
Core C++ 2025
19 Oct. 2025 :: Tel-Aviv
“Lock me up, Scotty!”
Ben Liderman & Igor Khanin

About Us

Ben Liderman

System Architect,
Security & Trust Products

Igor Khanin

Senior Core Engineer,
Wallet Solutions

About Fireblocks

Fireblocks powers companies of all sizes to confidently build, run, and grow their business on the blockchain.

$10T+
Transfers Secured

300M+
Wallets Created

100+
Blockchains

2,000+
Global Customers

Reminder: The Mutual Exclusion Problem

Given *N* concurrent threads and a critical section *C*:

- **Safety** Property - at most one thread inside C
- **Liveness** Property - one of the following:

No Deadlocks

No Starvation / Basic Fairness

Bounded Wait / Strong Fairness

(As per prof. Hagit Attiya)

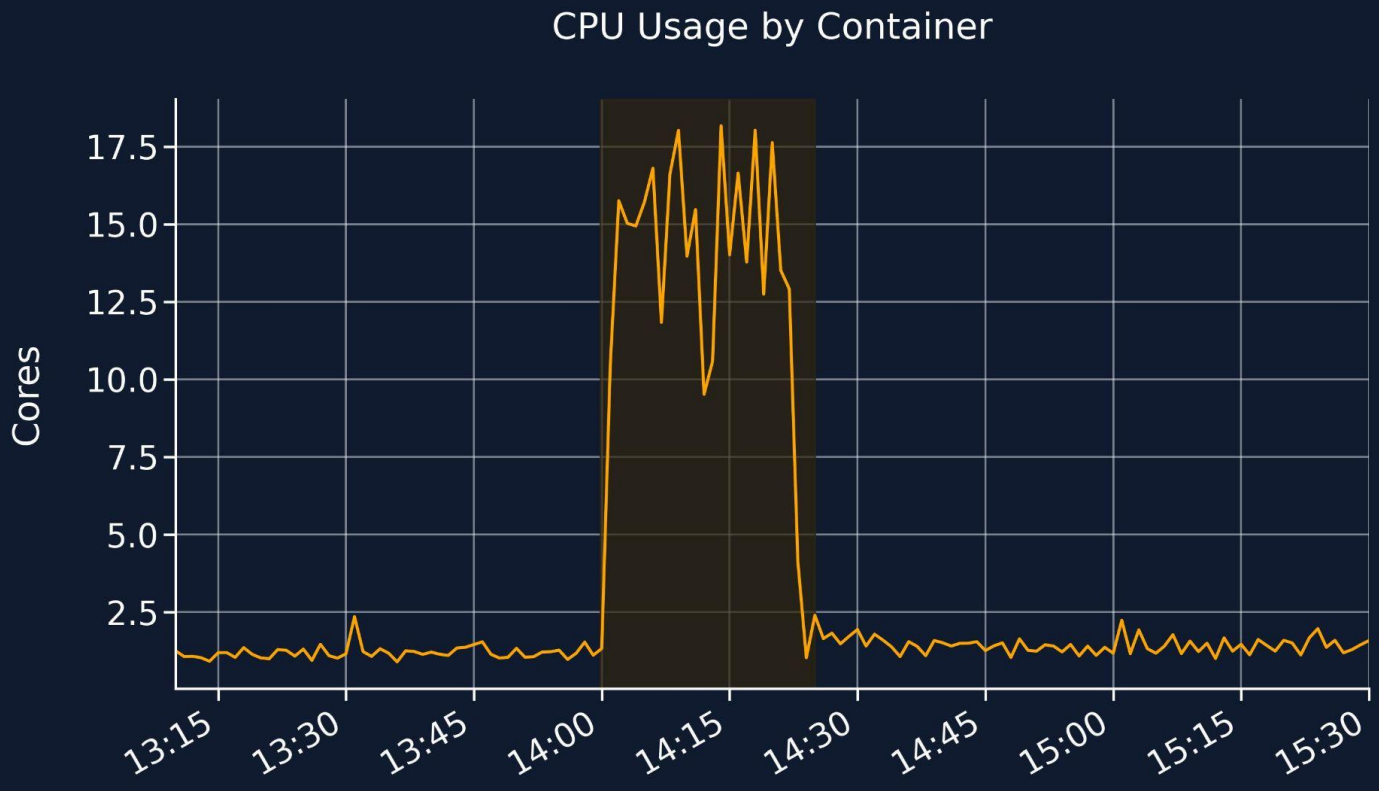
C++ has a Built-in Solution for the MEP

```cpp
static std::mutex transporter_lock;
static Engineer scotty;

void beam_up_crew(const std::string& crew_member)
{
    std::lock_guard lg { transporter_lock };

    scotty.beam_me_up(crew_member);
    std::cout << crew_member
              << " back on the Enterprise!"
              << std::endl;
}
```

It Worked, Until it didn’t…

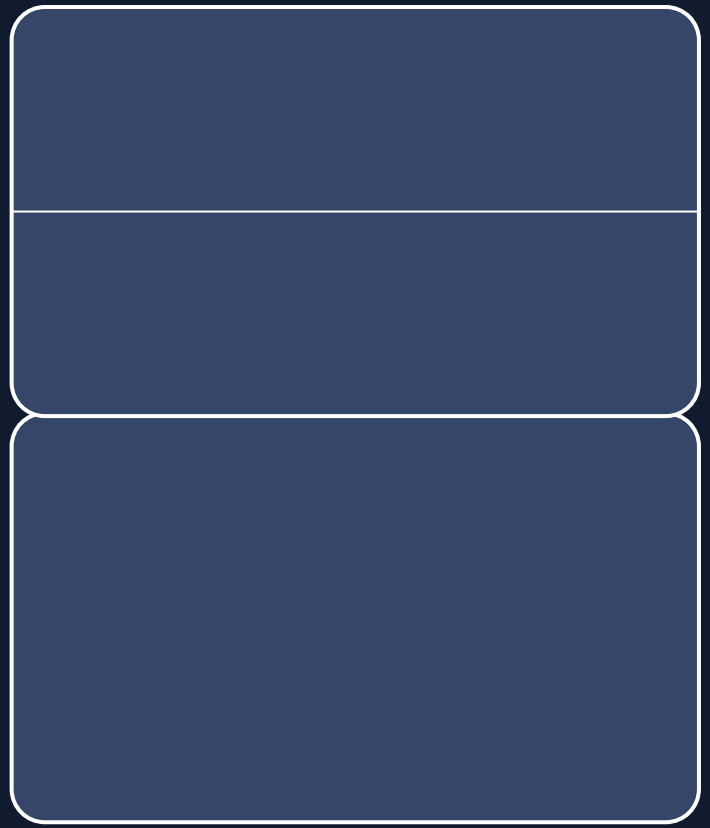


Software Guard eXtensions from 10,000 feet

Kernel

Userspace

Encrypted Memory



Untrusted Host

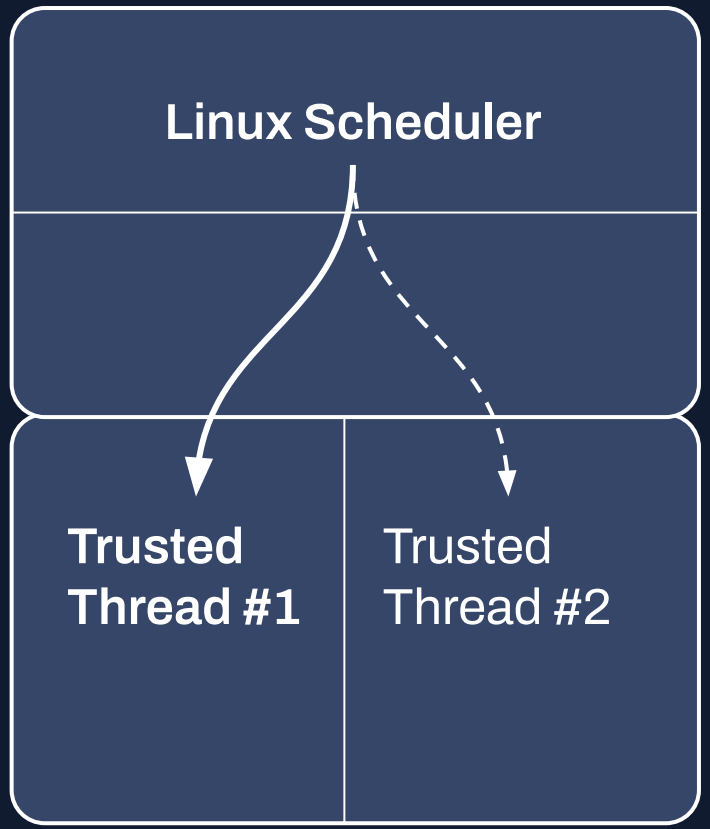
Trusted Enclave

Software Guard eXtensions from 10,000 feet

Kernel

Userspace

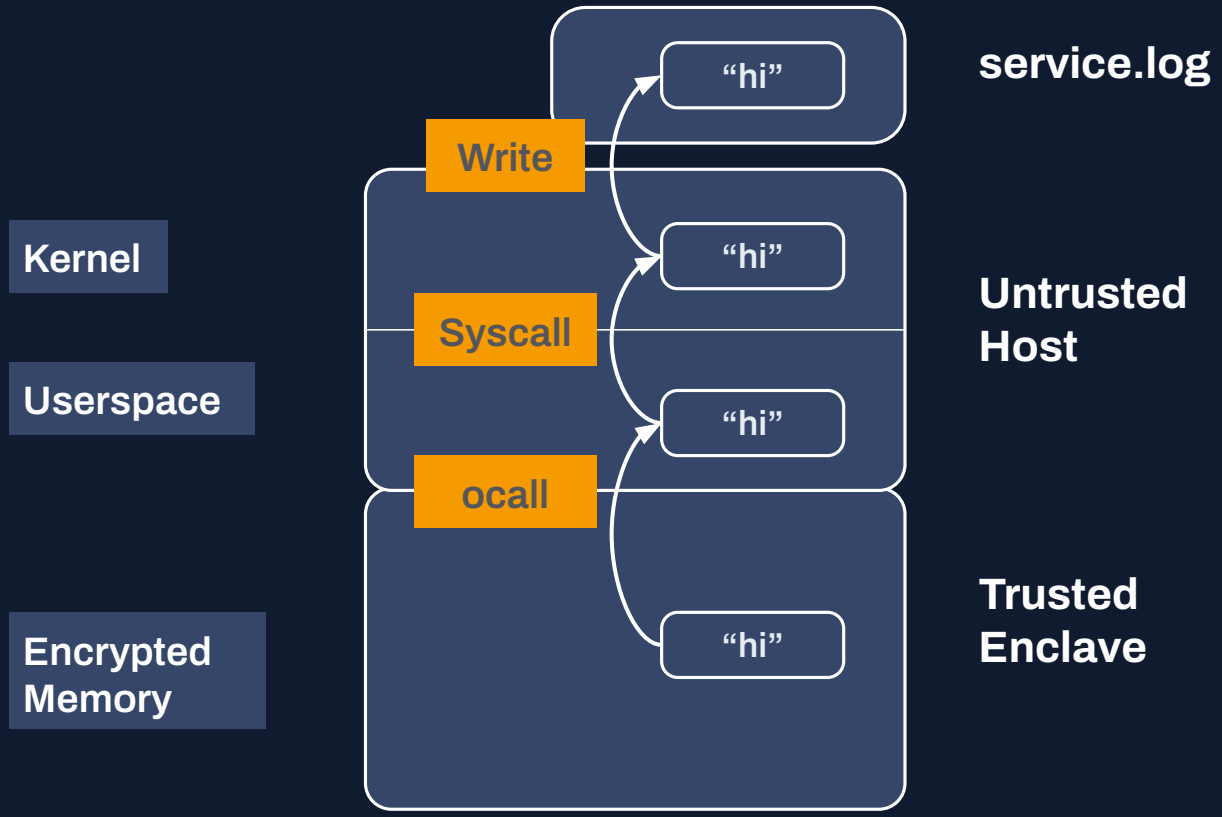
Encrypted Memory



Untrusted Host

Trusted Enclave

Software Guard eXtensions from 10,000 feet



Sources of Mutual Exclusion for Applications

- **Hardware support**
- **Operating system support**
- Pure software algorithms (not practical)

Mutual Exclusion with Hardware Support

```
std::atomic_flag is_locked = ATOMIC_FLAG_INIT;

void lock() {
    while (is_locked.test_and_set(std::memory_order_acquire))
    {
        // Spin!
    }
}

void unlock() {
    is_locked.clear(std::memory_order_release);
}
```

Mutual Exclusion with Hardware Support

- ✔ Safe and deadlock-free
- ✔ All state is inside the enclave
- ✔ Can be good under low contention

- ✗ Starvation
- ✗ Spinning wastes CPU
- ✗ Atomic access has overhead
- ✗ Breakdown under preemptive multitasking

Mutual Exclusion with OS Support (Linux)

```
constexpr uint32_t UNLOCKED = 0;
constexpr uint32_t LOCKED = 1;

alignas(4) uint32_t is_locked = UNLOCKED;
std::atomic_ref<uint32_t> is_locked_ref { is_locked };

void lock() {
    while (is_locked_ref.exchange(LOCKED, std::memory_order_acquire)
           != UNLOCKED) {
        syscall(SYS_futex, &is_locked, FUTEX_WAIT_PRIVATE, LOCKED, NULL);
    }
}

void unlock() {
    is_locked_ref.store(UNLOCKED, std::memory_order_release);
    syscall(SYS_futex, &is_locked, FUTEX_WAKE_PRIVATE, 1); // Wake one
}
```

A Balanced Approach

- We can mix the approaches
- Separate managing the lock state from thread control
- The “Parking Lot” design pattern

What we had

```
int sgx_thread_mutex_lock(sgx_thread_mutex_t *mutex) {
    while (1) {
        SPIN_LOCK(&mutex->m_lock);
        ...
        if (/*mutex is available*/) {
            ...
            mutex->m_owner = self;
            SPIN_UNLOCK(&mutex->m_lock);
            return 0;
        }
        QUEUE_INSERT_TAIL(&mutex->m_queue, self);
        SPIN_UNLOCK(&mutex->m_lock);
        sgx_thread_wait_untrusted_event_ocall(self);
    }
}
```

Critical Section Control

Thread Control

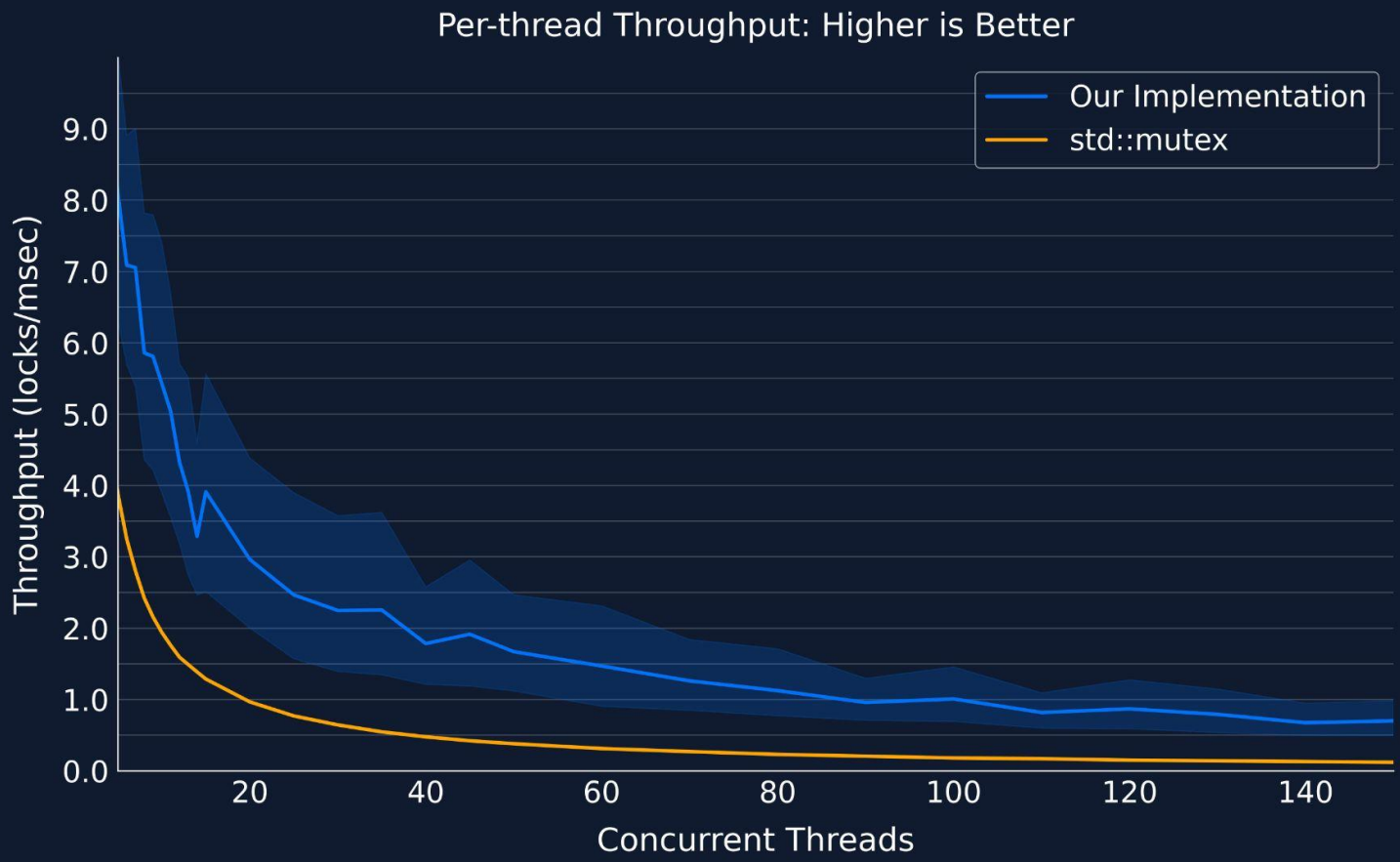
A First Improvement

- We can eliminate the inner spinlock by placing the state in a single trusted atomic variable
- But it doesn’t mean that we shouldn’t spin at all!
- **Adaptive Locking:** Try to set the lock state *N* times before asking the operating system to suspend the thread

Is Fairness Good?

- Satisfying a strong liveness property is a trade-off
- Strong fairness reduces variance of time for entering the critical section, at the cost of both throughput and average latency
- Most platforms do not have a fairness guarantee by default

What we Got


Per-thread Throughput: Higher is Better
Our Implementation
std::mutex
Throughput (locks/msec)
9.0
8.0
7.0
6.0
5.0
4.0
3.0
2.0
1.0
0.0
20
40
60
80
100
120
140
Concurrent Threads

Should I do that too?

- “Locking isn’t slow, contention is”
- Primary effort should be directed at architecture choices that reduce contention
- Understand the guarantees that your specific platform is giving, and compare them to your workloads
- “Insufficient facts always invite danger”

Fireblocks

Thank you!