Term Project Presentation
Topic: Synchronization

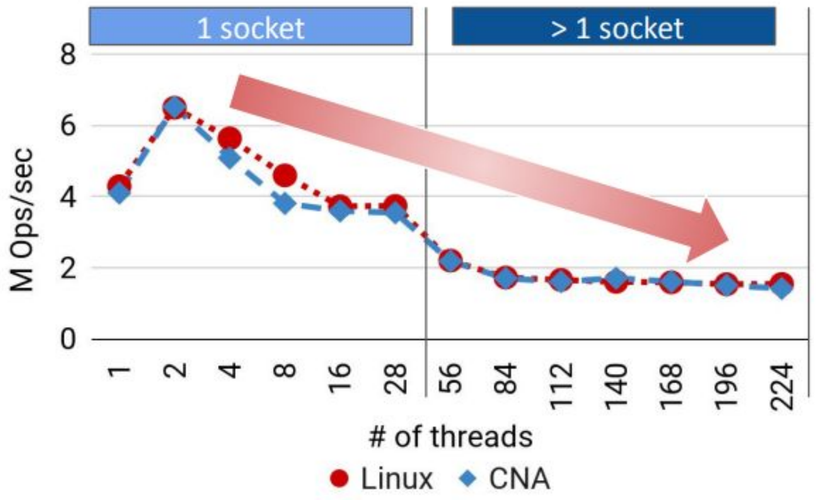
Group 2

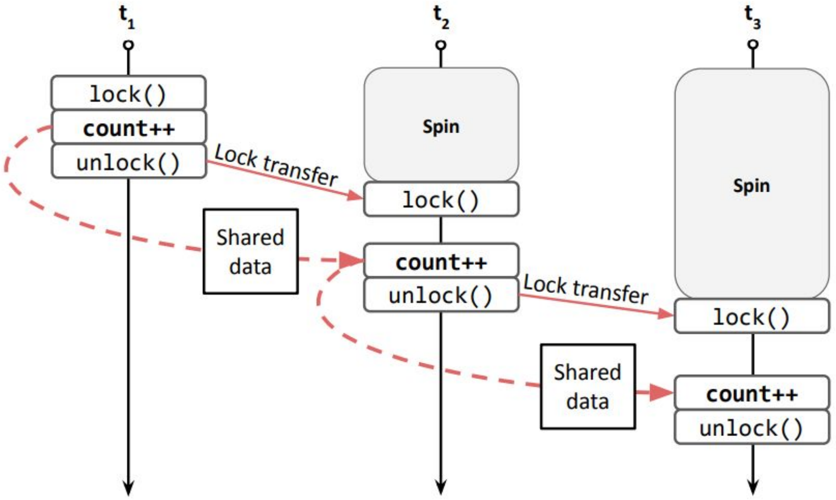
Chappidi Yoga Satwik	19CS30013
Rajas Bhatt	19CS30037
Seemant Achari	19CS30057

Ship your Critical Section, Not Your Data: Enabling Transparent Delegation with TCLocks

Chappidi Yoga Satwik

Traditional Lock

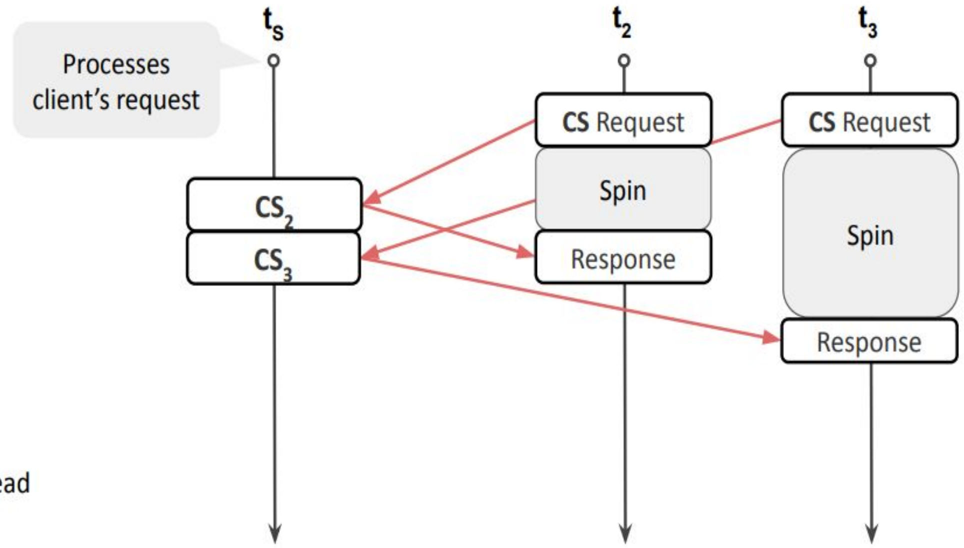




Benchmark: Each thread enumerates files in the system, with each directory having a lock.

Lock transfer leads to shared data movement.
Given, each thread is on different cores with caches.

Delegation-style locks



t_s: server thread
t_i: thread i
CS: critical section

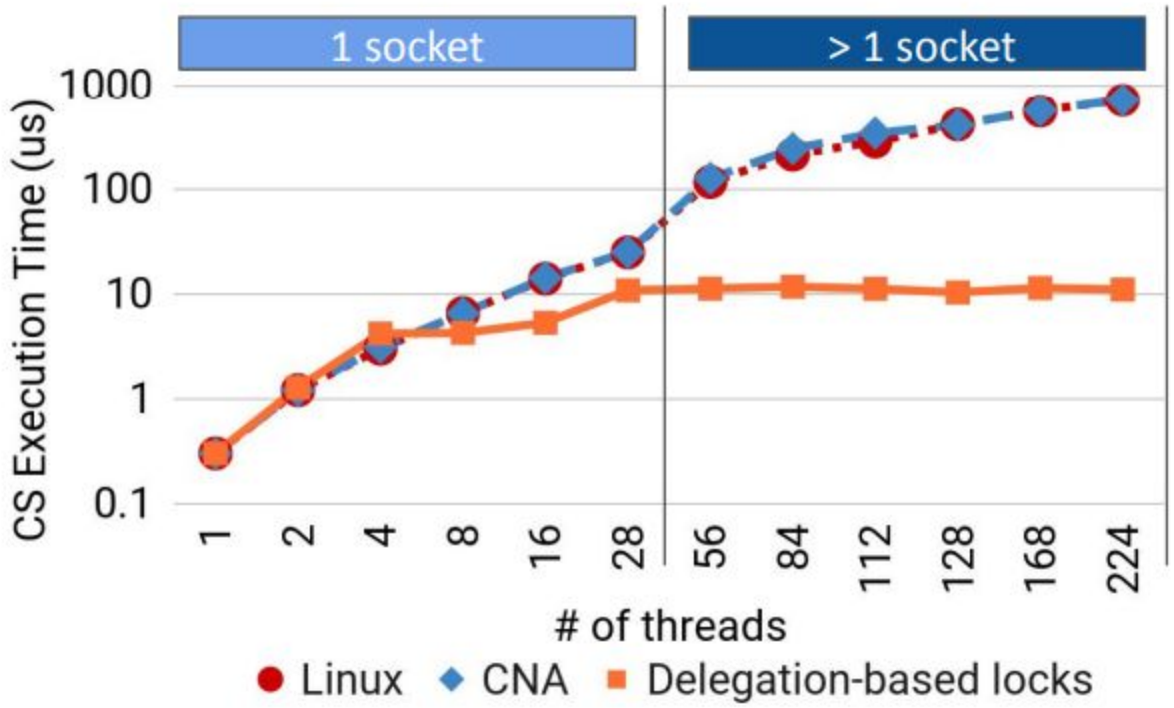
```
lock()
count++
unlock()
```

```
void incr_func() =
     count++

send_req_to_server(&incr_func)
```

Two types of threads:
Server: Holds lock, Keeps the shared data in its cache.
Client: Sends its critical section.

Difficult to modify large projects (Linux Kernel 180K Locks)

Delegation-style Locks

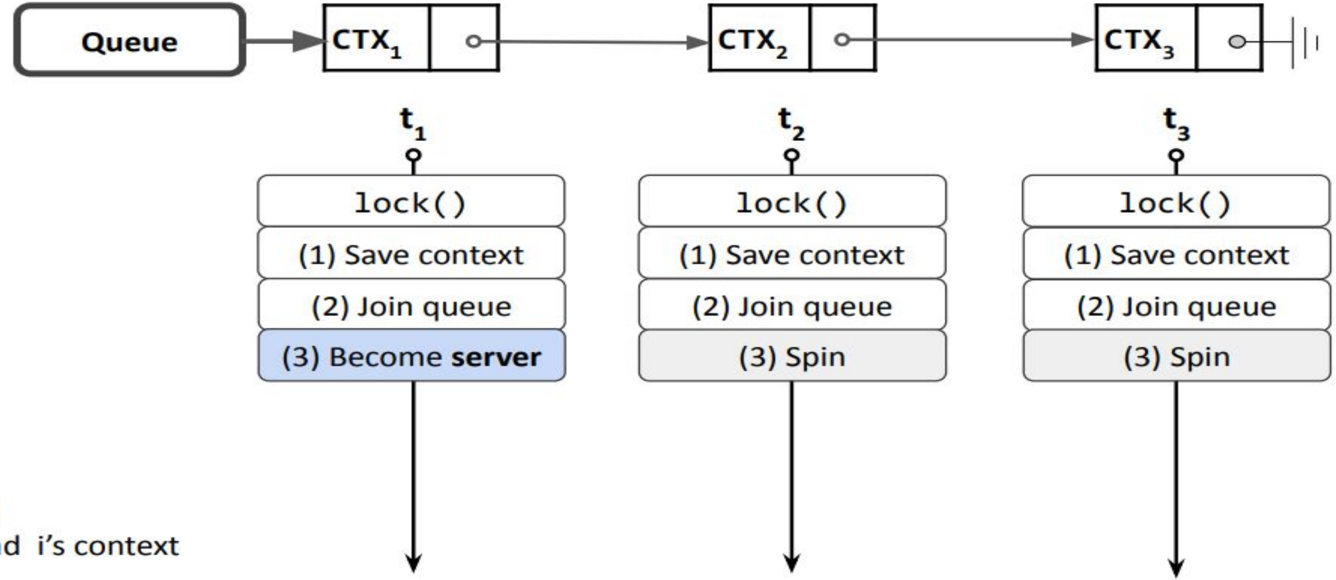


Time spent in CS is reduced.

Due to the minimal shared data movement

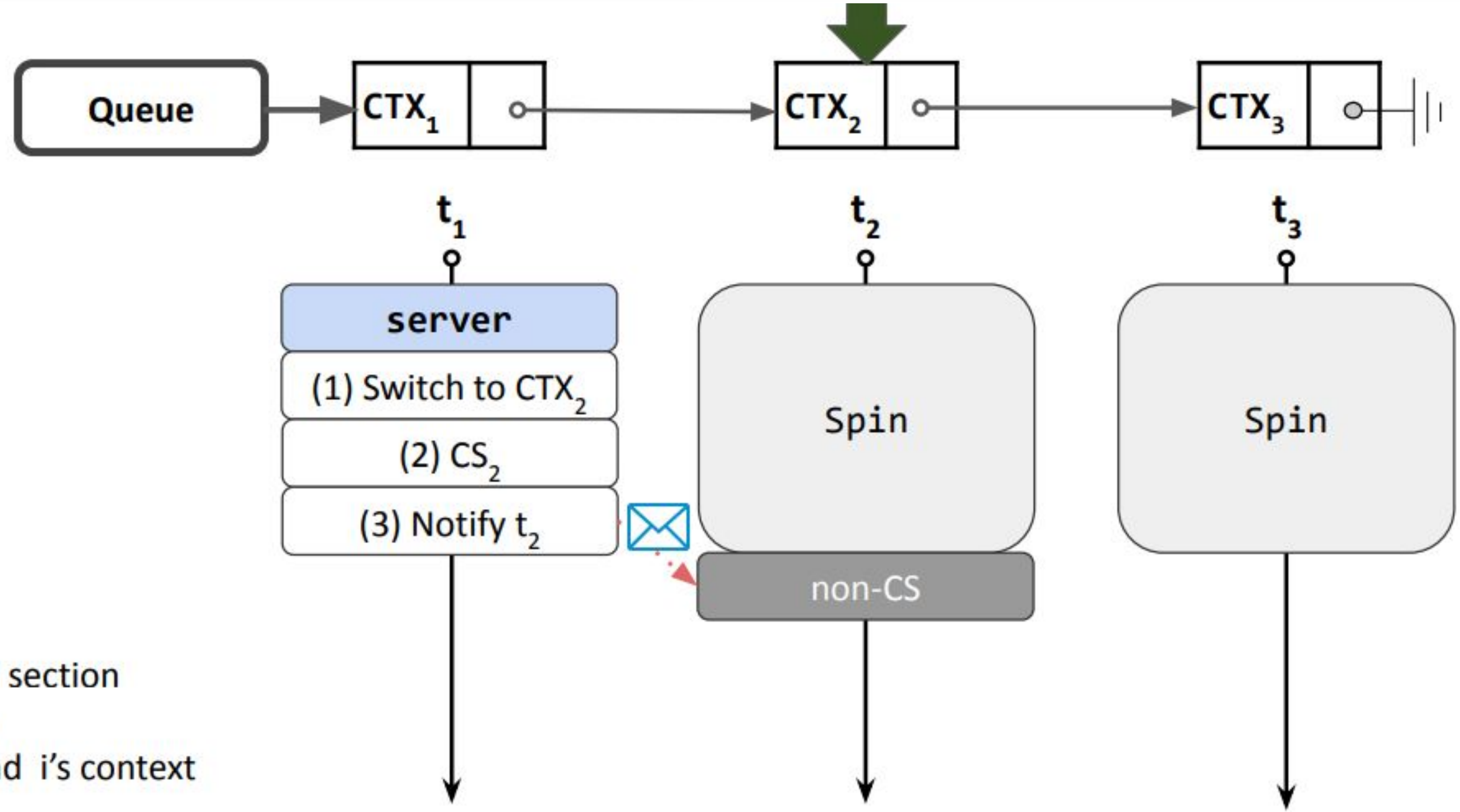
Transparent Delegation

Capture a thread's context, from the lock to unlock call
(Instruction pointer + stack pointer + general-purpose registers)



t_i: thread i
CTX_i: thread i's context

Transparent Delegation



CS: Critical section
t_i: thread i
CTX_i: thread i's context

Optimizations

While the waiters is spinning, there maybe be interrupts or signals.

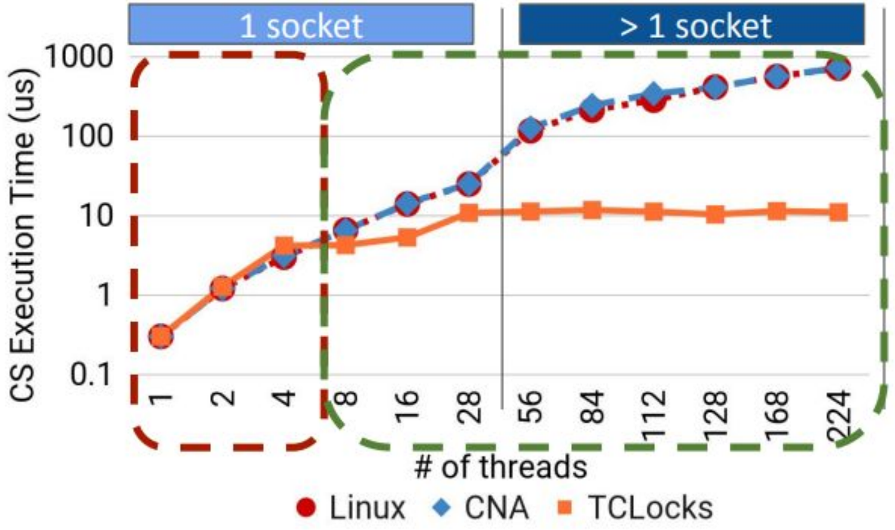
The server might be actively on using the waiter’s stack.

To enable a response to these signals, waiter has an **ephemeral stack** during critical section execution

More variants of TCLocks

- Blocking
  Waiter thread can sleep after joining the queue.
  Server thread needs to wake the waiter
- Reader-Writer
  Phase-based Lock (Read, write)
  Readers get to read as long as no writers.

Results



- > 4 threads

  Minimal shared data movement

- ≤ 4 threads

  Context-switch overhead

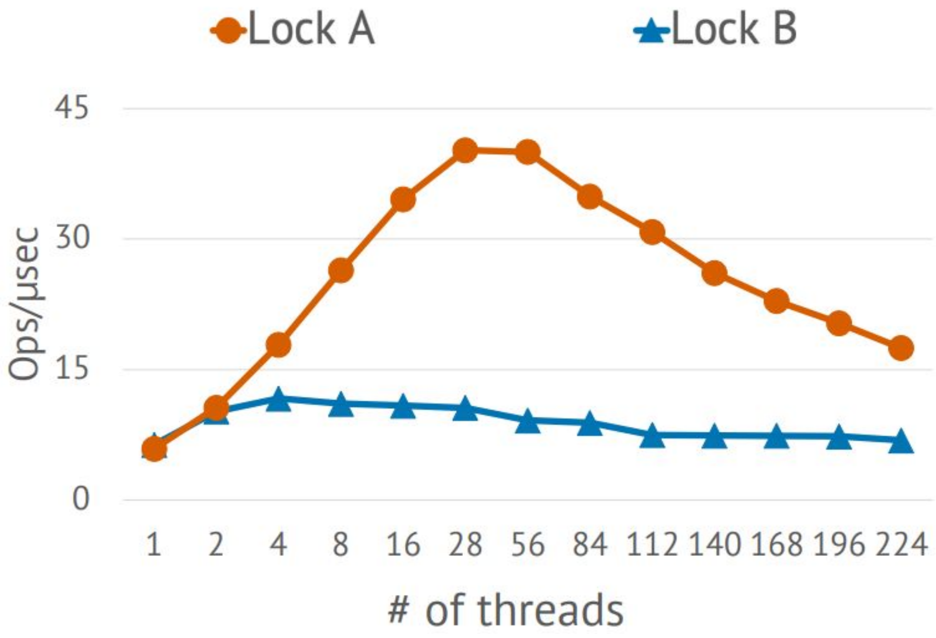
  Not enough batching

Applications can now use delegation-style locks without modification

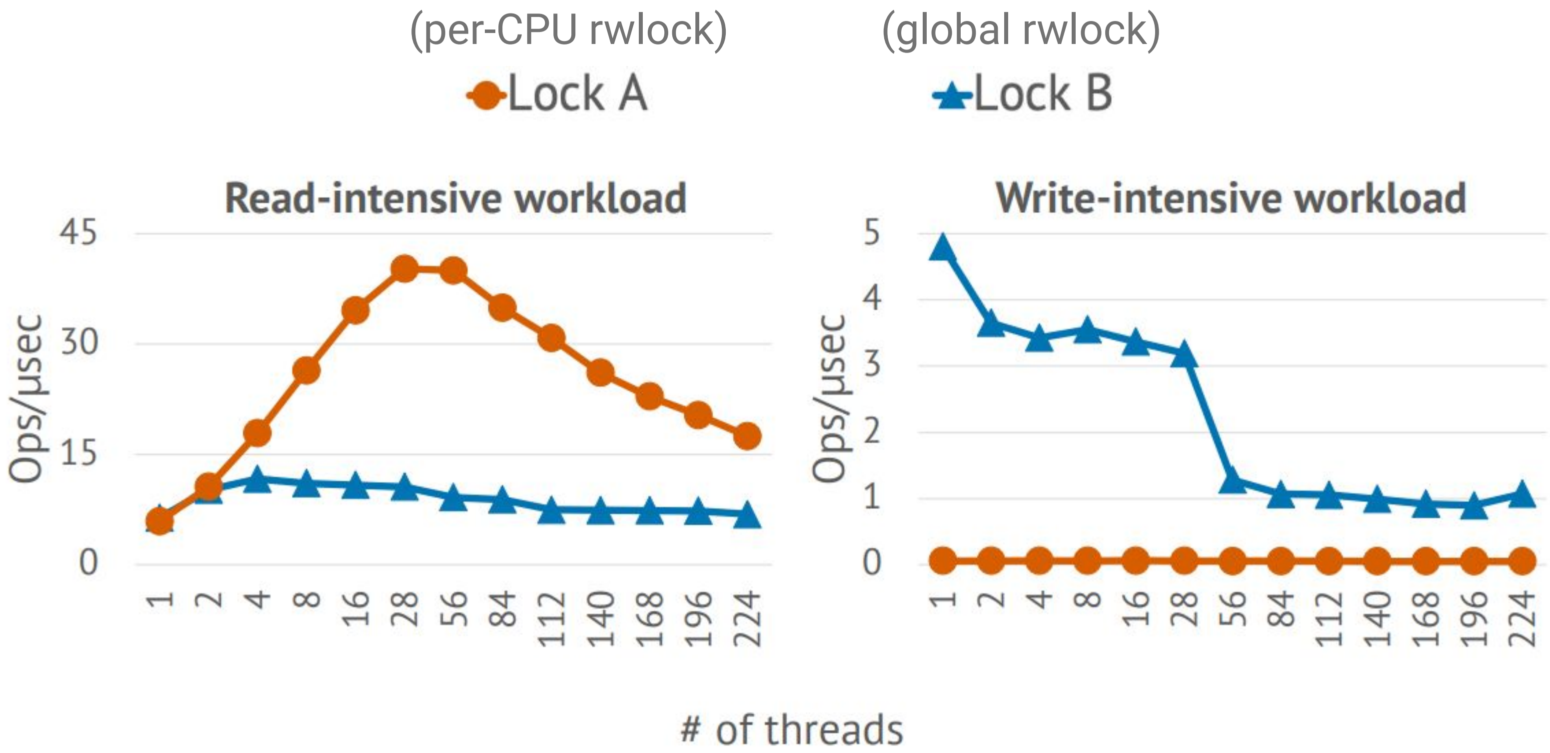
Application-Informed Kernel Synchronization Primitives

Rajas Bhatt

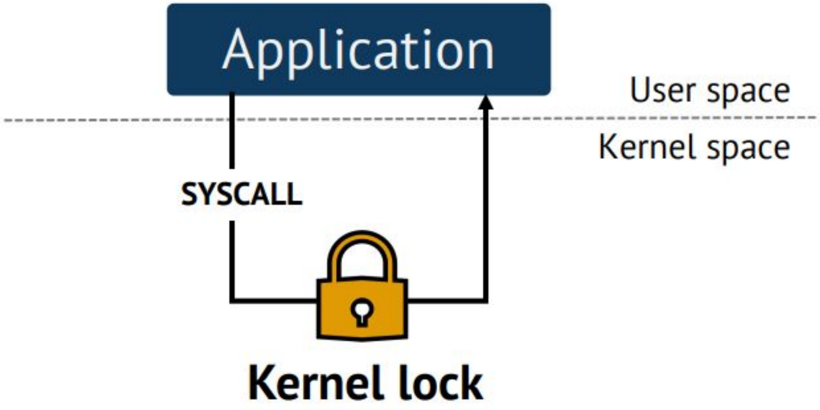
Locks are Critical for Application Performance



One lock is never the best in all cases



Kernel Locks affect application performance too!



Applications regularly ask for kernel support

System Calls!

But Kernel Locks are thought about as:

- Generic primitives good for all use cases
- Unsafe to customize/change
- Invisible to developers

Idea

- Kernel Locks show different performance for different workloads and different hardware configurations

- Software Developers should offer customizations/abstractions for kernel locks

- Due to a large number of workload/hardware scenarios, it is better to delegate this task safely to the end-user (using eBPF!)

SynCord

Generic primitives good for all use cases

Let application developers change locks on the fly using exposed APIs

Unsafe to customize/change

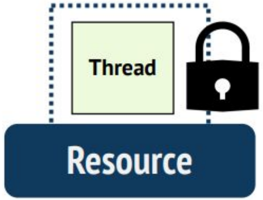
Use the eBPF verifier to ensure the safety of the user’s code

Invisible to developers

Profiling capability makes kernel lock design very much a part of application

Queue Based Locks

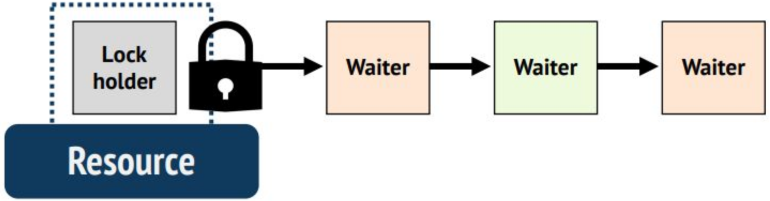
Low Contention



In a low contention scenario, lock is given almost instantaneously when asked for

What if we could which threads can access the fast path?

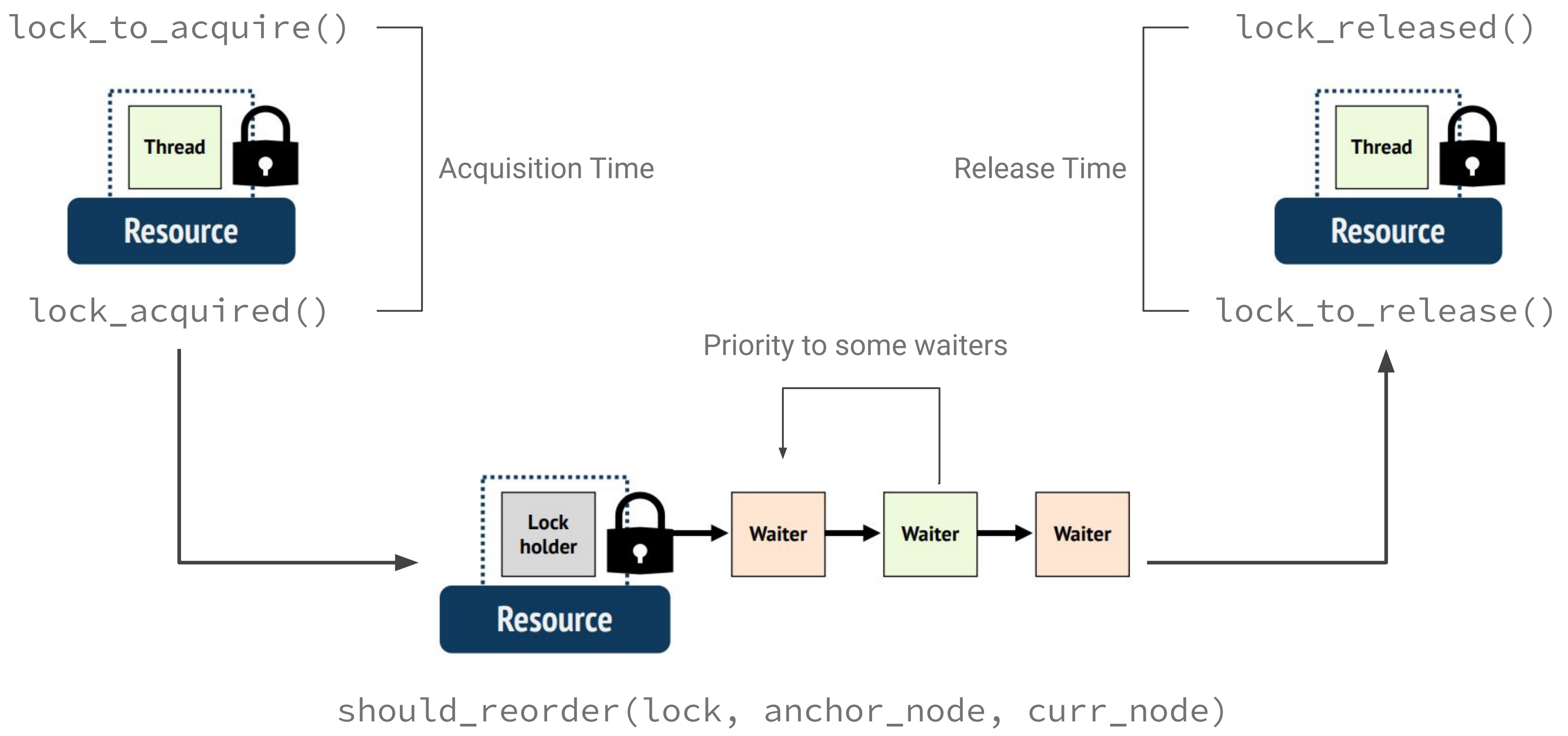
High Contention



In a high contention scenario, waiters push themselves into a contention queue

What if we could re-order waiters?

Exposure of APIs



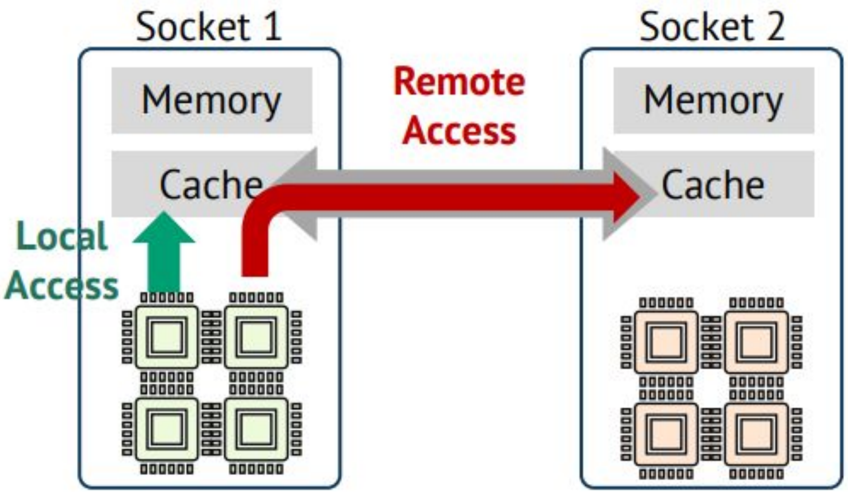
```python
def spin_lock(lock):
  lock_to_acquire(lock) # ① Hook the start of lock acquire

  if lock_bypass_acquire(lock): # ⑨ bypass lock acquire
    # lock acquisition is bypassed: used by lock experts
    return

  # If fastpath is enabled, first try to acquire the lock
  # instead of going into the wait queue
  if lock_enable_fastpath(lock) and # ⑥ can steal lock?
                  CAS(&lock.state, UNLOCK, LOCKED):
    lock_acquired(lock) # ② lock is acquired
    return

  node = Node() # A node to join the queue
  lock_to_enter_slowpath(lock, node) # ⑤ Hook before enqueuing
  queued_spin_lock_slowpath(lock, node) # Time to join the queue
  lock_acquired(lock) # ② Hook the start of critical section

def spin_unlock(lock):
  lock_to_release(lock) # ③ Hook the end of critical section

  if lock_bypass_release(lock): # ⑩ bypass lock release
    # lock release is bypassed; used along with ⑨
    return

  lock.state = UNLOCK # Lock released; critical section ends
  lock_released(lock) # ④ Hook right after critical section
```

Waiter Reordering is done inside this function

Reordering can lead to **starvation** of lower ‘priority’ waiters

`skip_reorder()` function which takes the queue back to FIFO operation

Use Case: NUMA-aware Spinlocks



Accessing local socket memory is faster than remote socket memory

NUMA-aware locks try to group lock requests from same socket together

Otherwise may lead to **bouncing of cache-lines**

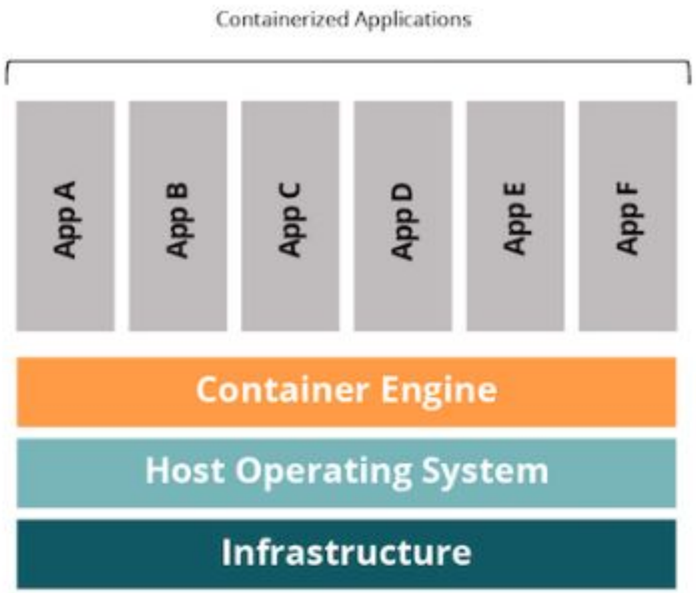
Store the socket ID of each thread and try to batch threads running at the same CPU together using `should_reorder(lock, anchor_node, curr_node)`

Use `skip_reorder` to randomly change the socket ID of the waiter thread to prevent starvation

Using Trātṛ to tame Adversarial Synchronization

Seemant Achari

Synchronisation primitives: A new avenue for attack on the kernel?



Shared elements:
- CPU
- Memory
- Network devices
- Data-structures?

Shared data-structures:
- pid-struct pid hashmap
- global open files table
- the red-black tree for CPU scheduling

Synchronisation primitives: A new avenue for attack on the kernel?

Synchronization Attack and Framing attack:

Setup: A shared kernel data-structure with **weak time complexity** guarantees protected by a synchronisation primitive

Premise: Increase the length of the critical section so that other tenants would have to wait longer to acquire the locks

Synchronisation primitives: A new avenue for attack on the kernel?

```
void insert(struct node **list, struct node *n) {
  lock();
  n->next = *list; *list = n;
  unlock();
}

struct node *find(struct node **list, int data) {
  lock();
  struct node *n = *list;
  while (n) {
    if (n->data == data) {
      unlock();
      return n;
    }
    n = n->next;
  }
  unlock();
  return NULL;
}
```

What happens to the critical section in *find* if we expand the linked list?

Synchronisation primitives: A new avenue for attack on the kernel?

Vulnerable data-structures in the kernel:

- *inode_cache:*

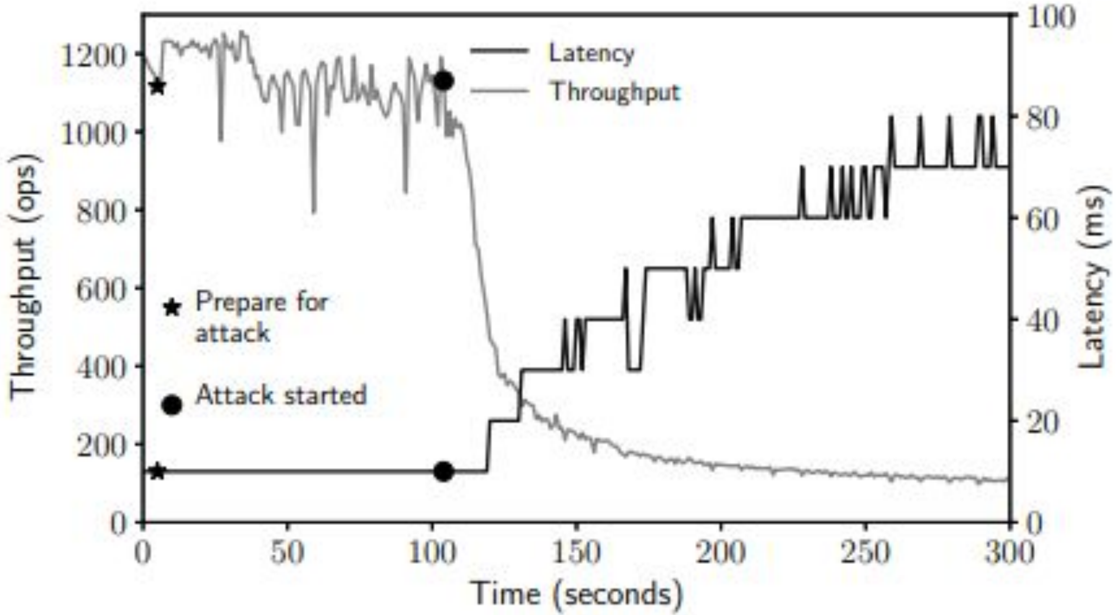
  input to the hash function: inode ID and superblock address

  synchronisation: a global lock for the table

  How is it attacked?

  - FUSE: File-system in user space (allocate inode IDs from the user space)
  - Break the hash and target a specific bucket, keep reading those inodes which collide with the bucket

Synchronisation primitives: A new avenue for attack on the kernel?



How do we stop such attacks?

What all do we need to do?

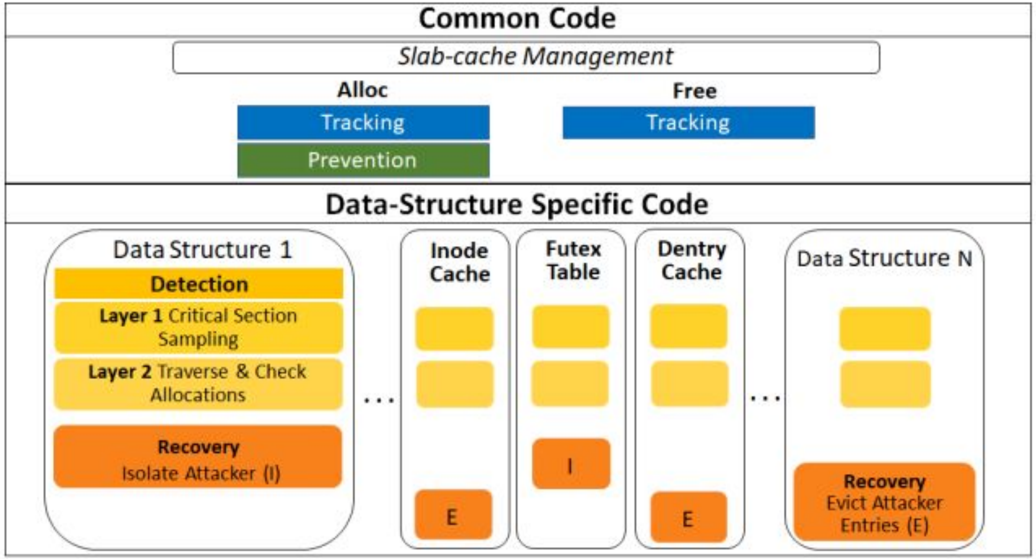
- Detect the attack
- Prevent it from happening further
- Recover from the attack

Why do we need to recover?

To stop the framing attacks or else the victims would keep reading the expanded data structure

How do we stop such attacks?

Detection and tracking

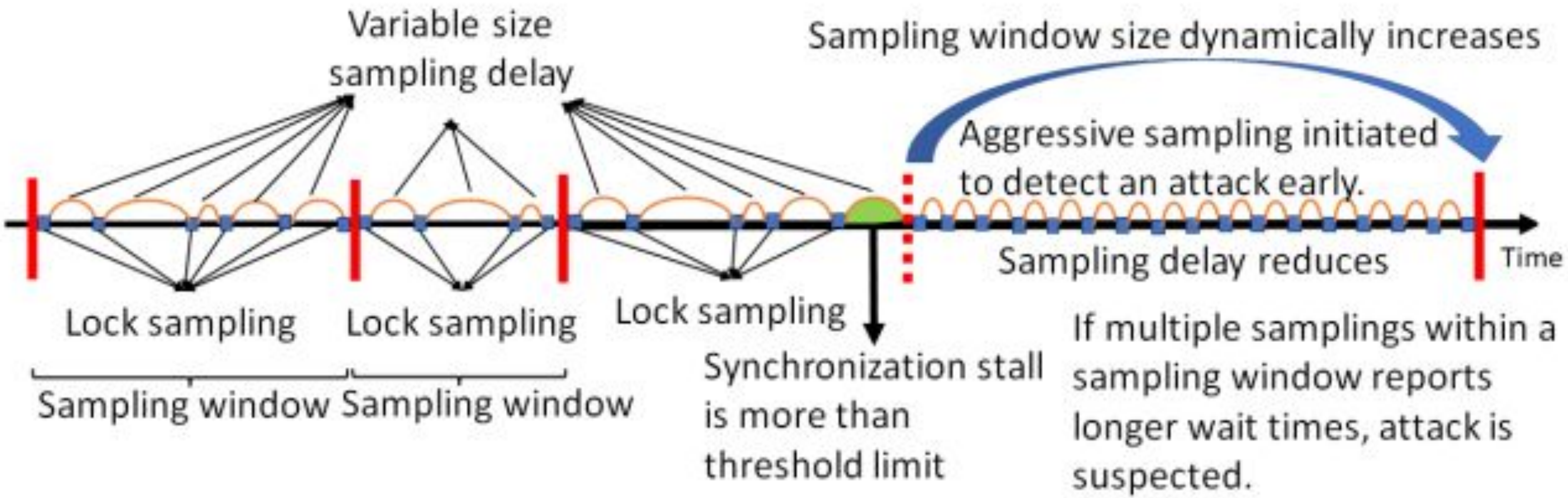


Tracking: Embed the objects with a user ID to identify which user allocated the object

Conditions to indicate:

- Long critical section
- High single user allocation

How do we stop such attacks?



How do we stop such attacks?

Prevention:

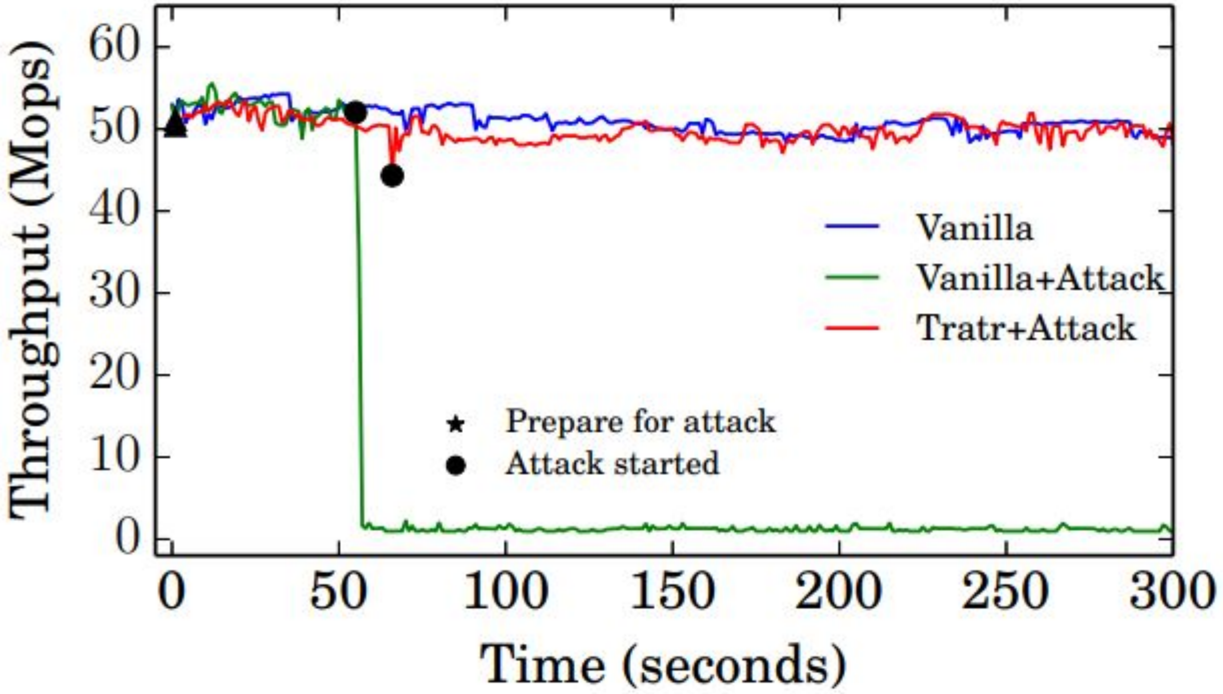
Don’t let the attacker allocate any new objects until a certain window of time

Why not suspend the process or kill the container?

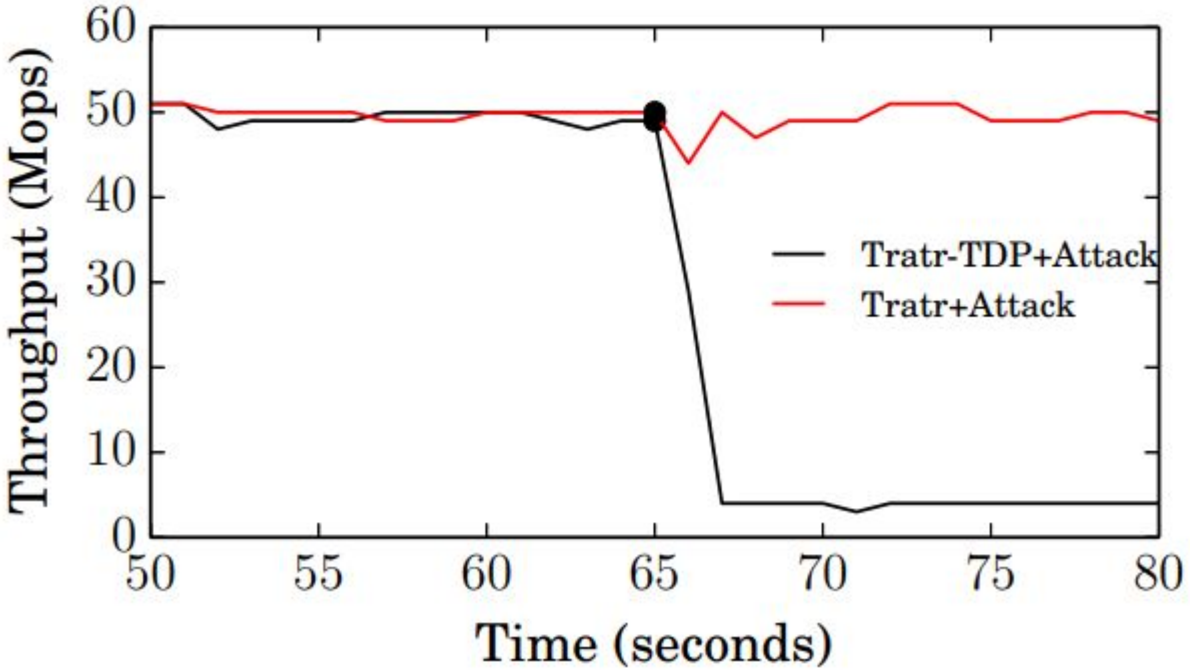
Recovery:

- For cache like data-structures where evicting the entries won’t affect the correctness of the application: remove the entries
- For other data structures create a copy of the data-structure with only entries from the attacker and remove those entries from the original data structure

How effective is Tratr?



How effective is Tratr?



Thank You!