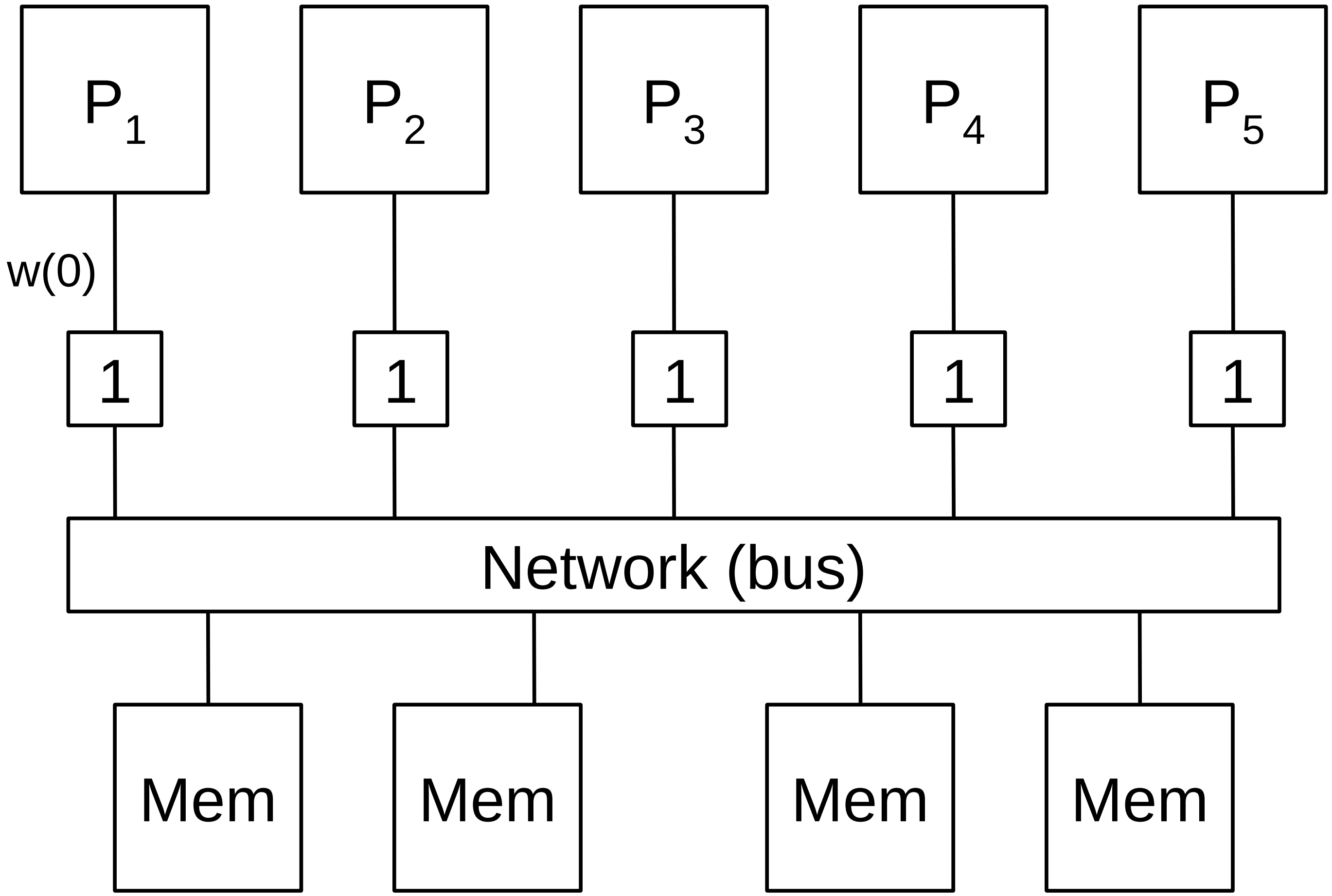
Test-and-test&set lock

- To help cope with high contention.
- Test-and-test&set:
  - First “test” (read).
  - Then, if the value is favorable (0), attempt test&set.
- Reduces network traffic (but it's still high!).

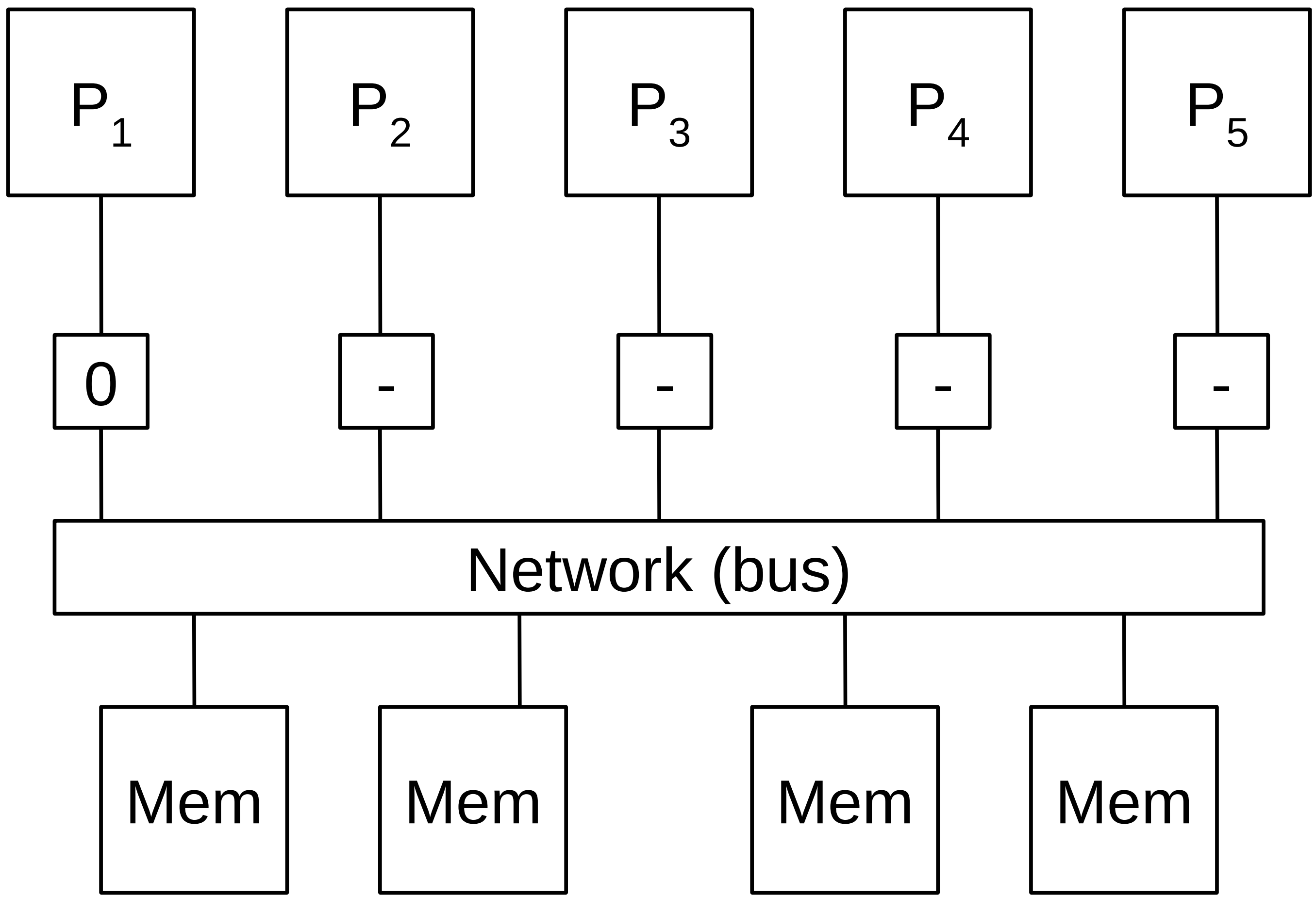
Test-and-test&set lock



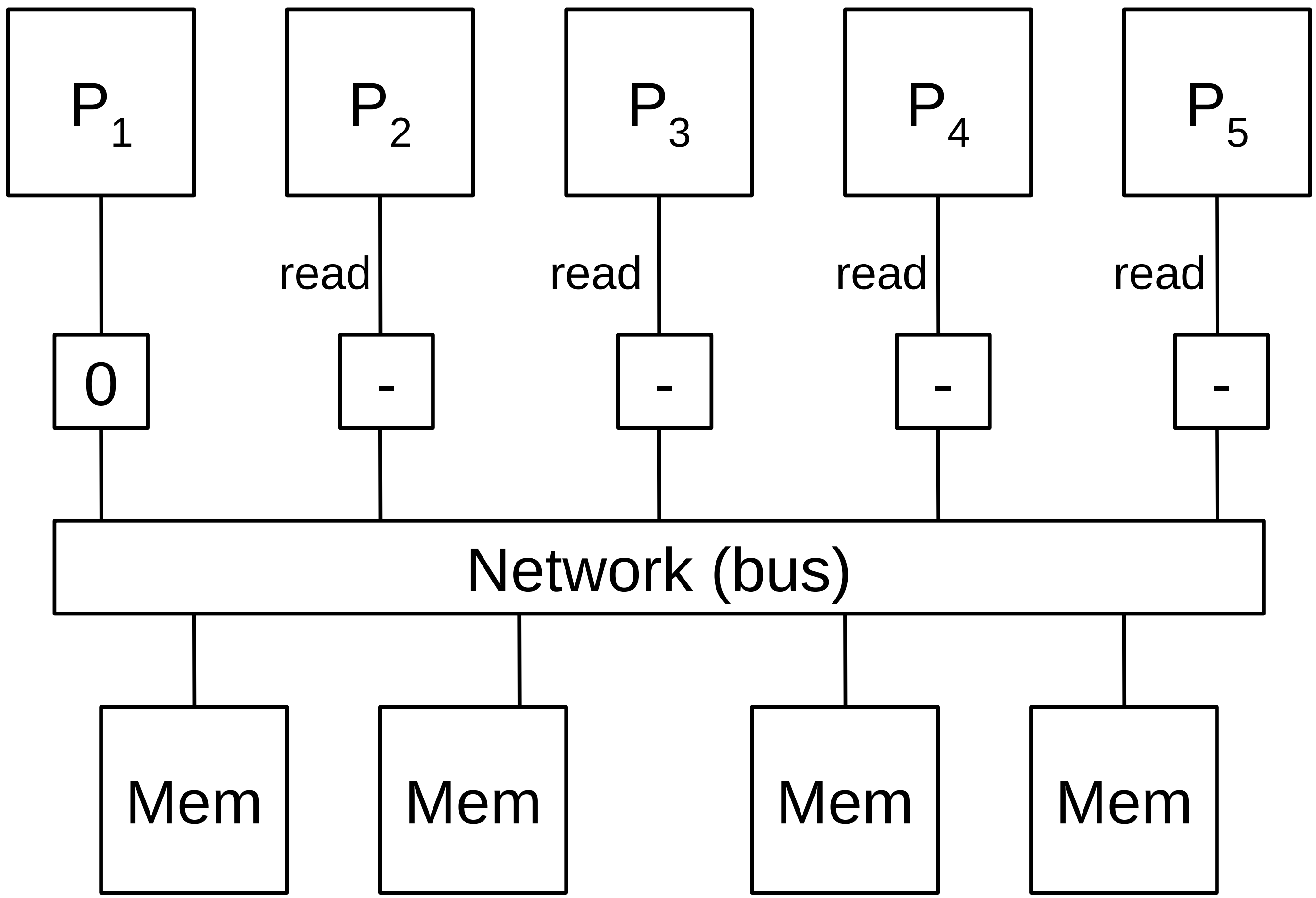
Test-and-test&set lock



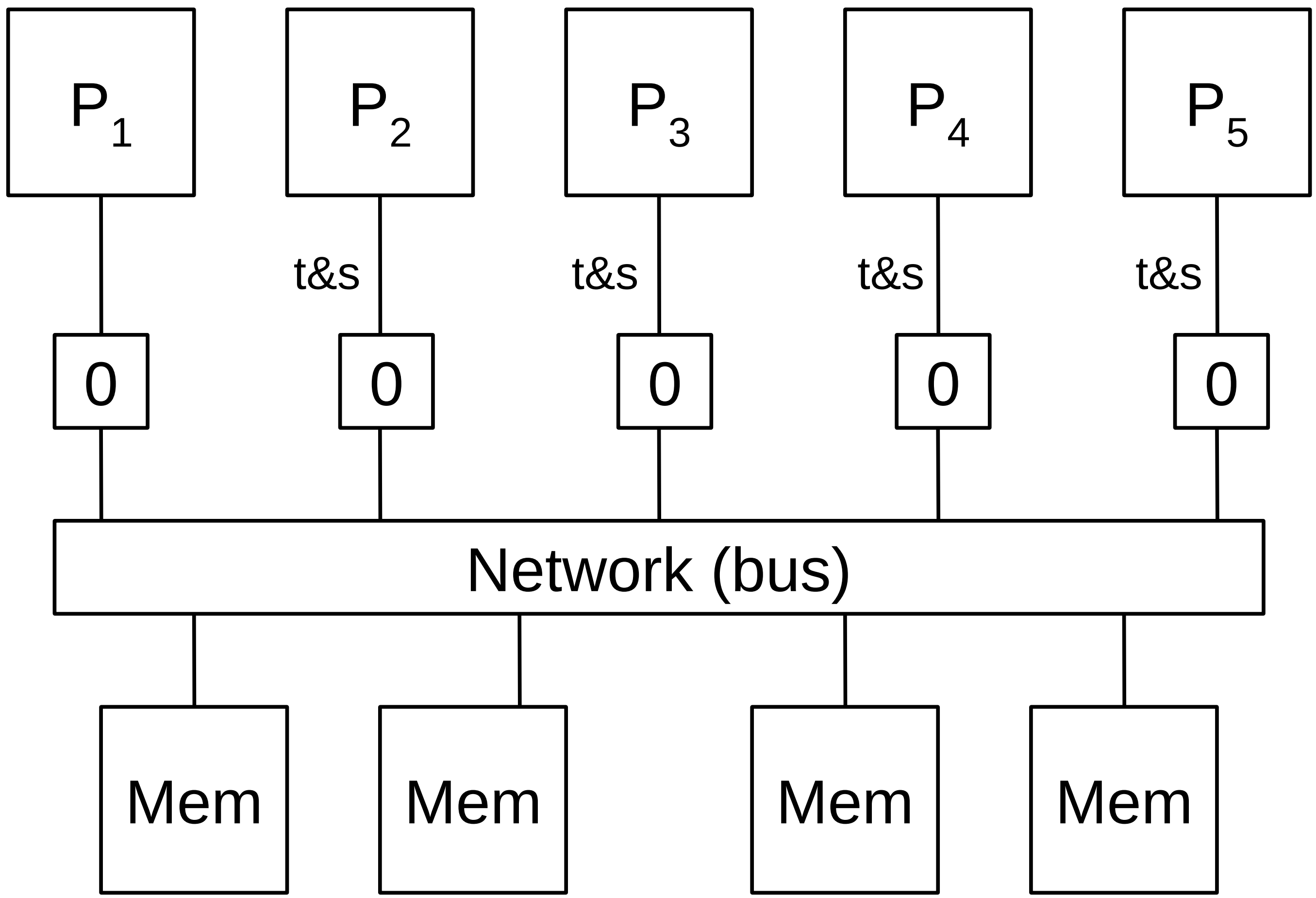
Test-and-test&set lock



Test-and-test&set lock



Test-and-test&set lock



Simple Test&Set lock with backoff

- More help coping with high contention.
- Recall: Test-and-test&set
  - Read before attempting Test&Set
  - Reduces network traffic.
  - But it's still high---especially when a cascade of requests arrives just after the lock is released.
- Test&Set with backoff
  - If Test&Set "fails" (returns 1), wait before trying again.
    - Makes success more likely.
    - Reduces network traffic (both read and write).
  - Exponential backoff seems to work best.
  - Obviates need for Test-and-test&set.

Ticket lock

next: integer; initially 0
granted: integer; initially 0

try_i
  ticket := f&i(**next**)
  waitfor(**granted** = ticket)
$crit_i$

$exit_i$
  f&i(**granted**)
rem_i

- Simple, low space cost, no bypass.
- Network traffic similar to Test-and-test&set (why?)
  - Not quite as bad, though.
- Can augment with backoff.
  - Proportional backoff seems best: delay depends on difference between ticket and granted.
  - Could introduce extra delays.

Queue Locks

- Processes form a FIFO queue.
  - Provides first-come first-serve fairness.
- Each process learns if its turn has arrived by checking whether its predecessor has finished.
  - Predecessor can notify the process when to check.
  - Improves utilization of the critical section.
- Each process spins on a different location.
  - Reduces invalidation traffic.

Several queue locks

- Array-based:
  - Anderson’s lock.
  - Graunke and Thakkar’s lock (skip this).
- Link-list-based:
  - Mellor-Crummey and Scott
  - Craig, Landin, Hagensten

Anderson's array lock

slots: array[0..N-1] of { front, not_front };
  initially (front, not_front, not_front,..., not_front)
next_slot: integer; initially 0

try_i
  my_slot := f&i(**next_slot**)
  waitfor(**slots**[my_slot] = front)
$crit_i$

$exit_i$
  slots[my_slot] := not_front
  slots[my_slot+1] := front
rem_i

- Entries are either "front" or "not-front" (of queue).
  - Exactly one "front" (except for short interval in exit region).
- Tail of queue indicated by next_slot.
  - Queue is empty if next_slot contains front.
- Each process spins on its own slot, reducing invalidation traffic.

Anderson's array lock

slots: array[0..N-1] of { front, not_front };
initially (front, not_front, not_front,..., not_front)
next_slot: integer; initially 0

try_i
my_slot := f&i(**next_slot**)
waitfor(**slots**[my_slot] = front)
$crit_i$

$exit_i$
slots[my_slot] := not_front
slots[my_slot+1] := front
rem_i

- Each process spins on its own slot, reducing invalidation traffic.
- Technicality: Separate slots should use different cache lines, to avoid "false sharing".
- This code allows only N competitors ever. But Anderson allows wraparound:

Anderson's array lock

slots: array[0..N-1] of { front, not_front };
initially (front, not_front, not_front,..., not_front)
next_slot: integer; initially 0

try_i
my_slot := f&i(**next_slot**)
if my_slot mod N = 0
atomic_add(**next_slot**, -N)
my_slot := my_slot mod N
waitfor(**slots**[my_slot] = front)
$crit_i$

$exit_i$
slots[my_slot] := not_front
slots[my_slot+1 mod N] :=
front
rem_i

- Wraps around to allow reuse of array entries.
- Still only N of competing processes at one time.
- High space cost: One location per lock per process.

Mellor-Crummey/Scott queue lock

- "…probably the most influential practical mutual exclusion algorithm of all time." ---2006 Dijkstra Prize citation
- Each process has its own "node".
  - Spins only on its own node, locally.
  - Others may write its node.
- Small space requirements.
  - Can "reuse" nodes for different locks.
  - Space overhead: O(L+N), for L locks and N processes, assuming each process accesses only one lock at a time.
  - Can allocate nodes as needed (typically upon process creation).
- May spin on exit.

Mellor-Crummey/Scott lock

node: array[1..N] of [next: 0..N, wait: Boolean]; initially arbitrary
tail: 0..N; initially 0

try_i
  node[i].next := 0
  pred := swap(**tail**,i)
  if pred ≠ 0
    node[i].wait := true
    node[pred].next := i
    waitfor(¬**node**[i].wait)
$crit_i$

$exit_i$
  if **node**[i].next = 0
    if CAS(**tail**,i,0) return
    waitfor(**node**[i].next ≠ 0)
  node[**node**[i].next].wait := false
rem_i

- Use array to model nodes.
- CAS: Change value, return true if expected value found.

Mellor-Crummey/Scott lock

try_i

  node[i].next := 0

  pred := swap(**tail**,i)

  if pred ≠ 0

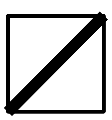
    node[i].wait := true

    node[pred].next := i

    waitfor(¬**node**[i].wait)

$crit_i$

tail



$exit_i$

  if **node**[i].next = 0

    if CAS(**tail**,i,0) return

    waitfor(**node**[i].next ≠ 0)

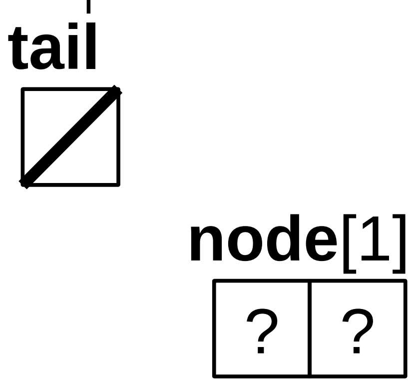
  node[**node**[i].next].wait := false

rem_i

Mellor-Crummey/Scott lock

try$_i$
node[i].next := 0
pred := swap(**tail**,i)
if pred ≠ 0
node[i].wait := true
node[pred].next := i
waitfor(¬**node**[i].wait)
crit$_i$

exit$_i$
if **node**[i].next = 0
if CAS(**tail**,i,0) return
waitfor(**node**[i].next ≠ 0)
node[**node**[i].next].wait := false
rem$_i$



Mellor-Crummey/Scott lock

try_i

  node[i].next := 0

  pred := swap(**tail**,i)

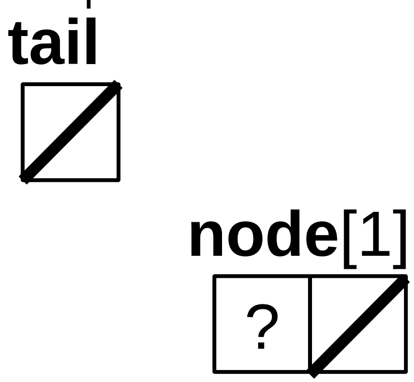
  if pred ≠ 0

    node[i].wait := true

    node[pred].next := i

    waitfor(¬**node**[i].wait)

crit_i



exit_i

  if **node**[i].next = 0

    if CAS(**tail**,i,0) return

    waitfor(**node**[i].next ≠ 0)

  node[**node**[i].next].wait := false

rem_i

Mellor-Crummey/Scott lock

try_i

node[i].next := 0

pred := swap(**tail**,i)

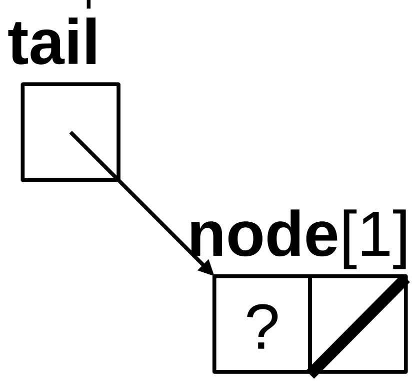
if pred ≠ 0

node[i].wait := true

node[pred].next := i

waitfor(¬**node**[i].wait)

$crit_i$



$exit_i$

if **node**[i].next = 0

if CAS(**tail**,i,0) return

waitfor(**node**[i].next ≠ 0)

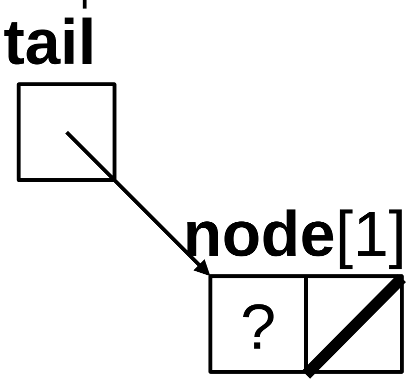
node[**node**[i].next].wait := false

rem_i

Mellor-Crummey/Scott lock

try_i
  node[i].next := 0
  pred := swap(**tail**,i)
  if pred ≠ 0
    node[i].wait := true
    node[pred].next := i
    waitfor(¬**node**[i].wait)
$crit_i$

$exit_i$
  if **node**[i].next = 0
    if CAS(**tail**,i,0) return
    waitfor(**node**[i].next ≠ 0)
  node[**node**[i].next].wait := false
rem_i

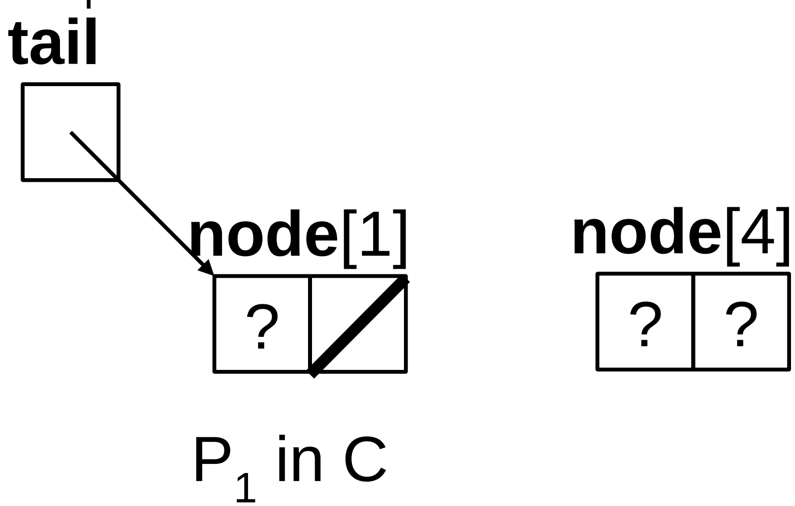


P_1 in C

Mellor-Crummey/Scott lock

try_i
  node[i].next := 0
  pred := swap(**tail**,i)
  if pred ≠ 0
    node[i].wait := true
    node[pred].next := i
    waitfor(¬**node**[i].wait)
$crit_i$

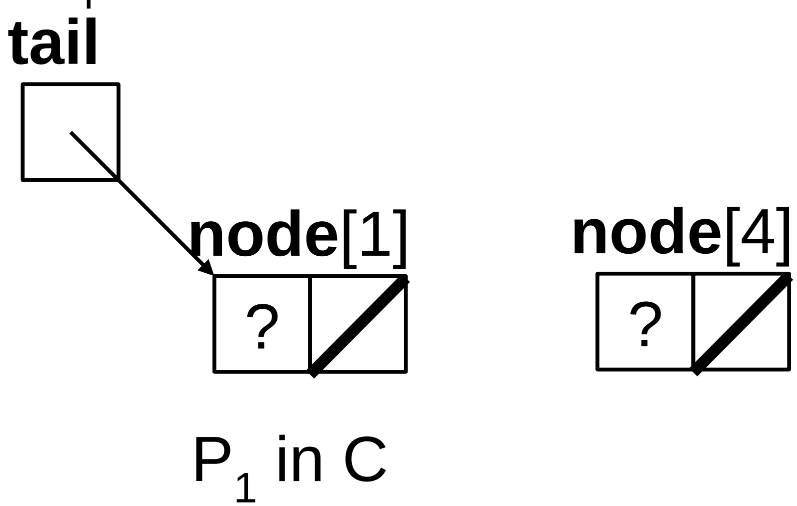
$exit_i$
  if **node**[i].next = 0
    if CAS(**tail**,i,0) return
    waitfor(**node**[i].next ≠ 0)
  node[**node**[i].next].wait := false
rem_i



Mellor-Crummey/Scott lock

```
try_i
  node[i].next := 0
  pred := swap(tail,i)
  if pred ≠ 0
    node[i].wait := true
    node[pred].next := i
    waitfor(¬node[i].wait)
crit_i
```

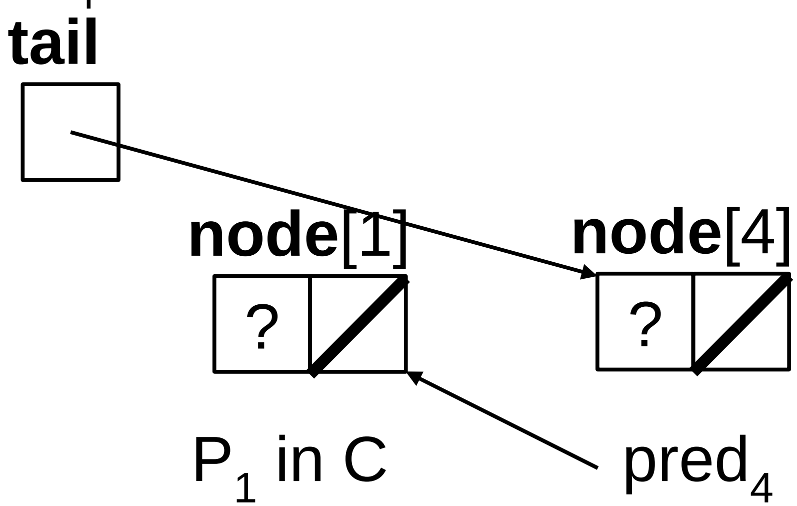
```
exit_i
  if node[i].next = 0
    if CAS(tail,i,0) return
    waitfor(node[i].next ≠ 0)
  node[node[i].next].wait := false
rem_i
```



Mellor-Crummey/Scott lock

try_i
  node[i].next := 0
  pred := swap(**tail**,i)
  if pred ≠ 0
    node[i].wait := true
    node[pred].next := i
    waitfor(¬**node**[i].wait)
$crit_i$

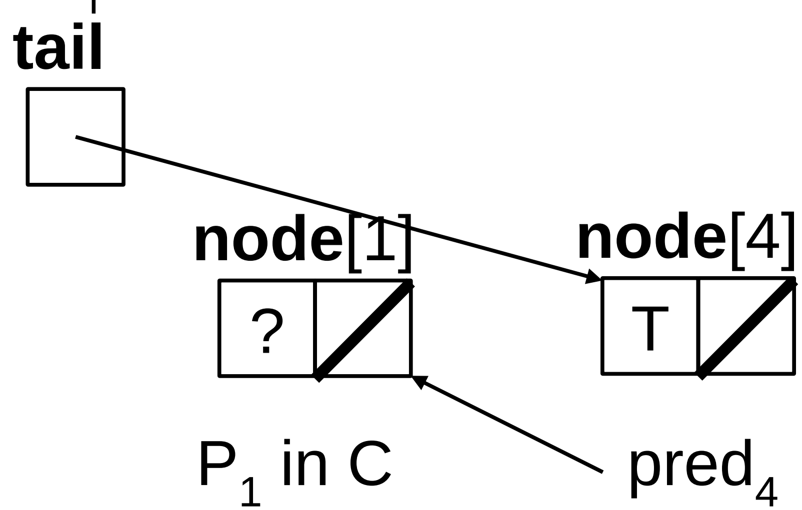
$exit_i$
  if **node**[i].next = 0
    if CAS(**tail**,i,0) return
    waitfor(**node**[i].next ≠ 0)
  node[**node**[i].next].wait := false
rem_i



Mellor-Crummey/Scott lock

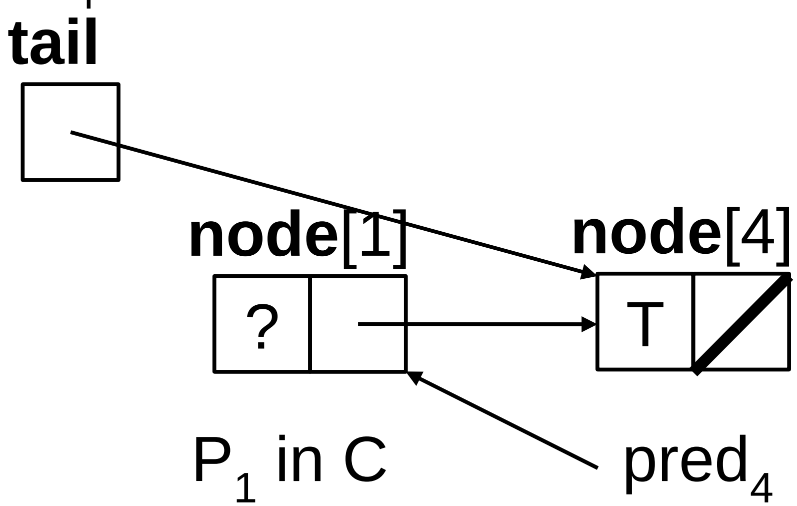
try_i
  node[i].next := 0
  pred := swap(**tail**,i)
  if pred ≠ 0
    node[i].wait := true
    node[pred].next := i
    waitfor(¬**node**[i].wait)
$crit_i$

$exit_i$
  if **node**[i].next = 0
    if CAS(**tail**,i,0) return
    waitfor(**node**[i].next ≠ 0)
  node[**node**[i].next].wait := false
rem_i



Mellor-Crummey/Scott lock

try_i
  node[i].next := 0
  pred := swap(**tail**,i)
  if pred ≠ 0
    node[i].wait := true
    node[pred].next := i
    waitfor(¬**node**[i].wait)
$crit_i$

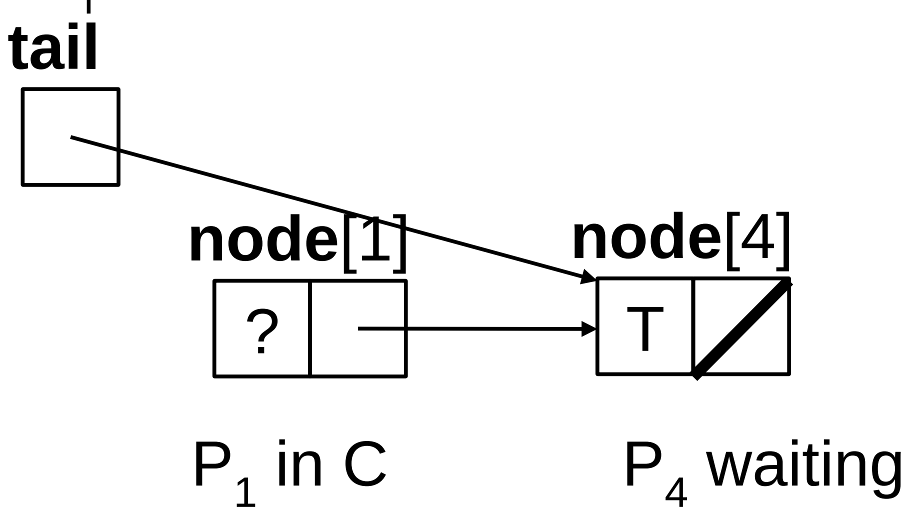


$exit_i$
  if **node**[i].next = 0
    if CAS(**tail**,i,0) return
    waitfor(**node**[i].next ≠ 0)
  node[**node**[i].next].wait := false
rem_i

Mellor-Crummey/Scott lock

try_i
  node[i].next := 0
  pred := swap(**tail**,i)
  if pred ≠ 0
    node[i].wait := true
    node[pred].next := i
    waitfor(¬**node**[i].wait)
$crit_i$

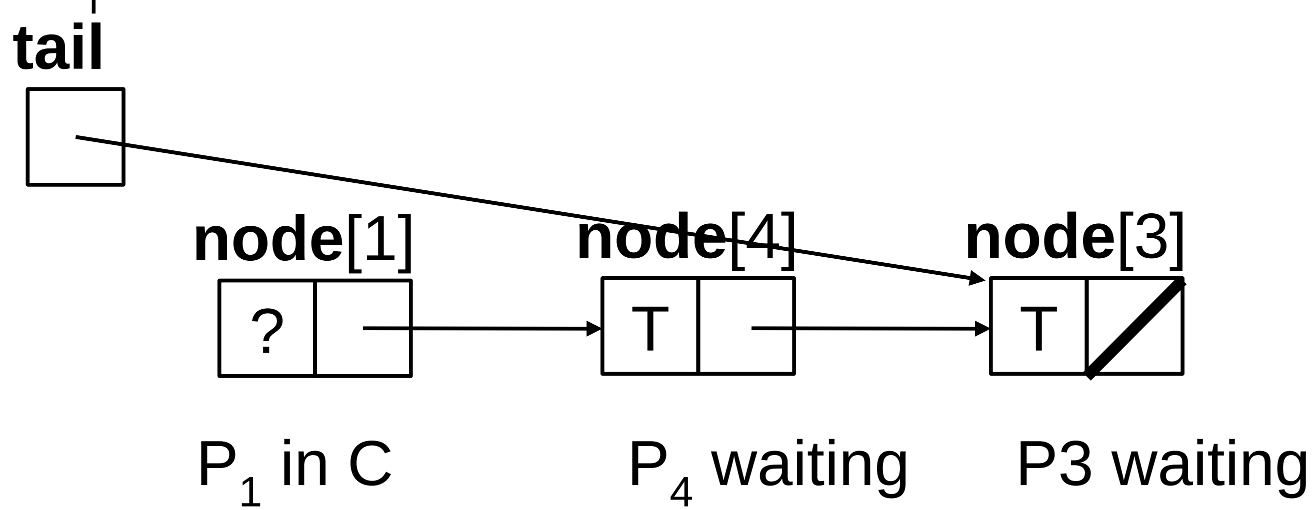
$exit_i$
  if **node**[i].next = 0
    if CAS(**tail**,i,0) return
    waitfor(**node**[i].next ≠ 0)
  node[**node**[i].next].wait := false
rem_i



Mellor-Crummey/Scott lock

try_i
  node[i].next := 0
  pred := swap(**tail**,i)
  if pred ≠ 0
    node[i].wait := true
    node[pred].next := i
    waitfor(¬**node**[i].wait)
$crit_i$

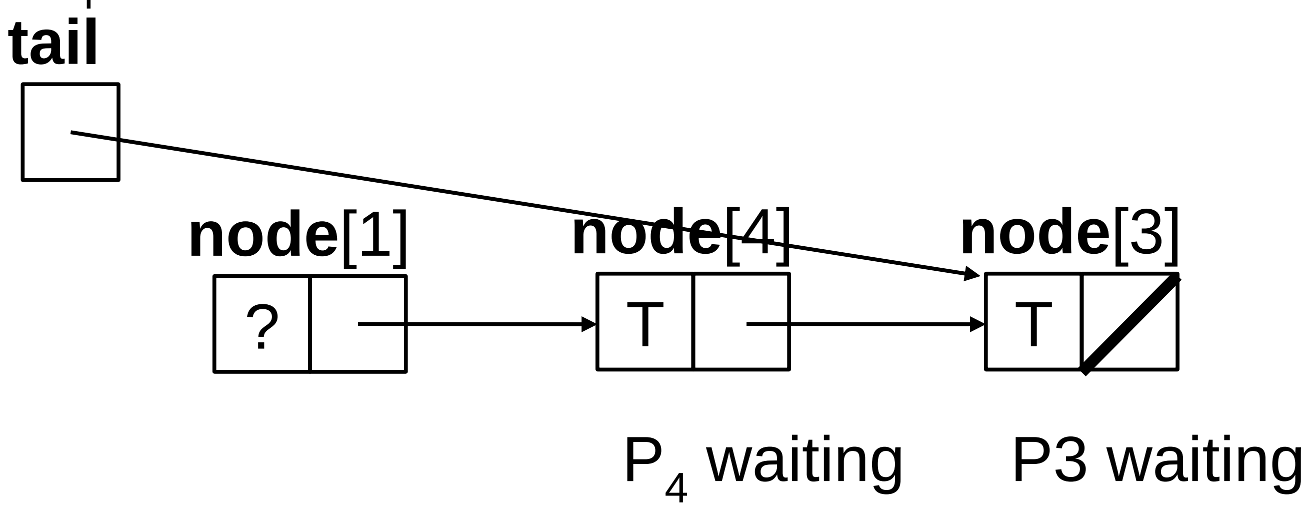
$exit_i$
  if **node**[i].next = 0
    if CAS(**tail**,i,0) return
    waitfor(**node**[i].next ≠ 0)
  node[**node**[i].next].wait := false
rem_i



Mellor-Crummey/Scott lock

```
try_i
  node[i].next := 0
  pred := swap(tail,i)
  if pred ≠ 0
    node[i].wait := true
    node[pred].next := i
    waitfor(¬node[i].wait)
crit_i
```

```
exit_i
  if node[i].next = 0
    if CAS(tail,i,0) return
    waitfor(node[i].next ≠ 0)
  node[node[i].next].wait := false
rem_i
```



Mellor-Crummey/Scott lock

try_i

node[i].next := 0

pred := swap(**tail**,i)

if pred ≠ 0

node[i].wait := true

node[pred].next := i

waitfor(¬**node**[i].wait)

$crit_i$

$exit_i$

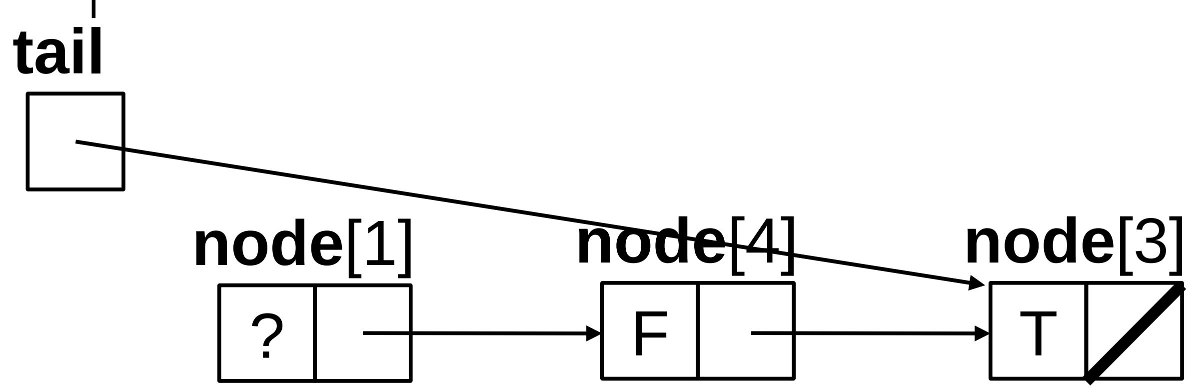
if **node**[i].next = 0

if CAS(**tail**,i,0) return

waitfor(**node**[i].next ≠ 0)

node[node[i].next].wait := false

rem_i

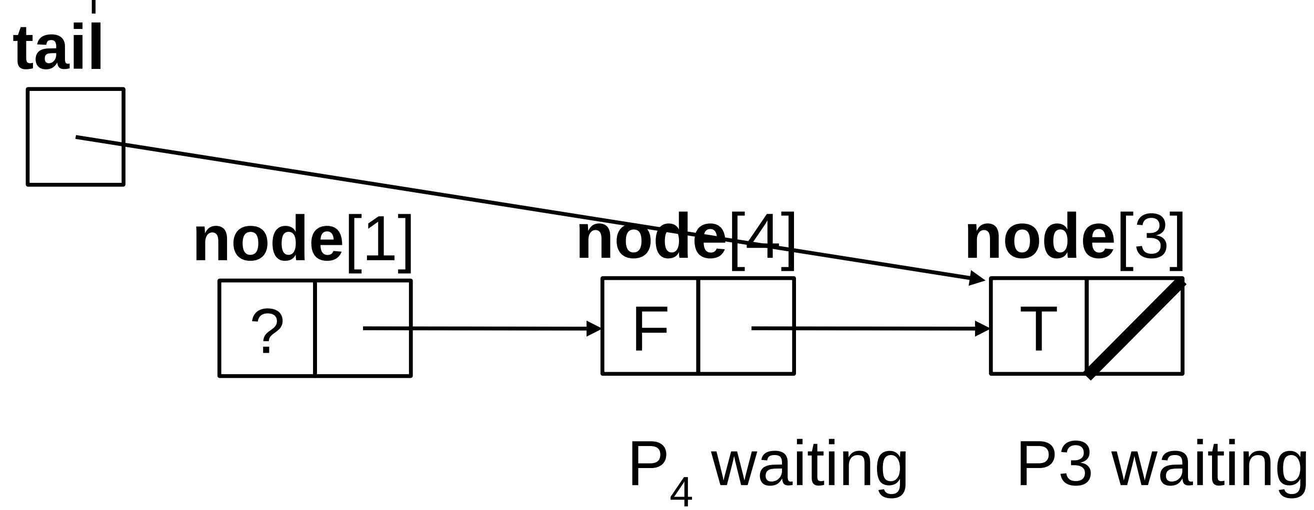


P_4 waiting P3 waiting

Mellor-Crummey/Scott lock

try$_i$
  node[i].next := 0
  pred := swap(**tail**,i)
  if pred ≠ 0
    node[i].wait := true
    node[pred].next := i
    waitfor(¬**node**[i].wait)
crit$_i$

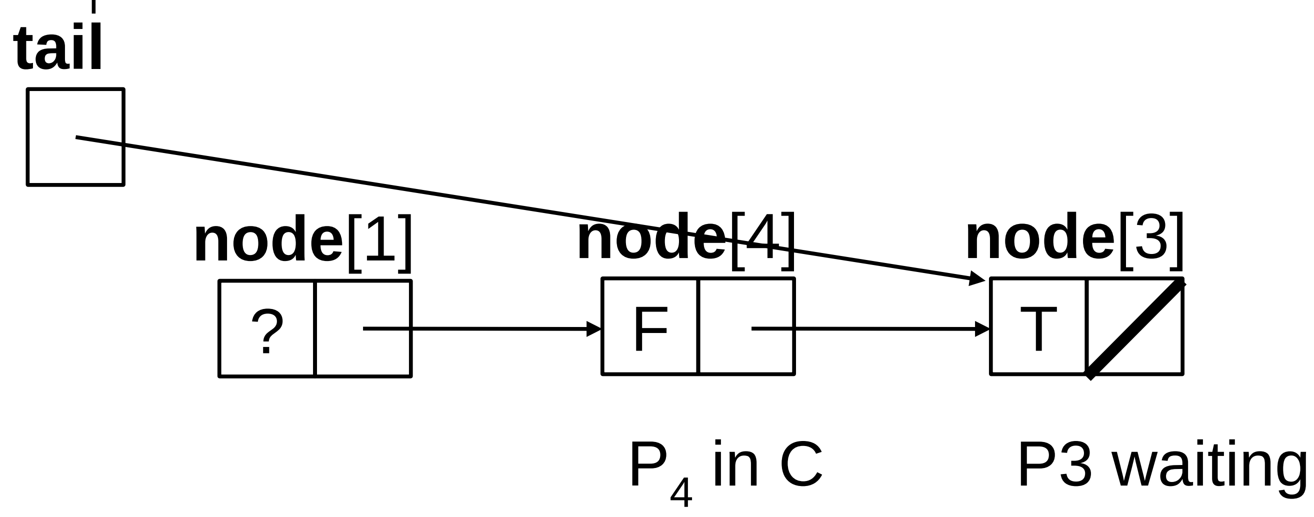
exit$_i$
  if **node**[i].next = 0
    if CAS(**tail**,i,0) return
    waitfor(**node**[i].next ≠ 0)
  node[**node**[i].next].wait := false
rem$_i$



Mellor-Crummey/Scott lock

try_i
node[i].next := 0
pred := swap(**tail**,i)
if pred ≠ 0
  node[i].wait := true
  node[pred].next := i
  waitfor(¬**node**[i].wait)
$crit_i$

$exit_i$
if **node**[i].next = 0
  if CAS(**tail**,i,0) return
  waitfor(**node**[i].next ≠ 0)
node[**node**[i].next].wait := false
rem_i



Craig/Landin/Hagersten lock

node: array[0..N] of {wait,done}; initially all done
tail: 0..N; initially 0

local to i: my_node: 0..N; initially i

try_i
  node[my_node] := wait
  pred := swap(**tail**,my_node)
  waitfor(**node**[pred] = done)
$crit_i$

$exit_i$
  node[my_node] := done
  my_node := pred
rem_i

- Even simpler than MCS.
- Has same nice properties, plus eliminates spinning on exit.
- Not as good on cacheless architectures, since nodes spin on locations that could be remote.

Craig/Landin/Hagersten lock

node: array[0..N] of {wait,done}; initially all done
tail: 0..N; initially 0

local to i: my_node: 0..N; initially i

try_i
  node[my_node] := wait
  pred := swap(**tail**,my_node)
  waitfor(**node**[pred] = done)
$crit_i$

$exit_i$
  node[my_node] := done
  my_node := pred
rem_i

- Queue structure information now distributed, not in shared memory.
- List is linked implicitly, via local pred pointers.
- Upon exit, processes acquire new node id (specifically, from predecessor).

Craig/Landin/Hagersten lock

node: array[0..N] of {wait,done}; initially all done

tail: 0..N; initially 0

local to i: my_node: 0..N; initially i

try_i

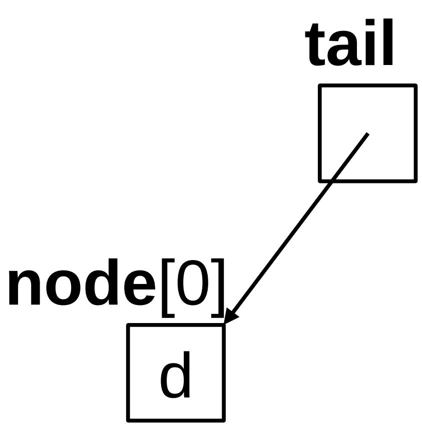
  node[my_node] := wait

  pred :=

swap(**tail**,my_node)

  waitfor(**node**[pred] = done)

$crit_i$



$exit_i$

  node[my_node] := done

  my_node := pred

rem_i

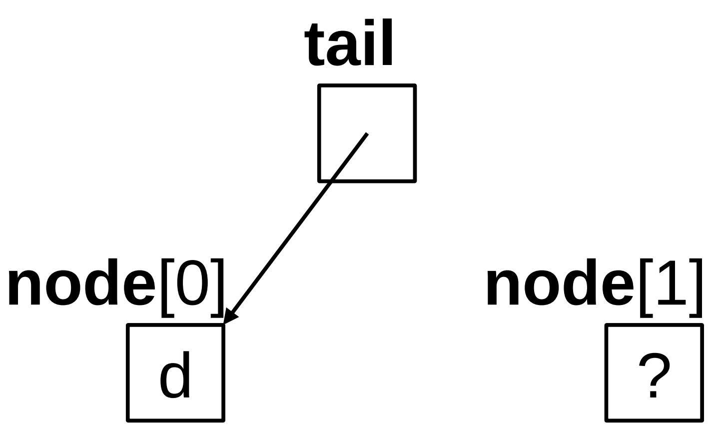
Craig/Landin/Hagersten lock

node: array[0..N] of {wait,done}; initially all done
tail: 0..N; initially 0

local to i: my_node: 0..N; initially i

try_i
  node[my_node] := wait
  pred := swap(**tail**,my_node)
  waitfor(**node**[pred] = done)
$crit_i$

$exit_i$
  node[my_node] := done
  my_node := pred
rem_i



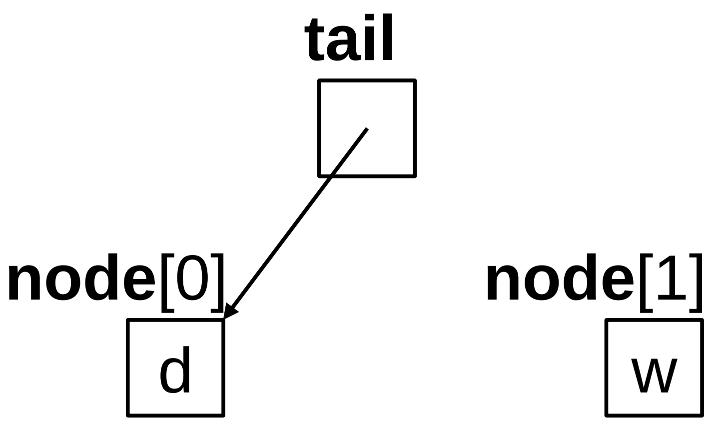
Craig/Landin/Hagersten lock

node: array[0..N] of {wait,done}; initially all done
tail: 0..N; initially 0

local to i: my_node: 0..N; initially i

try_i
  node[my_node] := wait
  pred := swap(**tail**,my_node)
  waitfor(**node**[pred] = done)
$crit_i$

$exit_i$
  node[my_node] := done
  my_node := pred
rem_i



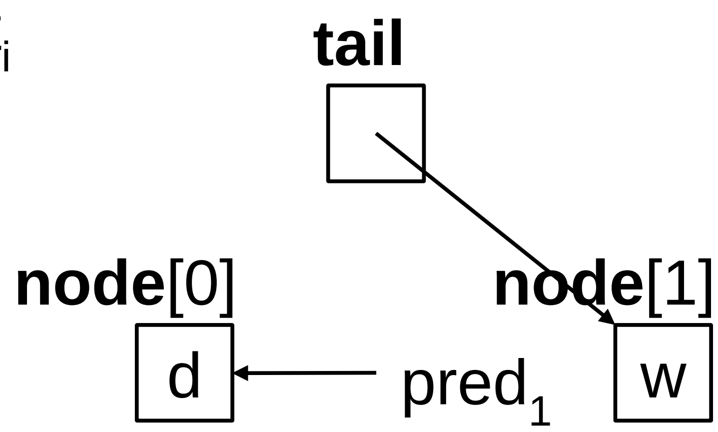
Craig/Landin/Hagersten lock

node: array[0..N] of {wait,done}; initially all done
tail: 0..N; initially 0

local to i: my_node: 0..N; initially i

try_i
  node[my_node] := wait
  pred :=
swap(**tail**,my_node)
  waitfor(**node**[pred] = done)
$crit_i$

$exit_i$
  node[my_node] := done
  my_node := pred
rem_i



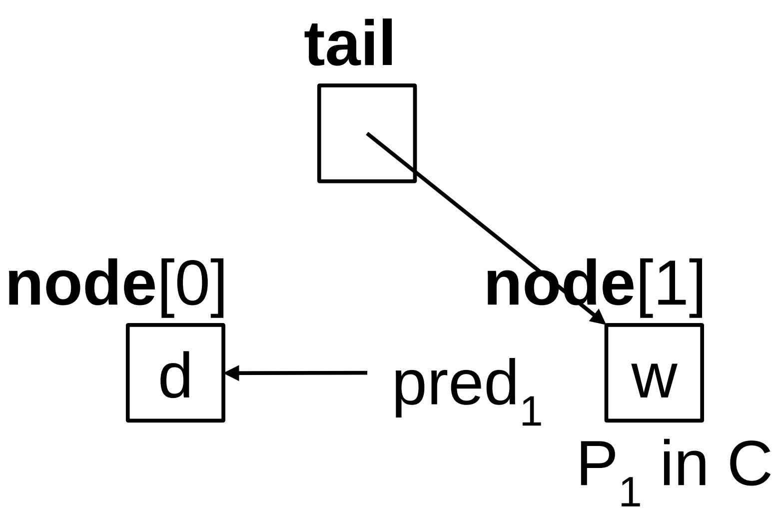
Craig/Landin/Hagersten lock

node: array[0..N] of {wait,done}; initially all done
tail: 0..N; initially 0

local to i: my_node: 0..N; initially i

try_i
  node[my_node] := wait
  pred := swap(**tail**,my_node)
  waitfor(**node**[pred] = done)
$crit_i$

$exit_i$
  node[my_node] := done
  my_node := pred
rem_i



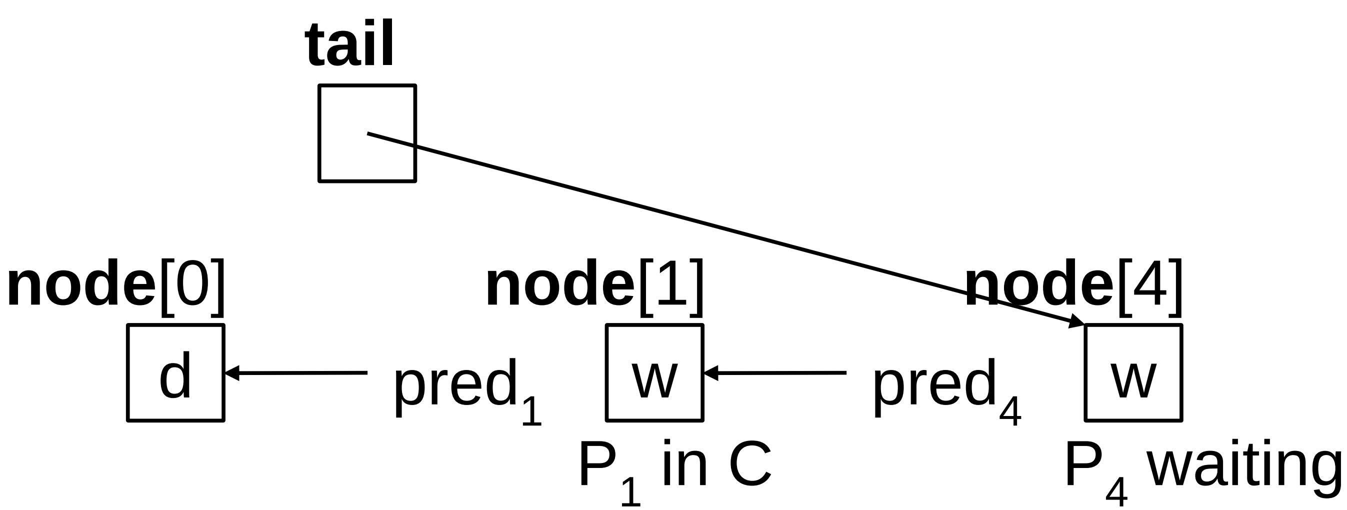
Craig/Landin/Hagersten lock

node: array[0..N] of {wait,done}; initially all done
tail: 0..N; initially 0

local to i: my_node: 0..N; initially i

try_i
  node[my_node] := wait
  pred := swap(**tail**,my_node)
  waitfor(**node**[pred] = done)
$crit_i$

$exit_i$
  node[my_node] := done
  my_node := pred
rem_i



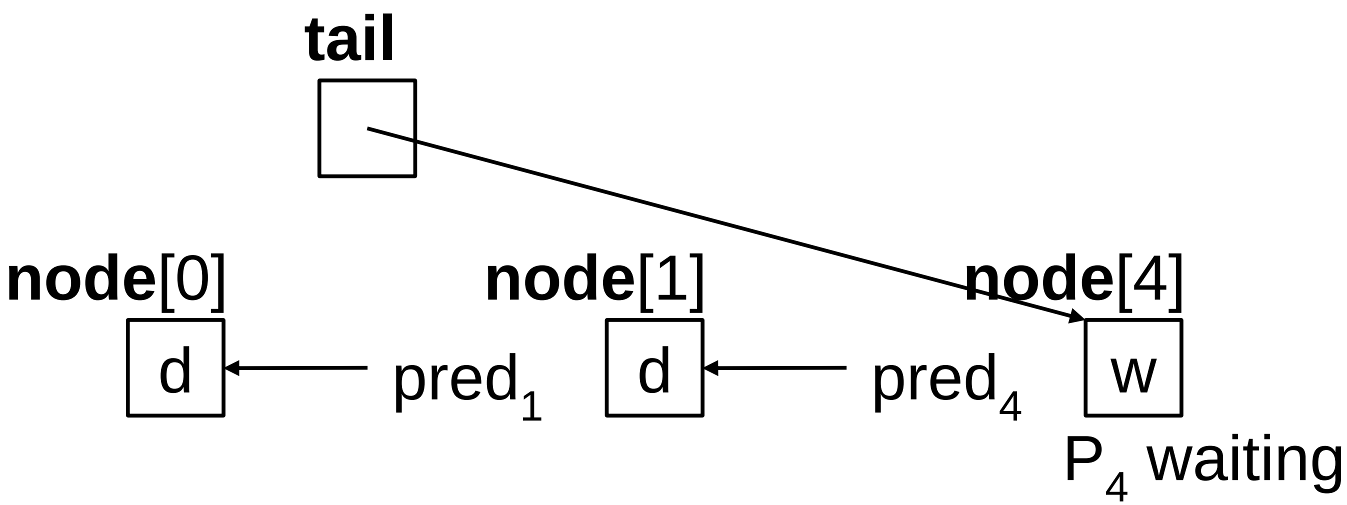
Craig/Landin/Hagersten lock

node: array[0..N] of {wait,done}; initially all done
tail: 0..N; initially 0

local to i: my_node: 0..N; initially i

try_i
  node[my_node] := wait
  pred := swap(**tail**,my_node)
  waitfor(**node**[pred] = done)
$crit_i$

$exit_i$
  node[my_node] := done
  my_node := pred
rem_i



Craig/Landin/Hagersten lock

node: array[0..N] of {wait,done}; initially all done

tail: 0..N; initially 0

local to i: my_node: 0..N; initially i

try_i

  node[my_node] := wait

  pred := swap(**tail**,my_node)

  waitfor(**node**[pred] = done)

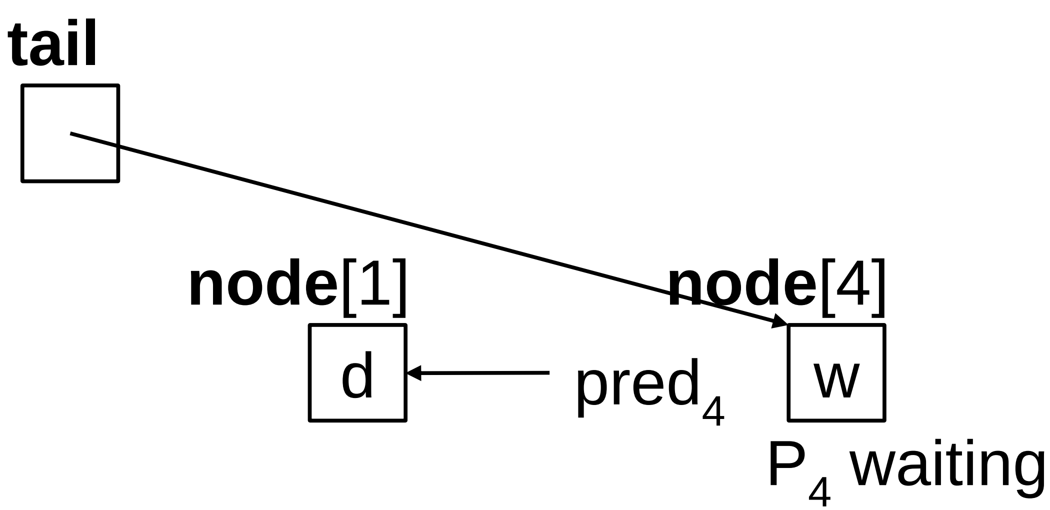
$crit_i$

$exit_i$

  node[my_node] := done

  my_node := pred

rem_i



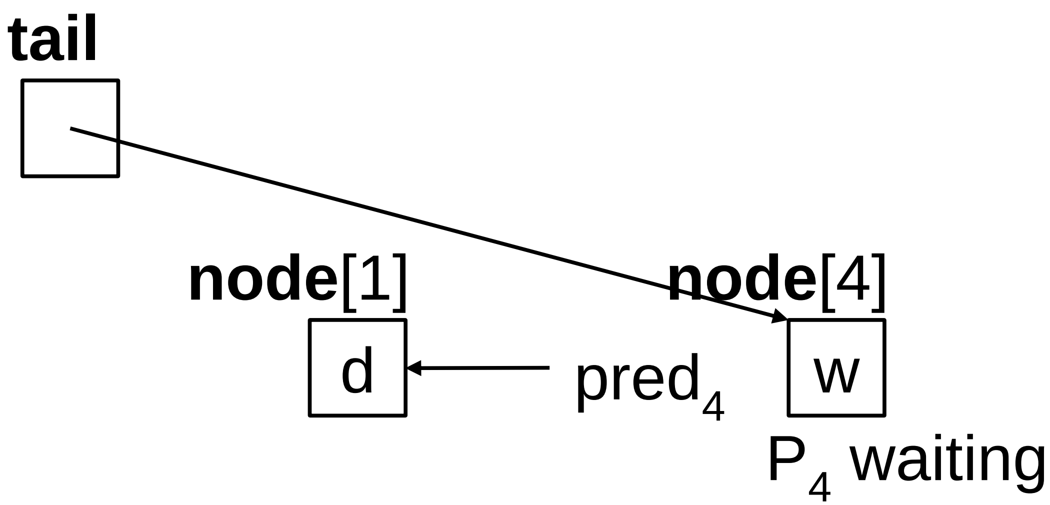
Craig/Landin/Hagersten lock

node: array[0..N] of {wait,done}; initially all done
tail: 0..N; initially 0

local to i: my_node: 0..N; initially i

try_i
  node[my_node] := wait
  pred := swap(**tail**,my_node)
  waitfor(**node**[pred] = done)
$crit_i$

$exit_i$
  node[my_node] := done
  my_node := pred
rem_i



Craig/Landin/Hagersten lock

node: array[0..N] of {wait,done}; initially all done

tail: 0..N; initially 0

local to i: my_node: 0..N; initially i

try_i

  node[my_node] := wait

  pred := swap(**tail**,my_node)

  waitfor(**node**[pred] = done)

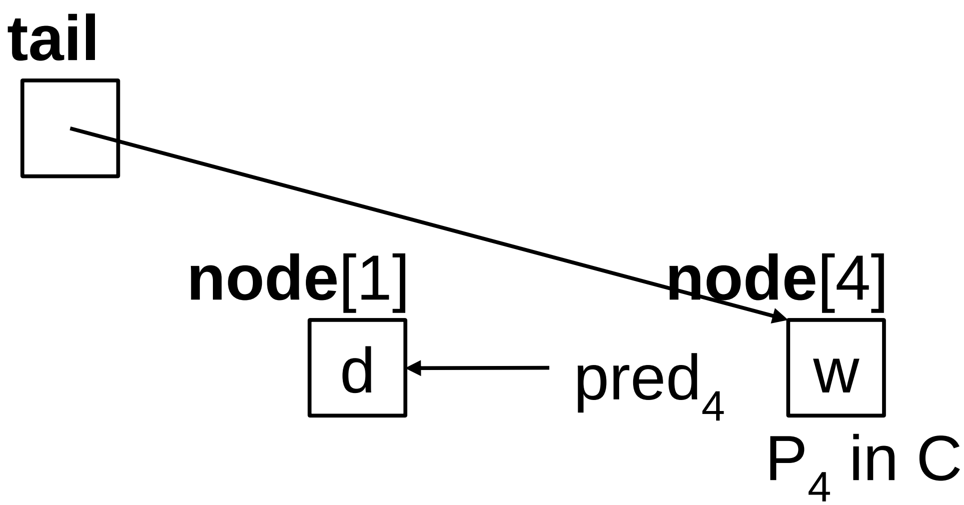
$crit_i$

$exit_i$

  node[my_node] := done

  my_node := pred

rem_i



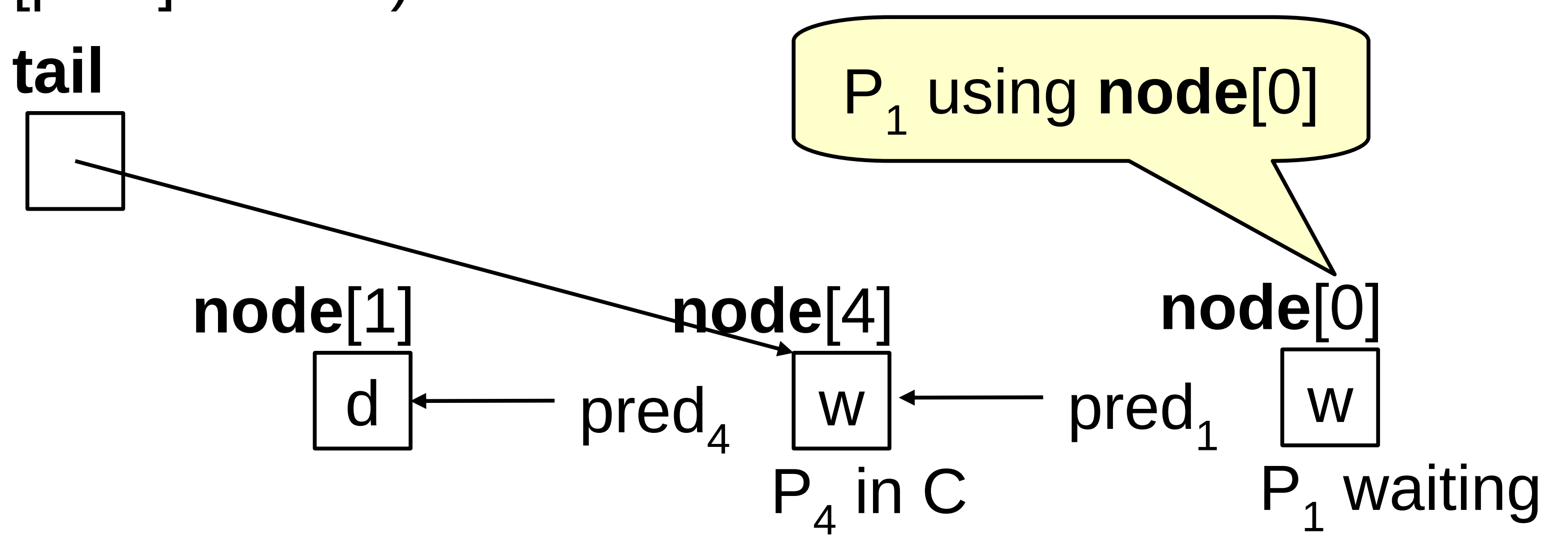
Craig/Landin/Hagersten lock

node: array[0..N] of {wait,done}; initially all done
tail: 0..N; initially 0

local to i: my_node: 0..N; initially i

try$_i$
  node[my_node] := wait
  pred := swap(**tail**,my_node)
  waitfor(**node**[pred] = done)
crit$_i$

exit$_i$
  node[my_node] := done
  my_node := pred
rem$_i$



Additional lock features

- Timeout (of waiting for lock)
  - Well-formedness implies you are stuck once you start trying.
  - May want to bow out (to reduce contention?) if taking too long.
  - How could we do this?
    - Easy for test&set locks; harder for queue locks (and ticket lock).
- Hierarchical locks
  - If machine is hierarchical, and critical section protects data, it may be better to schedule “nearby” processes consecutively.
- Reader/writer locks
  - Readers don't conflict, so many readers can be “critical” together
  - Especially important for “long” critical sections.

Generalized Resource Allocation

- A very quick tour
- Lynch, Chapter 11

Generalized resource allocation

- Mutual exclusion: Problem of allocating a single non-sharable resource.
- Can generalize to more resources, some sharing.
- Exclusion specification **E** (for a given set of users):
  - Any collection of sets of users, closed under superset.
  - Expresses which users are incompatible, can't coexist in the critical section.

- Example: k-exclusion (any k users are okay, but not k+1)

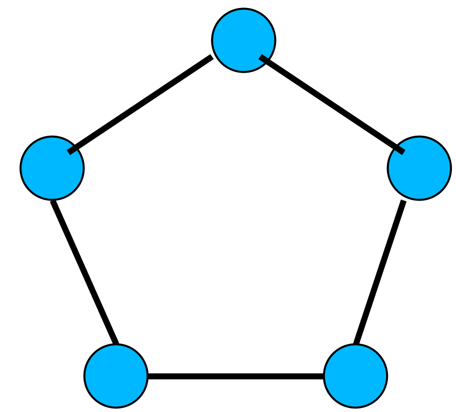
  E = { E : |E| > k }

- Example: Reader-writer locks
  - Relies on classification of users as readers vs. writers.

  E = { E : |E| > 1 and E contains a writer }

- Example: Dining Philosophers (Dijkstra)

  E = { E : E includes a pair of neighbors }

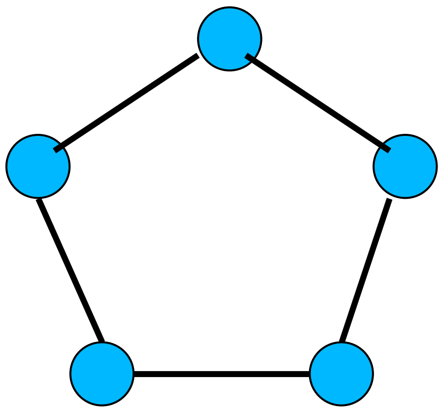


Resource specifications

- Some exclusion specs can be described conveniently in terms of requirements for concrete resources.
- Resource spec: Different users need different subsets of resources
  - Can't share: Users with intersecting sets exclude each other.

- Example: Dining Philosophers (Dijkstra)

  E = { E : E includes a pair of neighbors }
  Forks (resources) between adjacent philosophers; each needs both adjacent forks in order to eat.
  Only one can hold a particular fork at a time, so adjacent philosophers must exclude each other.

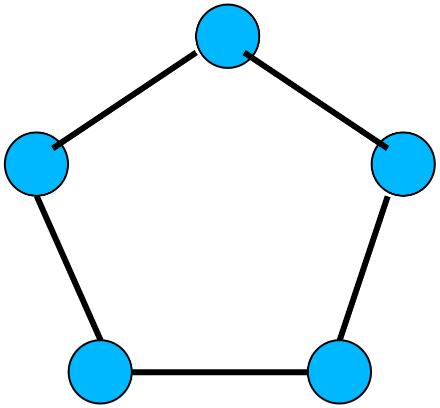


- Not every exclusion problem can be expressed in this way.
  - k-exclusion cannot.

Resource allocation problem, for a given exclusion spec **E**

- Same shared-memory architecture as for mutual exclusion (processes and shared variables, no buses, no caches).
- Well-formedness, as before.
- Exclusion: No reachable state in which the set of users in C is a set in **E.**
- Progress: As before.
- Lockout-freedom: As before.
- But these don't capture concurrency requirements.
  - Any lockout-free mutual exclusion algorithm also satisfies **E** (provided that **E** doesn't contain any singleton sets).
- Can add concurrency conditions, e.g.:
  - Independent progress: If $i \in T$ and every j that could conflict with i remains in R, then eventually $i \rightarrow C$. (LTTR)
  - Time bound: Obtain better bounds from $i \rightarrow T$ to $i \rightarrow C$, even in the presence of conflicts, than we can for mutual exclusion.

Dining Philosophers

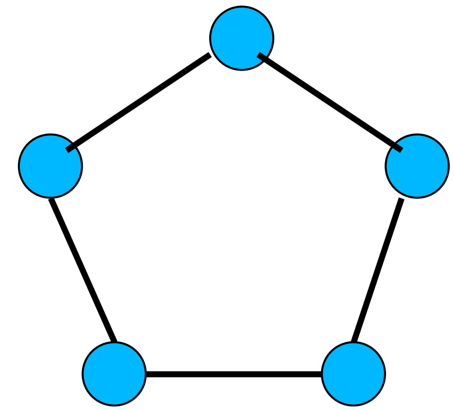


- Dijkstra's paper posed the problem, gave a solution using strong shared-memory model.
  - Globally-shared variables, atomic access to all of shared memory.
  - Not very distributed.
- More distributed version: Assume the only shared variables are on the edges between adjacent philosophers.
  - Correspond to forks.
  - Use RMW shared variables.
- Impossibility result: If all processes are identical and refer to forks by local names "left" and "right", and all shared variables have the same initial values, then we can't guarantee DP exclusion + progress.
- Proof: Show we can't break symmetry:
  - Consider subset of executions that work in synchronous rounds, prove by induction on rounds that symmetry is preserved.

Then by progress, someone $\rightarrow$ C.

So all do, violating DP exclusion.

Dining Philosophers



- Example: Simple symmetric algorithm where all wait for R fork first, then L fork.
  - Guarantees DP exclusion, because processes wait for both forks.
  - But progress fails---all might get R, then deadlock.
- So we need something to break symmetry.
- Solutions:
  - Number forks around the table, pick up smaller numbered fork first.
  - Right/left algorithm (Burns):
    - Classify processes as R or L (need at least one of each).
    - R processes pick up right fork first, L processes pick up left fork first.
    - Yields DP exclusion, progress, lockout freedom, independent progress, and good time bound (constant, for alternating R and L).
- Generalize to solve any resource problem
  - Nodes represent resources.
  - Edge between resources if some user needs both.
  - Color graph; order colors.
  - All processes acquire resources in order of colors.

Next time

- Impossibility of consensus in the presence of failures.
- Reading: Lynch, Chapter 12