Spinlocks

Samy Al Bahra Devon H. O'Dell

Message Systems, Inc.

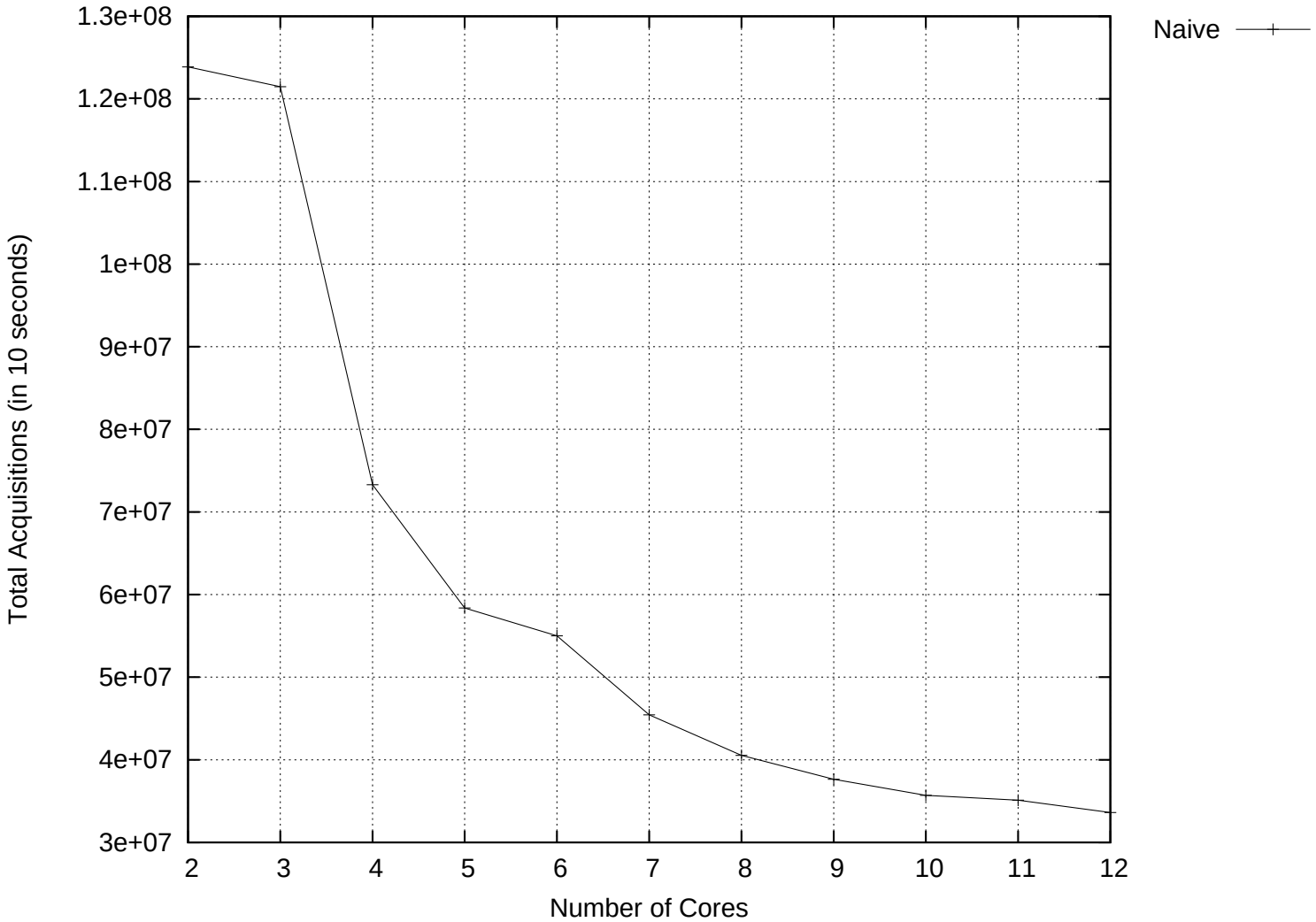
April 8, 2011

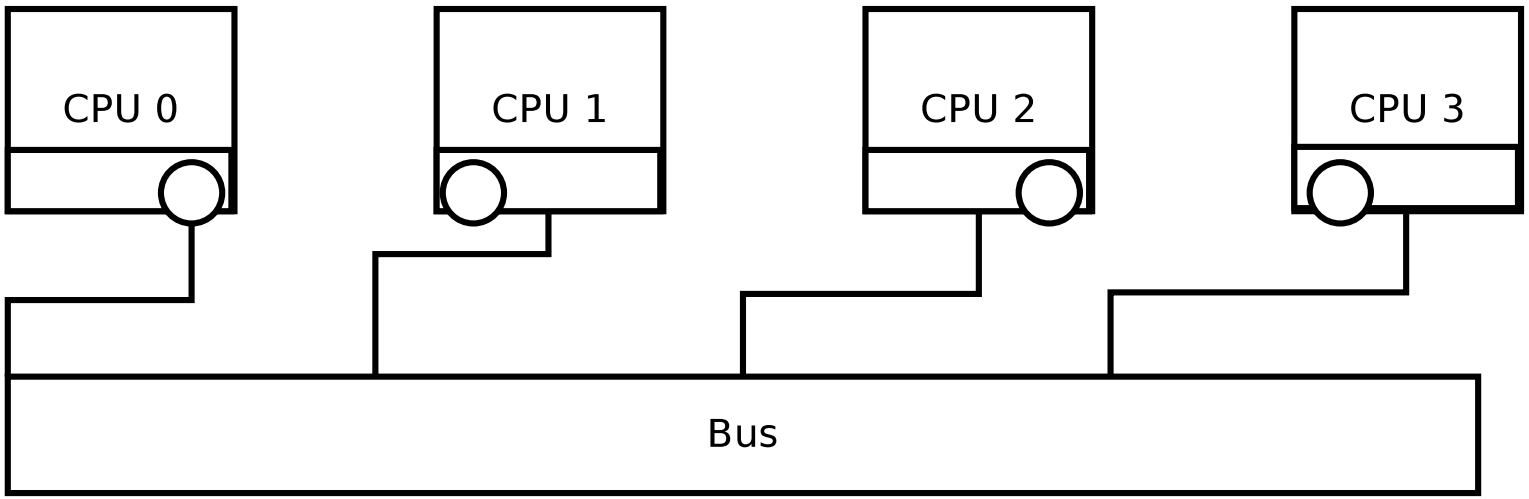
A mutex is an object which implements `acquire` and `relinquish` operations such that the execution following an acquire operation and up to the relinquish operation is executed in a mutually exclusive manner relative to the object implementing a mutex.

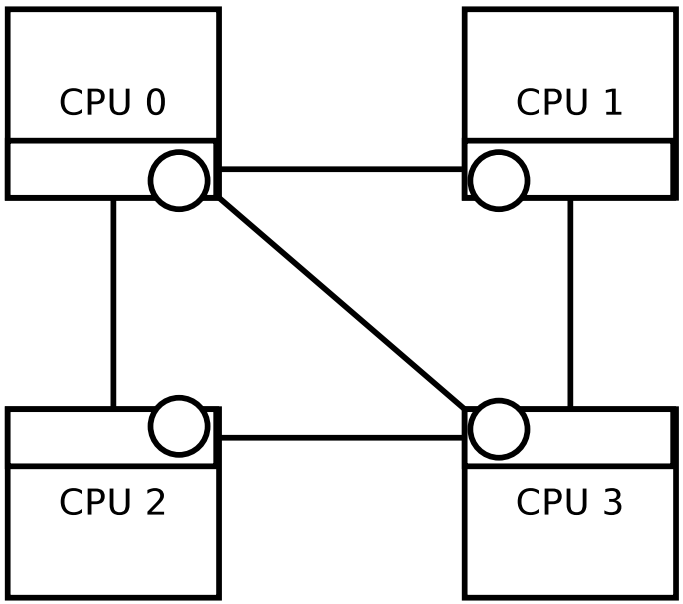
Locks are an implementation of a mutex.

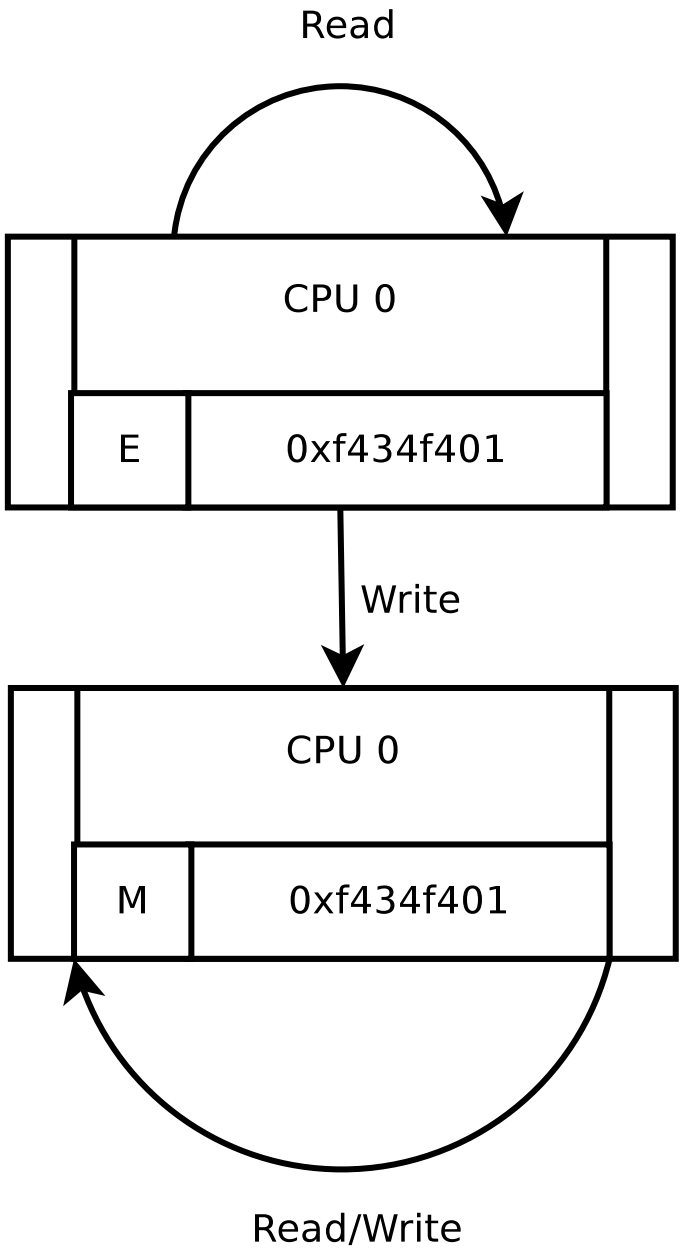
- Sleep lock
  - Any mutex type which deactivates processes that attempt to acquire a mutex that has already been acquired by another process until a relinquish operation on the mutex activates one or more of them.
- Spinlock
  - Any mutex type which forces callers of an acquire operation to spend an unbounded number of processor cycles re-evaluating the availability of the mutex until it has been acquired. The process that invokes acquire is never deactivated before the completion of the acquire operation.
- Spinlocks are preferred to sleep mutexes when the waiting time for a resource is less than the time for the scheduling overhead of process activation/deactivation or when scheduling simply is not possible.

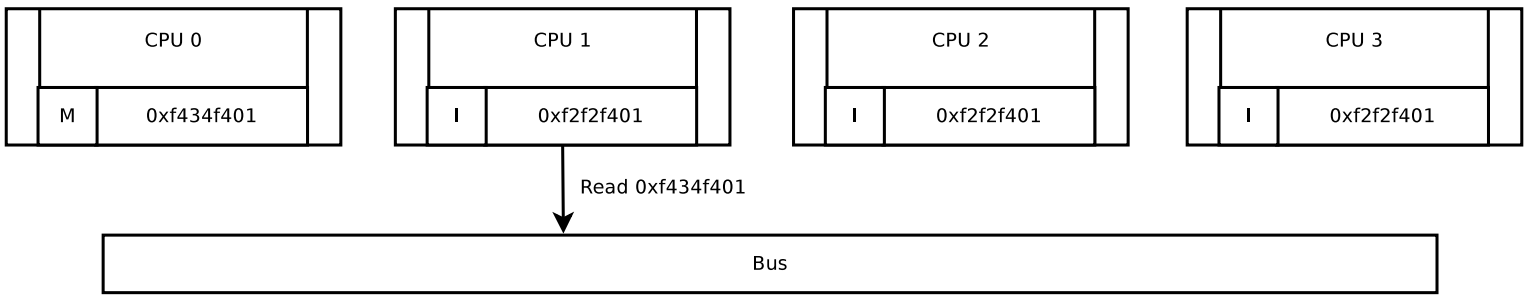
```
void
lock(uint32_t *mutex)
{

        while (ck_pr_fas_32(mutex, true) != false)
                ck_pr_stall();

        return;
}

void
unlock(uint32_t *mutex)
{

        *mutex = false;
        return;
}
```

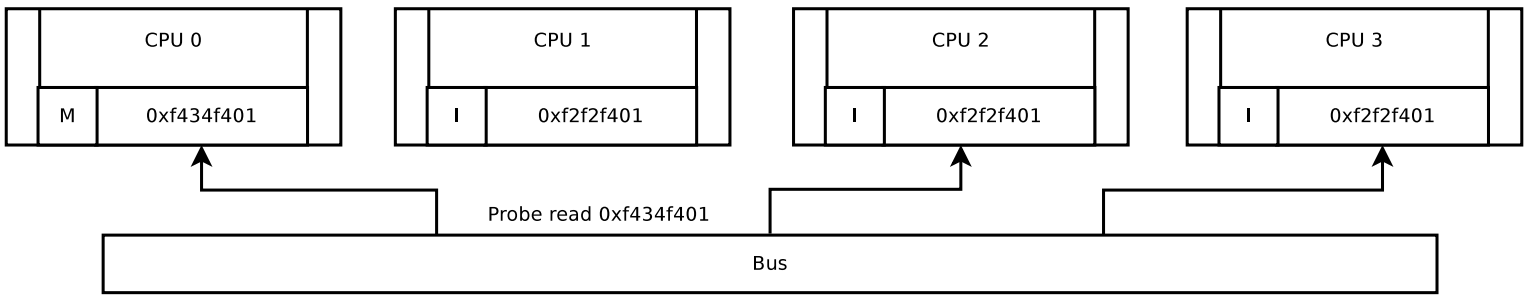

Total Acquisitions (in 10 seconds)
1.3e+08
1.2e+08
1.1e+08
1e+08
9e+07
8e+07
7e+07
6e+07
5e+07
4e+07
3e+07
2
3
4
5
6
7
8
9
10
11
12
Number of Cores
Naive

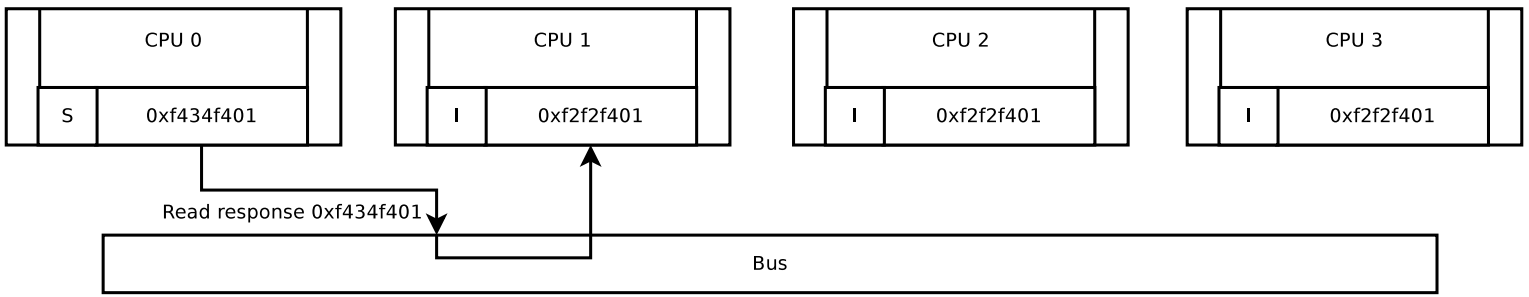

CPU 0
CPU 1
CPU 2
CPU 3
Bus

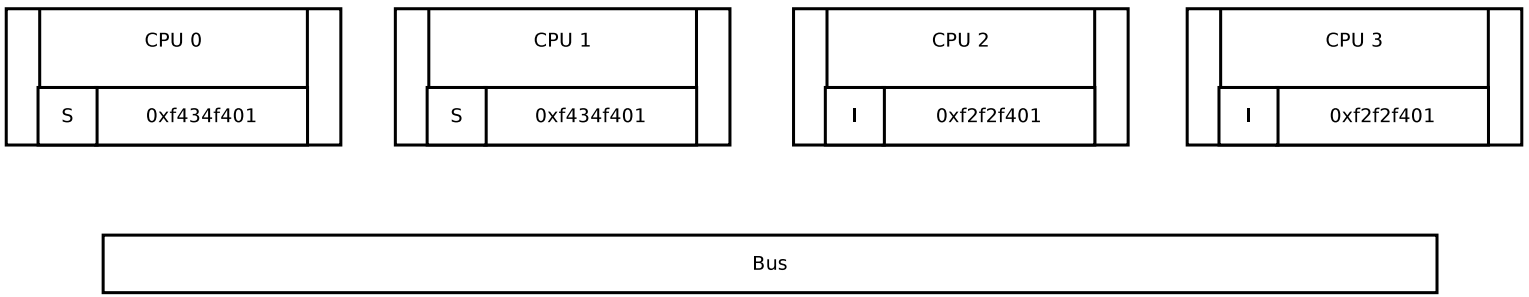

CPU 0
CPU 1
CPU 2
CPU 3

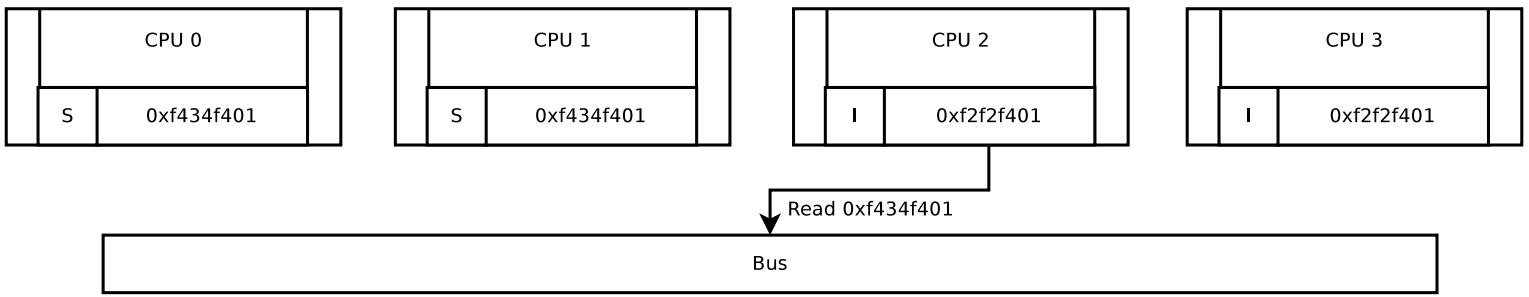

Read
CPU 0
E
0xf434f401
Write
CPU 0
M
0xf434f401
Read/Write

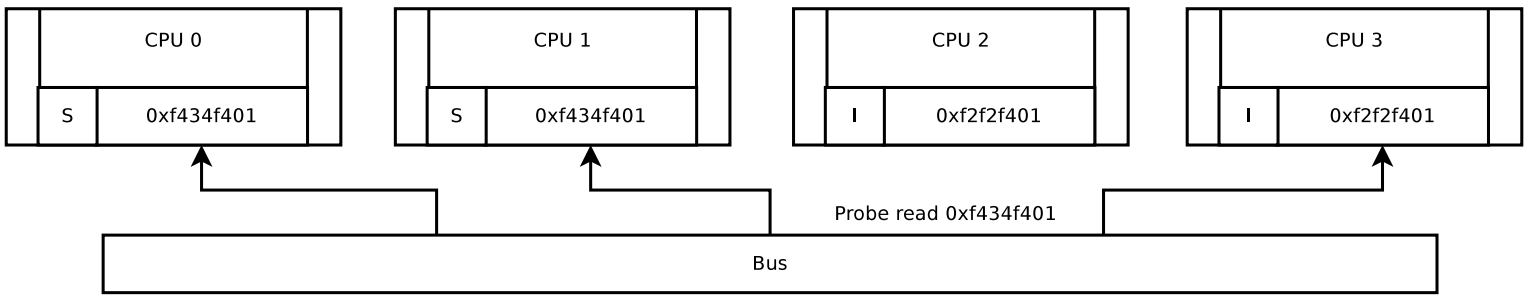

CPU 0
M
0xf434f401
CPU 1
I
0xf2f2f401
CPU 2
I
0xf2f2f401
CPU 3
I
0xf2f2f401
Read 0xf434f401
Bus

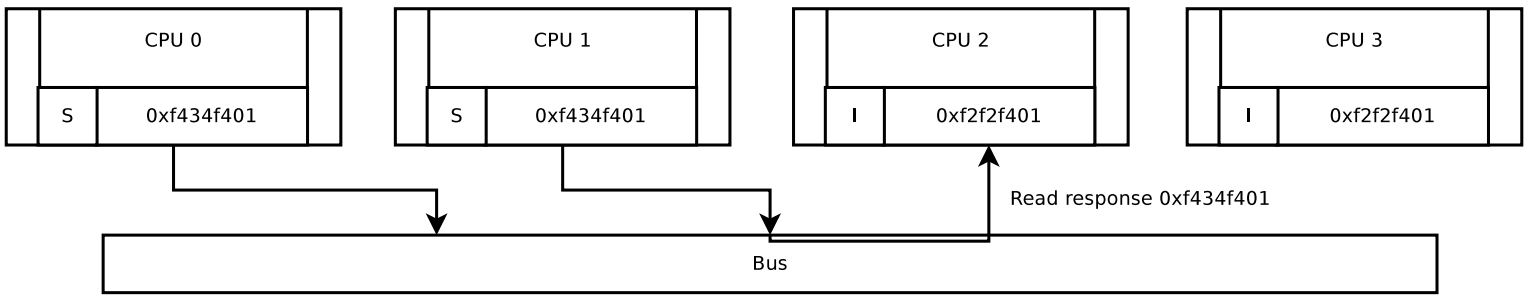

CPU 0
M
0xf434f401
CPU 1
I
0xf2f2f401
CPU 2
I
0xf2f2f401
CPU 3
I
0xf2f2f401
Probe read 0xf434f401
Bus

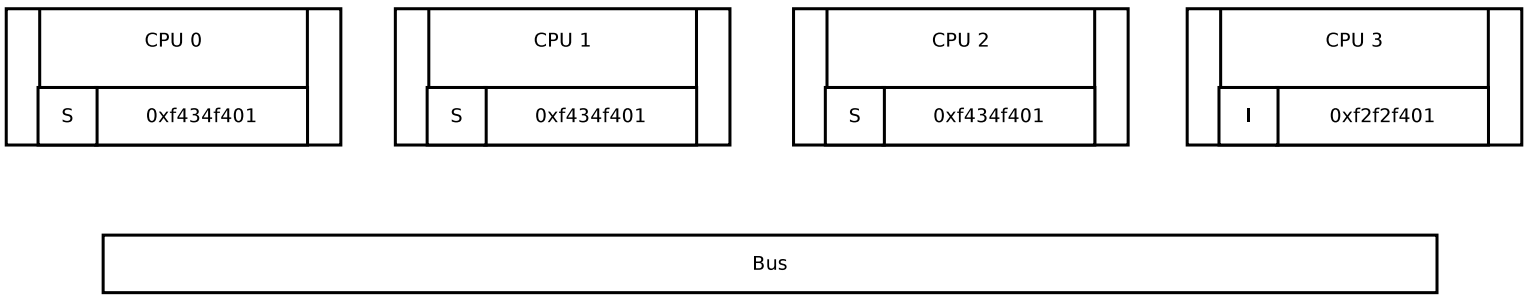

CPU 0
S
0xf434f401
CPU 1
I
0xf2f2f401
CPU 2
I
0xf2f2f401
CPU 3
I
0xf2f2f401
Read response 0xf434f401
Bus

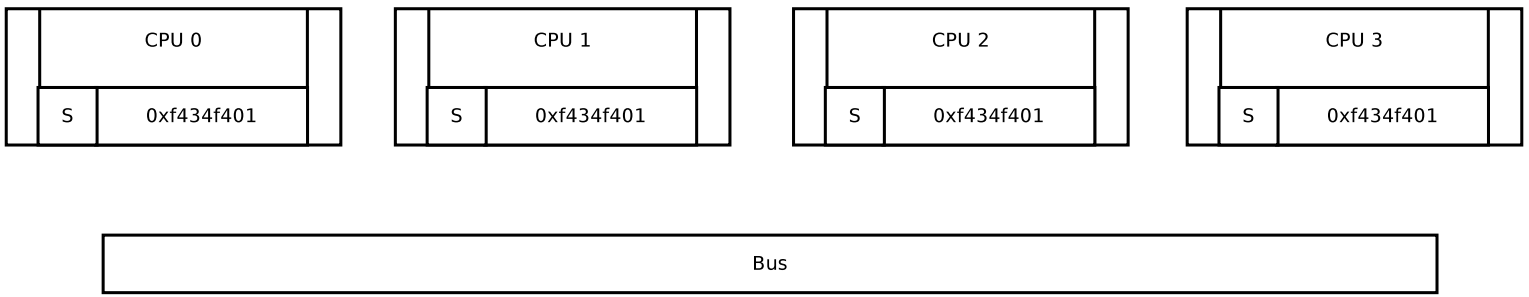

CPU 0
S
0xf434f401
CPU 1
S
0xf434f401
CPU 2
I
0xf2f2f401
CPU 3
I
0xf2f2f401
Bus

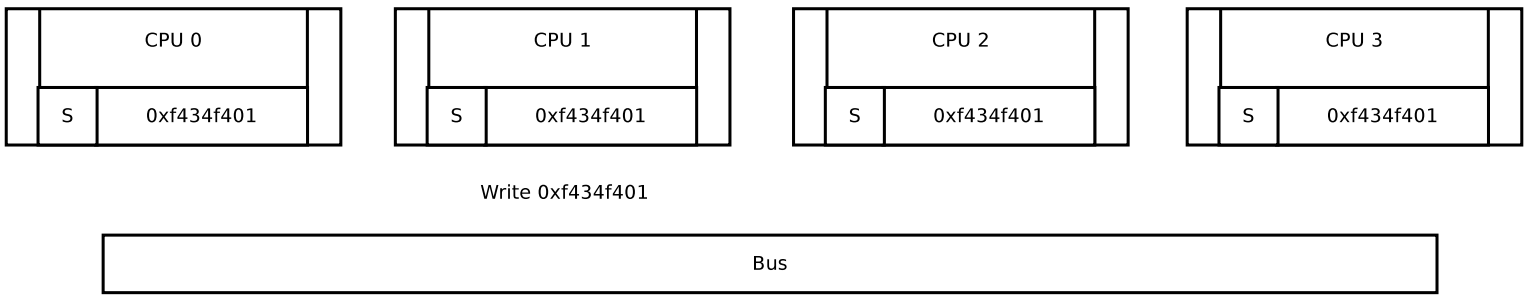

CPU 0
S
0xf434f401
CPU 1
S
0xf434f401
CPU 2
I
0xf2f2f401
CPU 3
I
0xf2f2f401
Read 0xf434f401
Bus

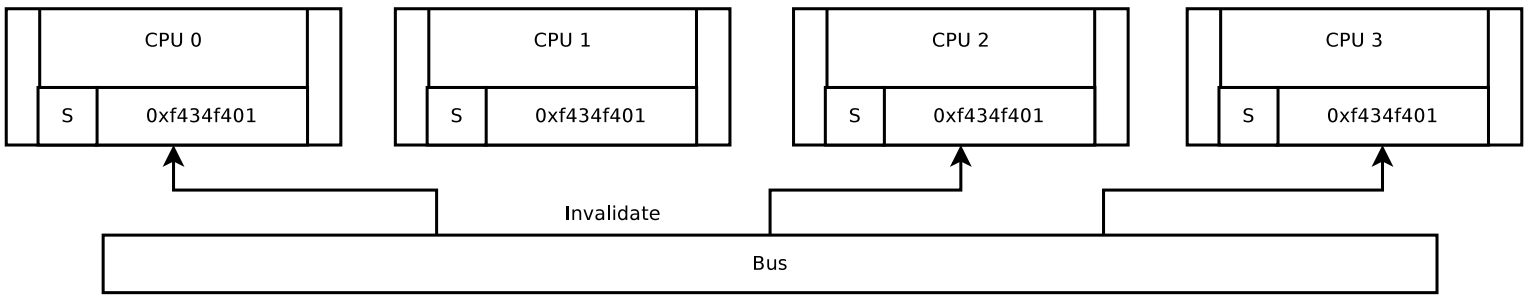

CPU 0
S
0xf434f401
CPU 1
S
0xf434f401
CPU 2
I
0xf2f2f401
CPU 3
I
0xf2f2f401
Probe read 0xf434f401
Bus

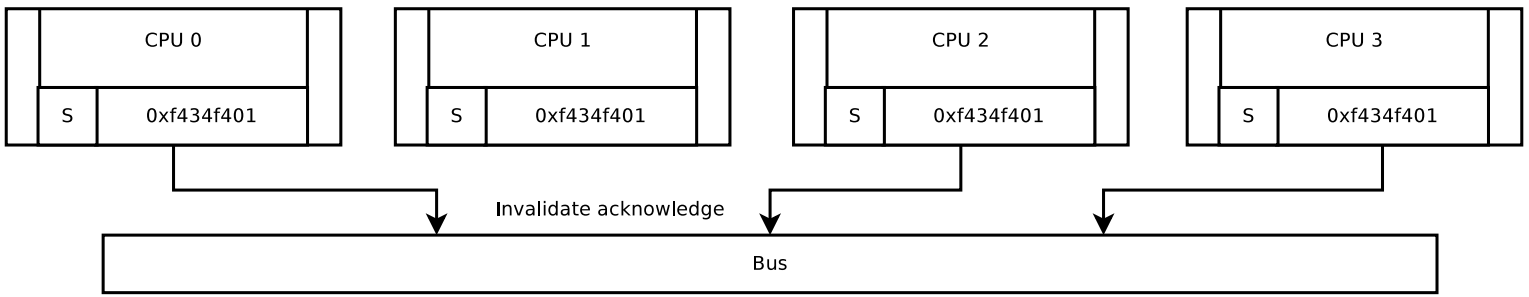

CPU 0
S
0xf434f401
CPU 1
S
0xf434f401
CPU 2
I
0xf2f2f401
CPU 3
I
0xf2f2f401
Read response 0xf434f401
Bus

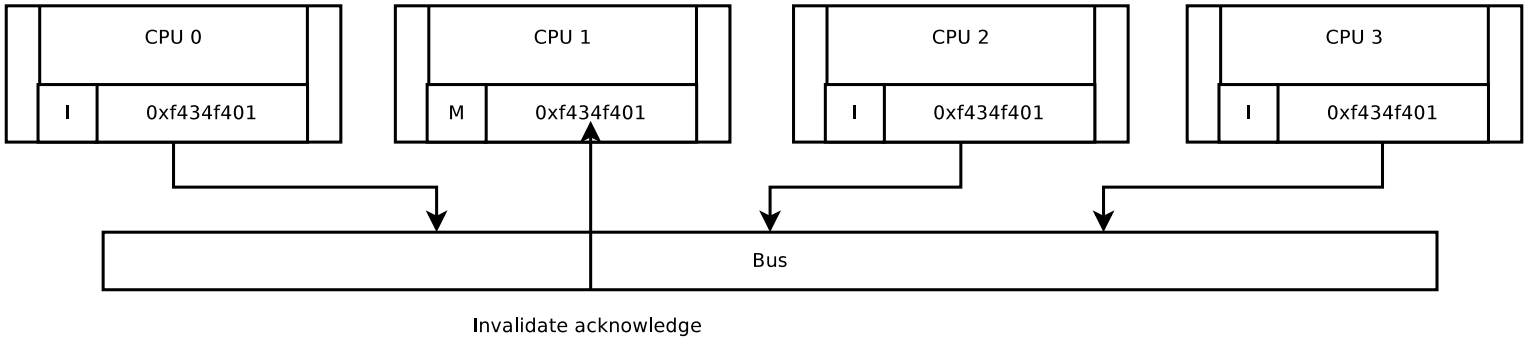

CPU 0
S
0xf434f401
CPU 1
S
0xf434f401
CPU 2
S
0xf434f401
CPU 3
I
0xf2f2f401
Bus


CPU 0
S
0xf434f401
CPU 1
S
0xf434f401
CPU 2
S
0xf434f401
CPU 3
S
0xf434f401
Bus

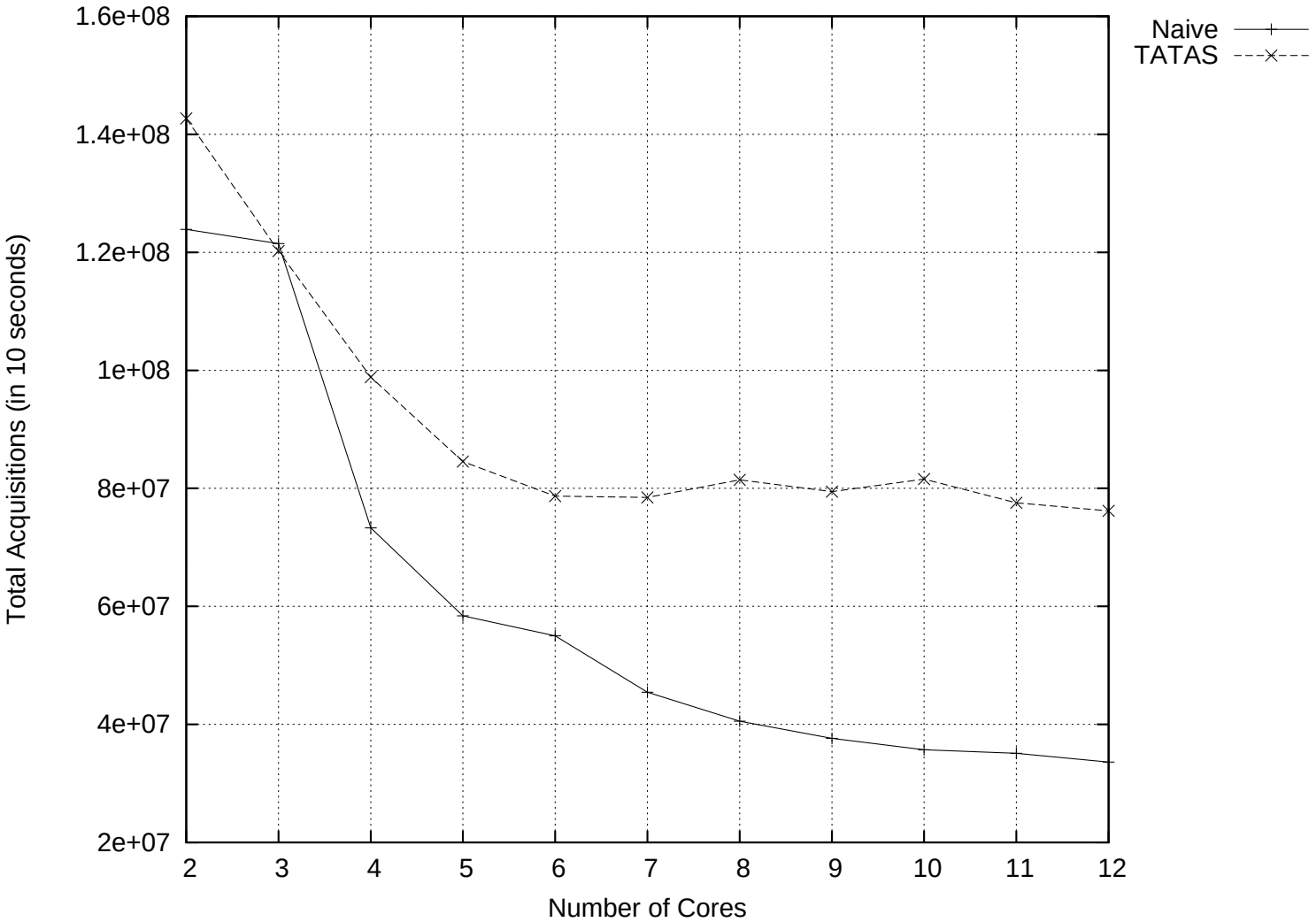

CPU 0
S
0xf434f401
CPU 1
S
0xf434f401
CPU 2
S
0xf434f401
CPU 3
S
0xf434f401
Write 0xf434f401
Bus


CPU 0
S
0xf434f401
CPU 1
S
0xf434f401
CPU 2
S
0xf434f401
CPU 3
S
0xf434f401
Invalidate
Bus

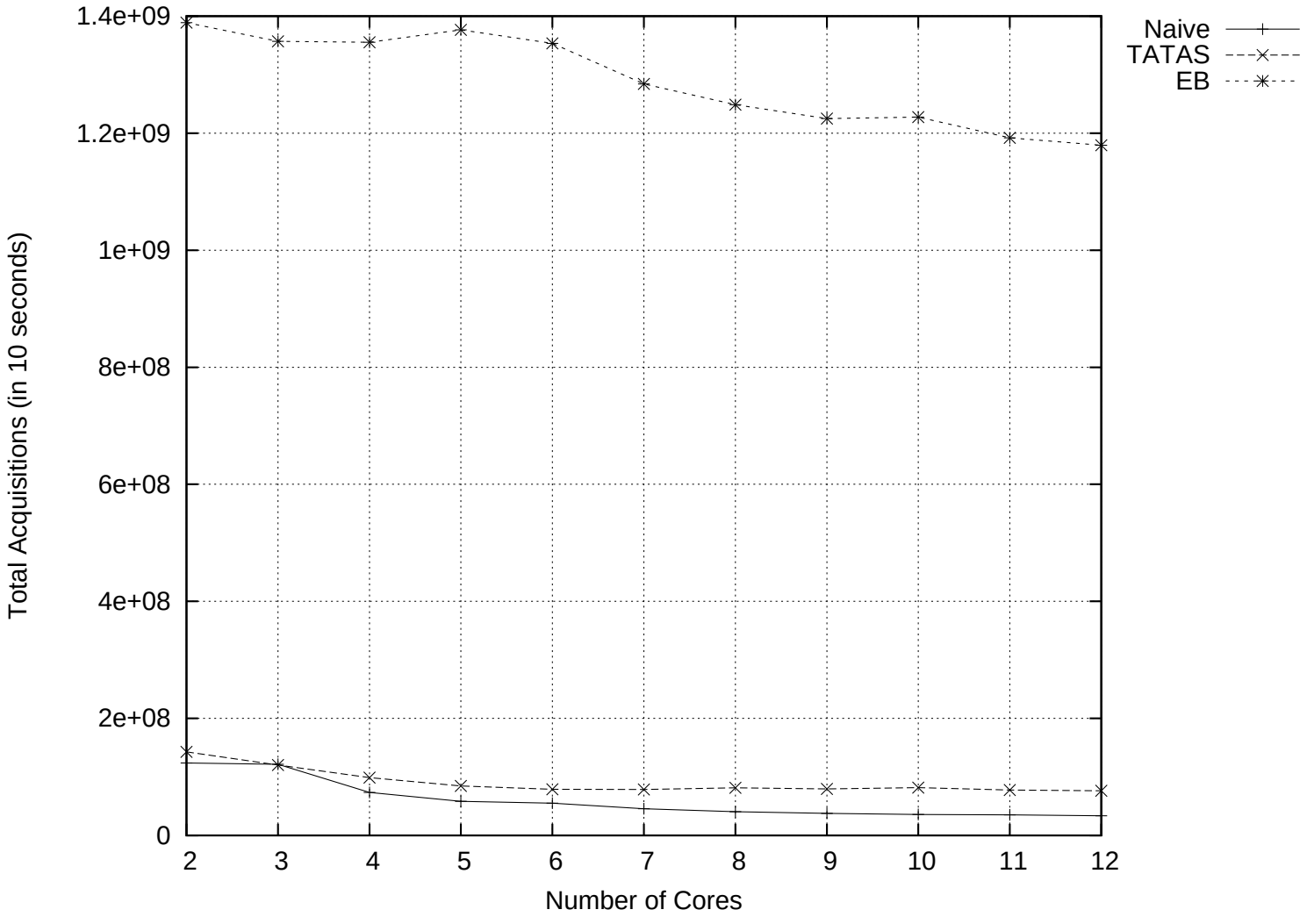

CPU 0
S
0xf434f401
CPU 1
S
0xf434f401
CPU 2
S
0xf434f401
CPU 3
S
0xf434f401
Invalidate acknowledge
Bus

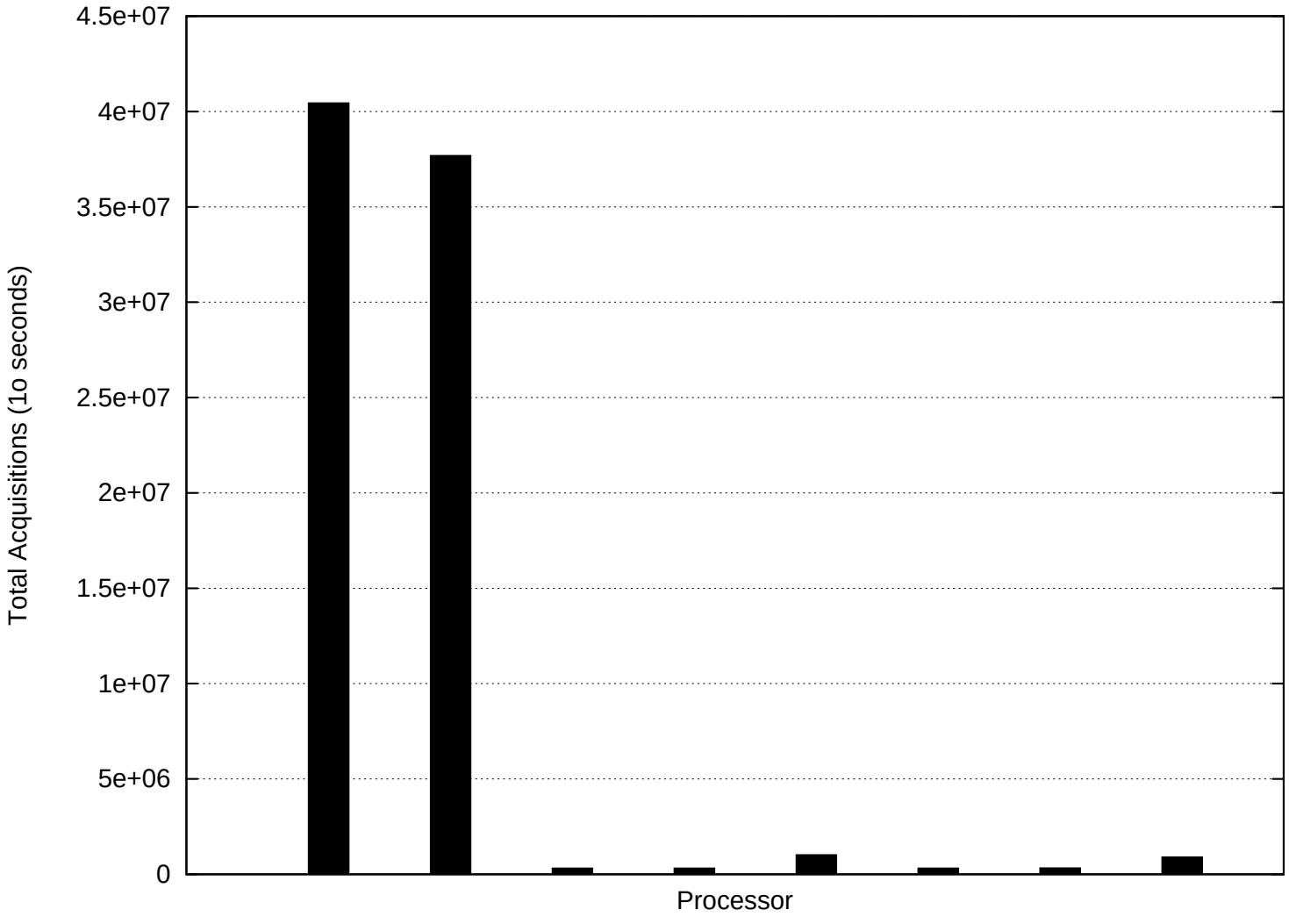

CPU 0
I
0xf434f401
CPU 1
M
0xf434f401
CPU 2
I
0xf434f401
CPU 3
I
0xf434f401
Bus
Invalidate acknowledge

```
void
lock(uint32_t *mutex)
{

        while (ck_pr_fas_32(mutex, true) != false) {
                while (ck_pr_load_32(mutex) == true)
                        ck_pr_stall();
        }

        return;
}

void
unlock(uint32_t *mutex)
{

        *mutex = false;
        return;
}
```

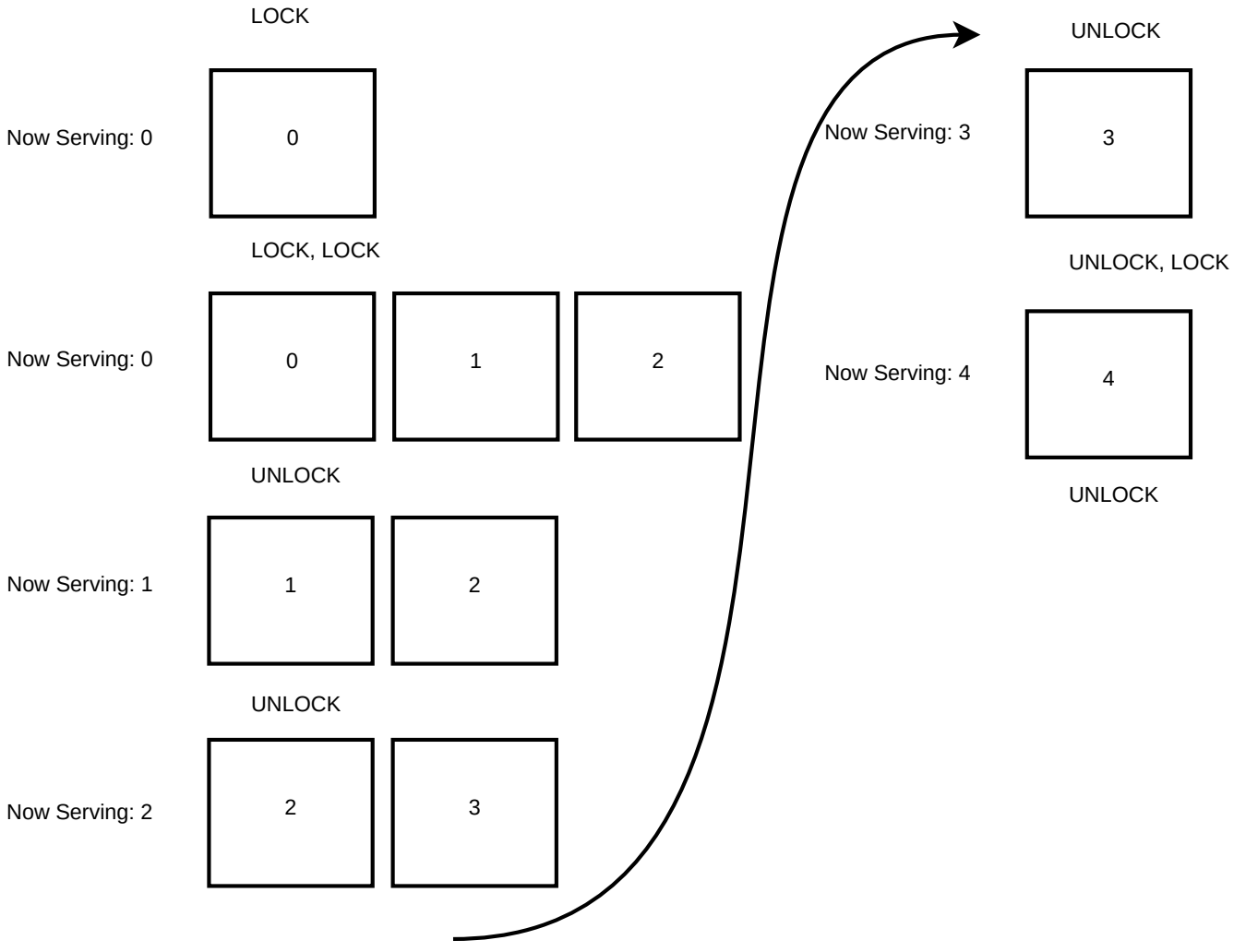

1.6e+08
1.4e+08
1.2e+08
1e+08
8e+07
6e+07
4e+07
2e+07
Total Acquisitions (in 10 seconds)
2
3
4
5
6
7
8
9
10
11
12
Number of Cores
Naive
TATAS

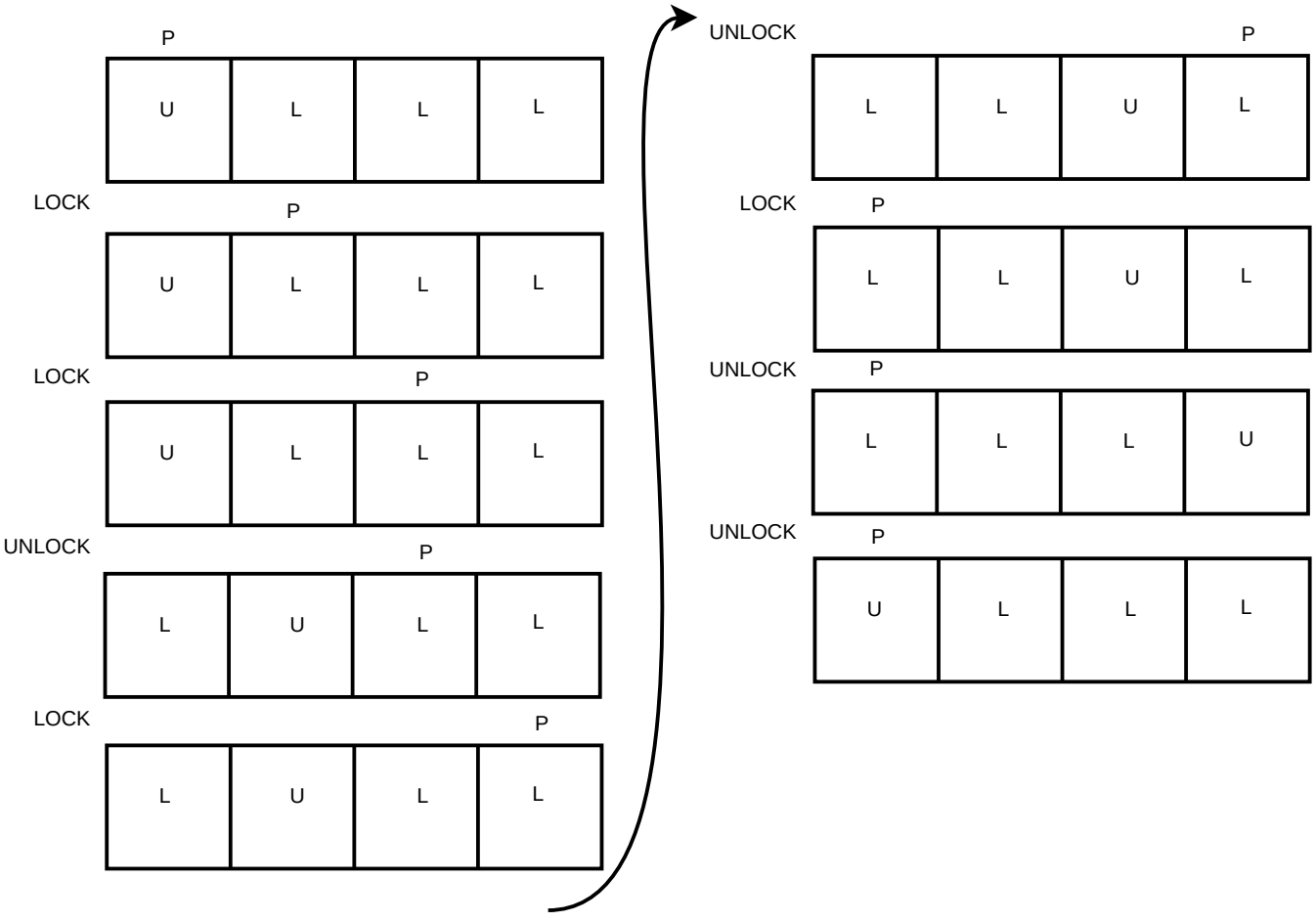
```
void
lock(uint32_t *mutex)
{

        ck_backoff_t backoff = CK_BACKOFF_INITIALIZER;

        while (ck_pr_fas_32(mutex, true) != false)
                ck_backoff_eb(&backoff);

        return;
}

void
unlock(uint32_t *mutex)
{

        *mutex = false;
        return;
}
```

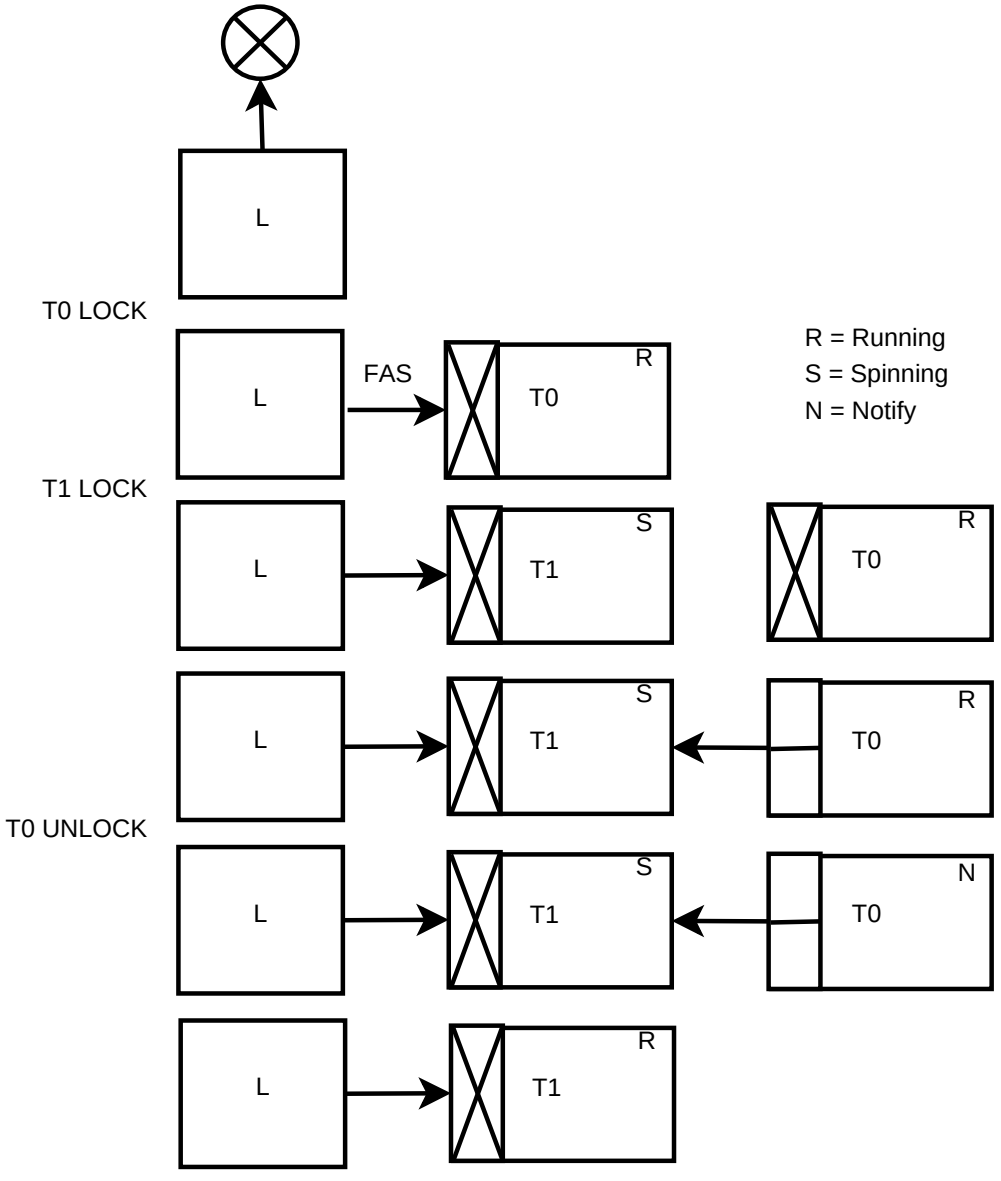

1.4e+09
1.2e+09
1e+09
8e+08
6e+08
4e+08
2e+08
0
Total Acquisitions (in 10 seconds)
2
3
4
5
6
7
8
9
10
11
12
Number of Cores
Naive
TATAS
EB

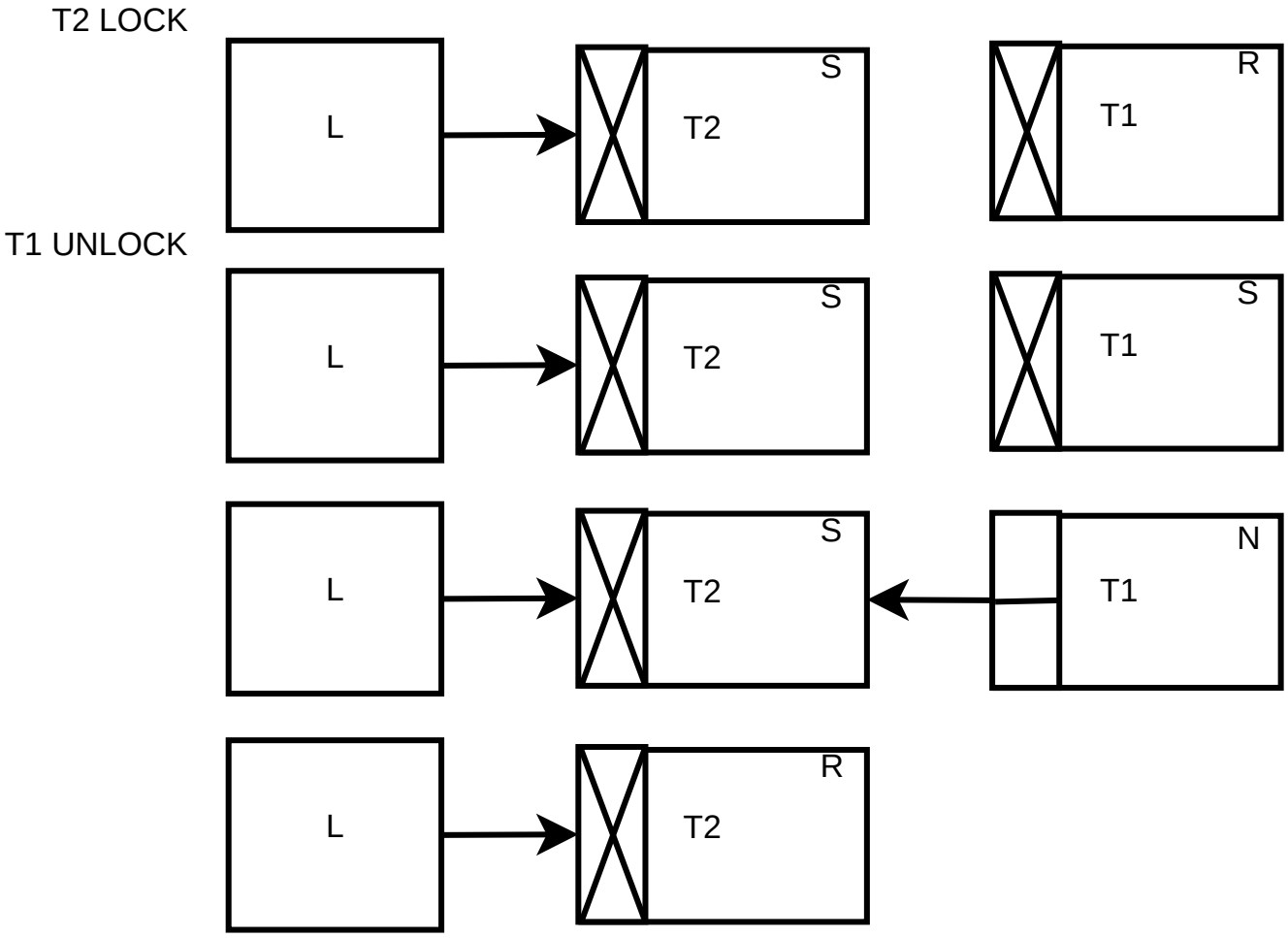

4.5e+07
4e+07
3.5e+07
3e+07
2.5e+07
2e+07
1.5e+07
1e+07
5e+06
0
Total Acquisitions (1o seconds)
Processor

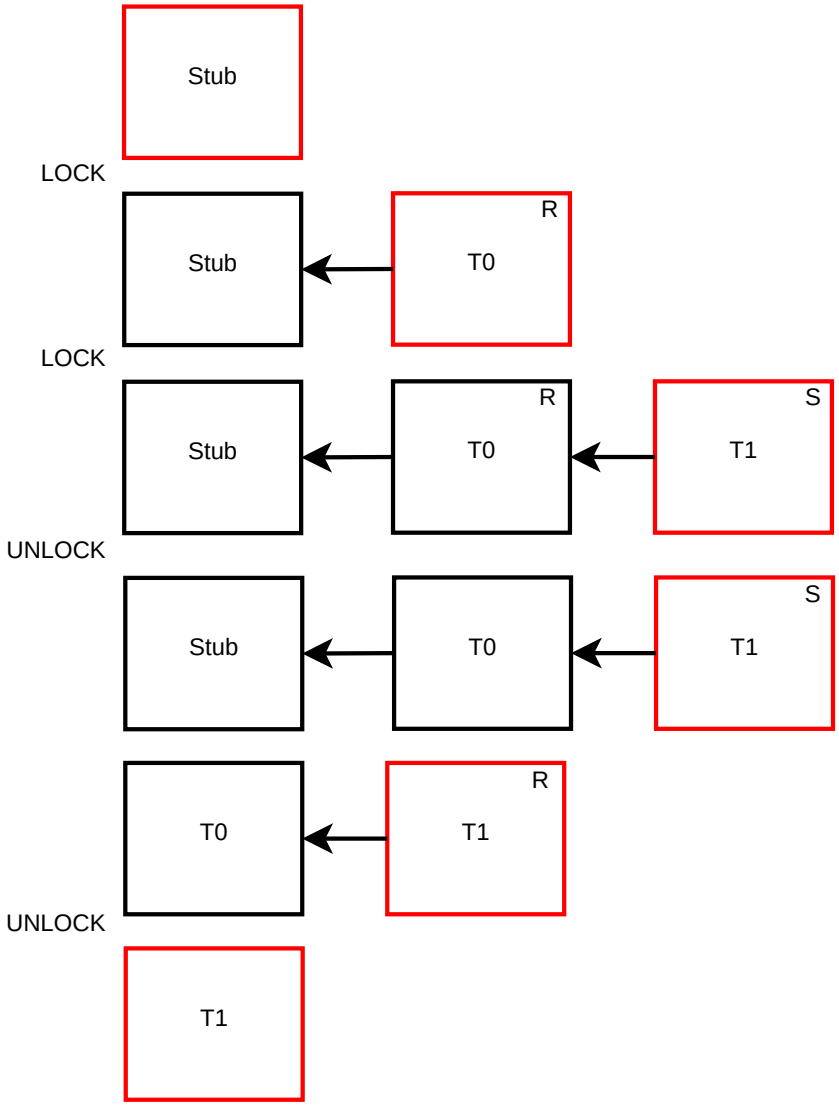
- Non-arbitrating spinlocks do not provide fairness (or starvation-freedom) guarantees.


LOCK
Now Serving: 0
0
LOCK, LOCK
Now Serving: 0
0
1
2
UNLOCK
Now Serving: 1
1
2
UNLOCK
Now Serving: 2
2
3
UNLOCK
Now Serving: 3
3
UNLOCK, LOCK
Now Serving: 4
4
UNLOCK


P
U L L L
LOCK
P
U L L L
LOCK
P
U L L L
UNLOCK
P
L U L L
LOCK
P
L U L L
UNLOCK
P
L L U L
LOCK
P
L L U L
UNLOCK
P
L L L U
UNLOCK
P
U L L L


L
T0 LOCK
L
FAS
T0
R
R = Running
S = Spinning
N = Notify
T1 LOCK
L
T1
S
T0
R
L
T1
S
T0
R
T0 UNLOCK
L
T1
S
T0
N
L
T1
R


T2 LOCK
L
T2
S
T1
R
T1 UNLOCK
L
T2
S
T1
S
L
T2
S
T1
N
L
T2
R


Stub
LOCK
Stub
R
T0
LOCK
Stub
R
T0
S
T1
UNLOCK
Stub
T0
S
T1
T0
R
T1
UNLOCK
T1

Fast path latency

See `http://concurrencykit.org/doc/appendixZ.html`

- Mutexes in general are not composable.
- Subtle ordering issues can lead to hard-to-detect deadlock conditions.
- Blocking synchronization is sensitive to preemption.

Questions?