PennState

Parsec: Fast, Scalable, and Secure Design with Wait-Free Parallelism

Ruslan Nikolaev, rnikola@psu.edu, The Pennsylvania State University, USA

Thread Synchronization

Thread Synchronization

- Modern systems are increasingly multicore
  - Mutual exclusion does not suffice => need better parallelism support

Thread Synchronization

- Modern systems are increasingly multicore
  - Mutual exclusion does not suffice => need better parallelism support
- Alternatives: fine-grained locks and read-copy-update (RCU)
  - These approaches are still blocking and need mutual exclusion

Thread Synchronization

- Modern systems are increasingly multicore
  - Mutual exclusion does not suffice => need better parallelism support
- Alternatives: fine-grained locks and read-copy-update (RCU)
  - These approaches are still blocking and need mutual exclusion
- **Non-blocking** data structures are becoming increasingly popular
  - ***Obstruction-freedom***: a thread always makes progress when executing without interference from other threads
  - ***Lock-freedom****:* at least one thread always makes progress (even with interference)
  - ***Wait-freedom****:* all threads always make progress

Lock-Freedom vs. Wait-Freedom

- **Deadlock-freedom** and **starvation-freedom** are well-known properties in the blocking world
  - **Lock-free** algorithms only guarantee global progress since individual threads can still starve
  - **Wait-free** algorithms prevent thread starvation

Lock-Freedom vs. Wait-Freedom

- **Deadlock-freedom** and **starvation-freedom** are well-known properties in the blocking world
  - **Lock-free** algorithms only guarantee global progress since individual threads can still starve
  - **Wait-free** algorithms prevent thread starvation

Blocking	Non-Blocking
Deadlock-free	Lock-free
Starvation-free	Wait-free

Lock-Freedom vs. Wait-Freedom

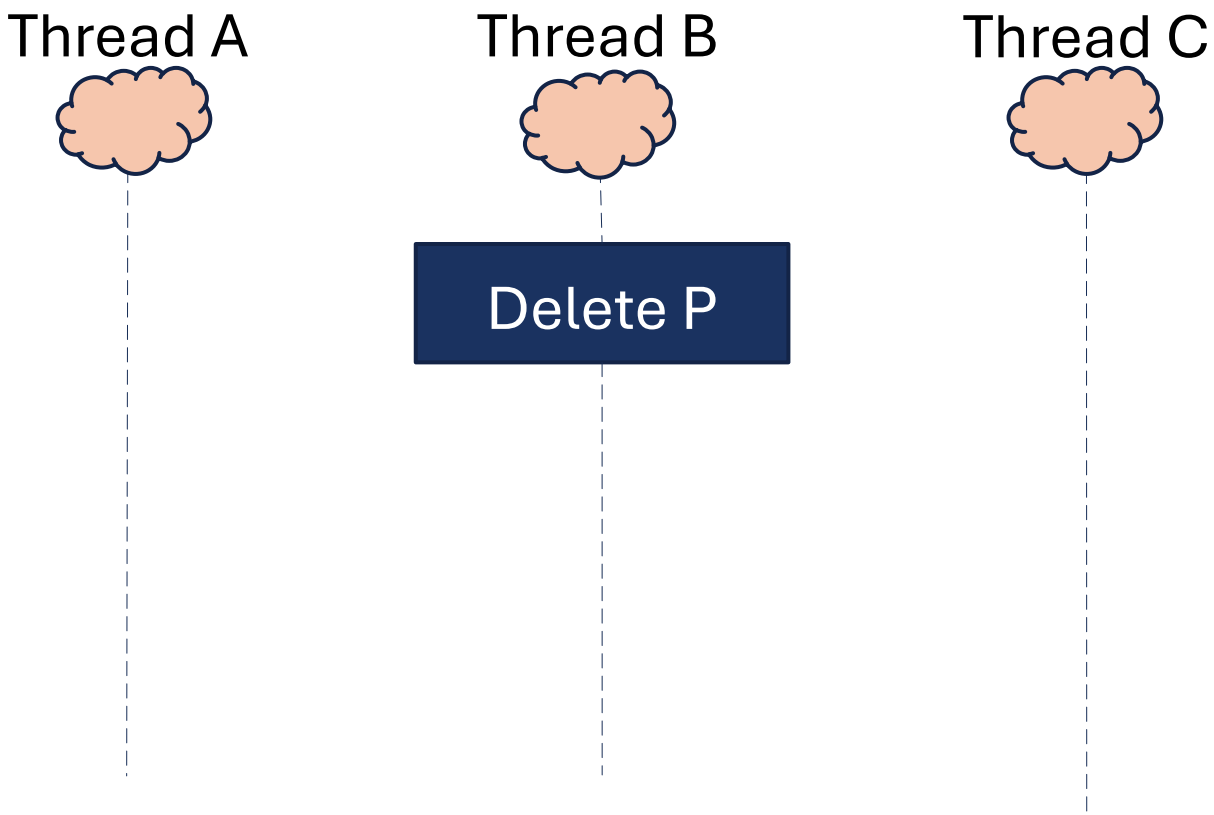
- **Deadlock-freedom** and **starvation-freedom** are well-known properties in the blocking world
  - **Lock-free** algorithms only guarantee global progress since individual threads can still starve
  - **Wait-free** algorithms prevent thread starvation

Blocking	Non-Blocking
Deadlock-free	Lock-free
Starvation-free 	Wait-free 

Background: Read-Copy-Update (RCU)

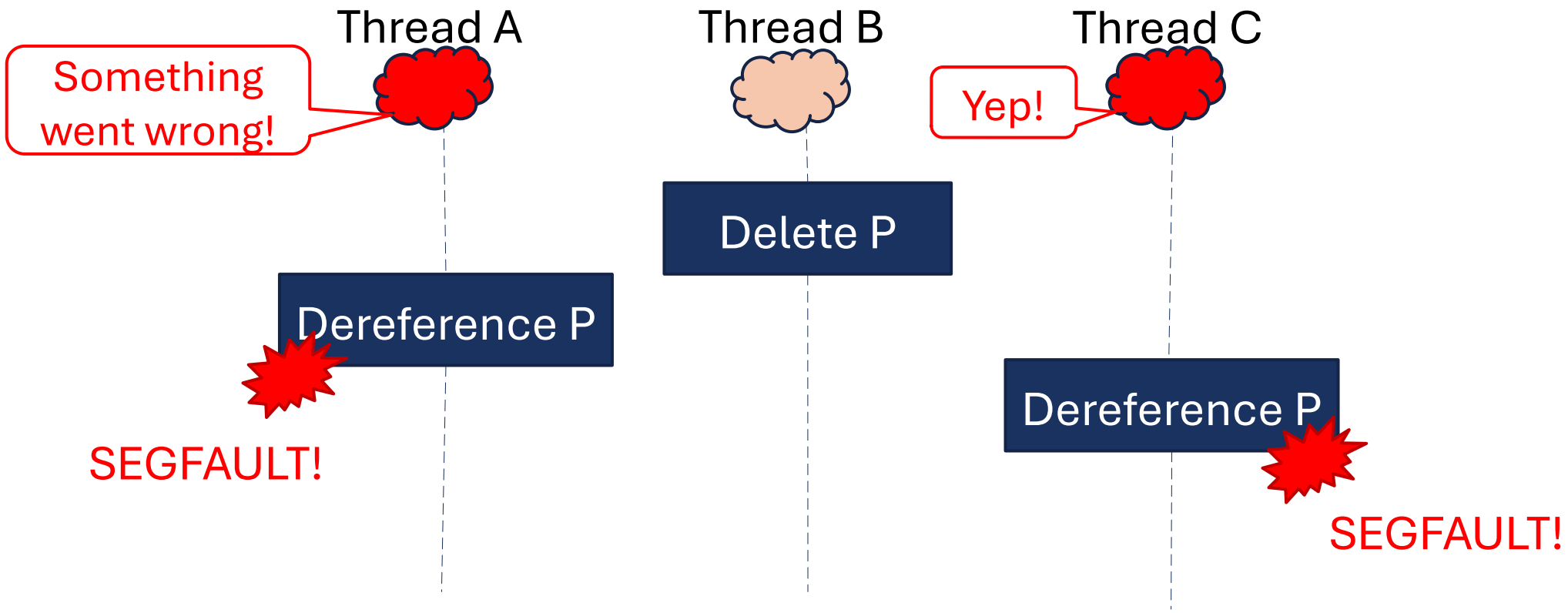
- Used widely in the Linux kernel
  - Avoids mutual exclusion for readers
  - But does not solve synchronization for writers unless it is trivial
- Great performance for reading-dominated workloads
- Has a built-in memory reclamation strategy
  - Can safely reclaim memory objects even though readers have stale pointers

Background: Read-Copy-Update (RCU)



One thread wants to de-allocate a memory object which is still reachable by concurrent threads

Background: Read-Copy-Update (RCU)



One thread wants to de-allocate a memory object which is still reachable by concurrent threads

Background: Read-Copy-Update (RCU)

- Used widely in the Linux kernel
  - Avoids mutual exclusion for readers
  - But does not solve synchronization for writers unless it is trivial
- Great performance for reading-dominated workloads
- Has a built-in memory reclamation strategy
  - Can safely reclaim memory objects even though readers have stale pointers
- … But RCU has a **known but rarely discussed problem**
  - Vulnerability to **denial-of-service (DoS)** attacks

Background: Read-Copy-Update (RCU)

- Used widely in the Linux kernel
  - Avoids mutual exclusion for readers
  - But does not solve synchronization for writers unless it is trivial
- Great performance for reading-dominated workloads
- Has a built-in memory reclamation strategy
  - Can safely reclaim memory objects even though readers have stale pointers
- … But RCU has a **known but rarely discussed problem**
  - Vulnerability to **denial-of-service (DoS)** attacks

RCU Vulnerability: Exhausting Memory

```
struct foo { struct rcu_head rh; };
struct foo *g;

void reader() {
    rcu_read_lock();
    cur_mem = rcu_dereference(g);
    ... // control-flow attack: unlock is
    rcu_read_unlock(); // skipped or delayed
}
```

RCU Vulnerability: Exhausting Memory

```
struct foo { struct rcu_head rh; };
struct foo *g;

void reader() {
    rcu_read_lock();
    cur_mem = rcu_dereference(g);
    ... // control-flow attack: unlock is
    rcu_read_unlock(); // skipped or delayed
}
```

```
void writer_block() {
    new_mem = malloc(sizeof(struct foo));
    old_mem = rcu_dereference(g);
    rcu_assign_pointer(g, new_mem);
    synchronize_rcu(); // Blocks indefinitely!
    kfree(old_mem); // Not reachable
}
```

RCU Vulnerability: Exhausting Memory

```
struct foo { struct rcu_head rh; };
struct foo *g;

void reader() {
    rcu_read_lock();
    cur_mem = rcu_dereference(g);
    ... // control-flow attack: unlock is
    rcu_read_unlock(); // skipped or delayed
}
                void writer_nonblock() {
                    for (i = 0; i < count; i++) {
                        new_mem = malloc(sizeof(struct foo));
                        old_mem = rcu_dereference(g);
                        rcu_assign_pointer(g, new_mem);
                        call_rcu(&old_mem->rh, callback_kfree);
                        // Exhausts memory because it allocates
                        // new memory without releasing anything!
                    }
                }
```

RCU Vulnerability: Exhausting Memory

RCU Vulnerability: Exhausting Memory

- Non-blocking **call_rcu** is problematic
  - Can easily exhaust memory, virtually no limit
  - High memory footprints: see “**The RCU-Reader Preemption Problem in VMs**” by Aravinda Prasad, K. Gopinath, and Paul E. McKenney **[ATC'17]**

RCU Vulnerability: Exhausting Memory

- Non-blocking **call_rcu** is problematic
  - Can easily exhaust memory, virtually no limit
  - High memory footprints: see “**The RCU-Reader Preemption Problem in VMs**” by Aravinda Prasad, K. Gopinath, and Paul E. McKenney **[ATC'17]**
- **synchronize_rcu** blocks the execution
  - Is it *better* than high memory consumption?
  - Has a high latency of at least 1 jiffy, slowdowns of several milliseconds

RCU Vulnerability: Exhausting Memory

- Non-blocking **call_rcu** is problematic
  - Can easily exhaust memory, virtually no limit
  - High memory footprints: see “**The RCU-Reader Preemption Problem in VMs**” by Aravinda Prasad, K. Gopinath, and Paul E. McKenney **[ATC'17]**
- **synchronize_rcu** blocks the execution
  - Is it *better* than high memory consumption?
  - Has a high latency of at least 1 jiffy, slowdowns of several milliseconds
- High latency of **synchronize_rcu** can mitigate DoS attacks
  - But not fully and is **not always acceptable**…
  - **synchronize_rcu_expedited** => more aggressive and vulnerable to DoS

What is the Solution?

What is the Solution?

- **Problem:** DoS is closely related to *progress* properties

What is the Solution?

- **Problem:** DoS is closely related to *progress* properties
- RCU is actually not lock-free, **blocking** *even for readers*
  - Recall: lock-freedom means at least one thread **always** makes progress
  - But when memory is exhausted, **no further progress can be made** (irrespective whether it is a reader or a writer)

What is the Solution?

- **Problem:** DoS is closely related to *progress* properties
- RCU is actually not lock-free, **blocking** *even for readers*
  - Recall: lock-freedom means at least one thread **always** makes progress
  - But when memory is exhausted, **no further progress can be made** (irrespective whether it is a reader or a writer)
- **Solution:** Use **non-blocking** approaches instead?
  - Note ***obstruction-free*** approaches are vulnerable to DoS because they depend on non-interference of threads
  - ***Can lock-free algorithms help with that?***

What is the Solution?

- **Problem:** DoS is closely related to *progress* properties
- RCU is actually not lock-free, **blocking** *even for readers*
  - Recall: lock-freedom means at least one thread **always** makes progress
  - But when memory is exhausted, **no further progress can be made** (irrespective whether it is a reader or a writer)
- **Solution:** Use **non-blocking** approaches instead?
  - Note ***obstruction-free*** approaches are vulnerable to DoS because they depend on non-interference of threads
  - ***Can lock-free algorithms help with that?***

Short answer: No

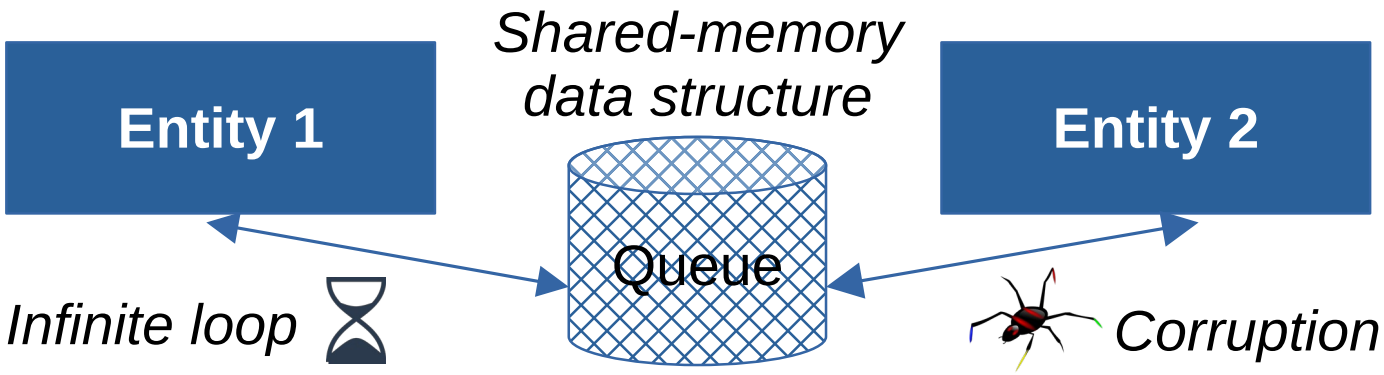
But **wait-free** algorithms can

What is the Problem with Lock-Freedom?

- A given thread may theoretically **never** complete due to starvation
  - Unlikely in practice due to randomness
  - Randomness can be lost when an attacker deliberately slows down atomic operations by invalidating L1 cache lines

What is the Problem with Lock-Freedom?

- A given thread may theoretically **never** complete due to starvation
  - Unlikely in practice due to randomness
  - Randomness can be lost when an attacker deliberately slows down atomic operations by invalidating L1 cache lines



How do we know that the delay is not transient and the loop is infinite (e.g., queue is corrupted)?

Wait-Free Approaches

Wait-Free Approaches

- Historically harder to implement
  - Now more feasible with Kogan-Petrank [PPoPP'12] “fast-path-slow-path” and similar methods
  - Threads collaborate to bound the number of operations for each thread

Wait-Free Approaches

- Historically harder to implement
  - Now more feasible with Kogan-Petrank [PPoPP'12] “fast-path-slow-path” and similar methods
  - Threads collaborate to bound the number of operations for each thread
- Provide a ***theoretical upper-bound*** for the number of iterations
  - When exceeding this threshold, we can declare that the data structure is corrupted by the other side
  - Assuming rigorous memory safety checks and this bound, we can avoid DoS => an insight that was not widely discussed in the literature

Hardware Primitives

Hardware Primitives

- RISC CPUs widely use a **pair** of instructions: Load-Link (LL) and Store-Conditional (SC)
  - Not guaranteed to ever succeed due to interrupts, false sharing, etc.

Hardware Primitives

- RISC CPUs widely use a **pair** of instructions: Load-Link (LL) and Store-Conditional (SC)
  - Not guaranteed to ever succeed due to interrupts, false sharing, etc.
- Compare-and-Swap (CAS)
  - A **single** CPU instruction => does not have the above problem

Hardware Primitives

- RISC CPUs widely use a **pair** of instructions: Load-Link (LL) and Store-Conditional (SC)
  - Not guaranteed to ever succeed due to interrupts, false sharing, etc.
- Compare-and-Swap (CAS)
  - A **single** CPU instruction => does not have the above problem
- Specialized instructions
  - Fetch-and-Add (FAA) and SWAP (XCHG)
  - Can be implemented via LL/SC and CAS

Hardware Primitives

Hardware Primitives

- CAS is considered inferior to LL/SC [Herlihy's Hierarchy]
  - ABA problem (false-positive match) is possible when objects are being recycled and pointers happen to be the same
  - FAA, SWAP, etc. is potentially more expensive via CAS
  - LL/SC while theoretically superior, prevents nesting and restricts types of operations in practice

Hardware Primitives

- CAS is considered inferior to LL/SC [Herlihy's Hierarchy]
  - ABA problem (false-positive match) is possible when objects are being recycled and pointers happen to be the same
  - FAA, SWAP, etc. is potentially more expensive via CAS
  - LL/SC while theoretically superior, prevents nesting and restricts types of operations in practice
- But these problems can be solved
  - Double-width CAS (cmpxchg16b), where the second word is a monotonically increasing tag, solves the ABA problem
  - **Wait-free** FAA and SWAP can be implemented natively in hardware

Issues with LL/SC

- “Strong” CAS implemented via LL/SC is problematic
  - Programmers expect CAS either succeed or fail after ***finite time***
  - But when implementing via LL/SC, we have a potentially infinite loop
- “Weak” CAS is safer for lock-free algorithms
  - But programmers are not necessarily aware of this
  - No ***bound*** for wait-free algorithms => **no wait-freedom**
- No “weak” FAA, etc.
  - Always dangerous to use

Issues with LL/SC

- “Strong” CAS implemented via LL/SC is problematic
  - Programmers expect CAS either succeed or fail after ***finite time***
  - But when implementing via LL/SC, we have a potentially infinite loop
- “Weak” CAS is safer for lock-free algorithms
  - But programmers are not necessarily aware of this
  - No ***bound*** for wait-free algorithms => **no wait-freedom**
- No “weak” FAA, etc.
  - Always dangerous to use

Conclusion: LL/SC is unsafe and bad even for RISC architectures!
Fortunately, AArch64 and RISC-V already fixed this problem

Evaluation Setup

Evaluation Setup

- AMD EPYC 9554, 64 cores, 128 hardware threads, 384 GiB of RAM

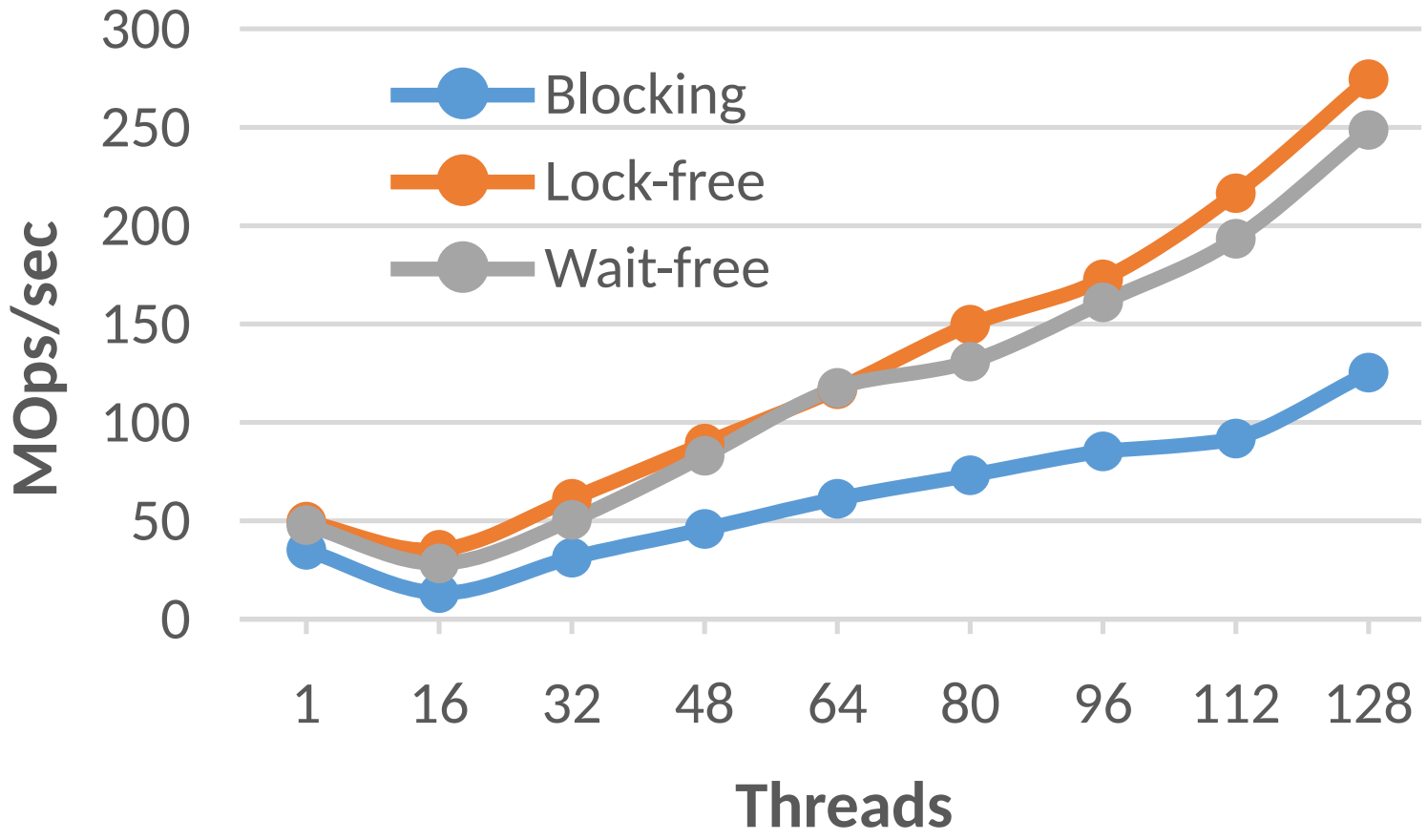
Evaluation Setup

- AMD EPYC 9554, 64 cores, 128 hardware threads, 384 GiB of RAM
- Go-like channels
  - Every thread sends to or receives from its own channel and from another channel for the next thread => at most **two** threads access any channel
  - Up to 512 messages in any channel

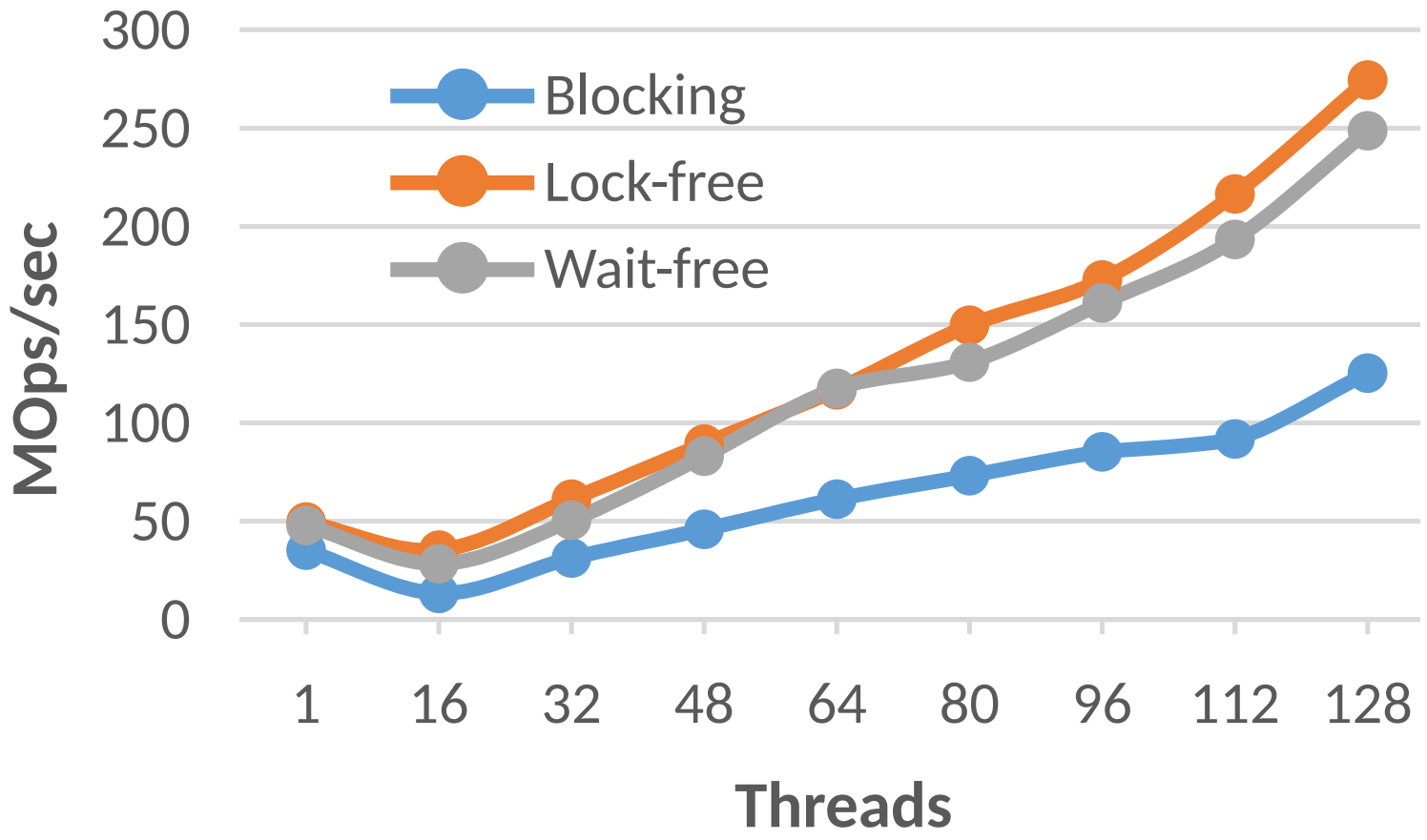
Evaluation Setup

- AMD EPYC 9554, 64 cores, 128 hardware threads, 384 GiB of RAM
- Go-like channels
  - Every thread sends to or receives from its own channel and from another channel for the next thread => at most **two** threads access any channel
  - Up to 512 messages in any channel
- Our C implementation
  - Straight-forward implementation using semaphores and buffer **locks**
  - Semaphores and a **lock-free** ring buffer by Nikolaev **[DISC'19]**
  - Semaphores and a **wait-free** ring buffer by Nikolaev & Ravindran **[SPAA'22]**
  - The latter two approaches are **non-blocking** unless sleeping (nothing to produce or to consume)

Evaluation


300
250
200
150
100
50
0
MOps/sec
Blocking
Lock-free
Wait-free
1
16
32
48
64
80
96
112
128
Threads

Evaluation



Despite low-contention, blocking version is **2x-3x** slower

*System calls are needed to synchronize even just **two** threads*

Code Availability

- More information and code to be released at:

  https://github.com/rusnikola/parsec

Code Availability

- More information and code to be released at:

  https://github.com/rusnikola/parsec

Thank You!

Questions?