6.852 Lecture 15

- Pragmatic issues for shared-memory multiprocessors
- Practical mutual exclusion algorithms
  - test-and-set locks
  - queue locks
- Generalized exclusion/resource allocation problems
- Reading:
  - Mellor-Crummey and Scott paper (Dijkstra prize winner)
  - Magnussen, Landin, Hagersten paper
  - Chapter 11

Next time

- Consensus
- Reading: Chapter 12

Mutual exclusion with RMW

- Quick review
  - shared-memory multiprocessors provide “atomic operations”
    - test&set, fetch&increment, swap, compare&swap (CAS), LL/SC
  - in practice, all mutual exclusion algorithms use these operations
    - one-variable test&set algorithm
    - queue lock: one queue with enqueue, dequeue and head
      - multiprocessors do **not** support queues in hardware
    - ticket lock algorithm: two fetch&inc variables

A note on terminology

- Different usage in “systems” and “theory” communities
  - blocking: yields processor
  - atomic operation: some kind of read-modify-write operation
  - implement: provide specified functionality??
  - simulation: experiment, or running on a (hardware) simulator
  - process vs thread
  - locks vs mutual exclusion
- Different emphasis and concerns
  - mechanism vs. abstraction: processors, locks, blocking
  - performance issues: caching, contention, etc.

Mutual exclusion in practice

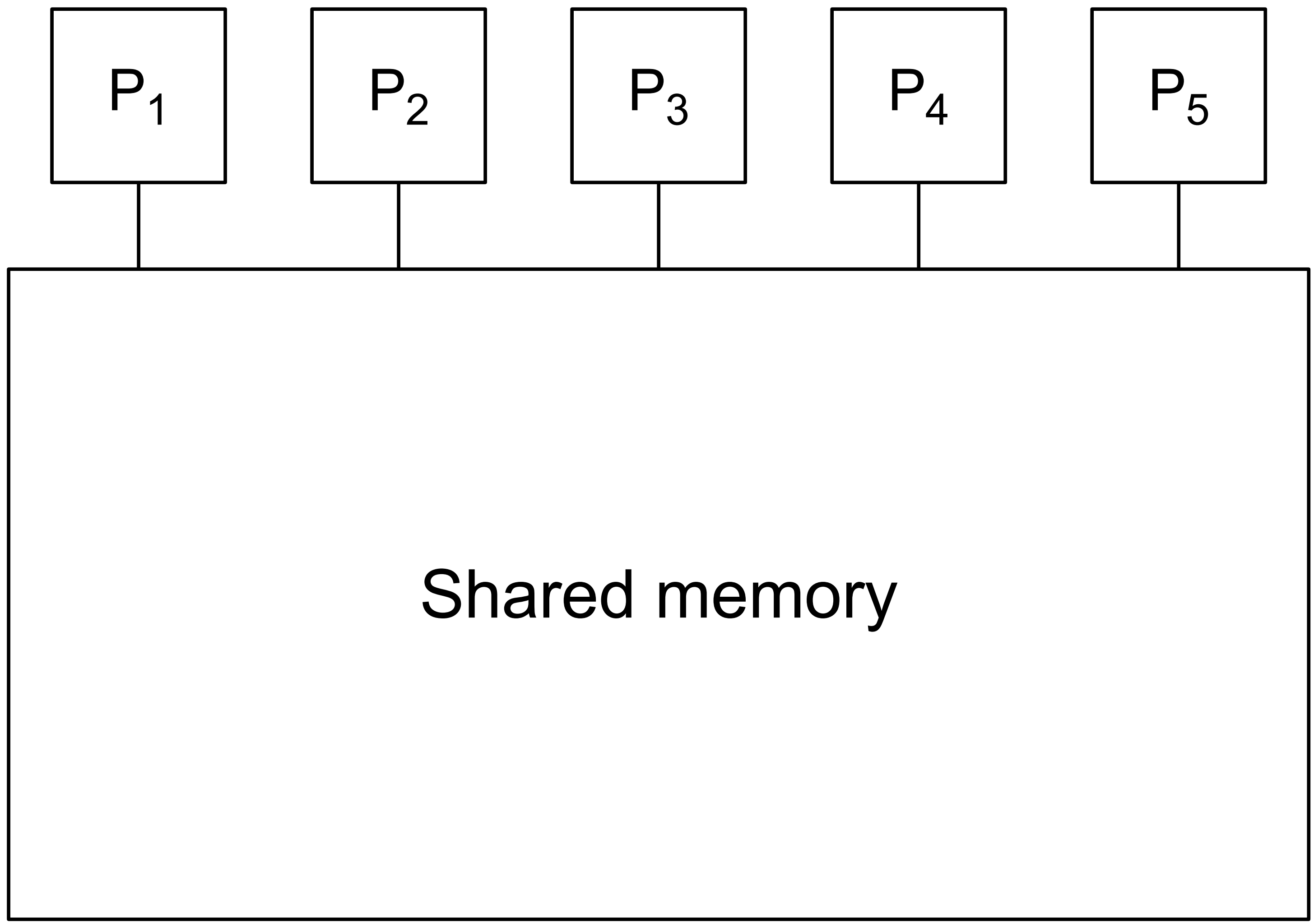
- What to do when lock is taken
  - “block”: deschedule process (yield processor)
    - OS reschedules it in future, often when some condition is satisfied
  - busy-wait/spin
    - don't yield process: repeatedly test for some condition
    - should be used only if waiting is expected to be very short

The choice of blocking vs spinning applies
to other synchronization constructs,
such as producer-consumer and barriers.

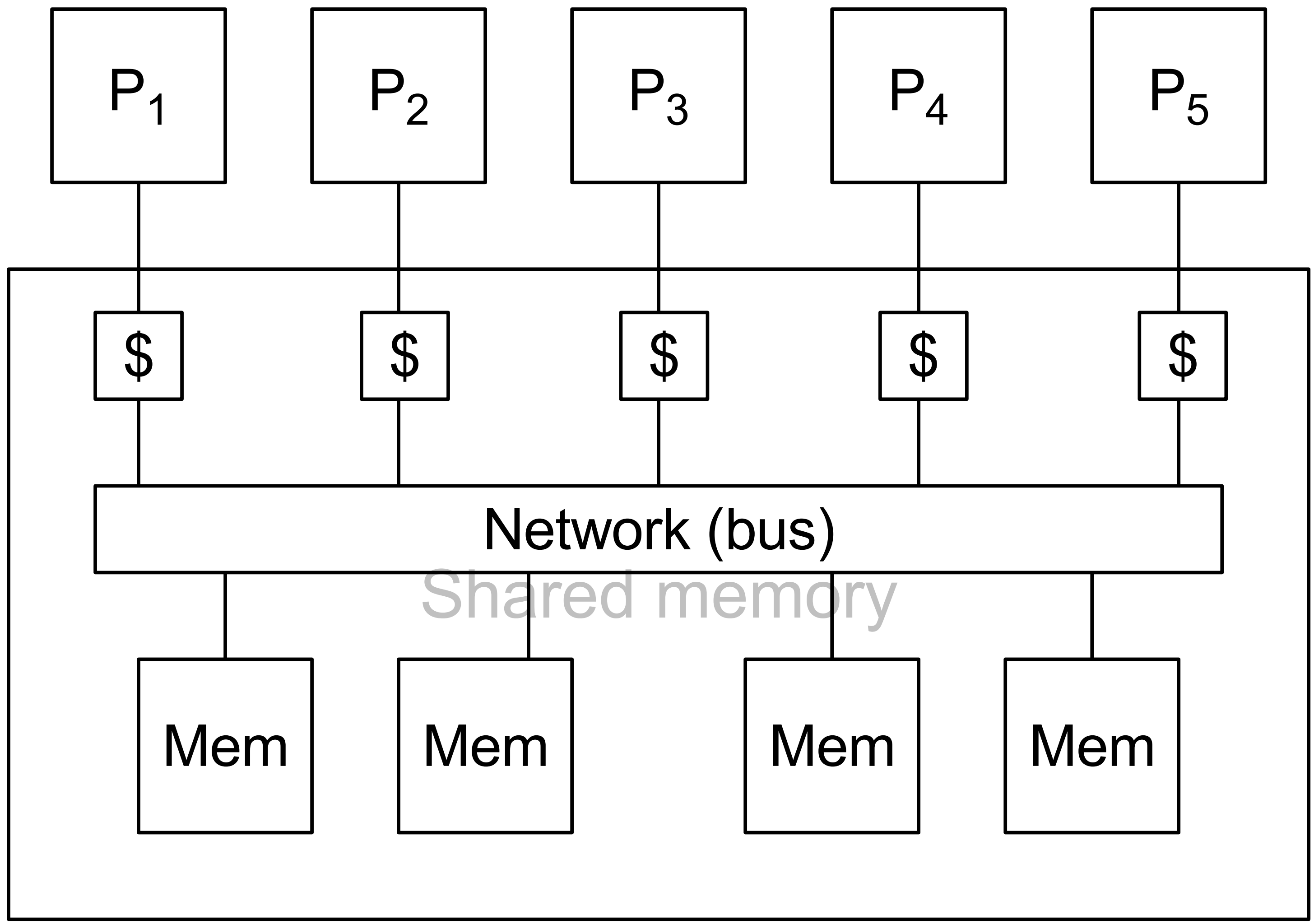
Mutual exclusion in practice

- Spin locks are very important
  - used in OS kernels
- Assume critical sections are very short
  - typically not nested (hold only one lock at a time)
- Performance is critical
  - must consider caching and contention effects
  - adaptive requirements/performance

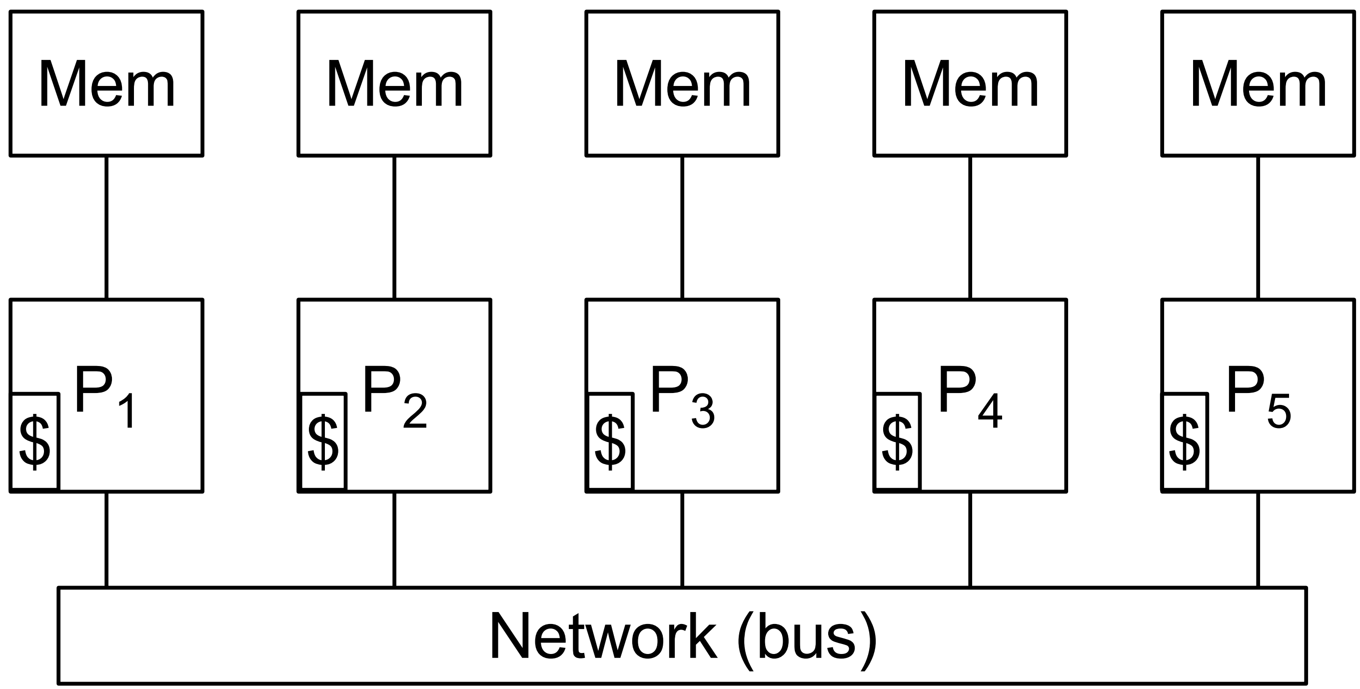
Shared-memory multiprocessors



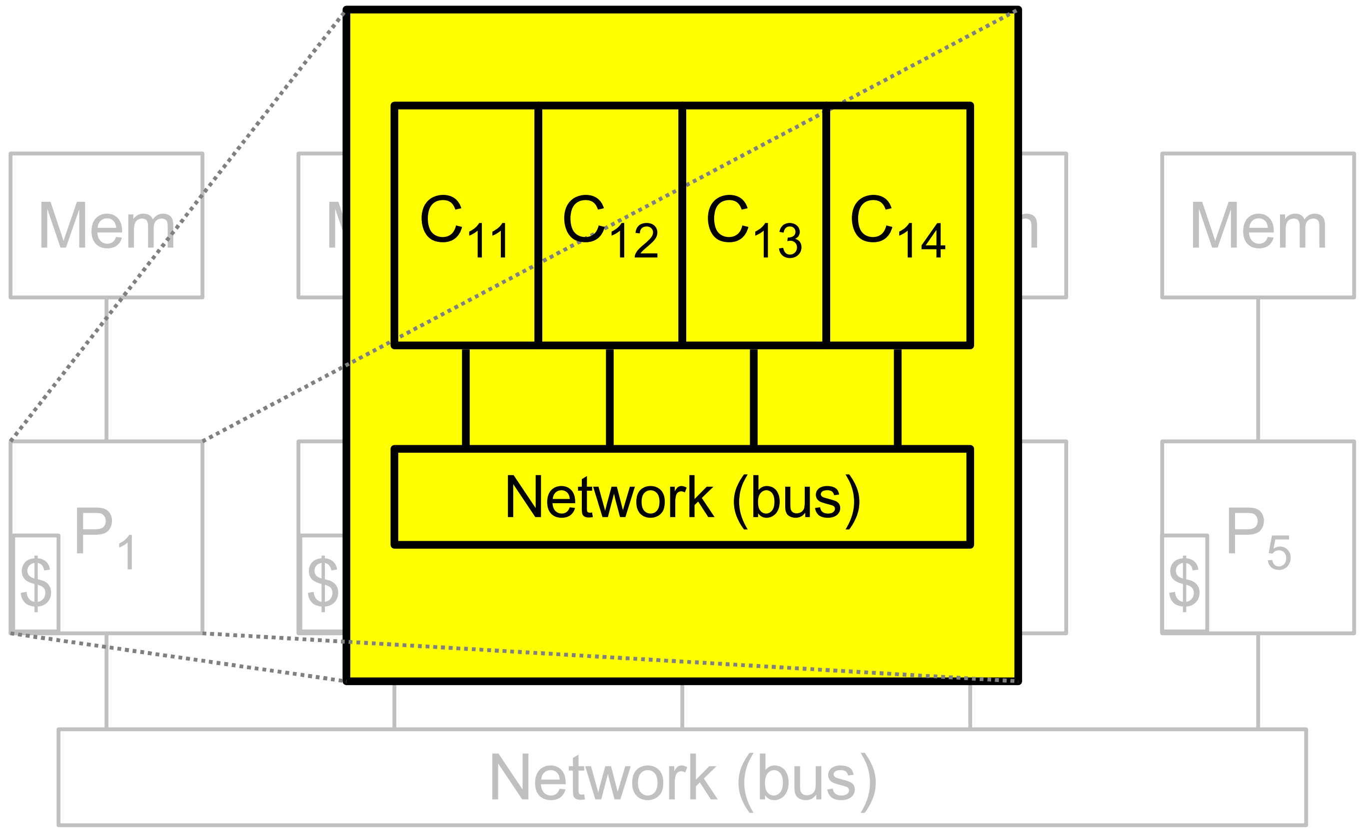
Shared-memory multiprocessors



Shared-memory multiprocessors



Shared-memory multiprocessors



Shared-memory multiprocessors

- Memory access does not have uniform cost
  - next-level cache access is ~10x more expensive
  - remote-memory access produces network traffic
    - network bandwidth can be bottleneck
  - writes invalidate caches
    - every processor that wants to read must request again
    - can typically share read access
  - all memory is multiwriter, but most is reserved for a process

Mutual exclusion in practice

- Critical sections are very short
  - typically hold only one lock at a time
  - critical processes are not swapped out
    - assume no multiprogramming for now (one thread per processor)
- Caching and contention are important

Practical 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

Simple test&set lock

lock: {0,1}; initially 0

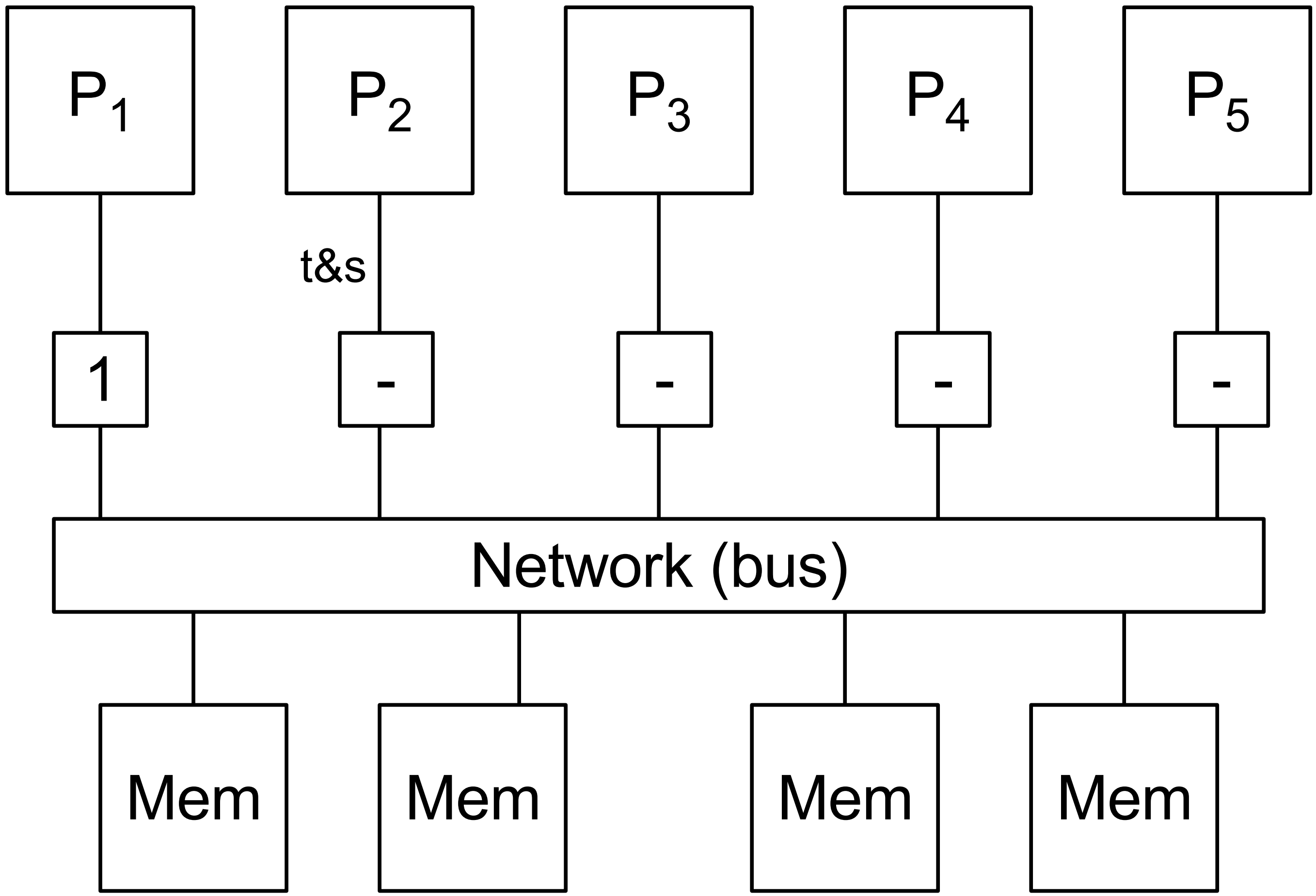
try_i
  waitfor(test&set(**lock)** = 0)
$crit_i$

$exit_i$
  lock := 0
rem_i

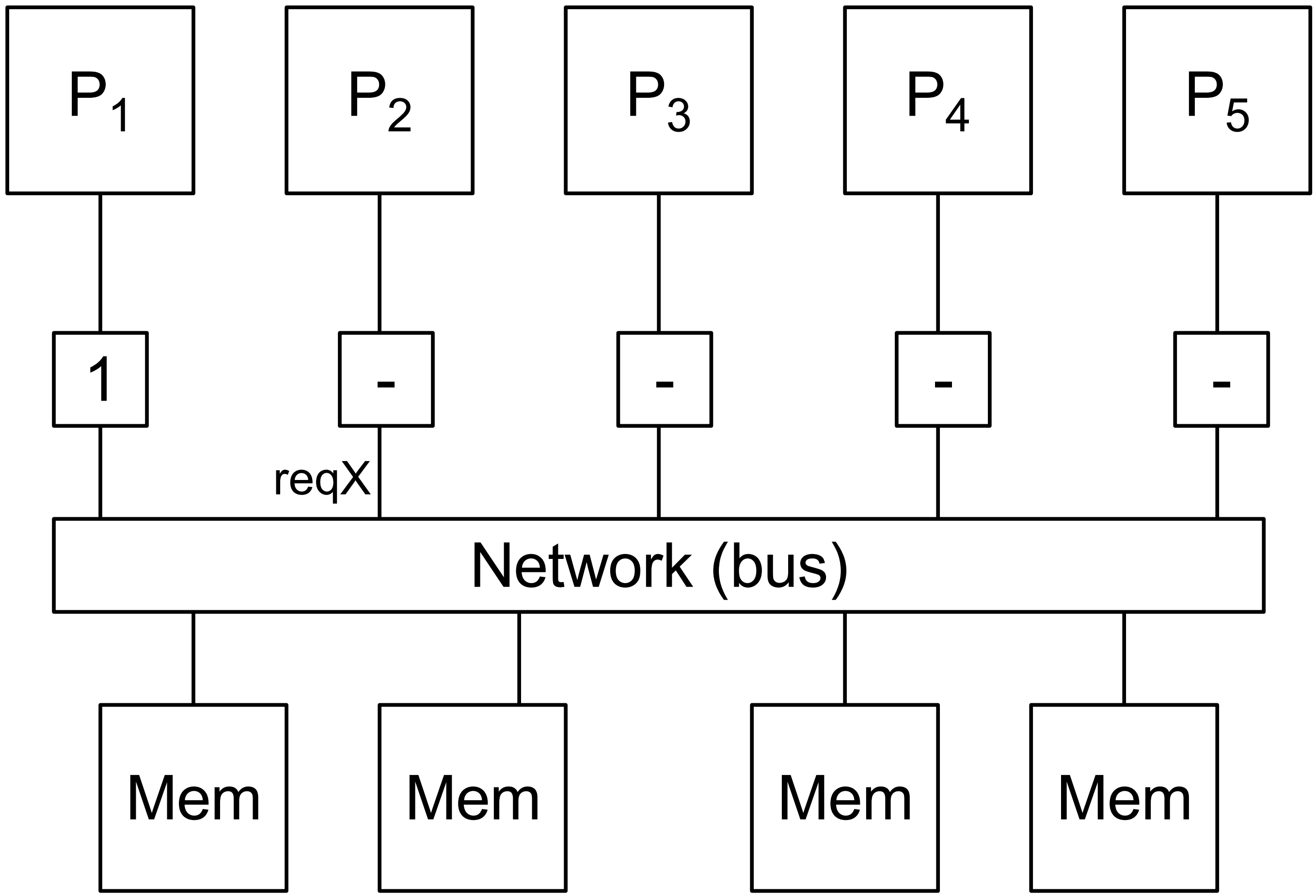
- Simple
- Low space cost
- But lots of network traffic if highly contended

many processes waiting for lock to become free

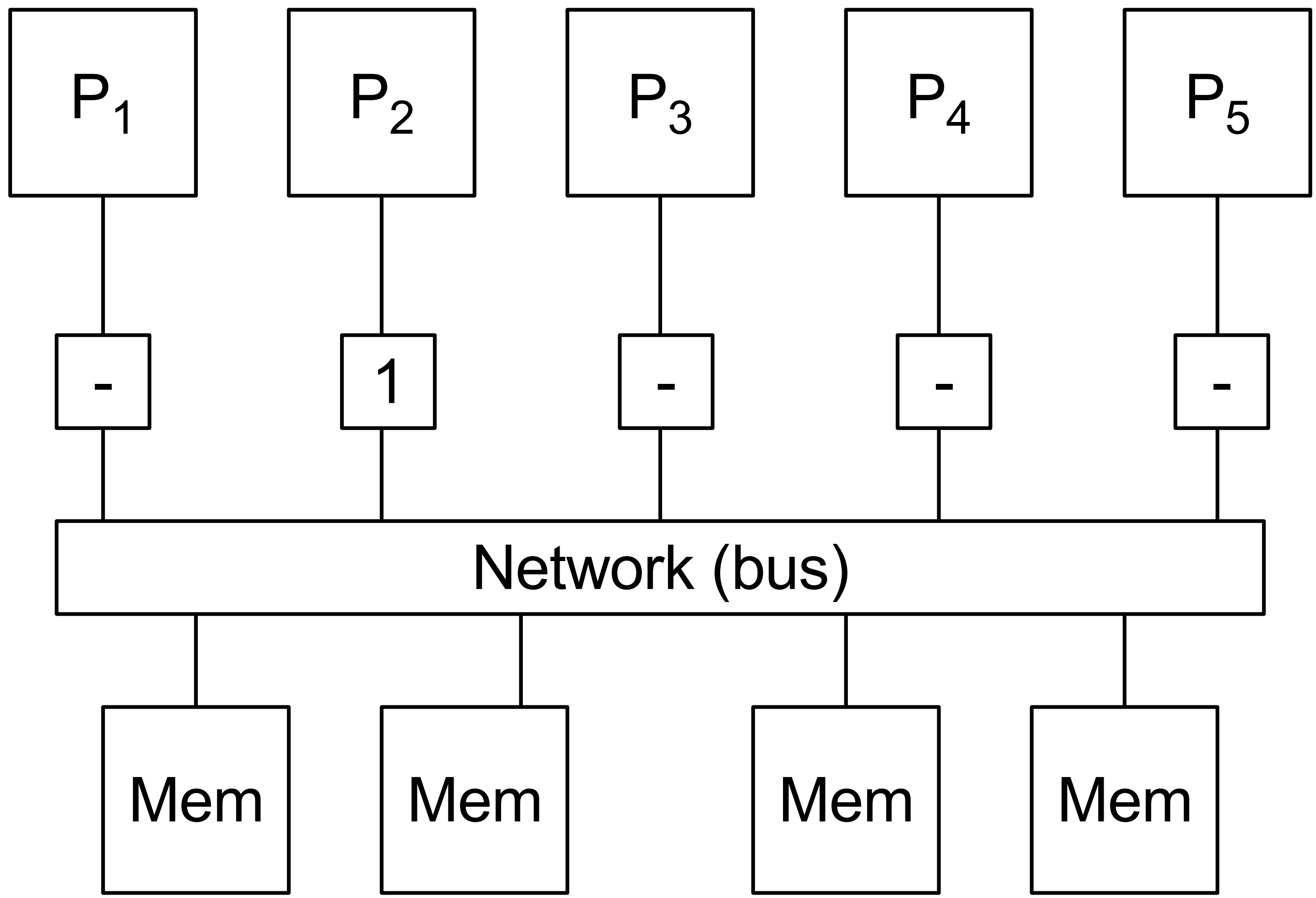
Simple test&set lock



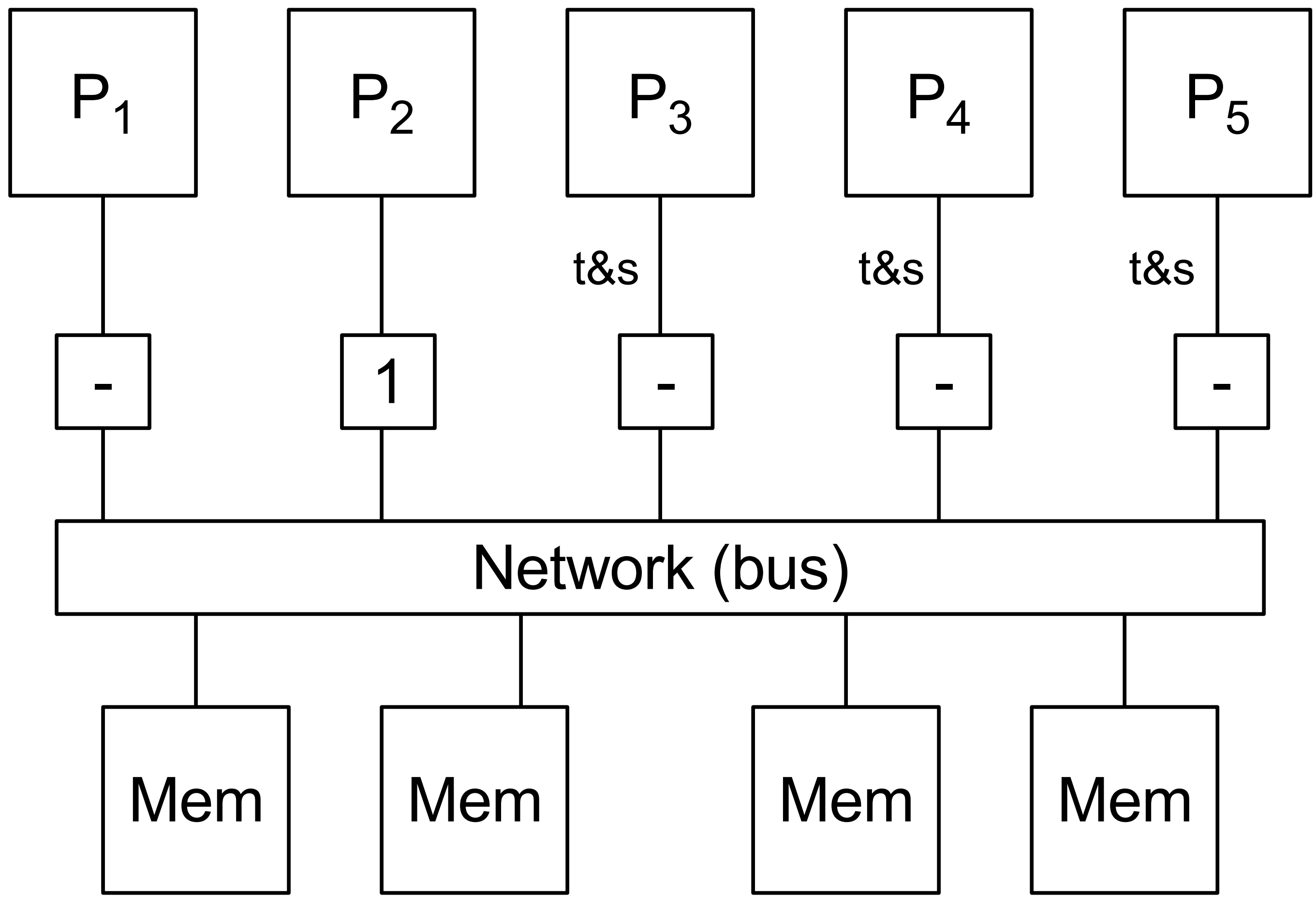
Simple test&set lock



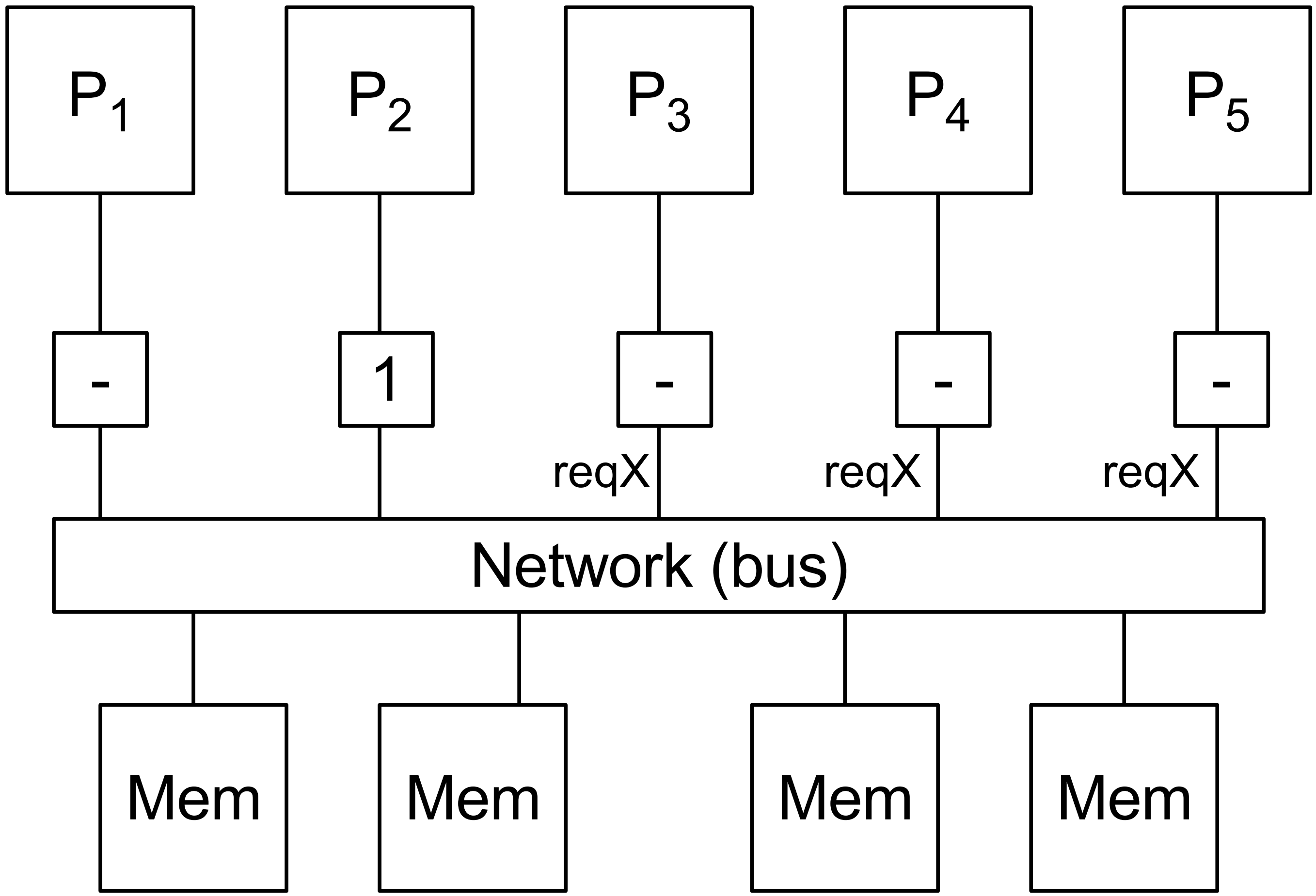
Simple test&set lock



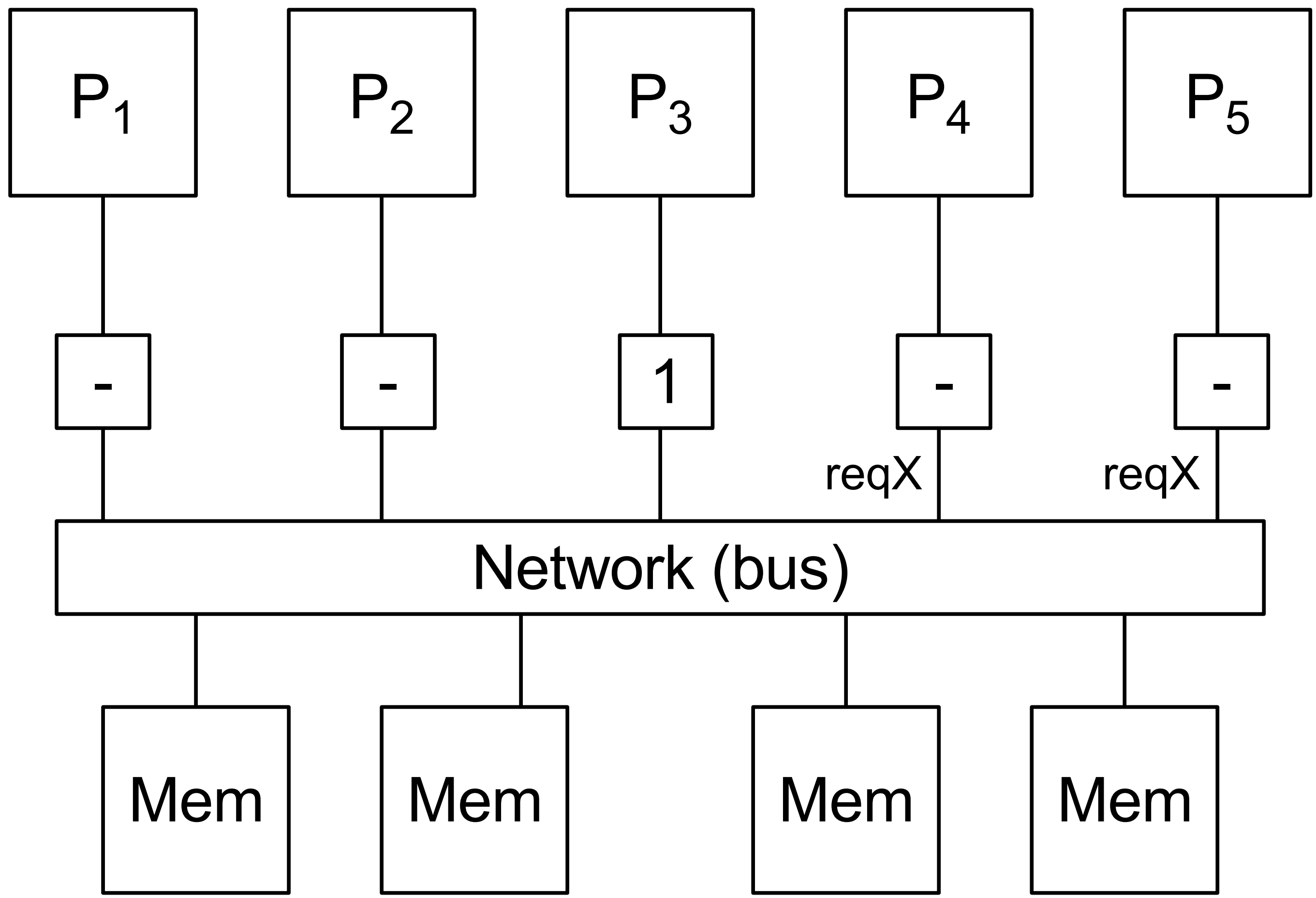
Simple test&set lock



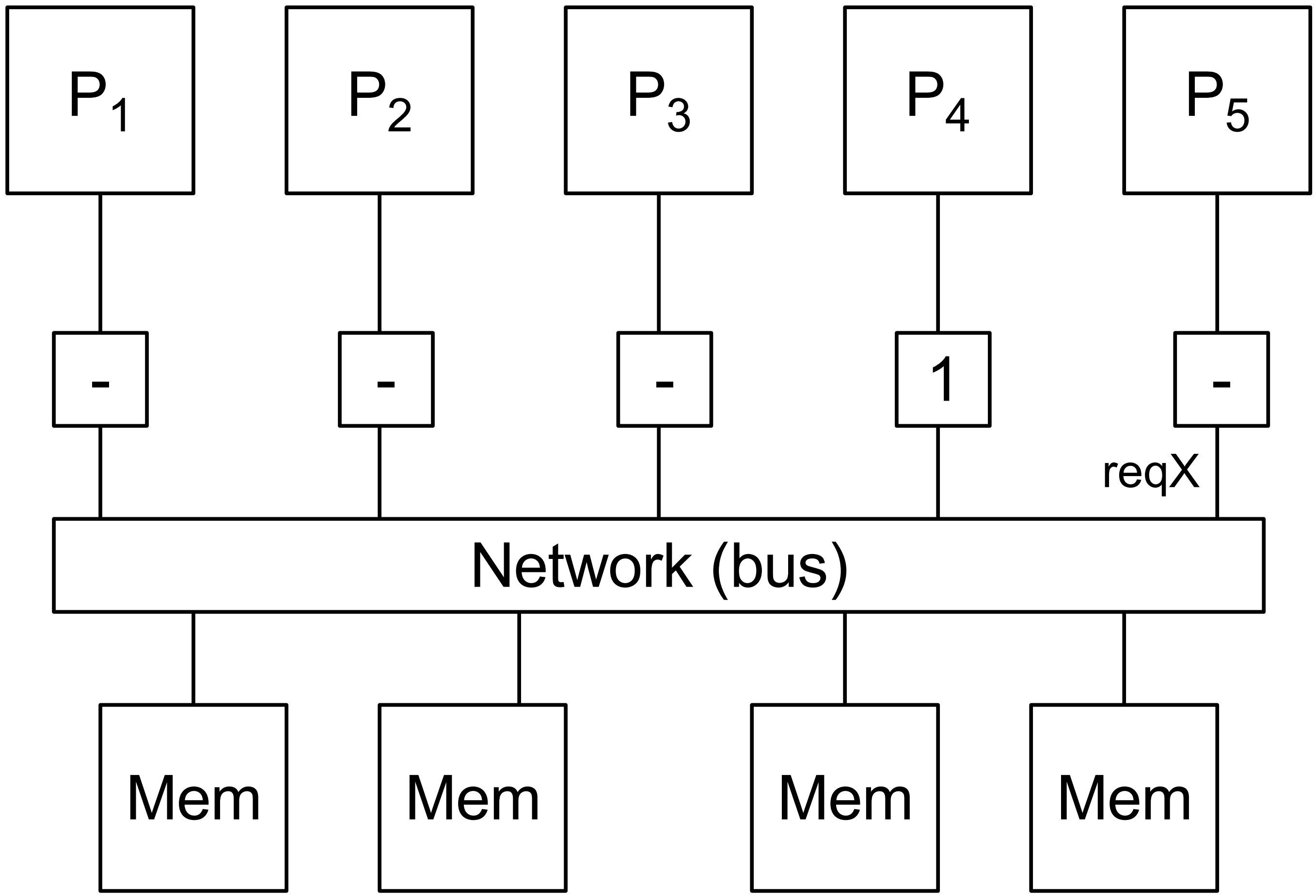
Simple test&set lock



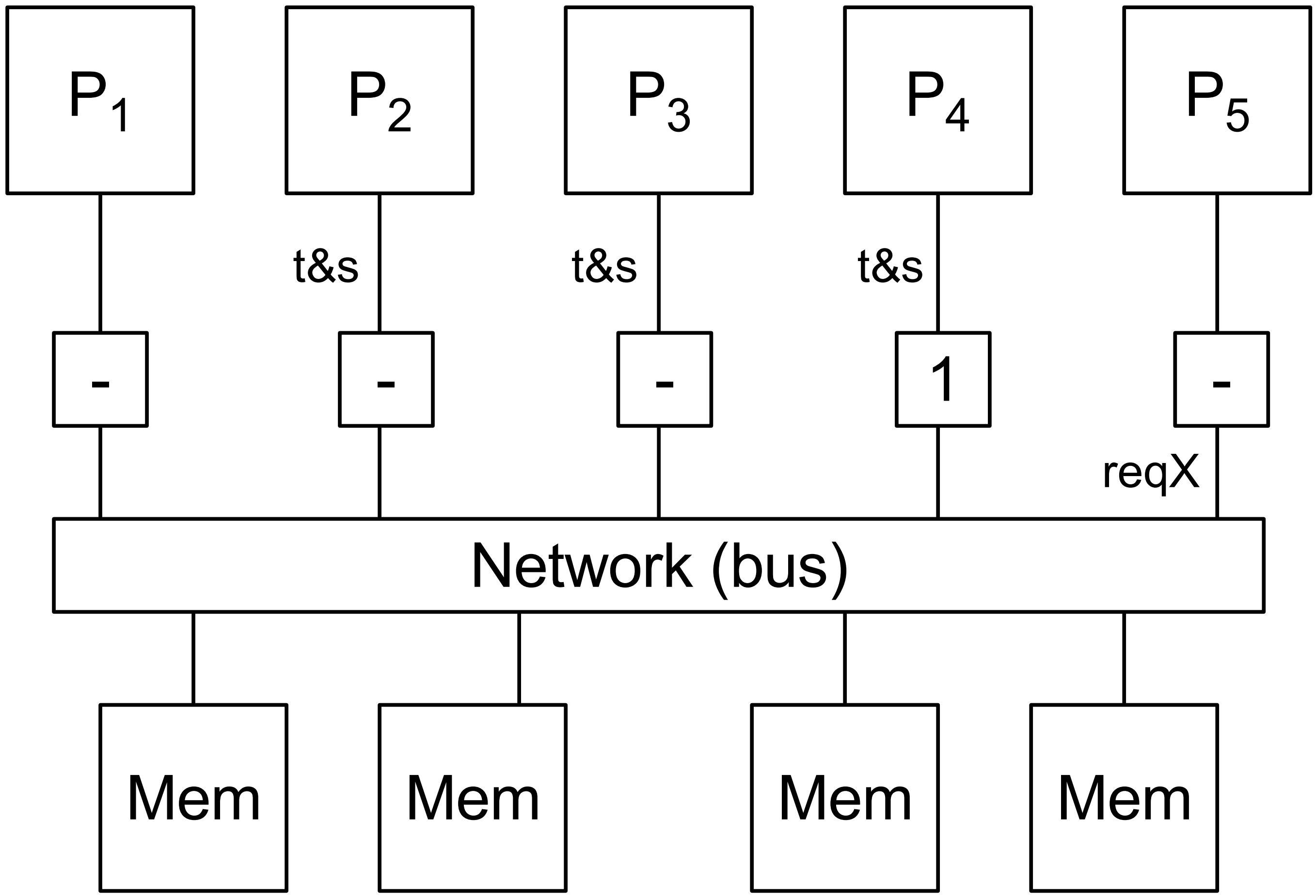
Simple test&set lock



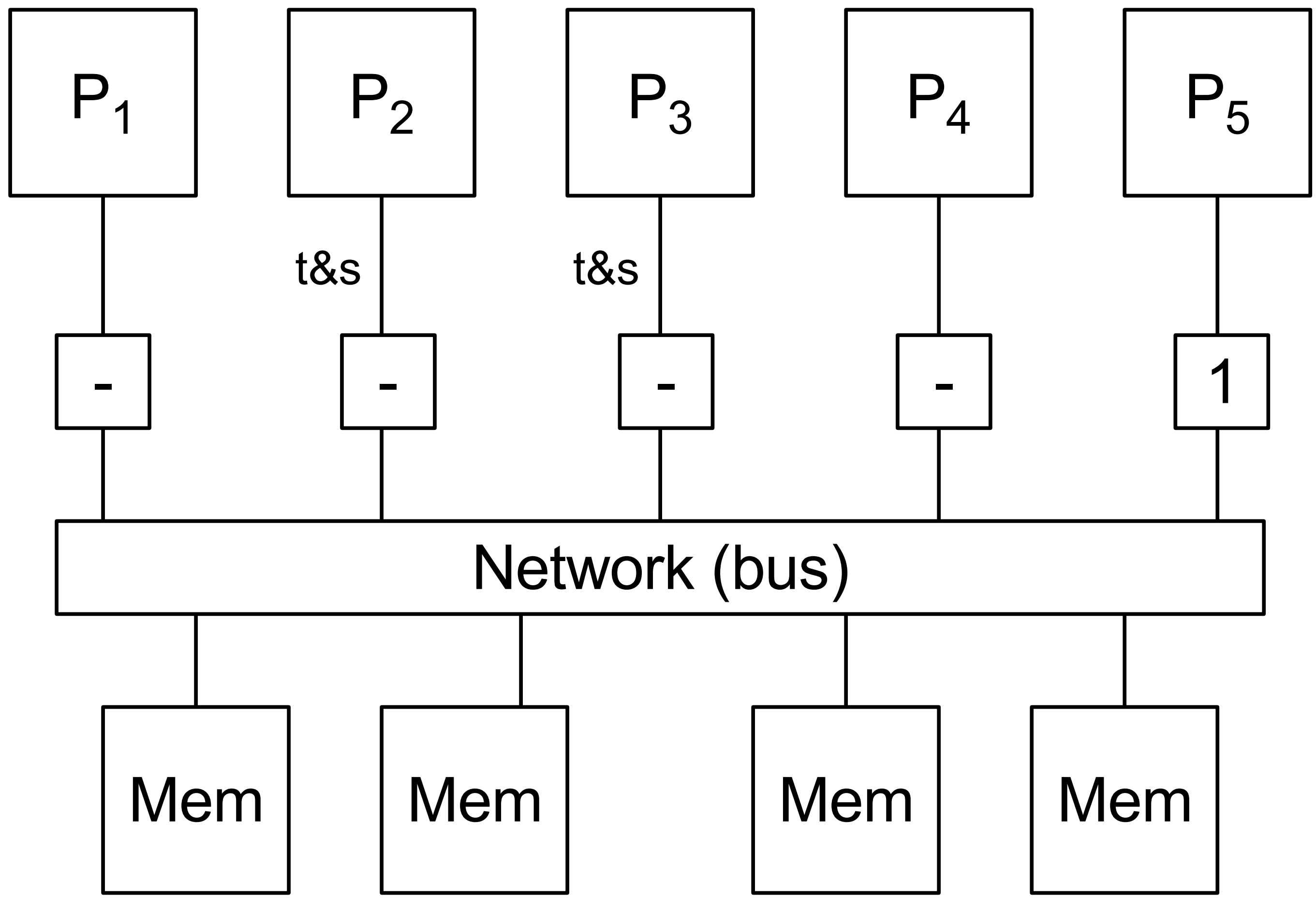
Simple test&set lock



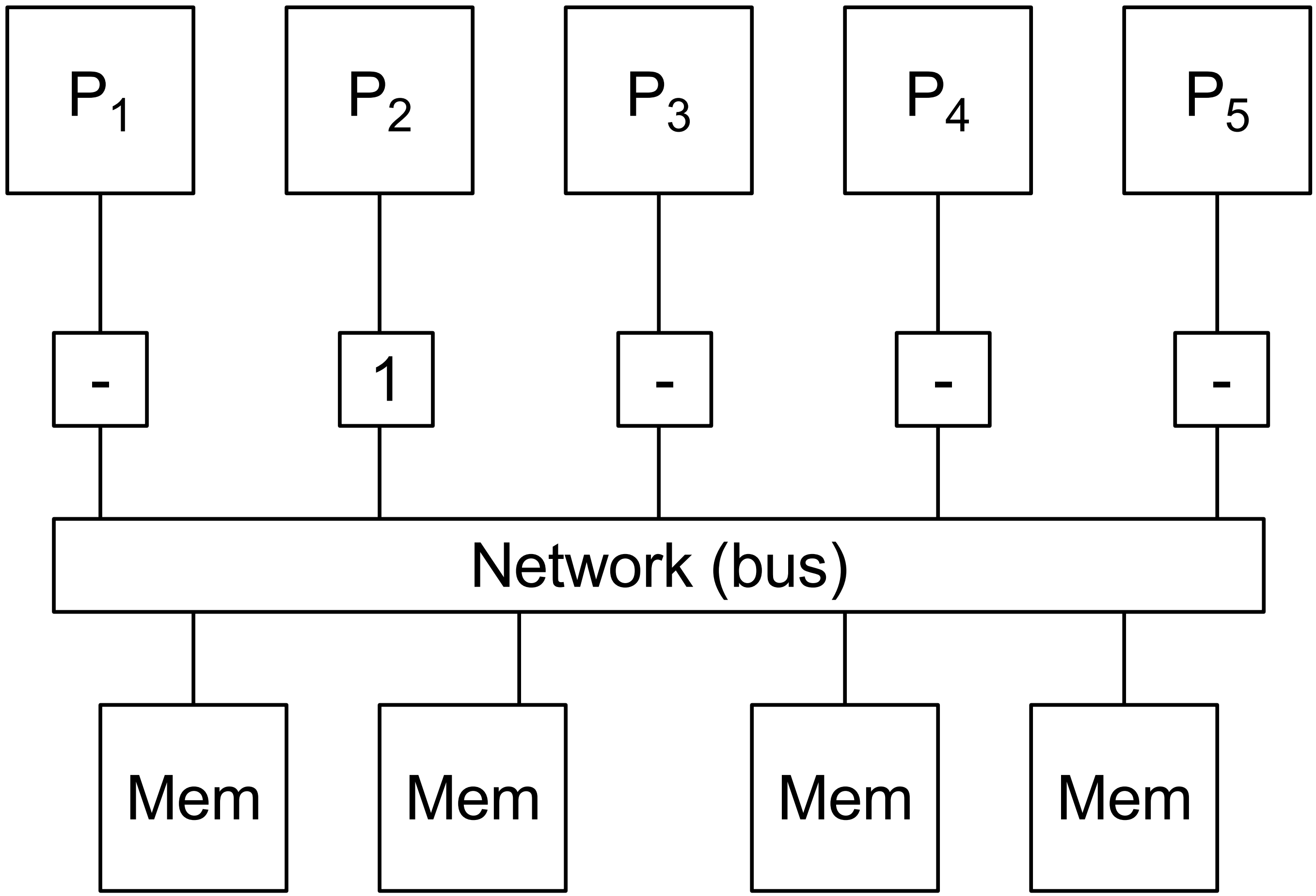
Simple test&set lock



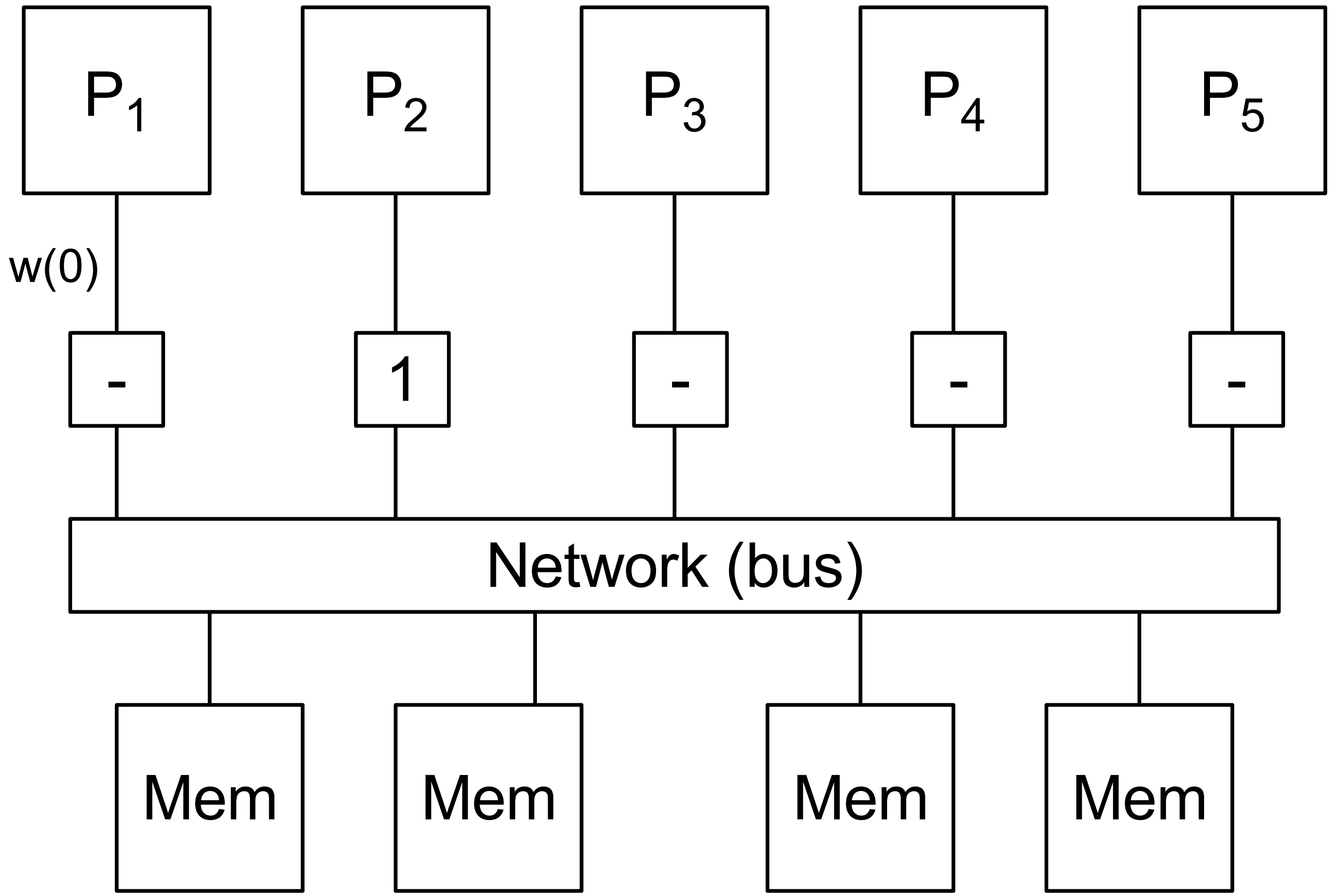
Simple test&set lock



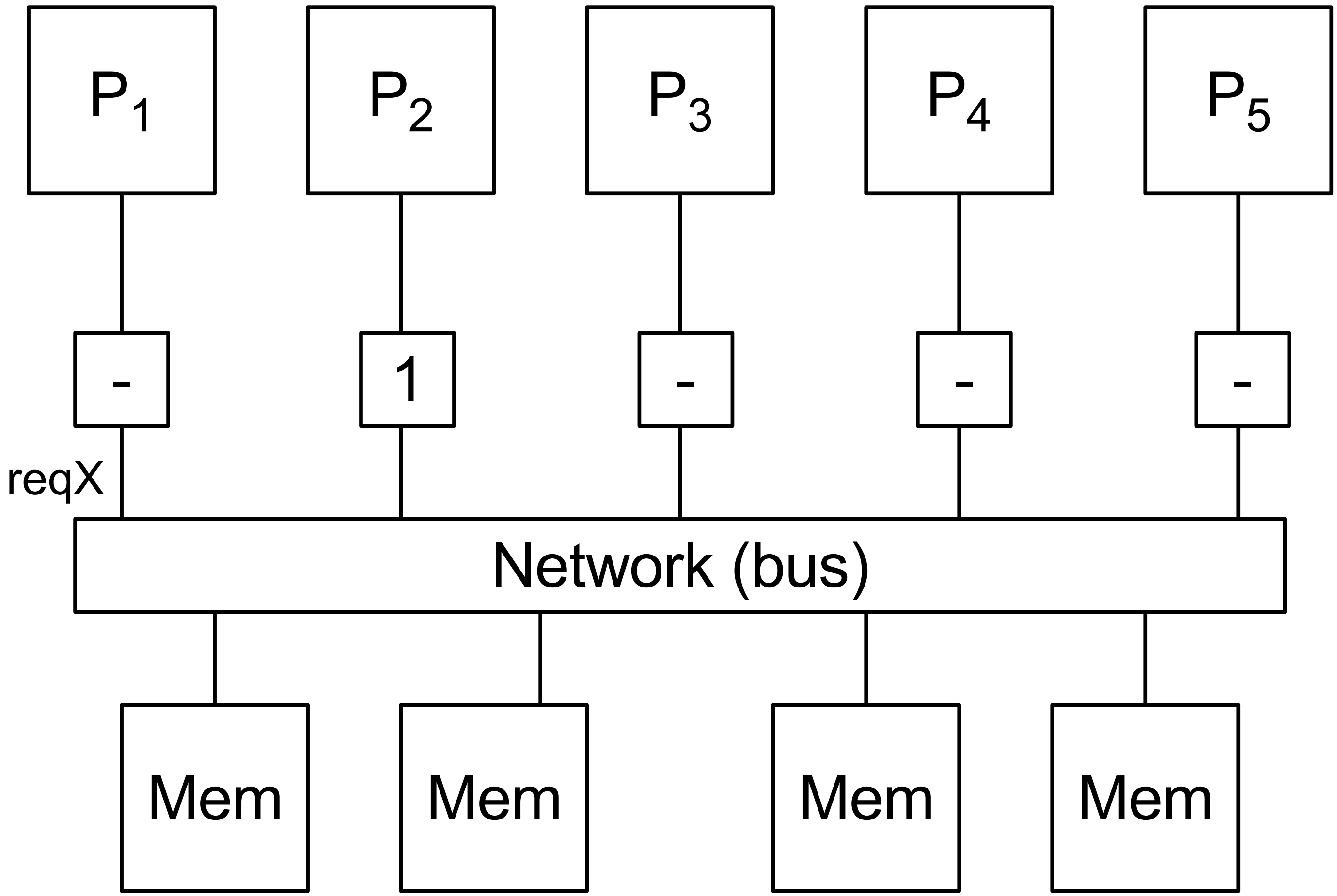
Simple test&set lock



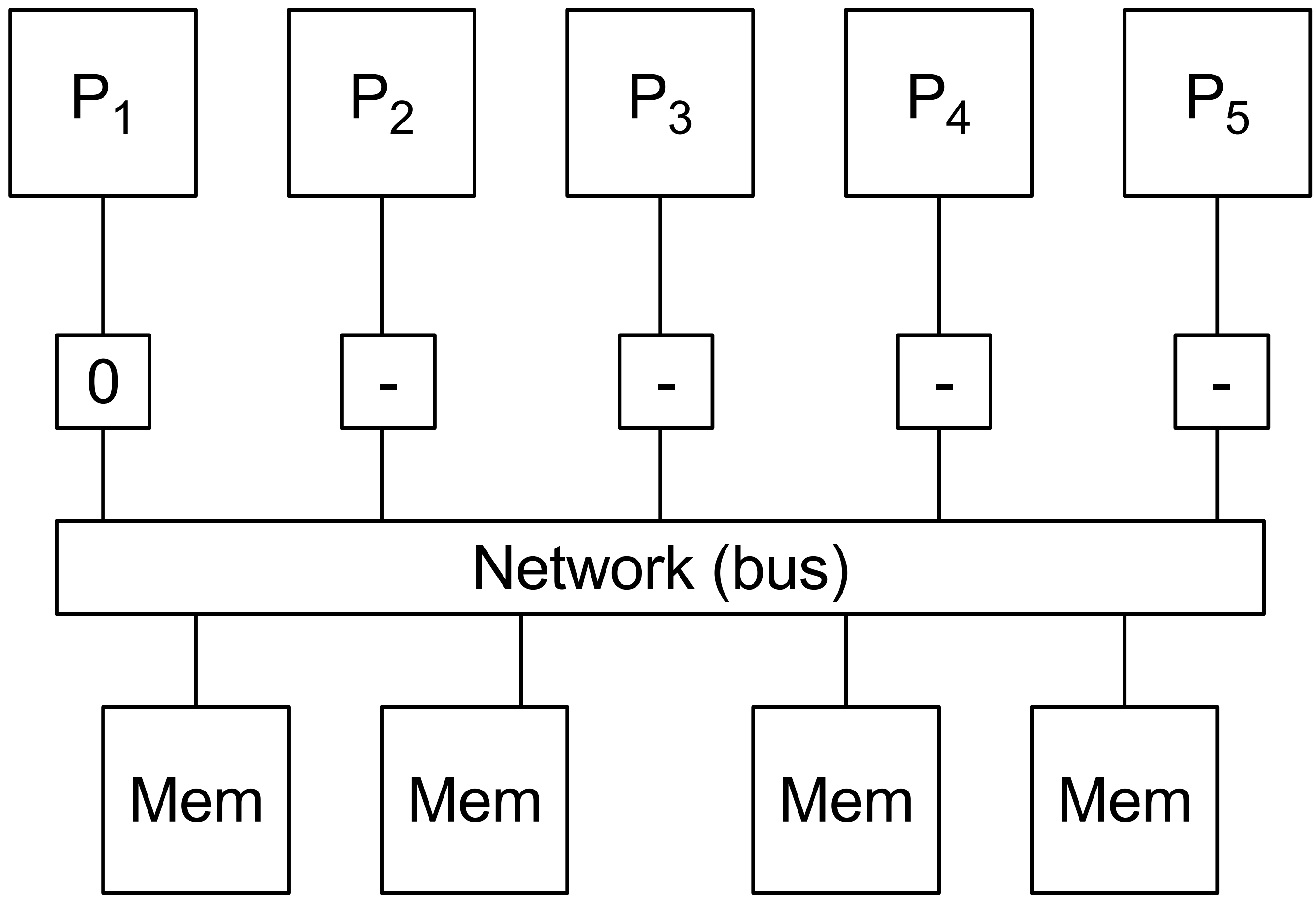
Simple test&set lock



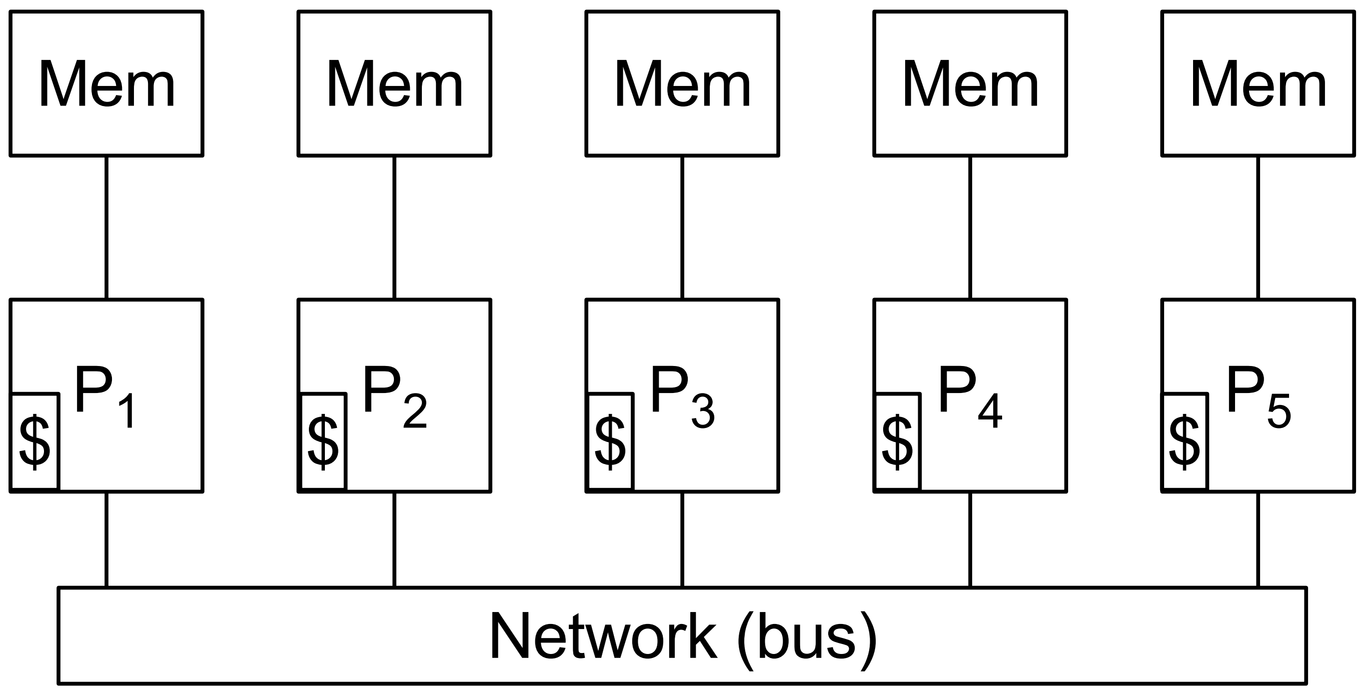
Simple test&set lock



Simple test&set lock



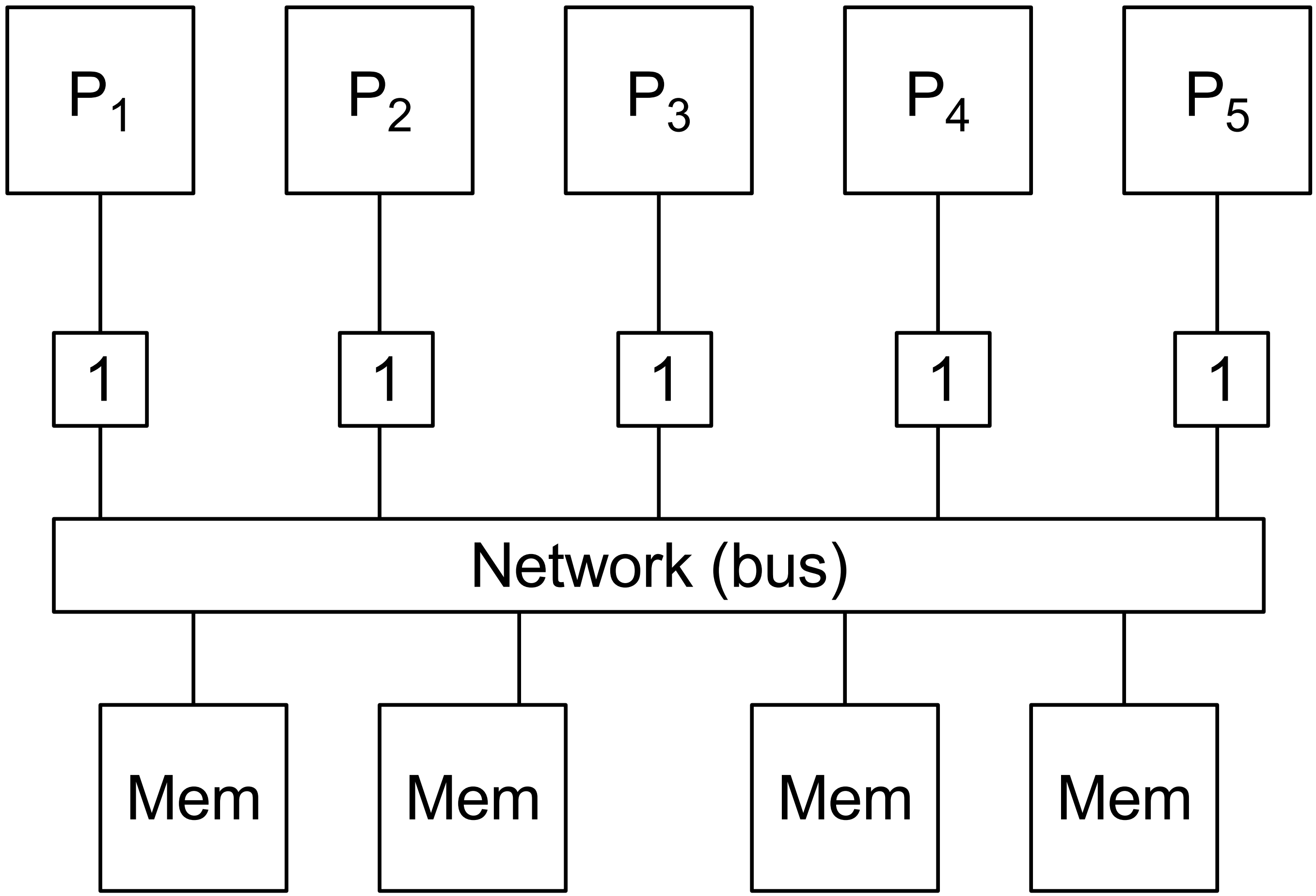
Simple test&set lock



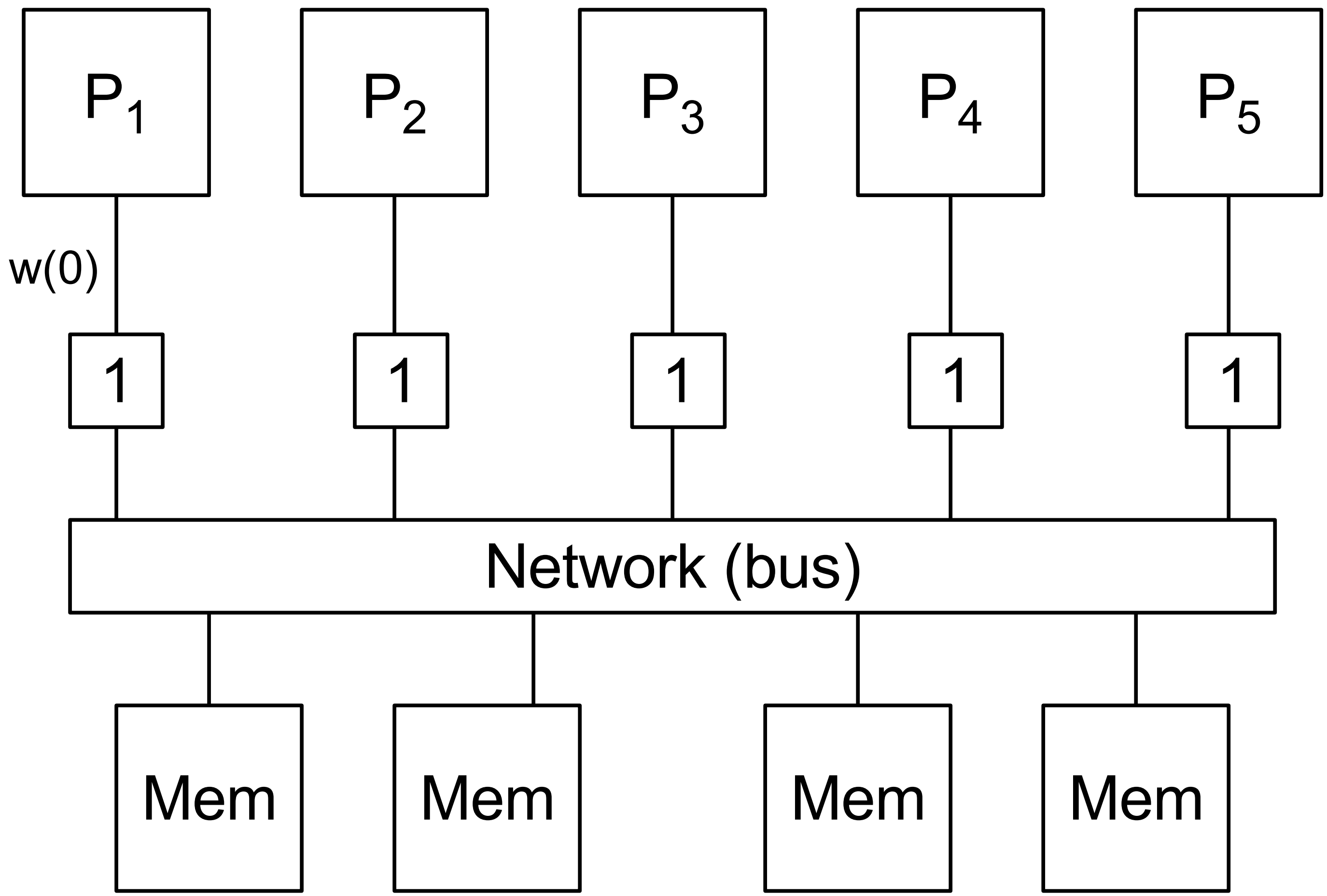
Test-and-test&set lock

- dealing with high contention
  - test-and-test&set
    - read before attempting test&set
    - reduces network traffic (but it's still high!)

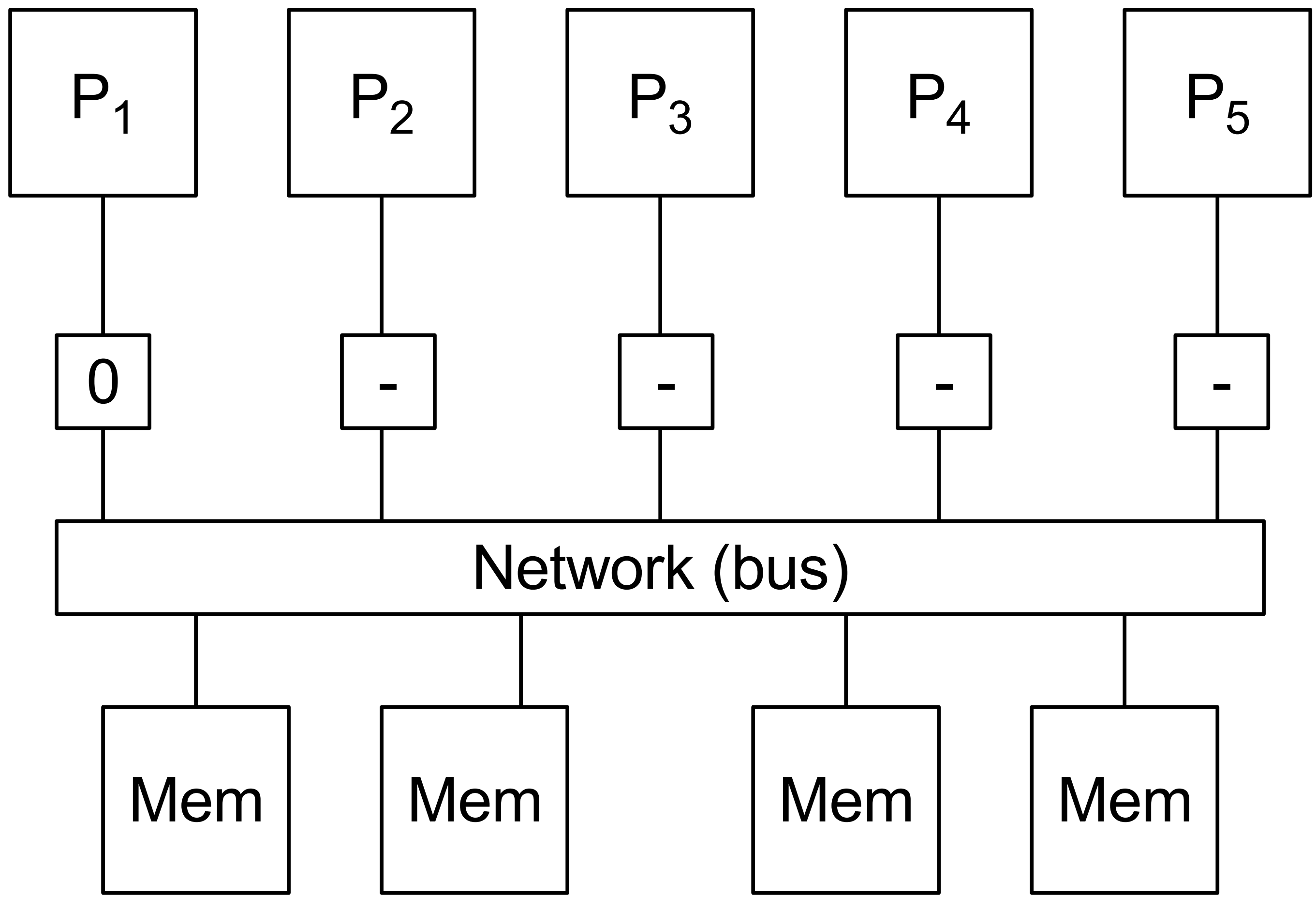
Test-and-test&set lock



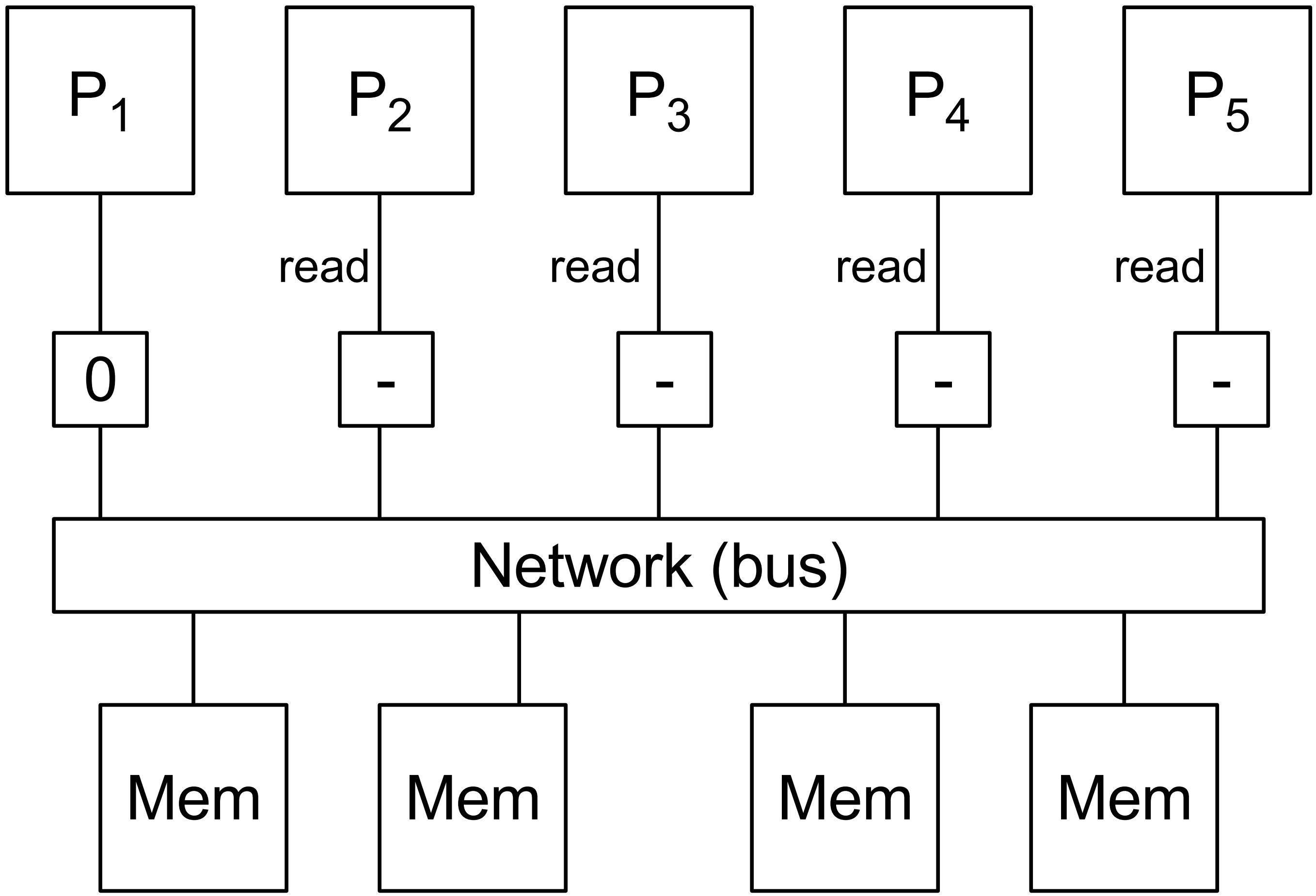
Test-and-test&set lock



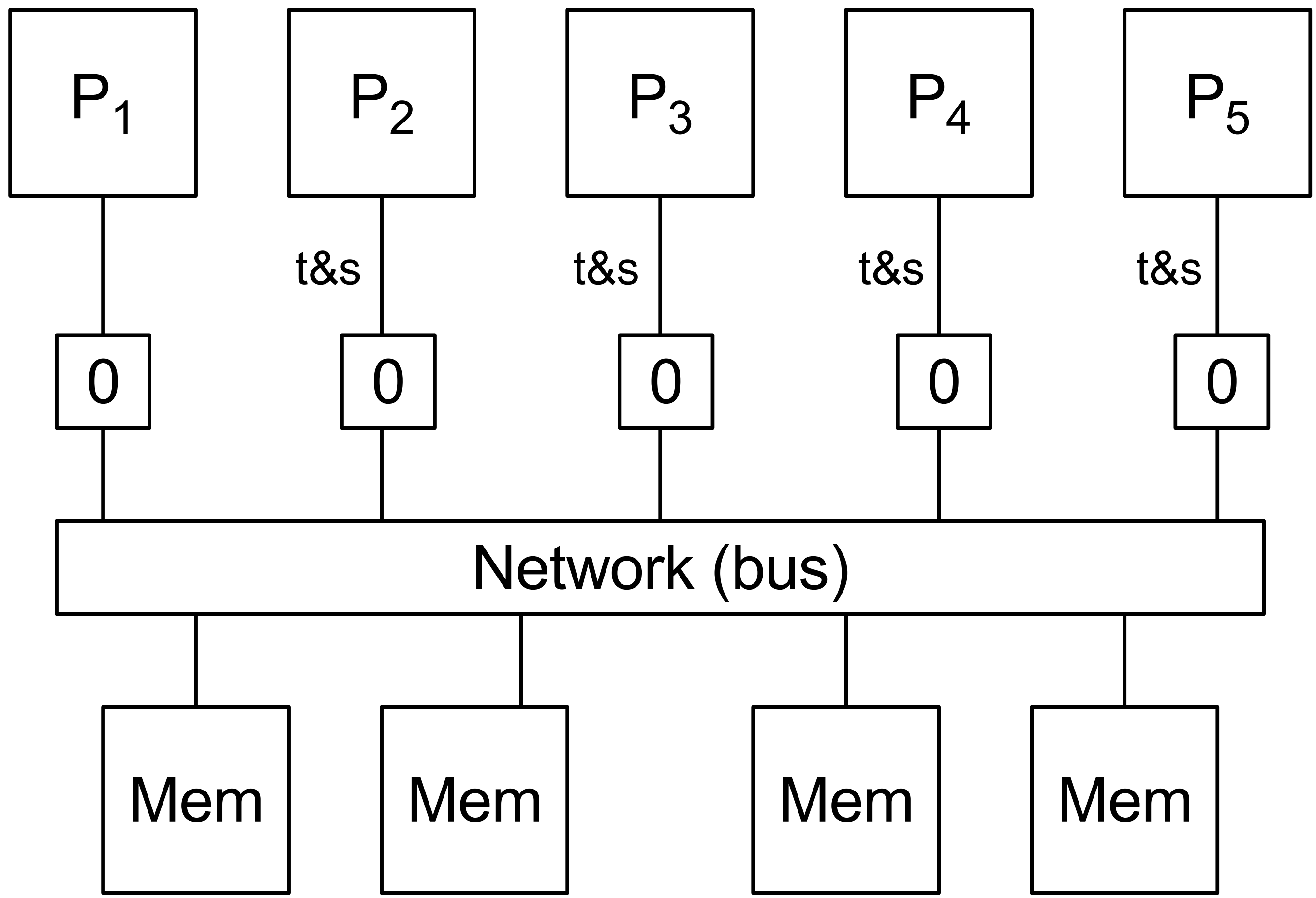
Test-and-test&set lock



Test-and-test&set lock



Test-and-test&set lock



Simple test&set lock with backoff

- dealing with high contention
  - test-and-test&set
    - read before attempting test&set
    - reduces network traffic (but it's still high!)
  - test&set with backoff
    - if test&set “fails” (returns 1), wait before trying again
      - 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 use backoff: but delay potentially more costly
  - proportional backoff seems best
    - delay depends on difference between ticket and **granted**

Array-based queue locks

- Each process spins on a different location
  - reduces invalidation traffic
    - each entry in array must be in separate cache line
  - high space cost: one location (cache line) per lock per process
    - **not** adaptive

Anderson 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 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

Anderson 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

Graunke/Thakkar lock

lockval: array[1..N] of {0,1}; initially all 1
tail: (1..N, {0,1}); initially (X,0) (X means "don't care")

try_i
  (pred,locked) := swap(**tail**,(i,**lockval**[i]))
  waitfor(**lockval**[pred] ≠ locked)
$crit_i$

$exit_i$
  lockval[i] := 1-**lockval**[i]
rem_i

- each entry belongs to some process (single-writer)
  - contains a bit indicating whether in T or C, or done
  - meaning of bit toggles
- tail contains last process in queue and meaning of bit
  - could use pointer instead of process name for linked list
  - but can't use "node" for other purposes (why?)

Mellor-Crummey/Scott lock

"probably the most influential practical mutual exclusion algorithm of all time." -- 2006 Dijkstra Prize citation

- each process has its own "node"
  - but others may write its node
  - spin only on local node (good for "cacheless" architectures)
- can "reuse" node for different locks (or free space)
  - space overhead: O(L+N) or O(L+kN), k = #locks held at once
  - can allocate nodes as needed (typically thread creation)
- can 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

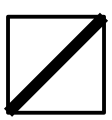
- as with GT, use array to model nodes
- CAS: change value, return true if expected value found
  - alternatively, return value seen regardless

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
```

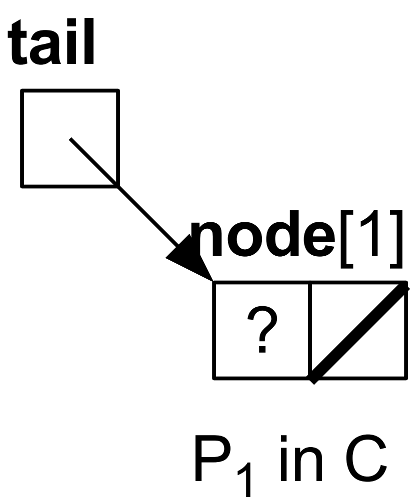
tail



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$_i$

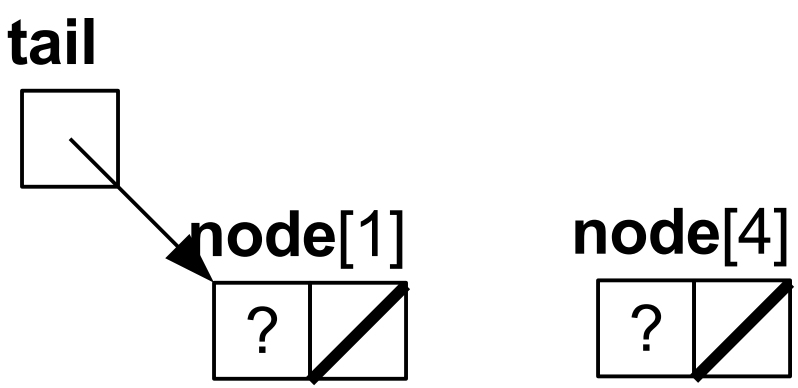
exit$_i$
  if **node**[i].next = 0
    if CAS(**tail**,i,0) return
    waitfor(**node**[i].next ≠ 0)
  node[**node**[i].next].wait := false
rem$_i$



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$_i$

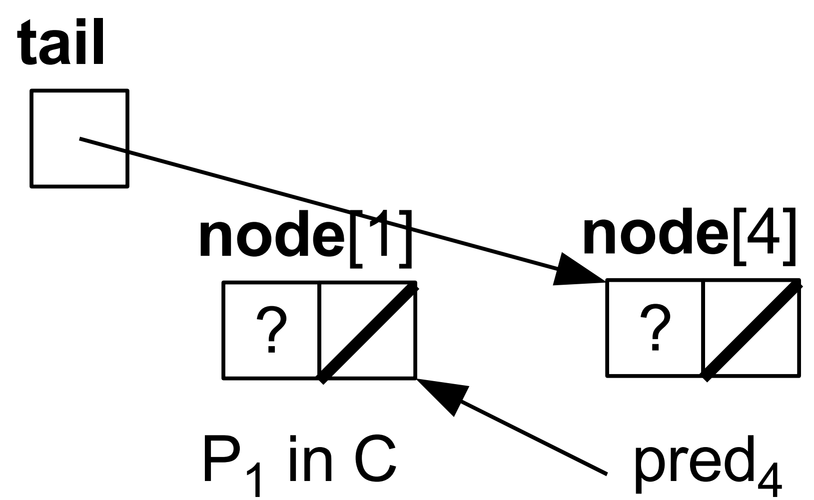


P_1 in C

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)
$crit_i$

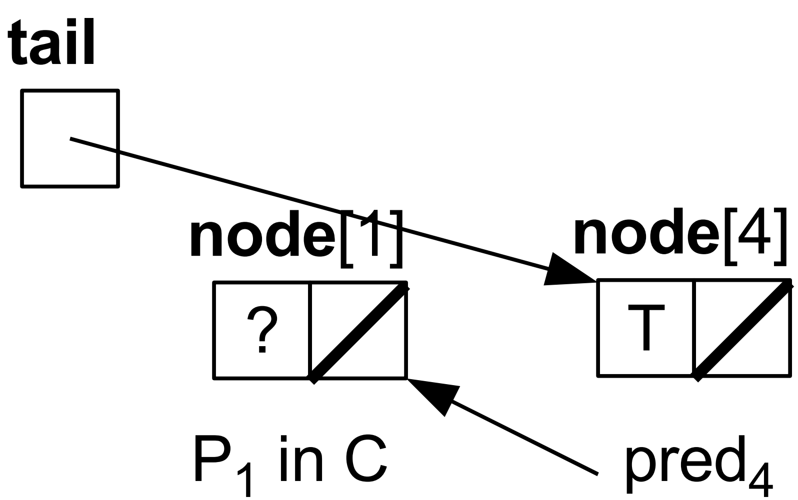
$exit_i$
  if **node**[i].next = 0
    if CAS(**tail**,i,0) return
    waitfor(**node**[i].next ≠ 0)
  node[**node**[i].next].wait := false
rem_i



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
```

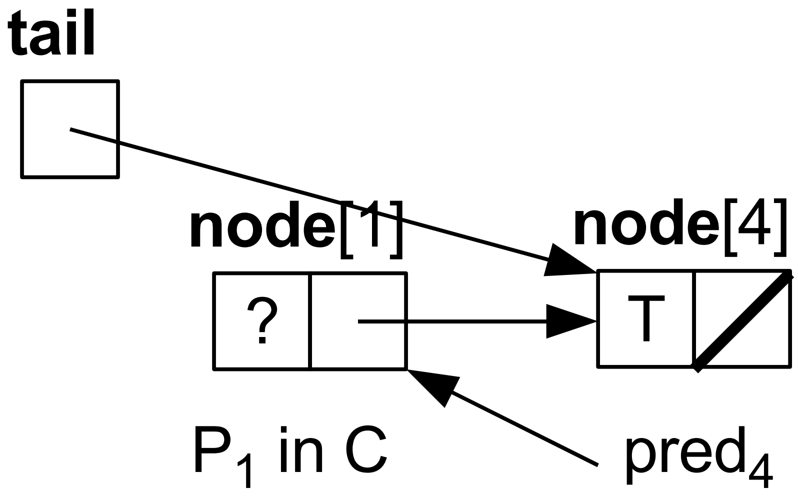
```
exit_i
  if node[i].next = 0
    if CAS(tail,i,0) return
    waitfor(node[i].next ≠ 0)
  node[node[i].next].wait := false
rem_i
```



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)
$crit_i$

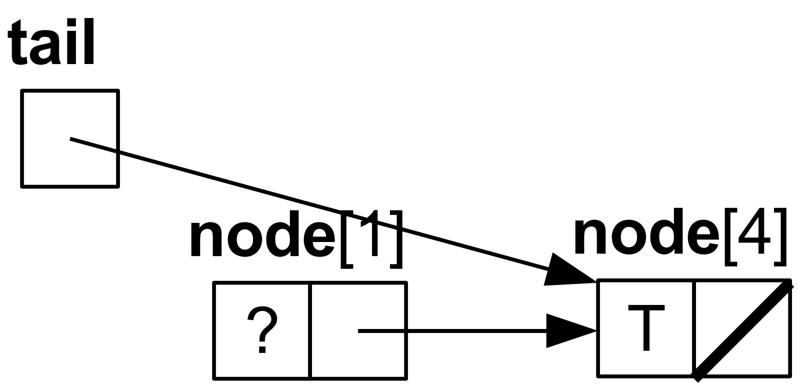
$exit_i$
  if **node**[i].next = 0
    if CAS(**tail**,i,0) return
    waitfor(**node**[i].next ≠ 0)
  node[**node**[i].next].wait := false
rem_i



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$_i$



P_1 in C

P_4 waiting

Mellor-Crummey/Scott lock

try_i

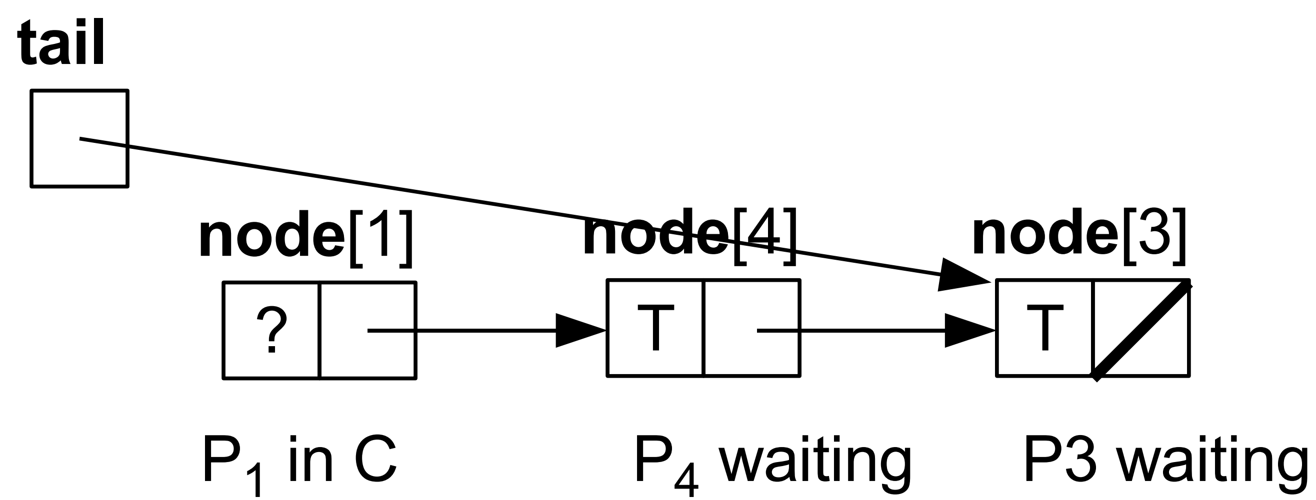
  node[i].next := 0
  pred := swap(**tail**,i)
  if pred ≠ 0
    node[i].wait := true
    node[pred].next := i
    waitfor(¬**node**[i].wait)

$crit_i$

$exit_i$

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

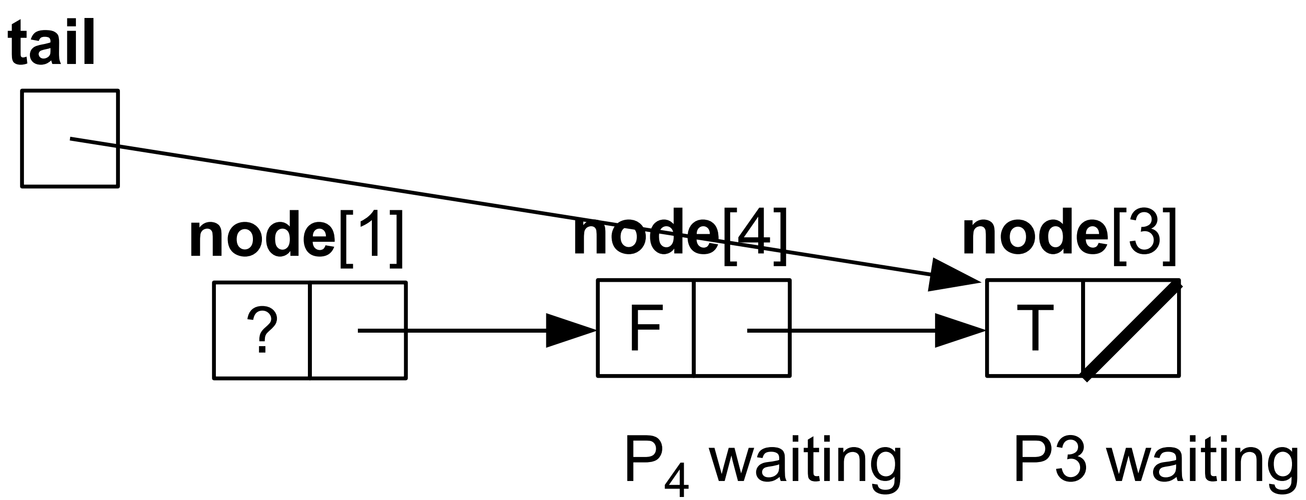
rem_i



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



Mellor-Crummey/Scott lock

try$_i$

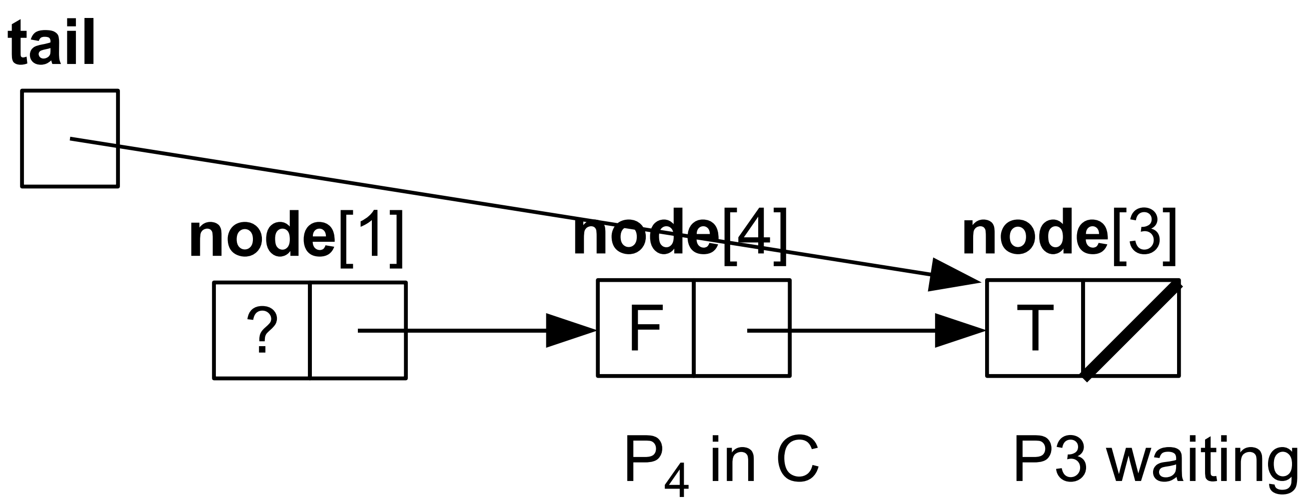
  node[i].next := 0
  pred := swap(**tail**,i)
  if pred ≠ 0
    node[i].wait := true
    node[pred].next := i
    waitfor(¬**node**[i].wait)

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

- eliminates spinning on exit by looking at pred node
  - list is linked “backwards” (only implicitly via local pred)
  - needs one node always at lock; take predecessor on exit
  - not good on cacheless architectures

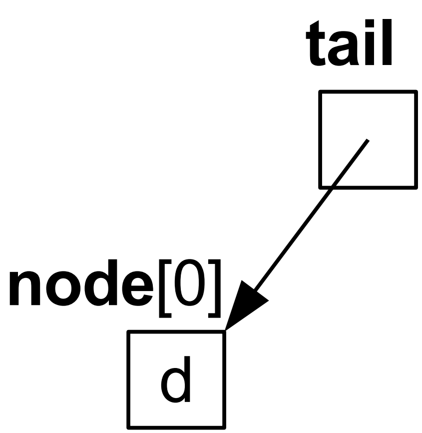
Craig/Landin/Hagersten lock

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$_i$



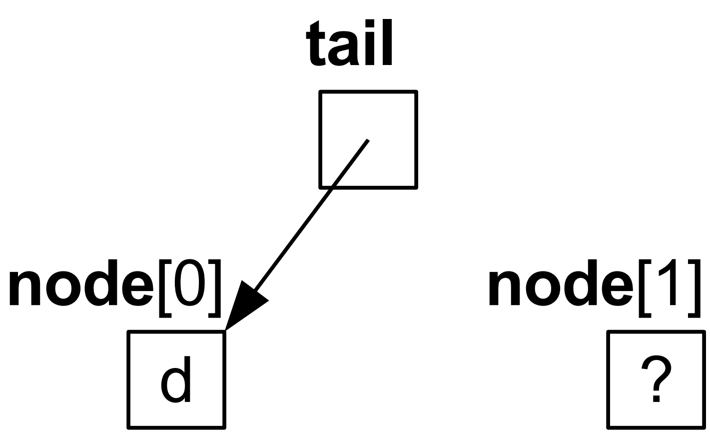
Craig/Landin/Hagersten lock

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$_i$



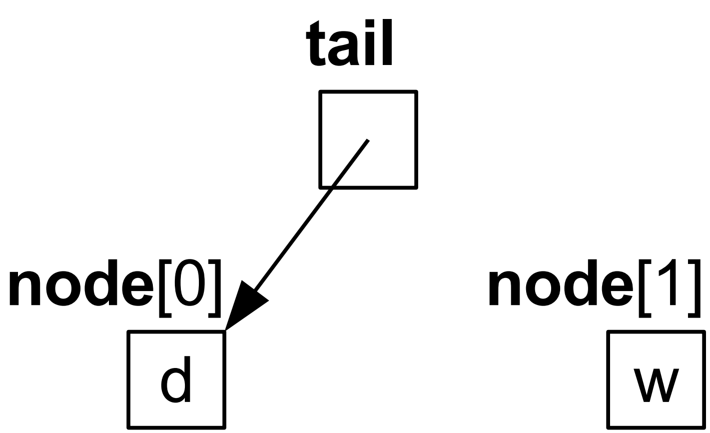
Craig/Landin/Hagersten lock

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



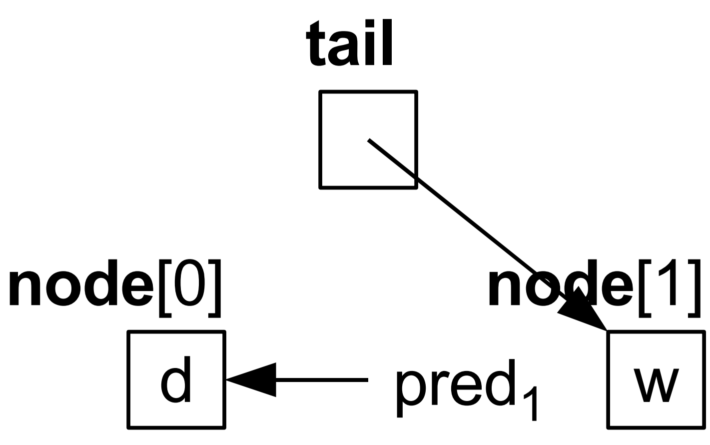
Craig/Landin/Hagersten lock

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



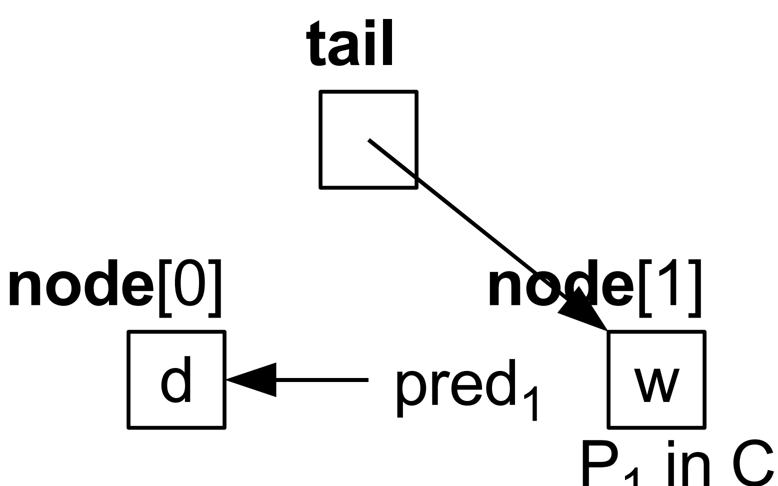
Craig/Landin/Hagersten lock

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



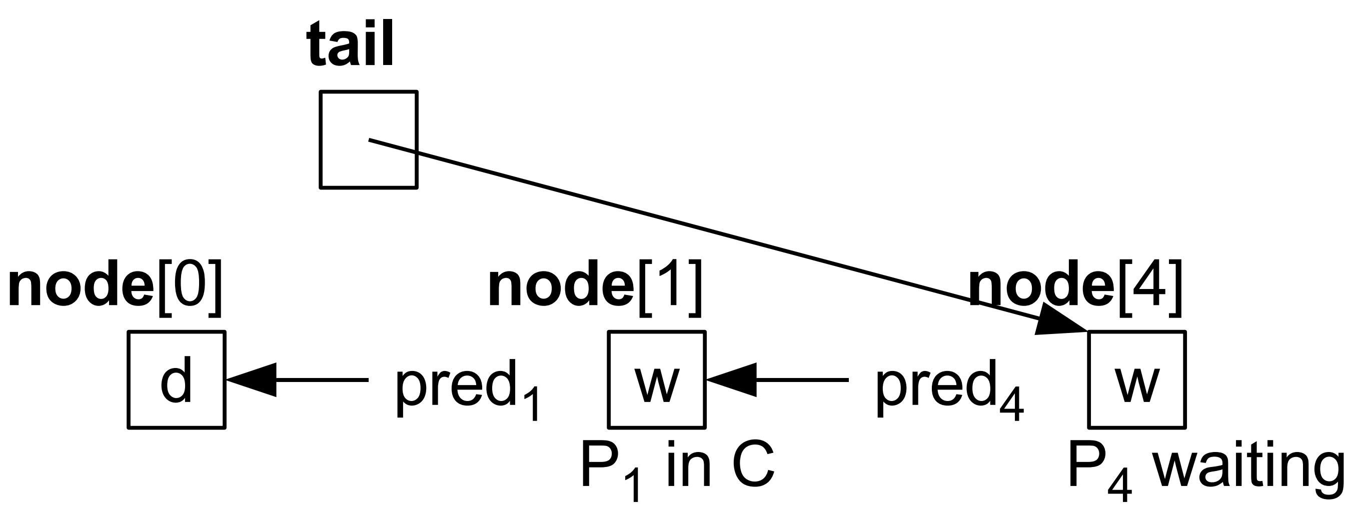
Craig/Landin/Hagersten lock

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$_i$



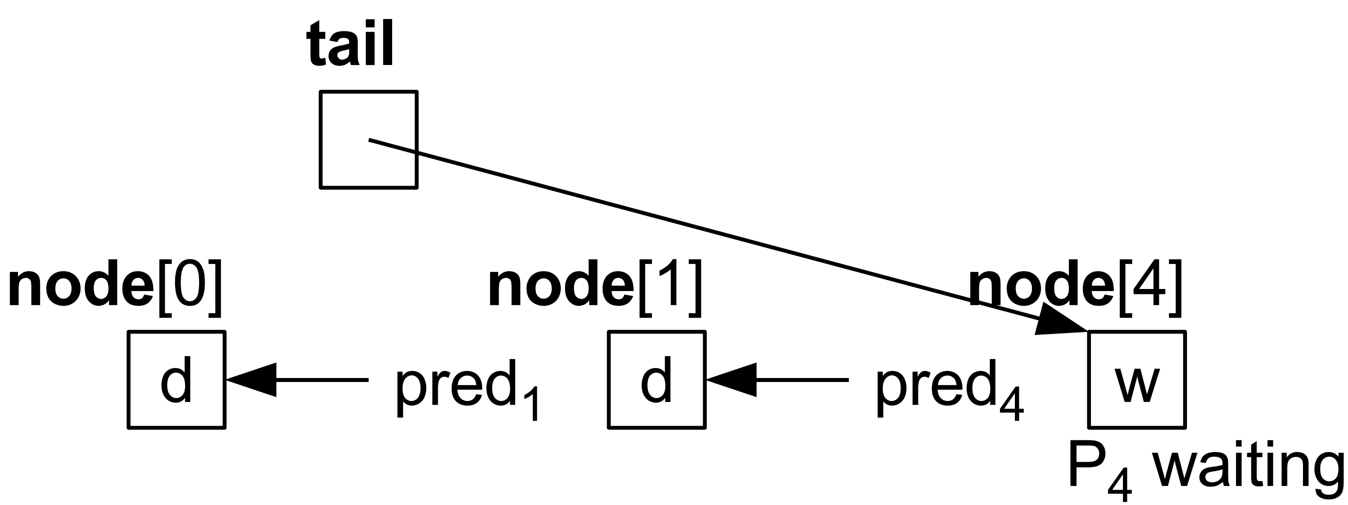
Craig/Landin/Hagersten lock

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



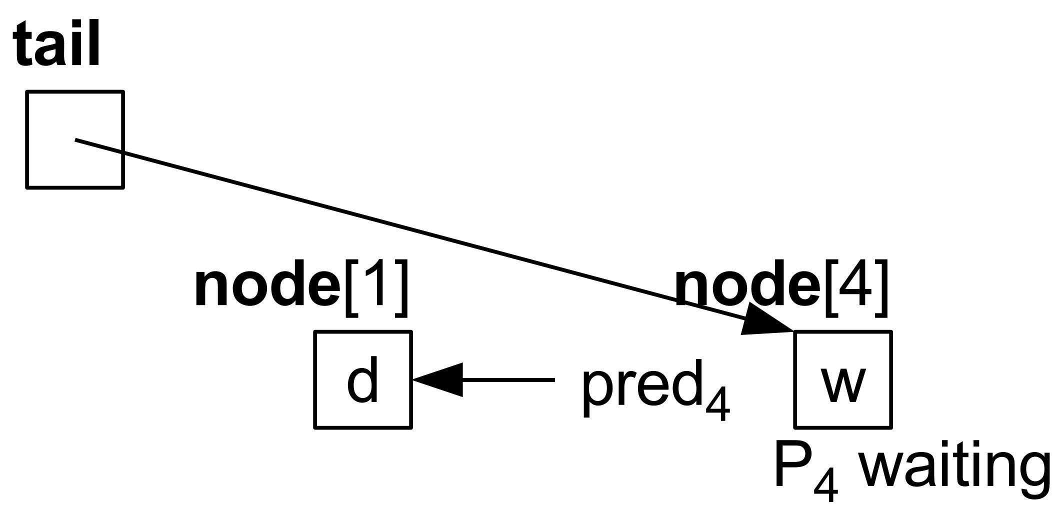
Craig/Landin/Hagersten lock

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



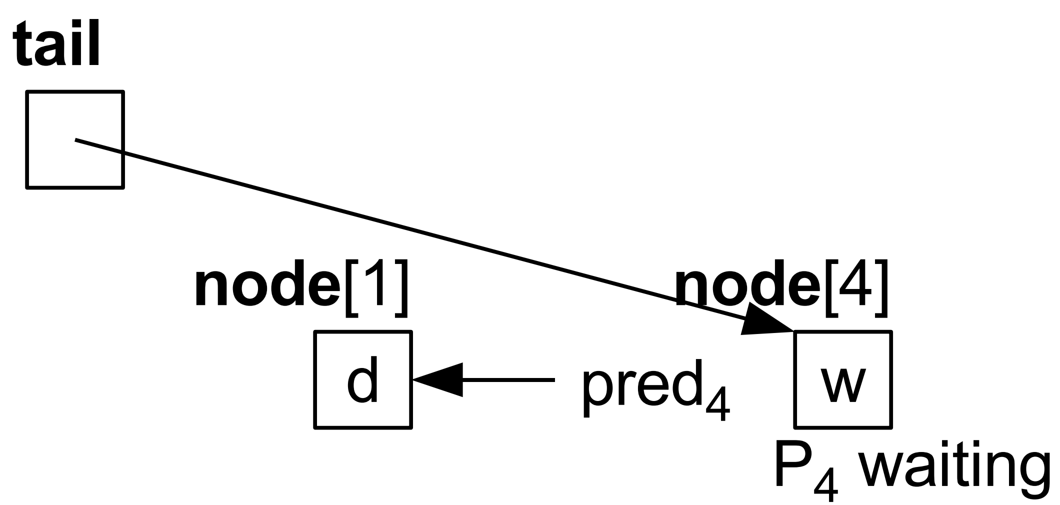
Craig/Landin/Hagersten lock

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



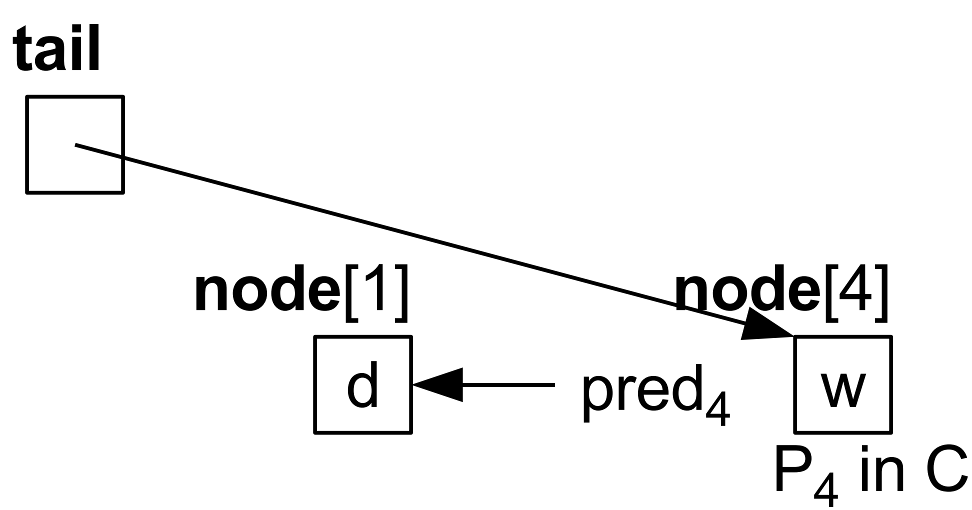
Craig/Landin/Hagersten lock

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$_i$



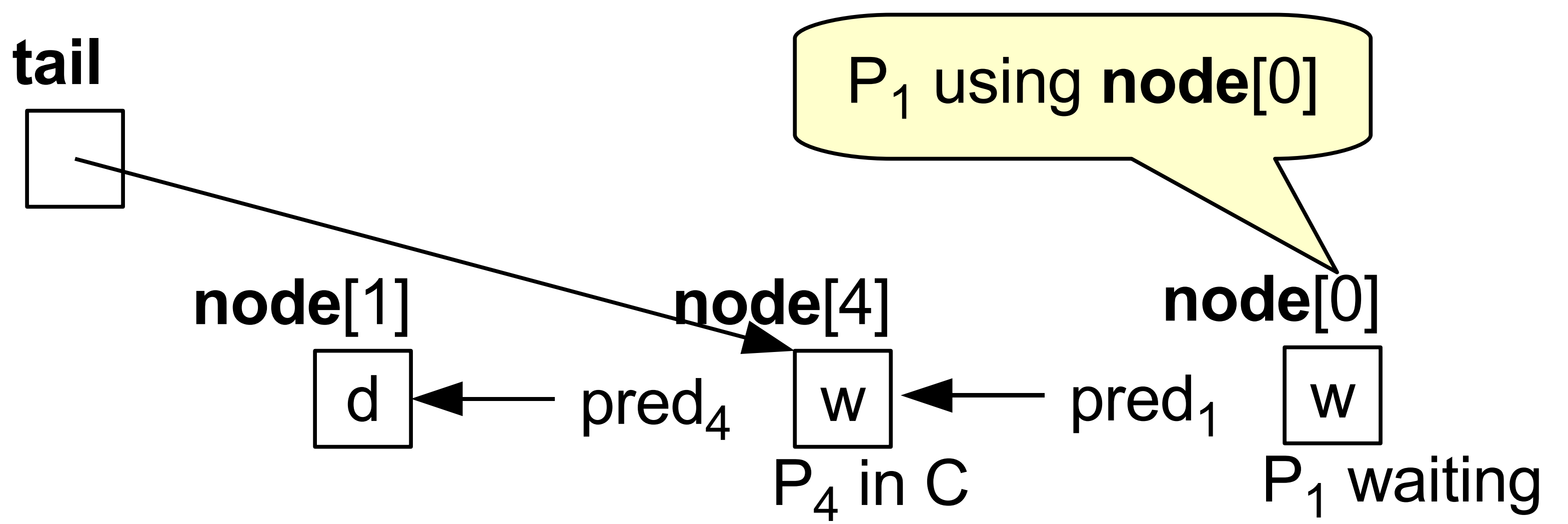
Craig/Landin/Hagersten lock

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 can we do this?
    - easy for test&set locks; harder for queue locks (including 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

- Two ways to generalize mutual exclusion
  - resource spec: different users need different subsets of resources
    - can't share: users with intersecting sets exclude each other
  - exclusion spec: incompatible sets of users
    - more general (any resource spec can be written as exclusion spec)
- Sample problems
  - Dining Philosophers (Dijkstra)
  - k-exclusion (any k users okay, but not k+1)
  - reader/writer locks
    - need further generalization: distinguish different user operations

Generalized resource allocation

- Dining Philosophers
  - neighboring philosophers share a fork
  - need fork on both sides to eat
  - no one should starve
  - can't solve without some symmetry breaking (why?)
  - solutions:
    - number forks around the table; get “smaller” fork first
    - left-right algorithm
- Generalize to solve any resource allocation problem
  - nodes represent resources
  - edge between resources if some user needs both
  - color graph; order colors