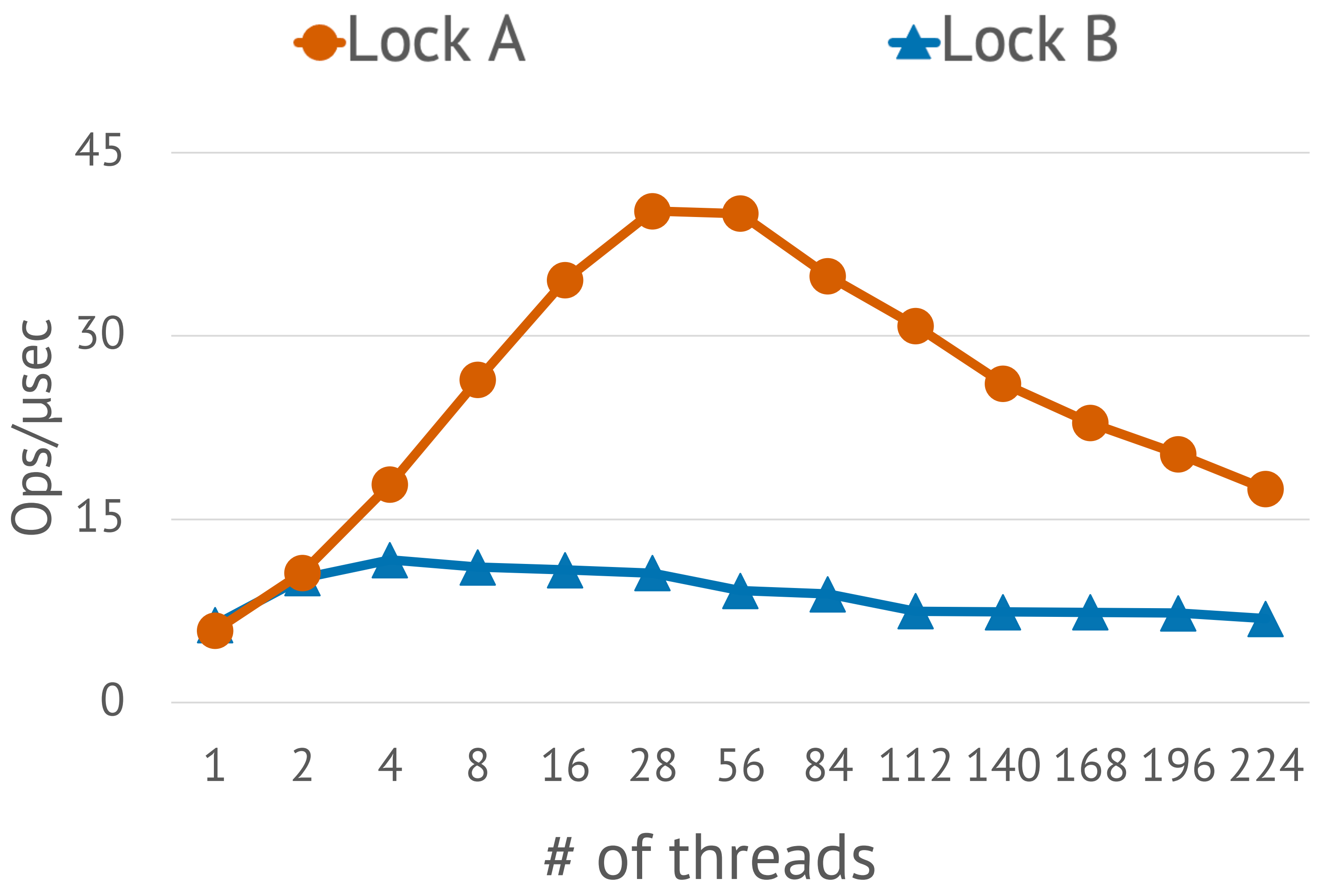
Application-Informed Kernel Synchronization Primitives

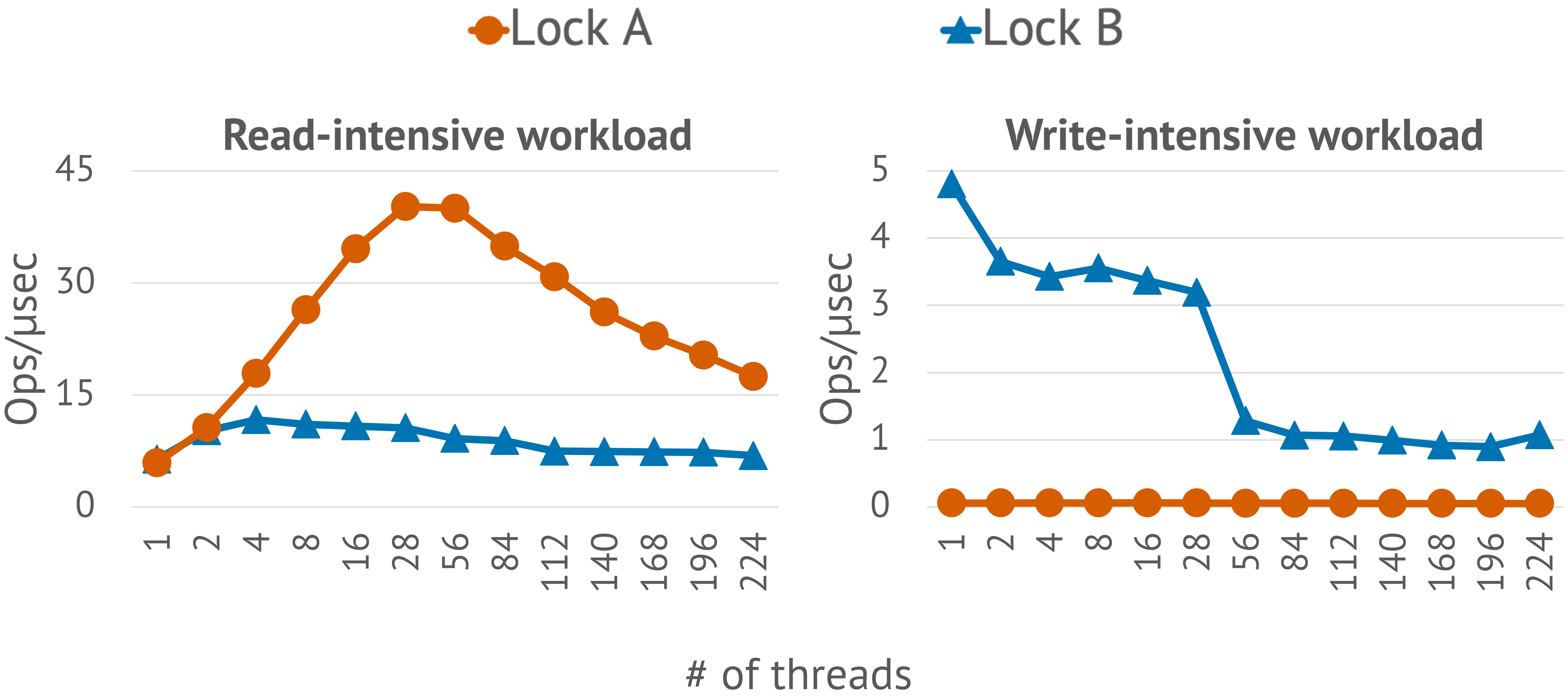
Sujin Park Diyu Zhou Irina Calciu Yuchen Qian Taesoo Kim Sanidhya Kashyap

Georgia Tech EPFL graft

Locks are critical for application performance



One lock cannot rule all scenarios



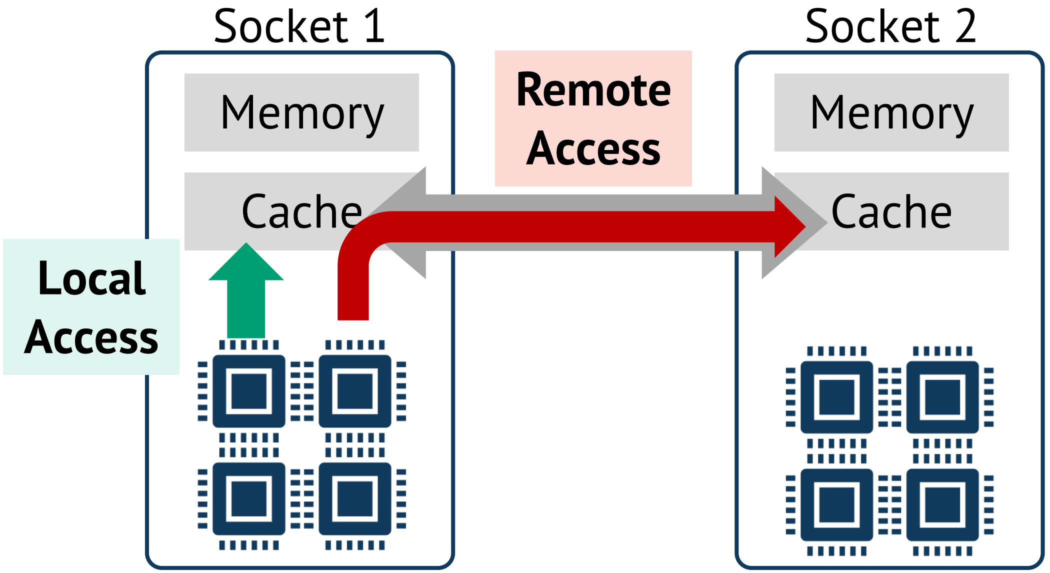
Depending on scenarios, different lock perform best

Hardware

Software

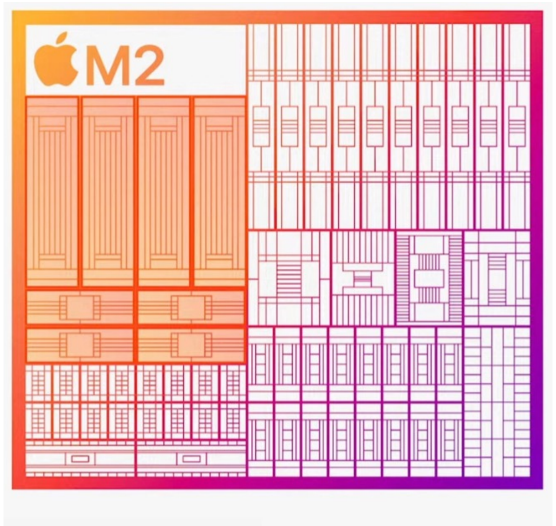
Locks considering hardware

NUMA (non-uniform memory access)



Accessing local socket data is faster than remote socket data

AMP (Asymmetric multicore processors)

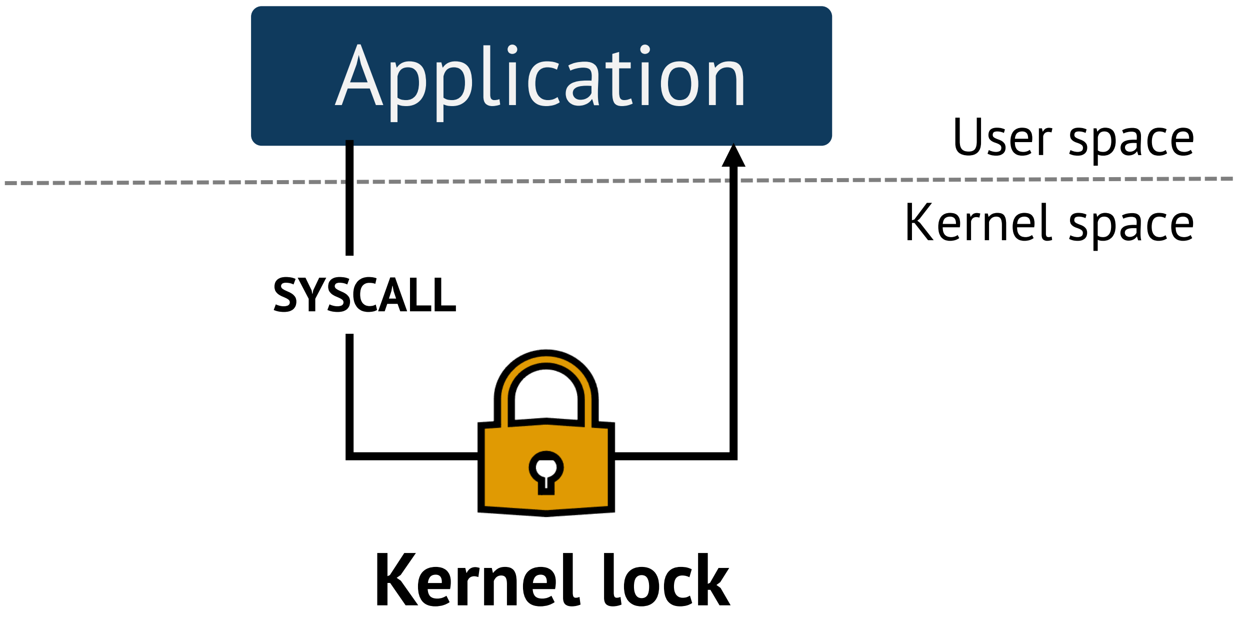


Faster performance cores and slower efficiency cores in one processor

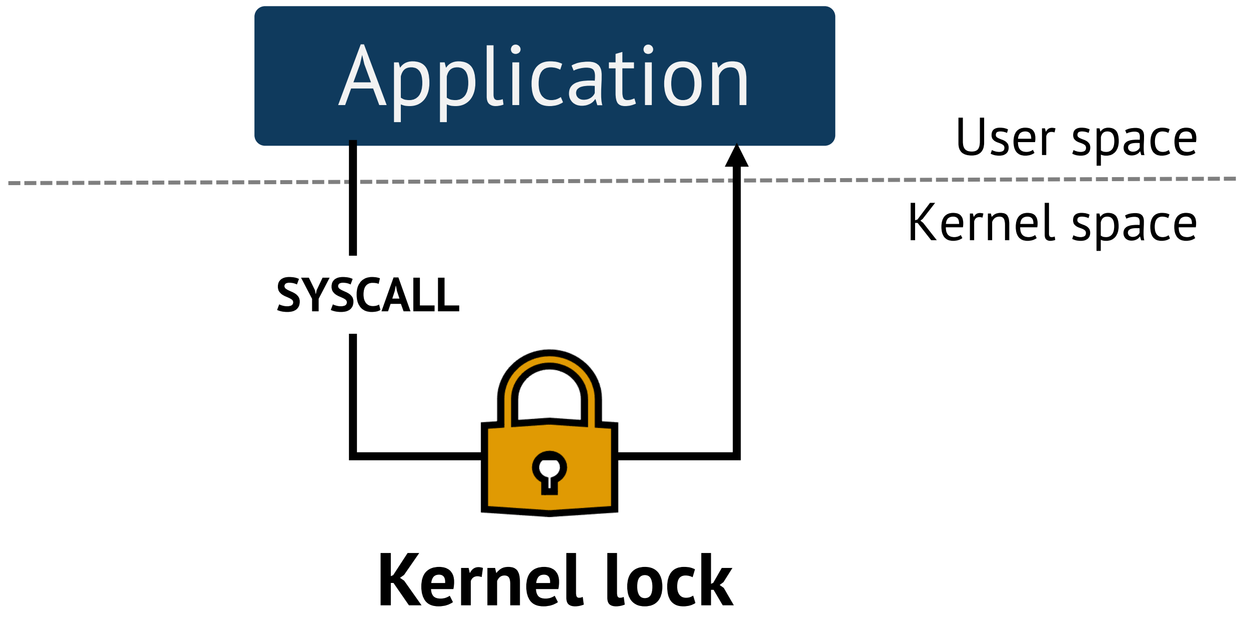
Locks considering software requirements

- Read / write ratio?
- Length of critical section?
- Any specific threads need to be prioritized?

Kernel locks also affect application performance

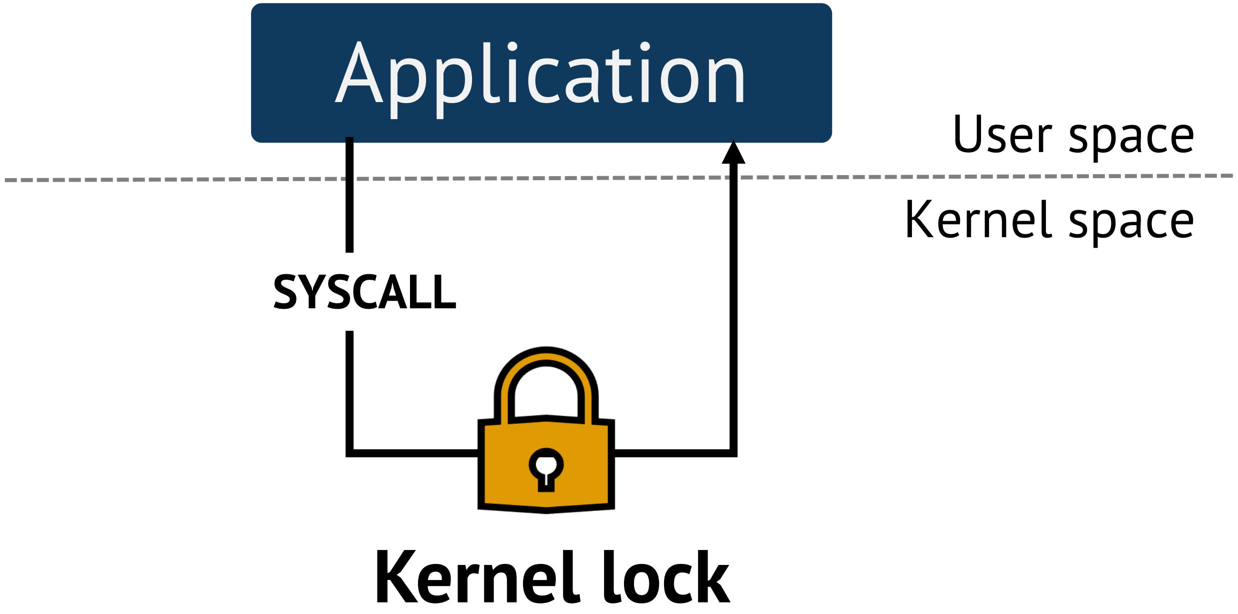


Kernel locks also affect application performance

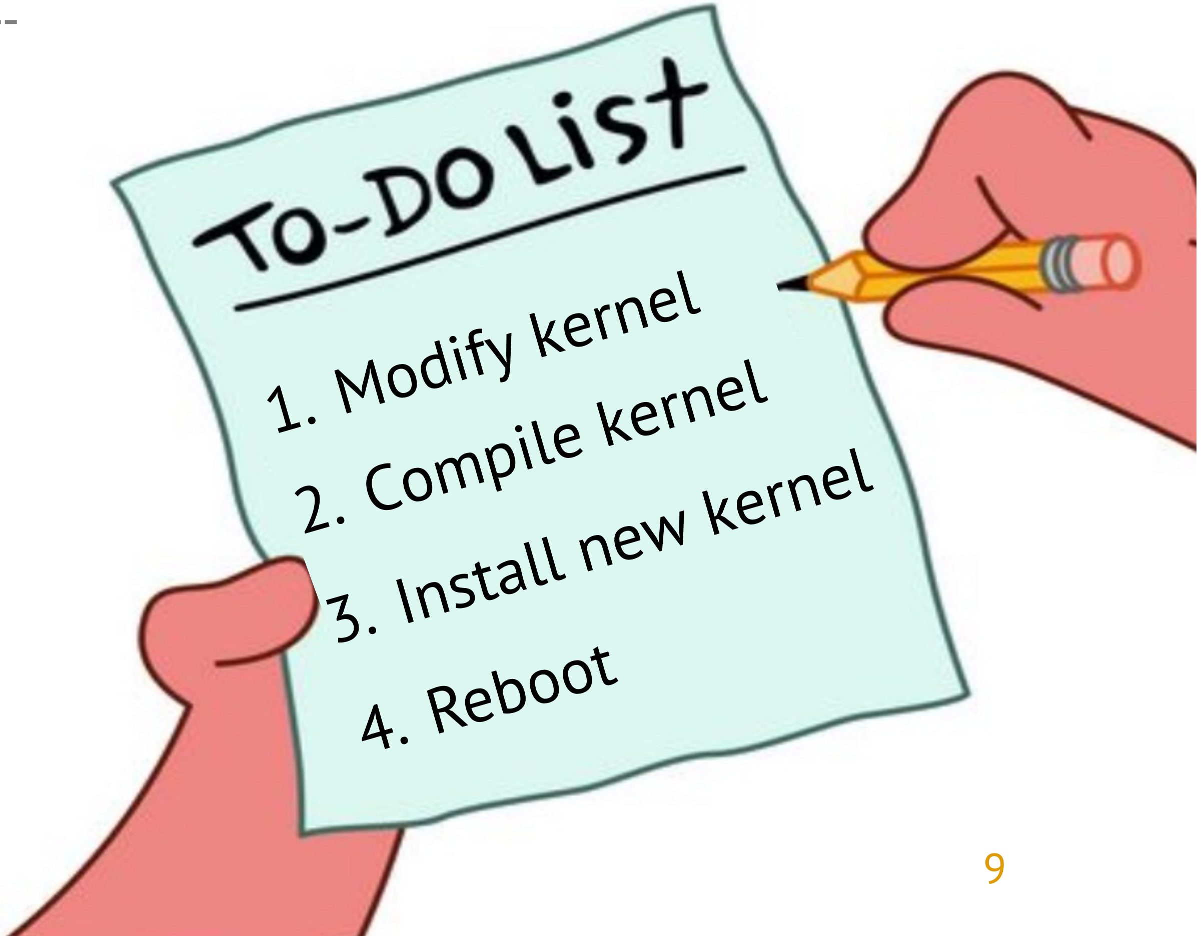


- Application-agnostic
- Invisible to application developers
- Generic design to support common cases

...But difficult to change



- Application-agnostic
- Invisible to application developers
- Generic design to support common cases



Issue with current kernel locks

Lock implementations are application agnostic

Only a few locks contend for given application

Difficult to implement a new lock design

The solution – SynCord

Lock implementations are application agnostic

→ Let application developers **safely** change locks in the kernel **on the fly**

Only a few locks contend for given application

→ Modify set of locks at **various granularities**

Difficult to implement a new lock design

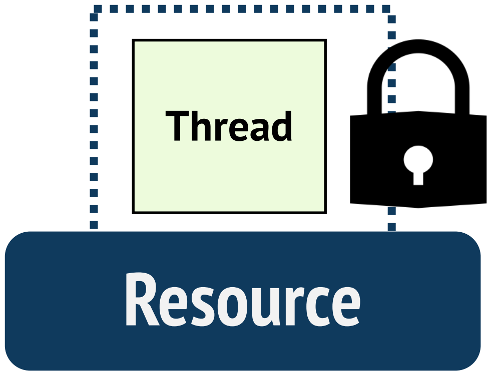
→ Expose set of **APIs** to easily write various lock policies

Key behavior of queue-based lock

To access shared resource, **thread** needs to acquire **lock**

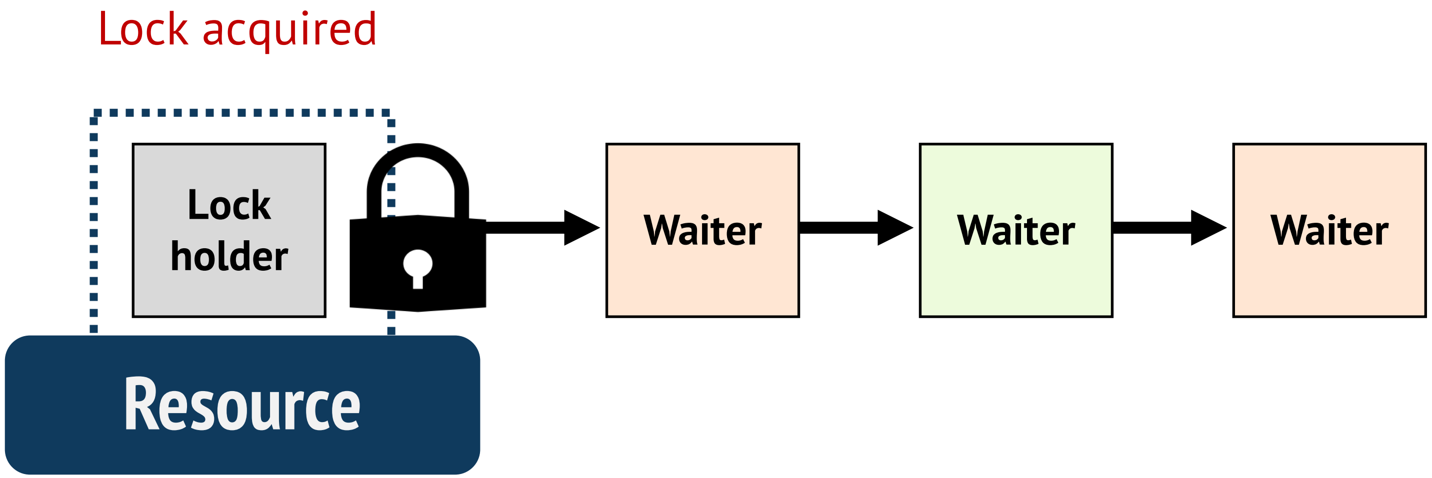
Key behavior of queue-based lock

Lock acquired



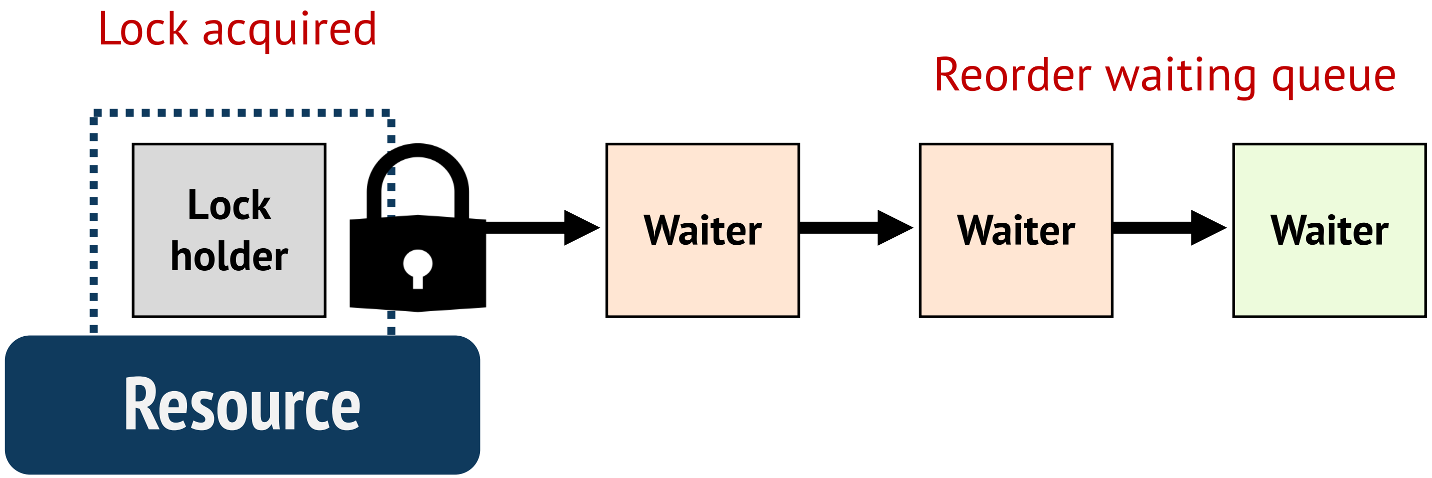
If lock is free, thread directly **acquires** lock

Key behavior of queue-based lock



Since lock is already held, other threads **join waiting queue**

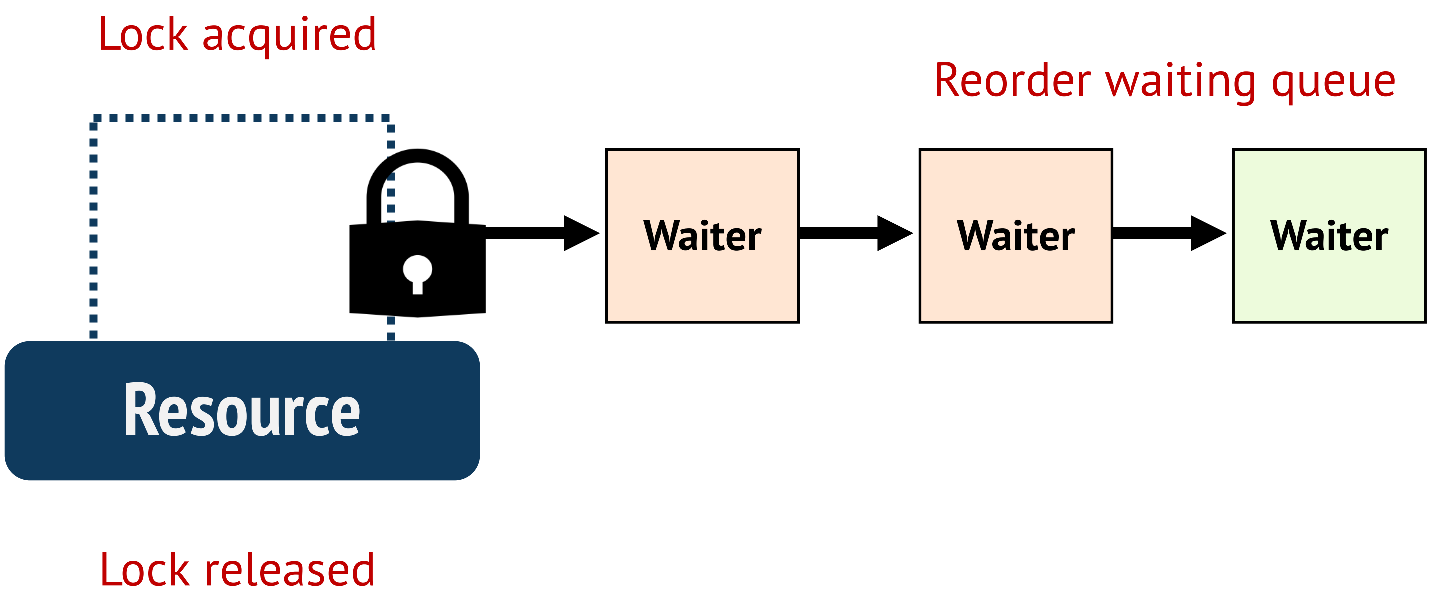
Key behavior of queue-based lock



Reorder waiters in the queue to group waiters from same socket (ShflLock[1], CNA[2])

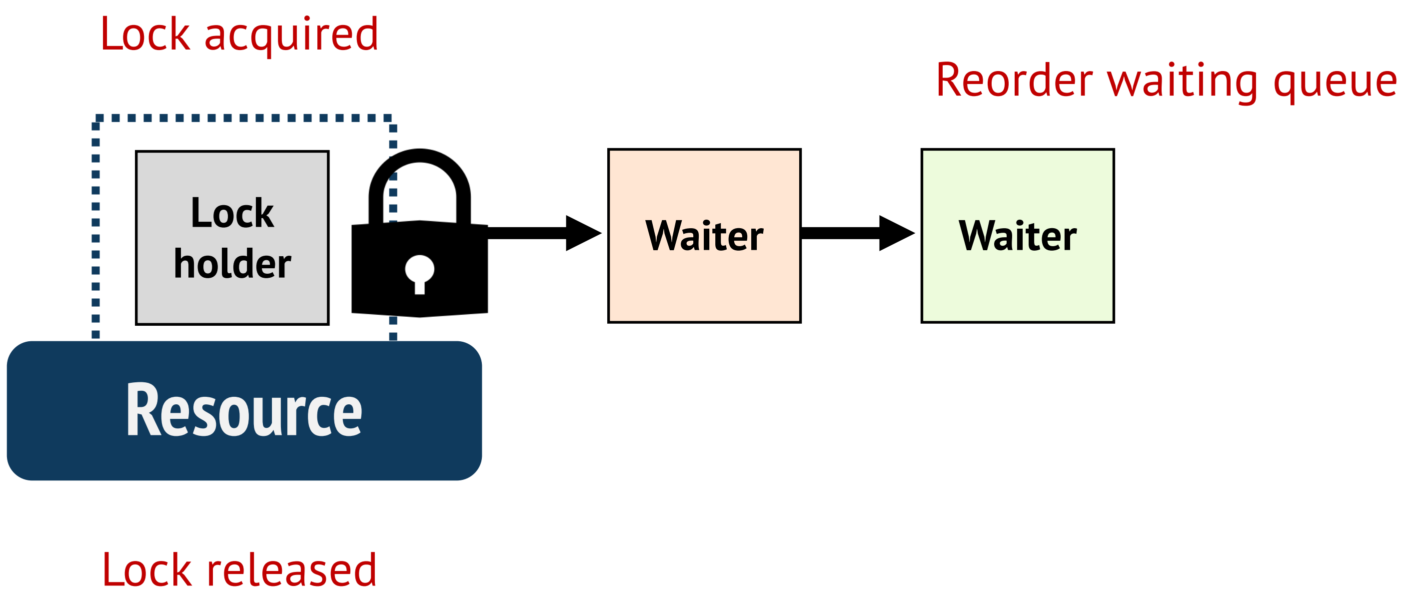
1. Scalable and Practical Locking With Shuffling. SOSP '19
2. Compact NUMA-aware Locks. EuroSys '19

Key behavior of queue-based lock



Release lock when thread finishes using resource

Key behavior of queue-based lock



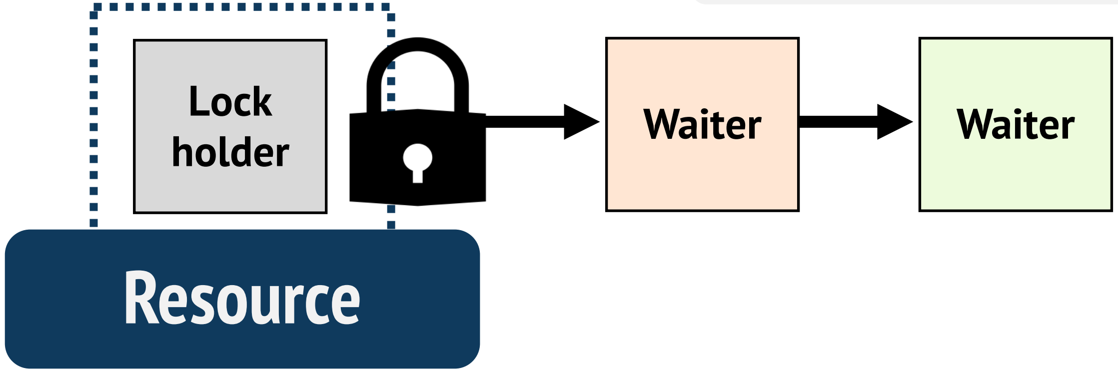
Next waiter acquire lock

SynCord exposes kernel locks' key behaviors as APIs

`lock_acquired (lock)`

move if `true`

`bool should_reorder (lock, anchor_node, curr_node)`

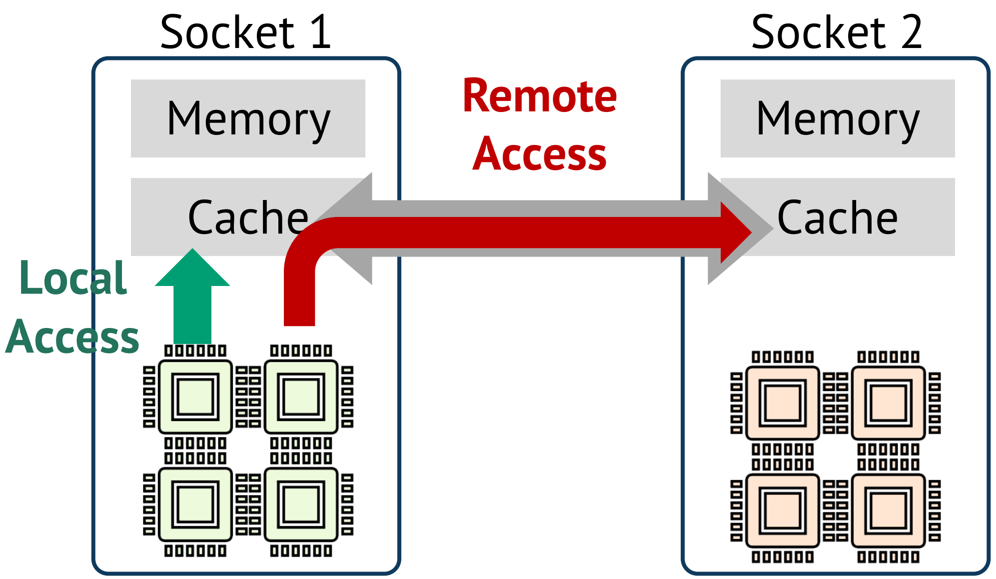


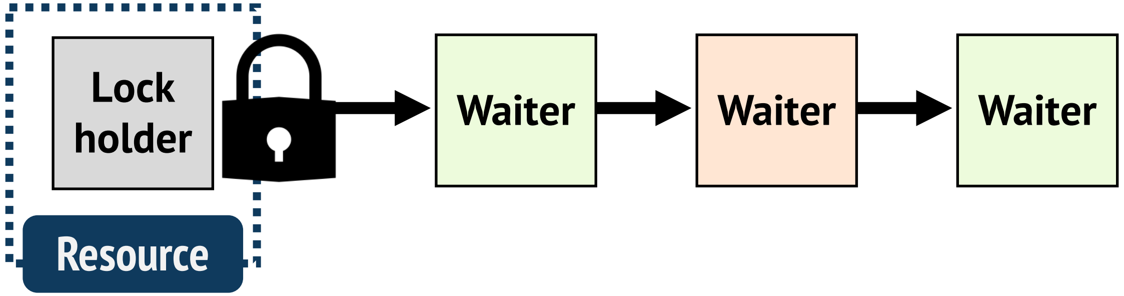
`lock_released (lock)`

And 7 more APIs!

SynCord overview with NUMA-aware example

- NUMA (non-uniform memory access)

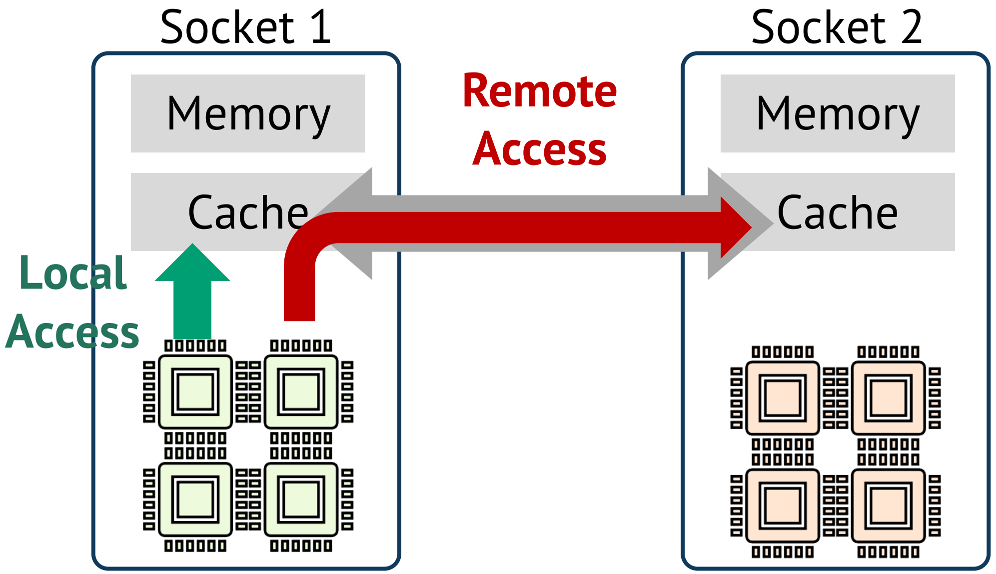




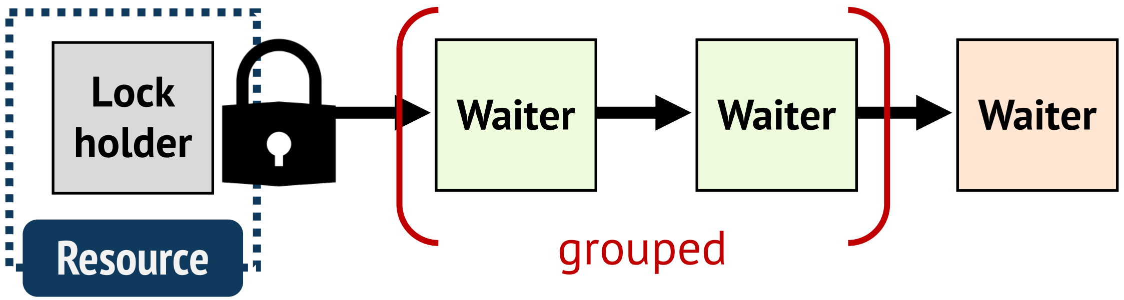
Accessing local socket memory is faster than remote socket memory

SynCord overview with NUMA-aware example

- NUMA (non-uniform memory access)



Accessing local socket memory is faster than remote socket memory



Minimize cache line bouncing

SynCord overview with NUMA-aware example

❶ User writes custom lock policy
and specify target point

```
bool should_reorder(lock *lock, node *anchor, node *curr)
{
    return (anchor->socket_id == curr->socket_id);
}
```

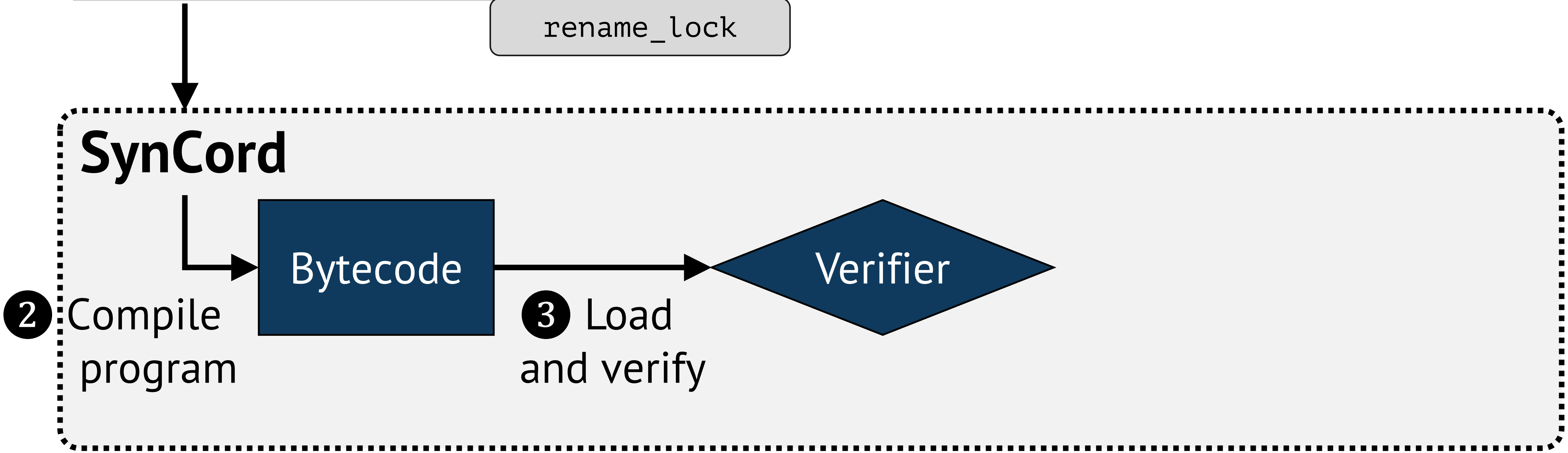
Target point: `rename_lock`

SynCord overview with NUMA-aware example

❶ User writes custom lock policy
and specify target point

```
bool should_reorder(lock *lock, node *anchor, node *curr)
{
    return (anchor->socket_id == curr->socket_id);
}
```

`rename_lock`

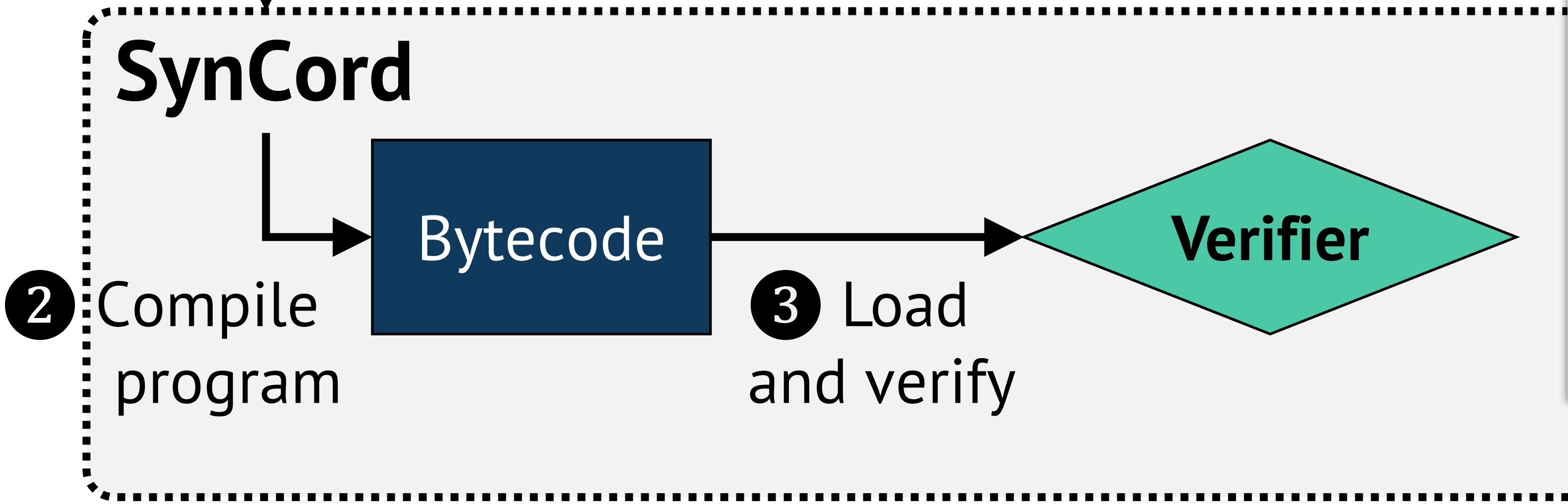


SynCord overview with NUMA-aware example

❶ User writes custom lock policy and specify target point

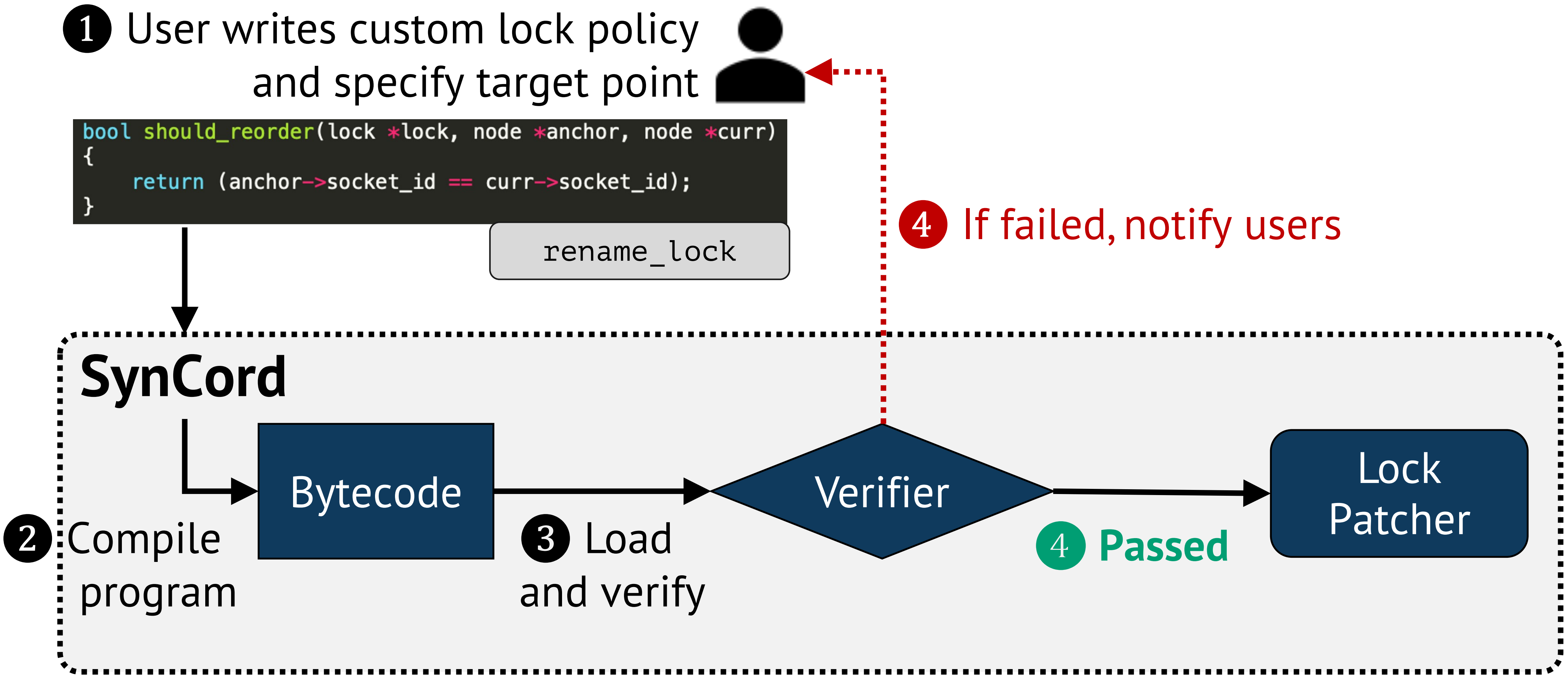
```
bool should_reorder(lock *lock, node *anchor, node *curr)
{
    return (anchor->socket_id == curr->socket_id);
}
```

`rename_lock`

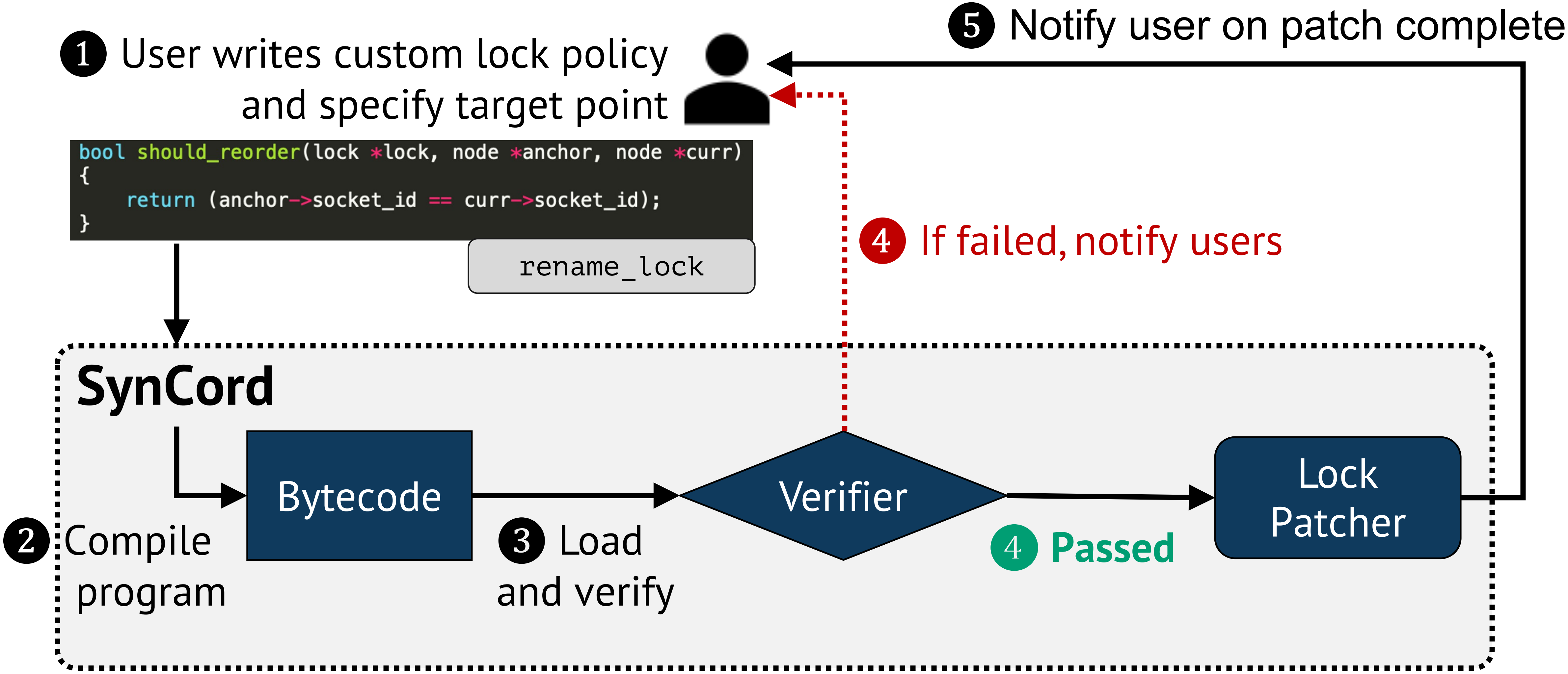


- ✓ **memory access**
  - → No arbitrary memory update
- ✓ **helper functions**
  - → Only allowlisted functions can be called
- ✓ **code termination**
  - → Lock policy must not hang

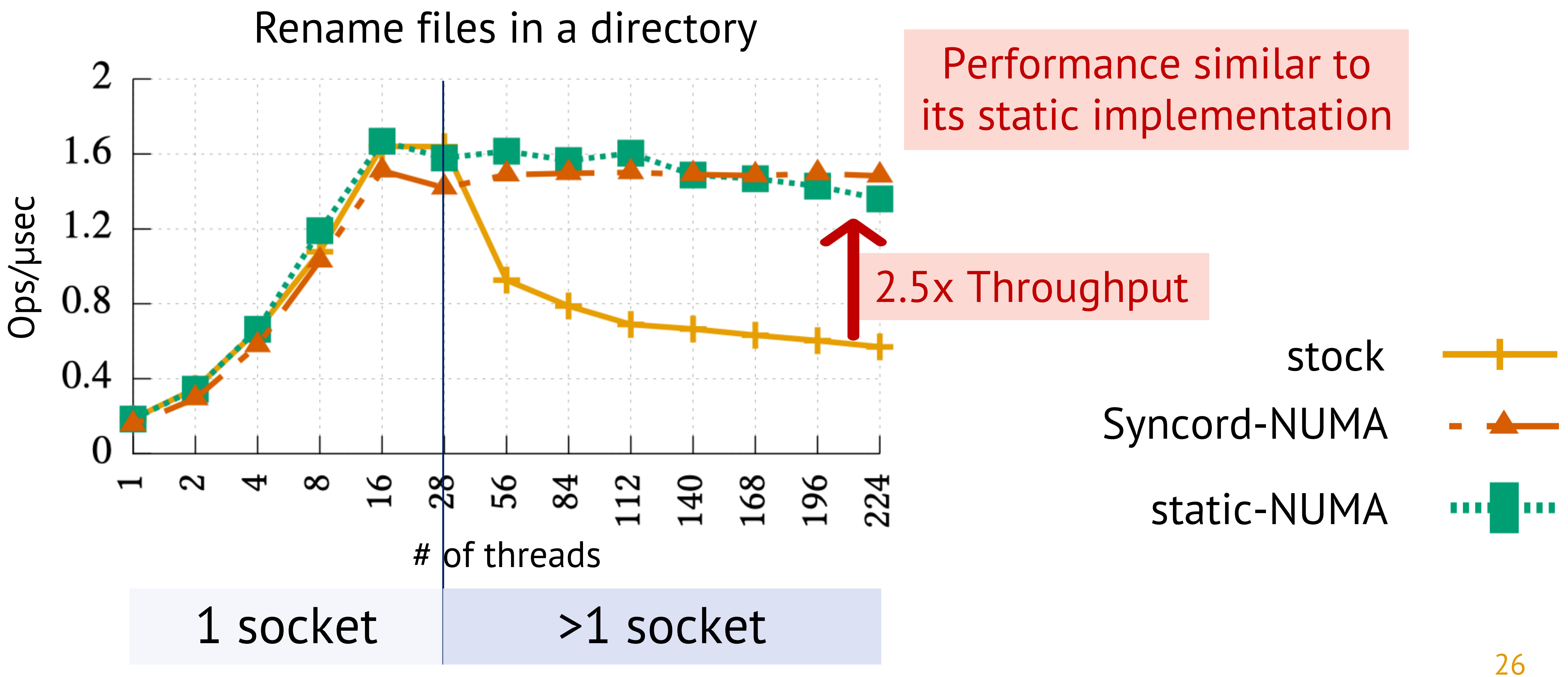
SynCord overview with NUMA-aware example



SynCord overview with NUMA-aware example

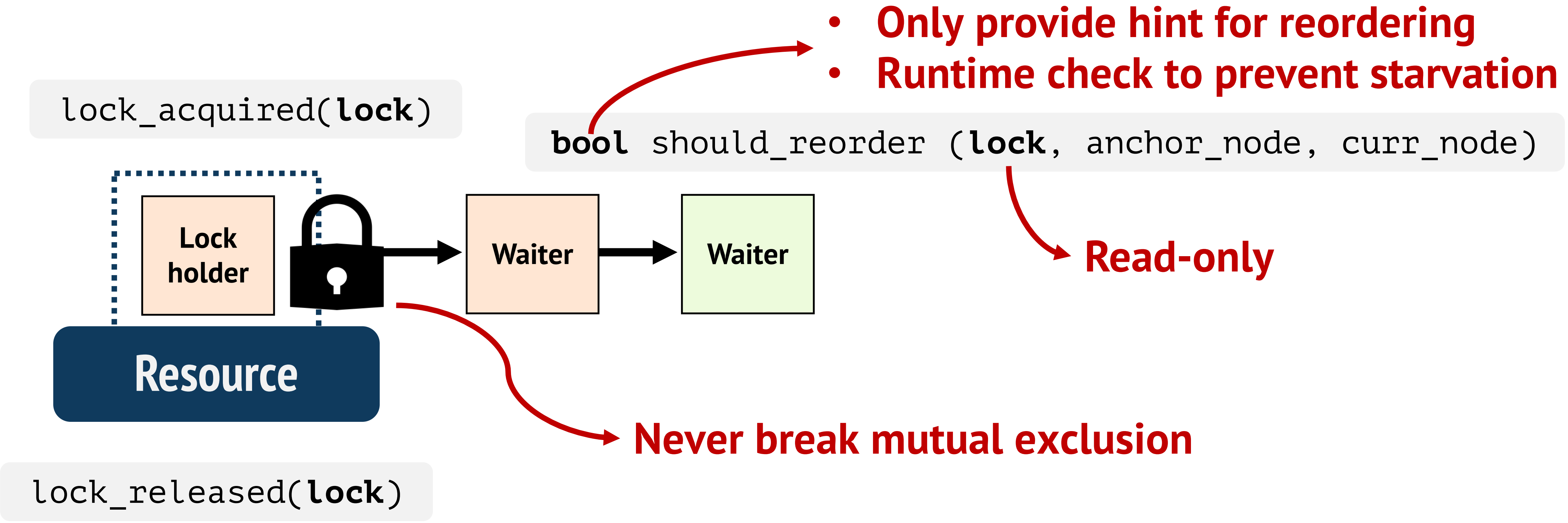


SynCord overview with NUMA-aware example



What if a user provide wrong code?

- Verifier + API design → sandboxed impact
- Mechanism remains intact



What user can do & can't do with SynCord

Can do

Prioritize/penalize specific threads

Run additional code blocks in hooking points

Affect performance

Affect fairness

Can't do

Break mutual exclusion

Change underlying mechanism

Change lock type

Usecases

Customized for

1. NUMA-aware lock — **HW: NUMA**
2. Asymmetric multicore lock — **HW: AMP+NUMA**
3. Scheduler-cooperative lock — **SW: Length of CS** **HW: NUMA**
4. Biased per-CPU readers-writer lock — **SW: Read-intensive** **HW: NUMA**
5. Dynamic lock profiling

Dynamic lock profiling

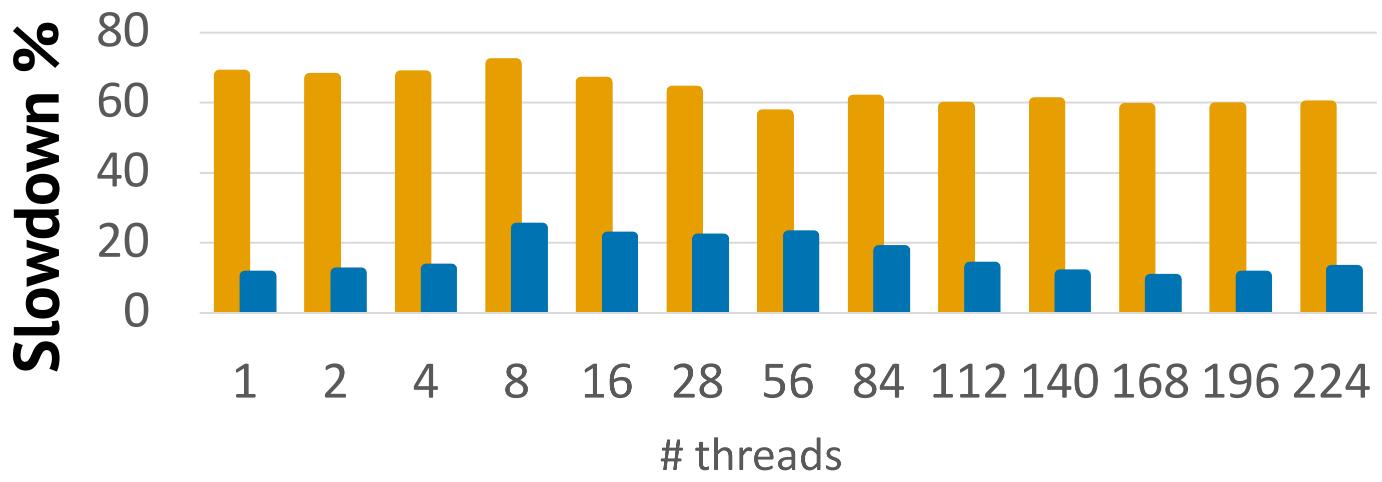
Lockstat	vs	Dynamic lock profiling

Lockstat

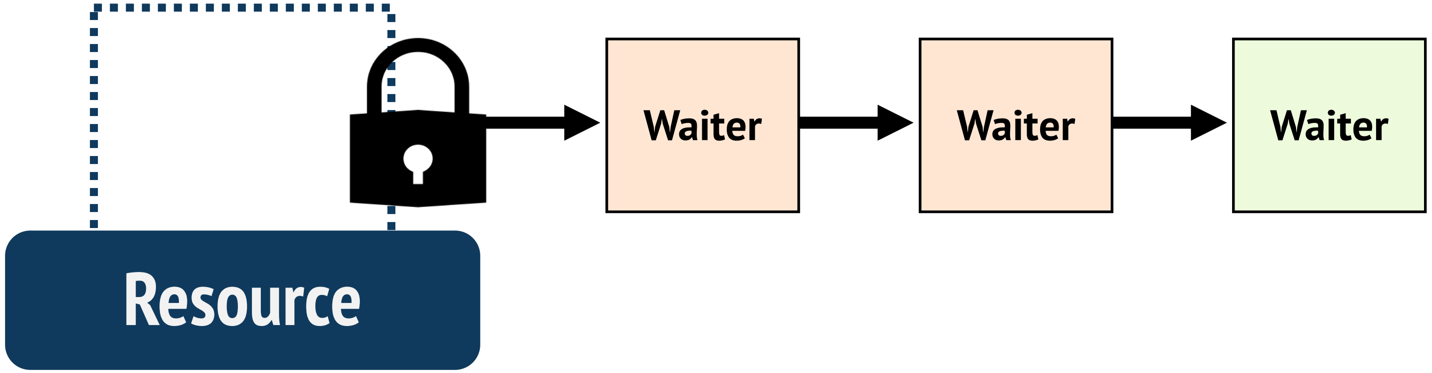
- In-kernel lock statistic tool
- System-wide tracing
- Enabled in compile time
- More memory usage from booting

Dynamic lock profiling

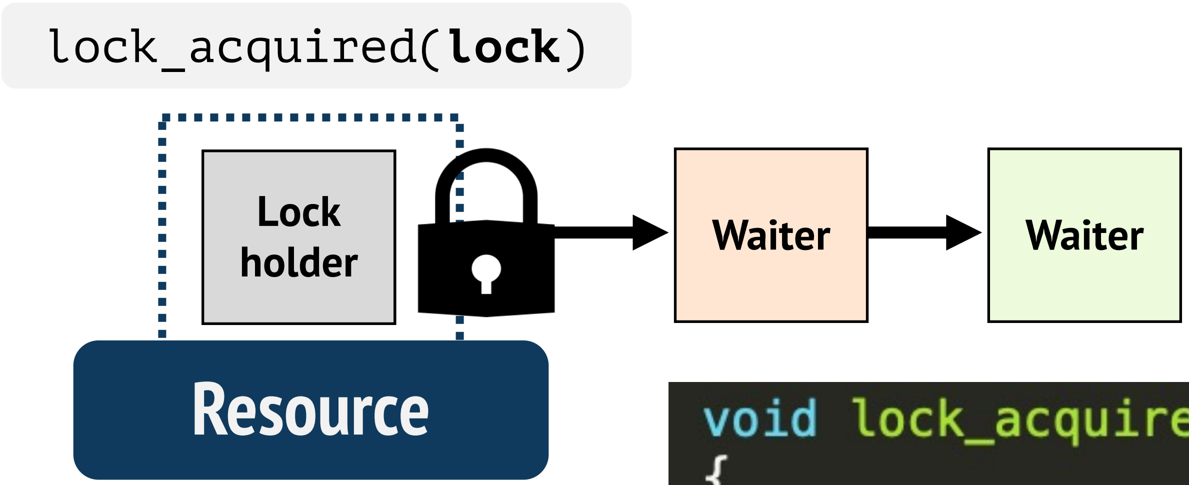
- Implemented with SynCord APIs
- Can trace single lock instance
- Dynamically enabled
- No memory overhead once disabled



Dynamic lock profiling: avg critical section length



Dynamic lock profiling: avg critical section length

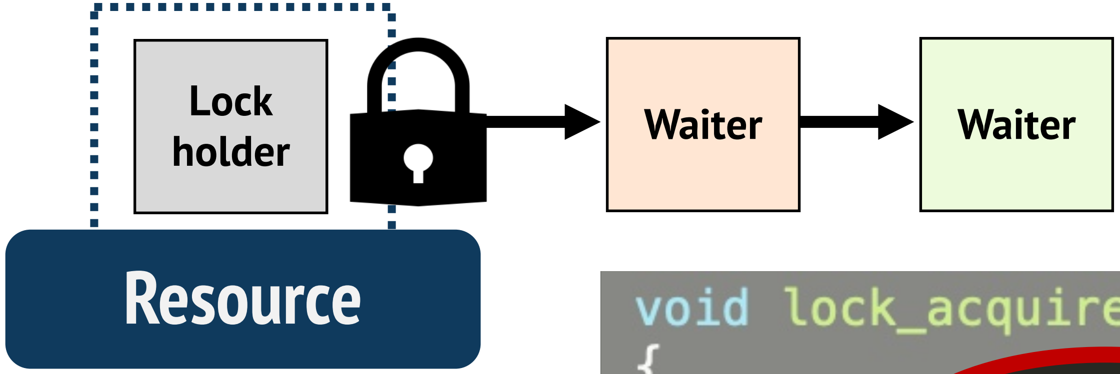



```
void lock_acquired(lock *lock)
{
      lock->acquisitions++;
      lock->hold_start = get_time();
}
```

```
void lock_released(lock *lock)
{
      lock->holdtime += (lock->hold_start - get_time());
}
```

Dynamic lock profiling: avg critical section length

```
lock_acquired(lock)
```




```
void lock_acquired(lock *lock)
{
    lock->acquisitions++;
    lock->hold_start = get_time();
}
```

```
void lock_released(lock *
{
    lock->holdtime
}
```

Auxiliary data structures

Conclusion

- Kernel locks are basic building of concurrent OSes
  - Affect performance and scalability of applications
  - Out of reach of application developers
- SYNCORD Framework
  - Allow users to fine-tune locking primitives dynamically
  - Exposes a set of user implementable APIs
  - No need to reinstall the kernel or reboot the system
- Application can now address pathological locking cases

Thank you!