HPC & Parallel Programming

Pthread Lab

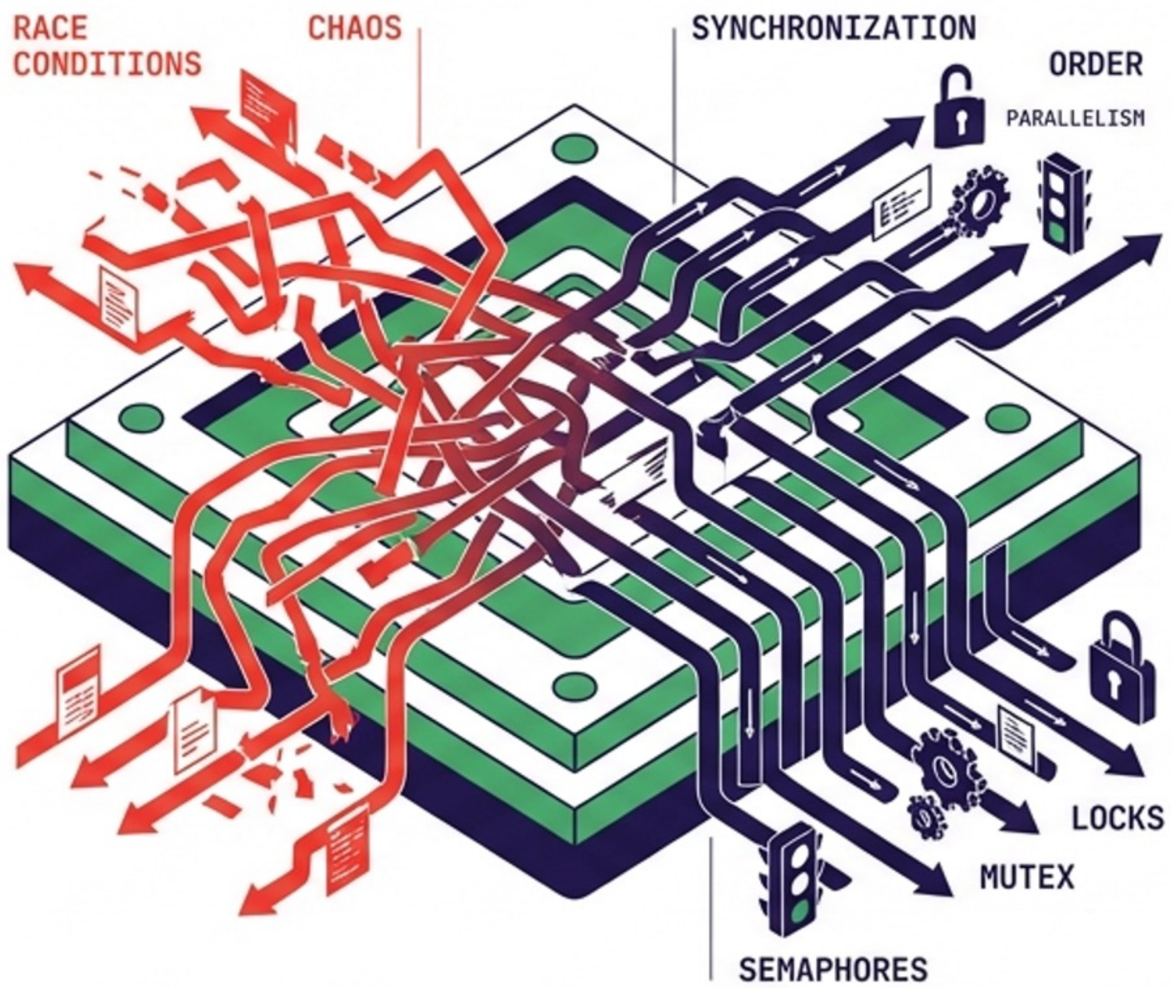
Kun Suo

Computer Science, Kennesaw State University

https://kevinsuo.github.io/

High Performance Computing: Pthread Project

Parallel Array Summation & Synchronization Analysis

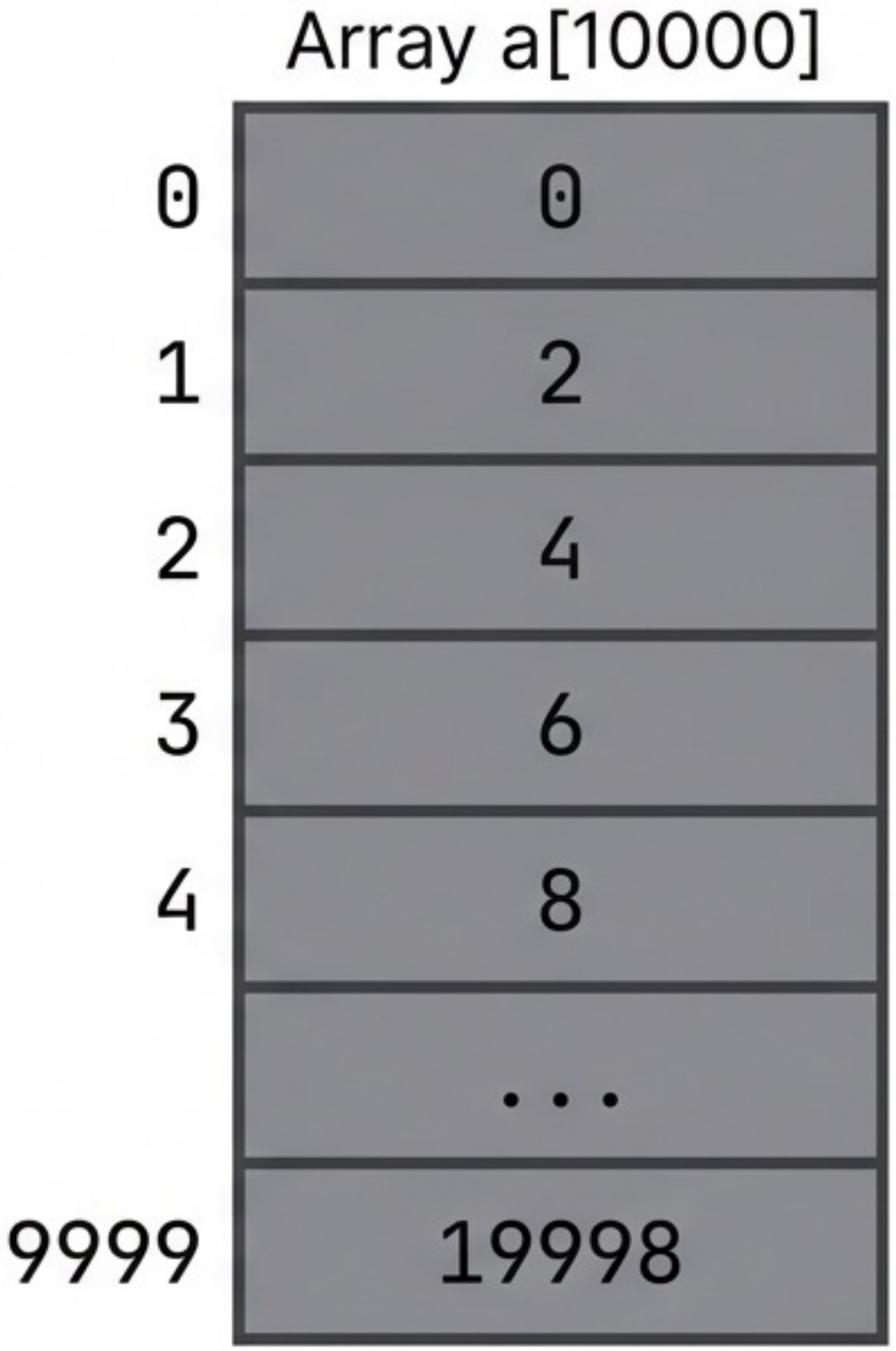


COURSE: HPC & PARALLEL PROGRAMMING

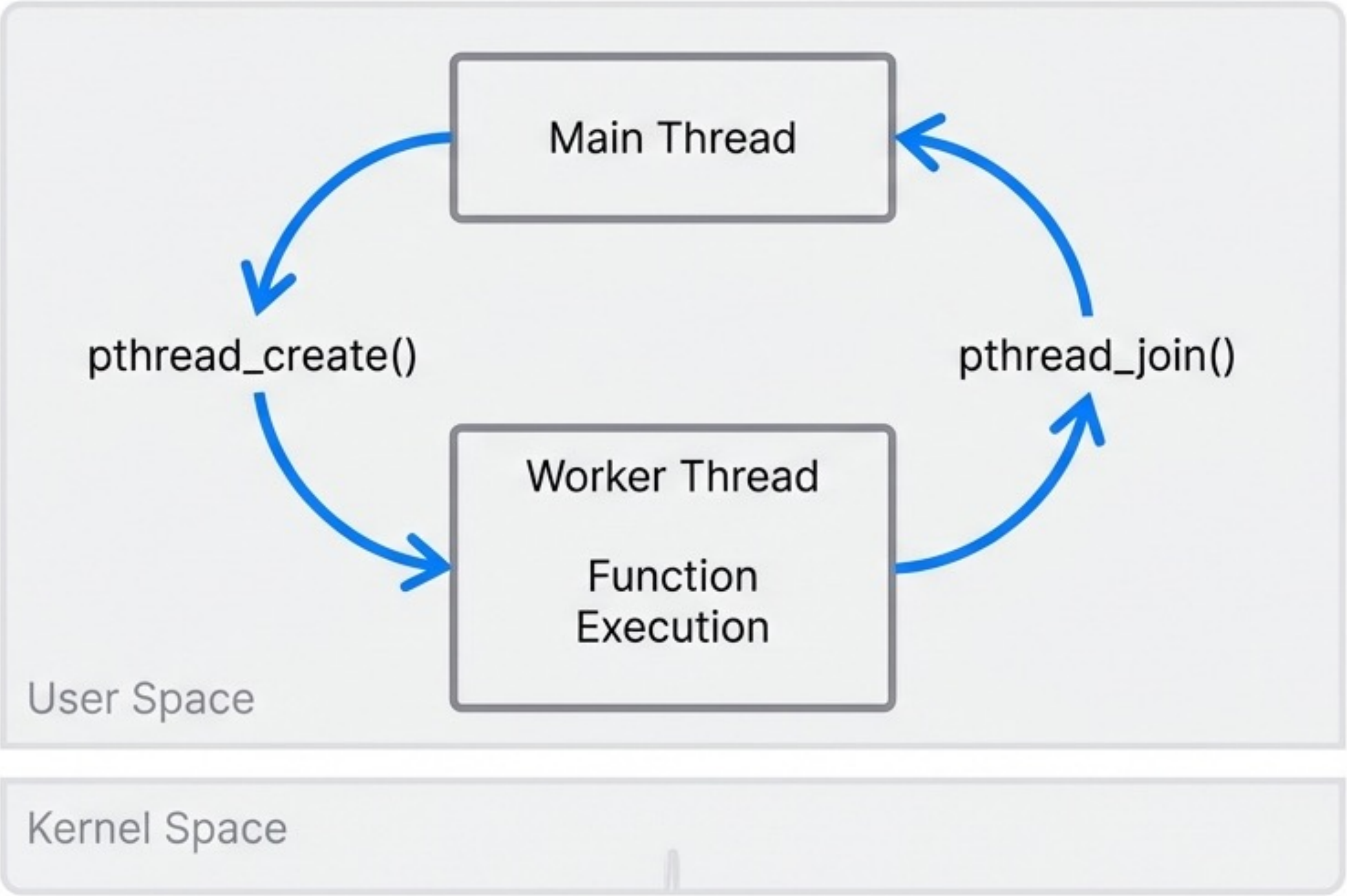
INSTRUCTOR: KUN SUO / KENNESAW STATE UNIVERSITY

The Objective

- **Task:** Sum array elements where `a[i] = 2*i`
- **Scale:** 10,000 elements
- **Threads:** Variable (1 to 10)
- **Output:** Execution Time + Global Sum



The Toolkit: Pthread Lifecycle



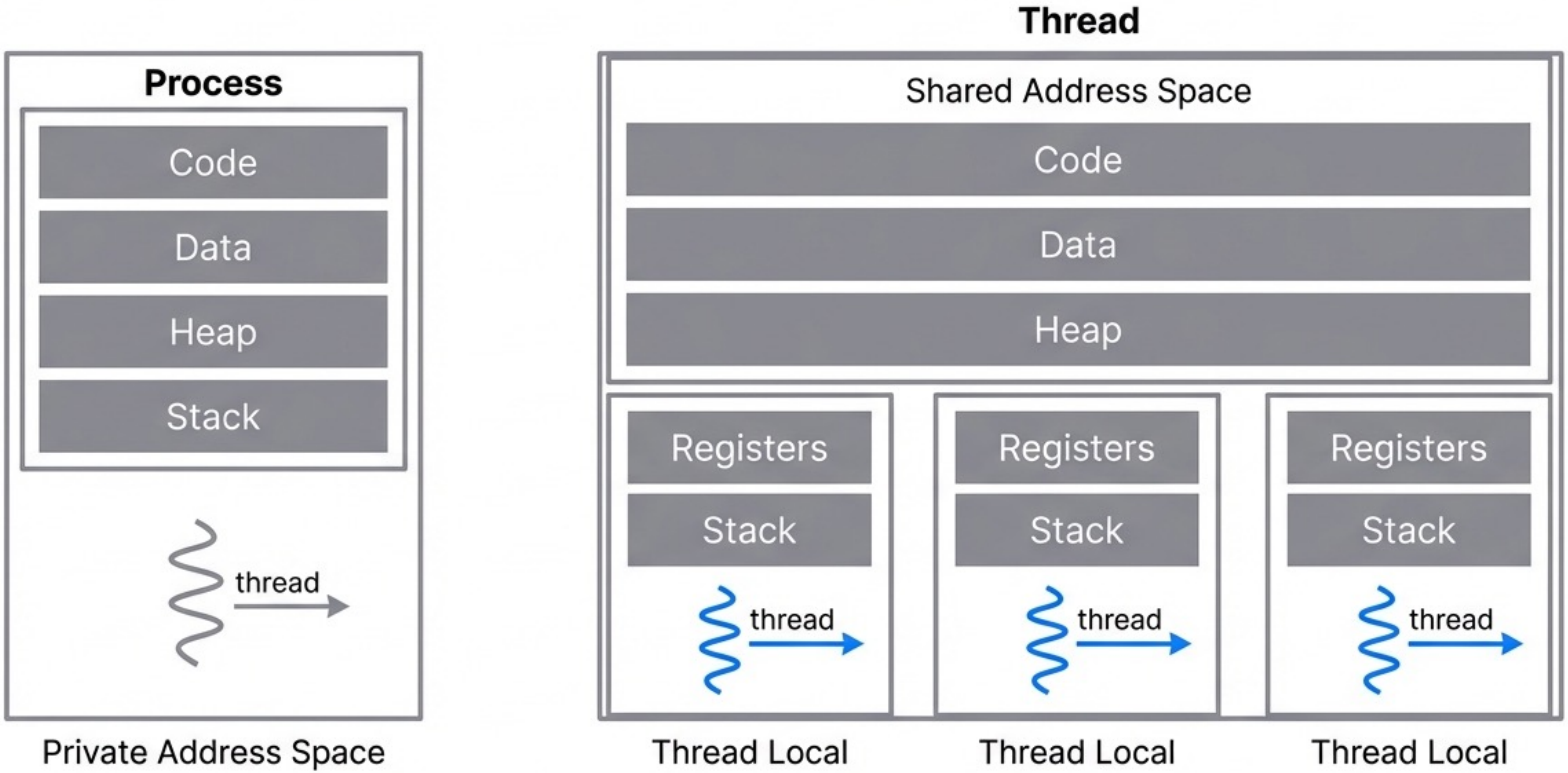

```
pthread_create(&thread, NULL,
  function, arg);
```

```
pthread_join(thread, &status);
```

```
pthread_exit(NULL);
```

The Shared Memory Model

Process vs. Thread



THE PHANTOM BUG: WHEN 50,000 DOES NOT EQUAL 50,000

THE EXPECTATION

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int counter = 0;

void *compute()
{
        int i = 0;
        while (i < 10000) {
                counter = counter + 1;
                i++;
        }
        printf("Counter value: %d\n", counter);
}

int main()
{
        pthread_t thread1, thread2, thread3, thread4, thread5;

        pthread_create(&thread1, NULL, compute, (void *)&thread1);
        pthread_create(&thread2, NULL, compute, (void *)&thread2);
        pthread_create(&thread3, NULL, compute, (void *)&thread3);
        pthread_create(&thread4, NULL, compute, (void *)&thread4);
        pthread_create(&thread5, NULL, compute, (void *)&thread5);

        pthread_exit(NULL);
        exit(0);
}
```

Expected Result: **50,000**

THE REALITY

```
pi@raspberrypi ~/Downloads> ./race_condition.o
Counter value: 14467
Counter value: 10410
Counter value: 12080
Counter value: 22745
Counter value: 32725
Counter value: 32,725
```

A race condition occurs when concurrent threads access shared data simultaneously. The order of access creates unpredictable results.

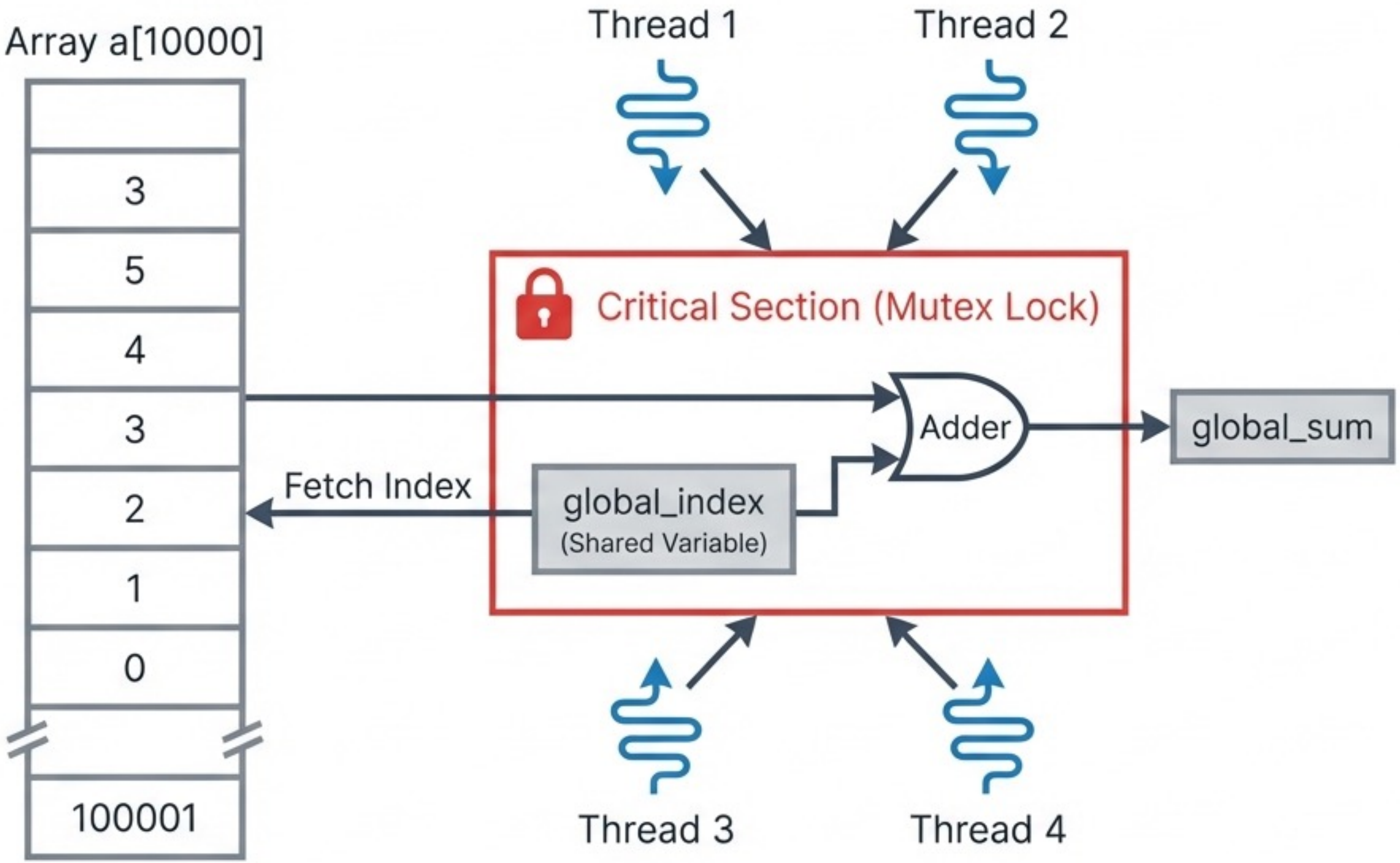
Diagnosing the Root Cause: The Critical Section

```
while (i < 100) {
    counter = counter + 1;
    i++;
}
```

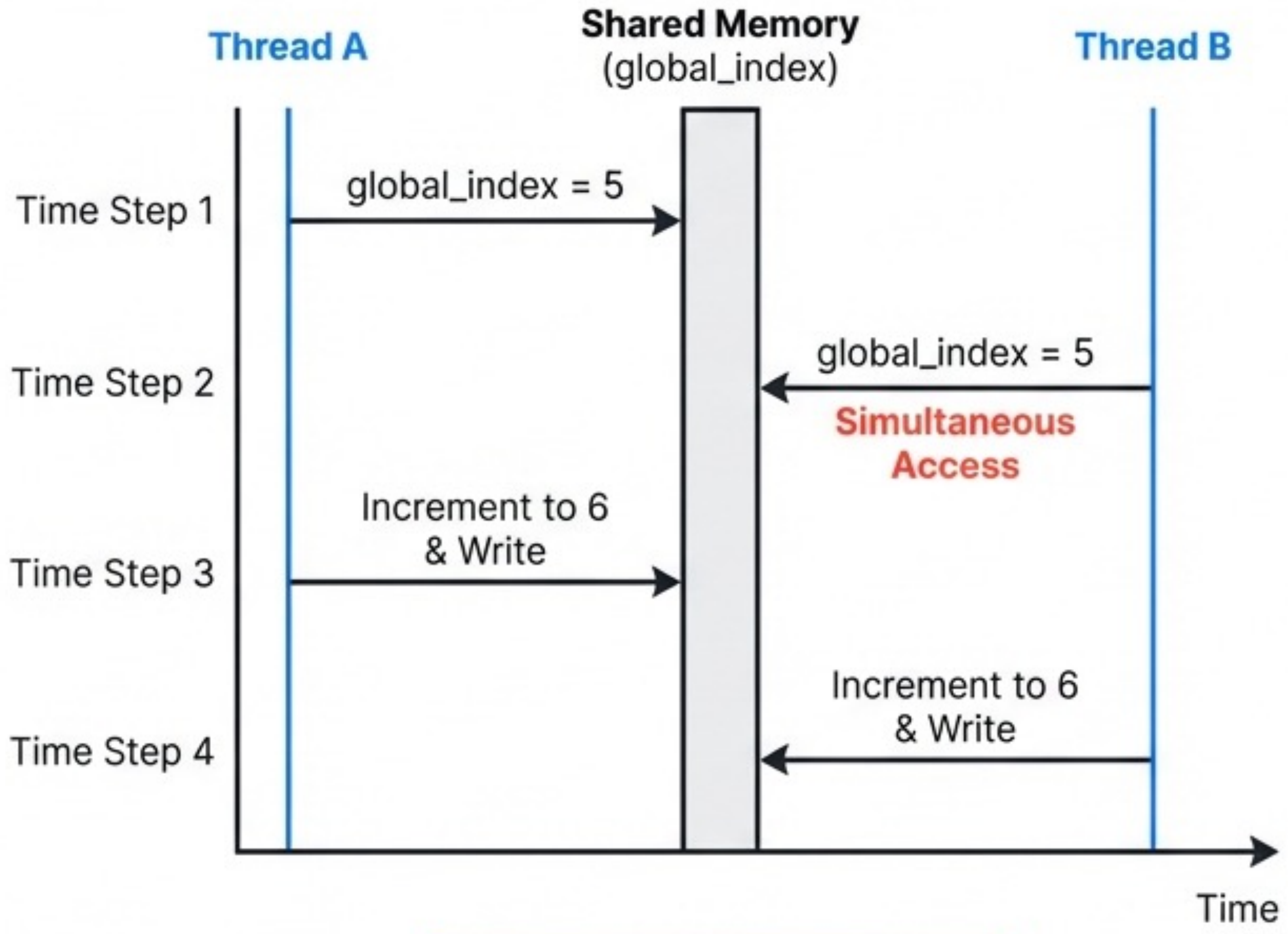
The Danger Zone

Critical Section: A section of code where a thread accesses a shared resource. If not protected, this is where the race condition lives.

Task 1: The Naive Architecture



The Danger: Race Conditions

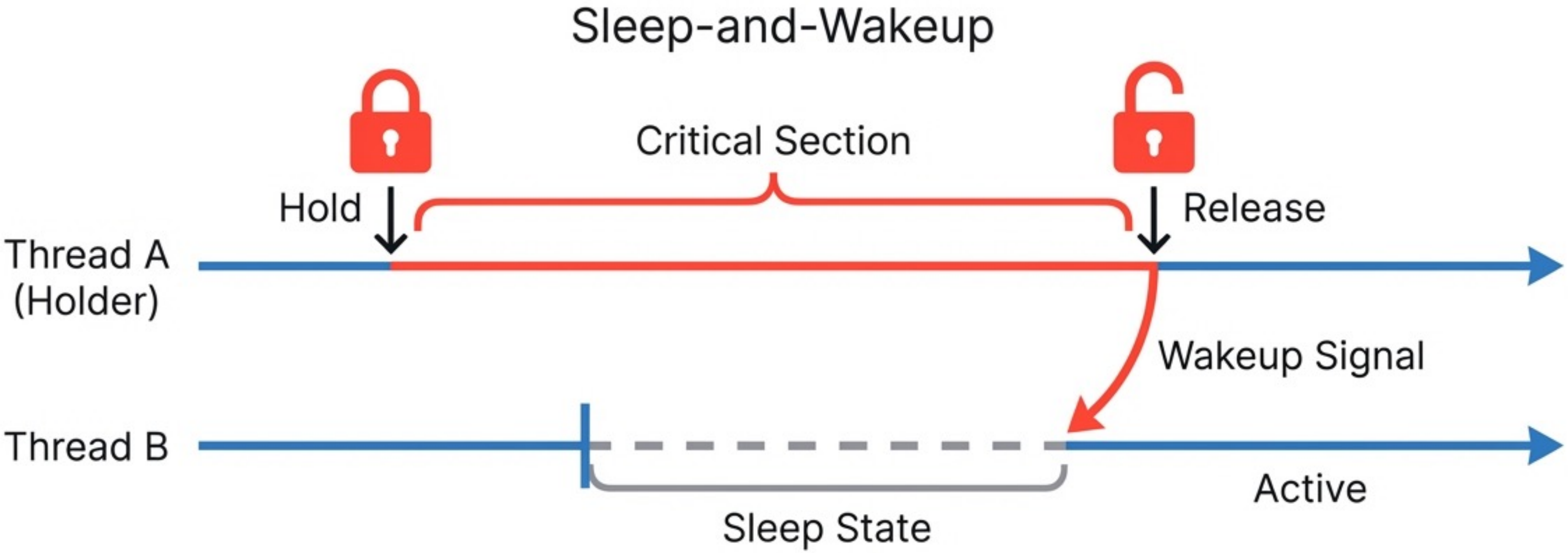


Result: One increment is lost.
Final Value: 6 (Expected: 7)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int counter = 0;

void *compute()
{
    int i = 0;
    while (i < 100) {
        counter = counter + 1;
        i++;
    }
    printf("Counter value: %d\n", counter);
}
```

Unprotected Shared Access

The Solution: Mutex Locks



Task 1 Implementation Flow

```
while (global_index < 10000) {
    pthread_mutex_lock(&lock);
        idx = global_index;
        global_index++;
    pthread_mutex_unlock(&lock);
    val = 2 * idx;
    // Add val to global_sum (requires lock)
}
```

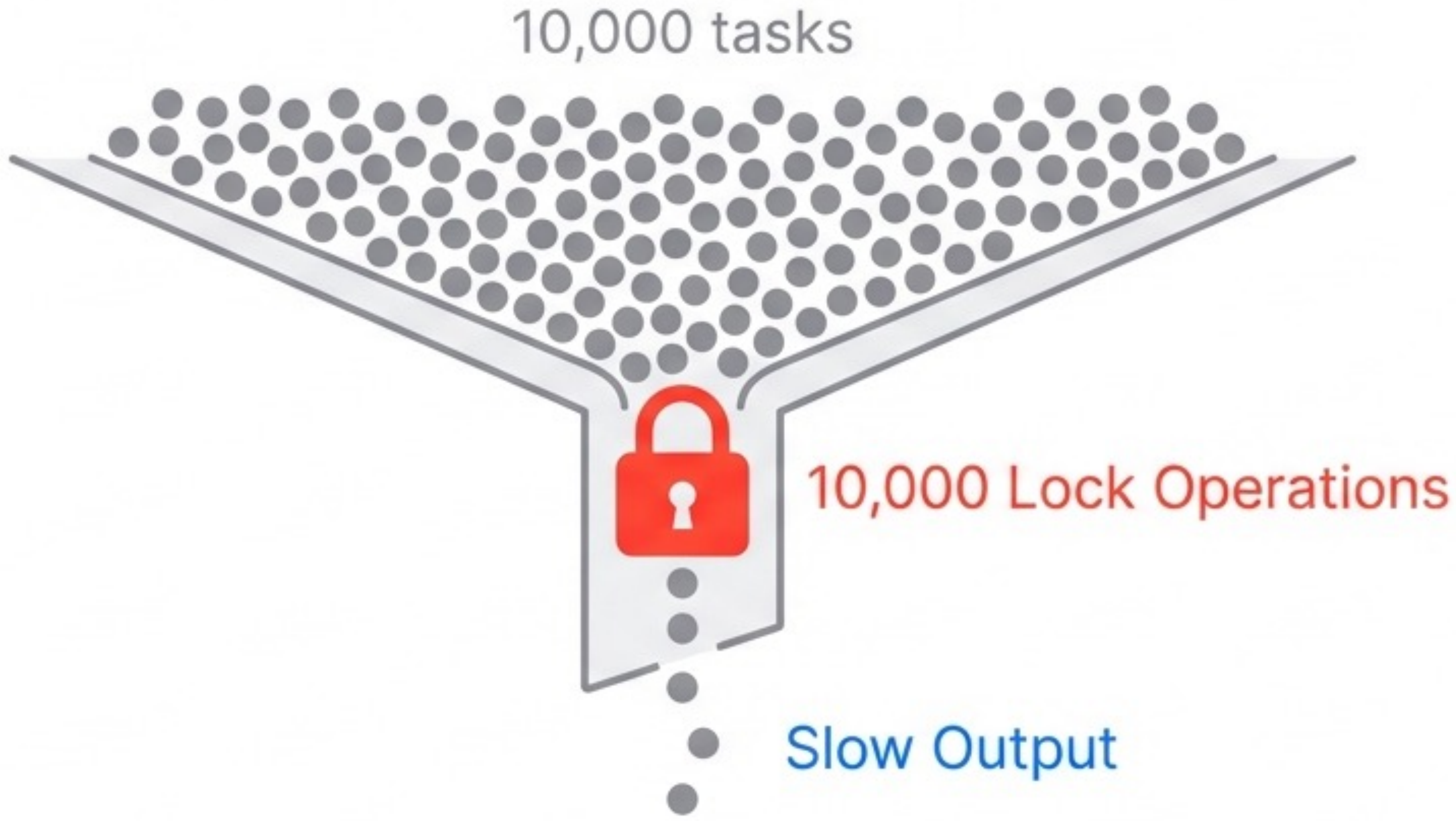
Serialization Point

Critical Section (Very Short)

Serialization Point

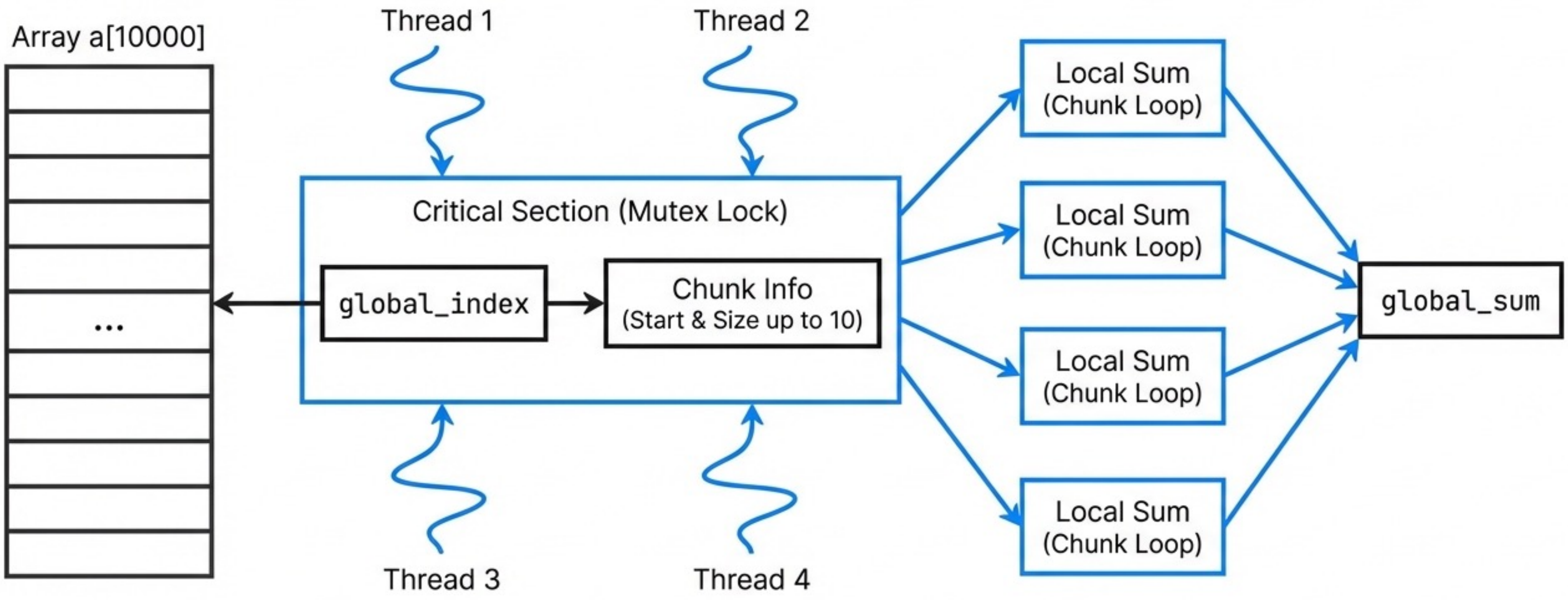
Parallel Execution

The Bottleneck: High Contention



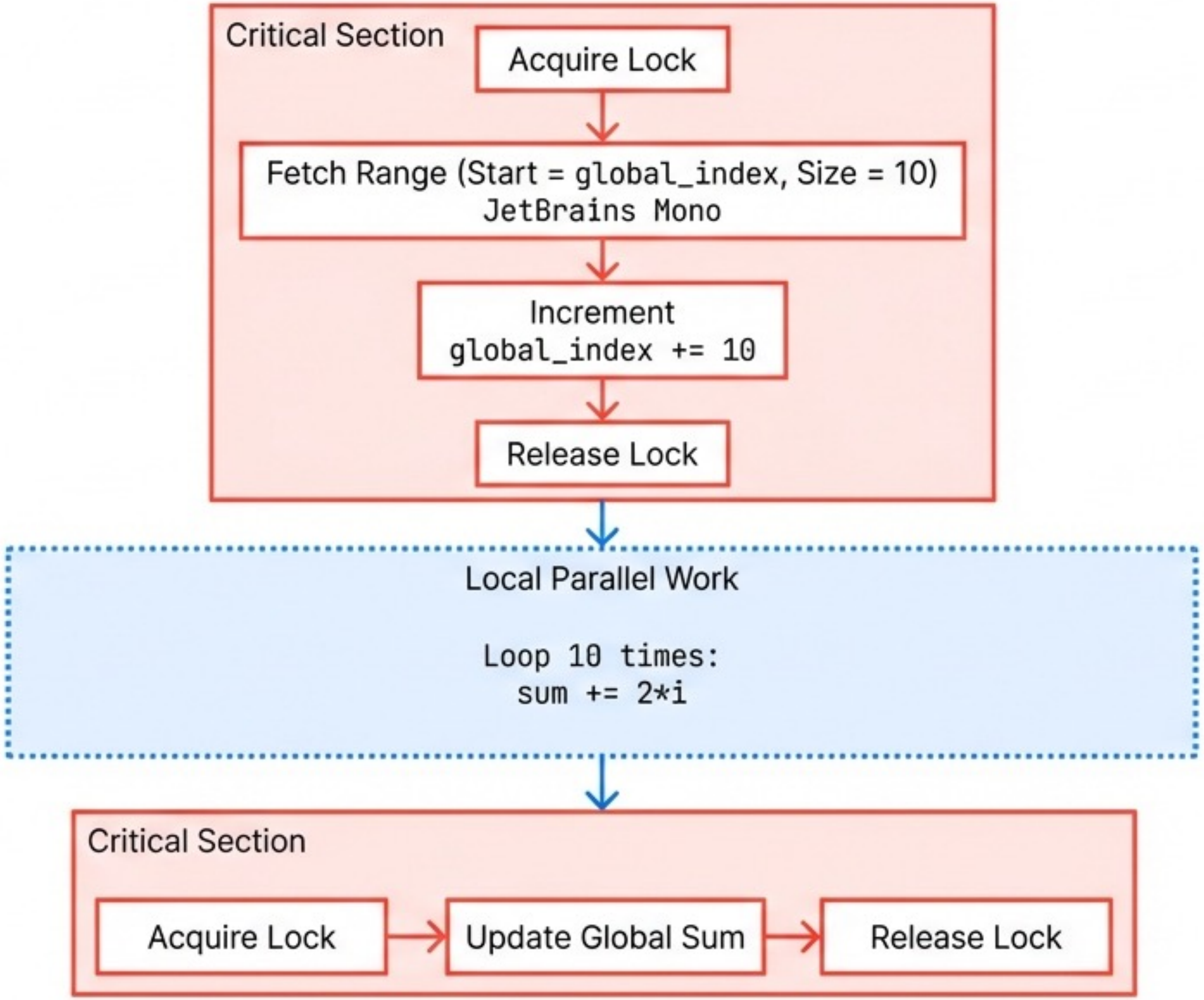
Threads spend more time fighting for the lock than computing.
The ratio of synchronization overhead to actual work is too high.

Task 2 Strategy: Chunking

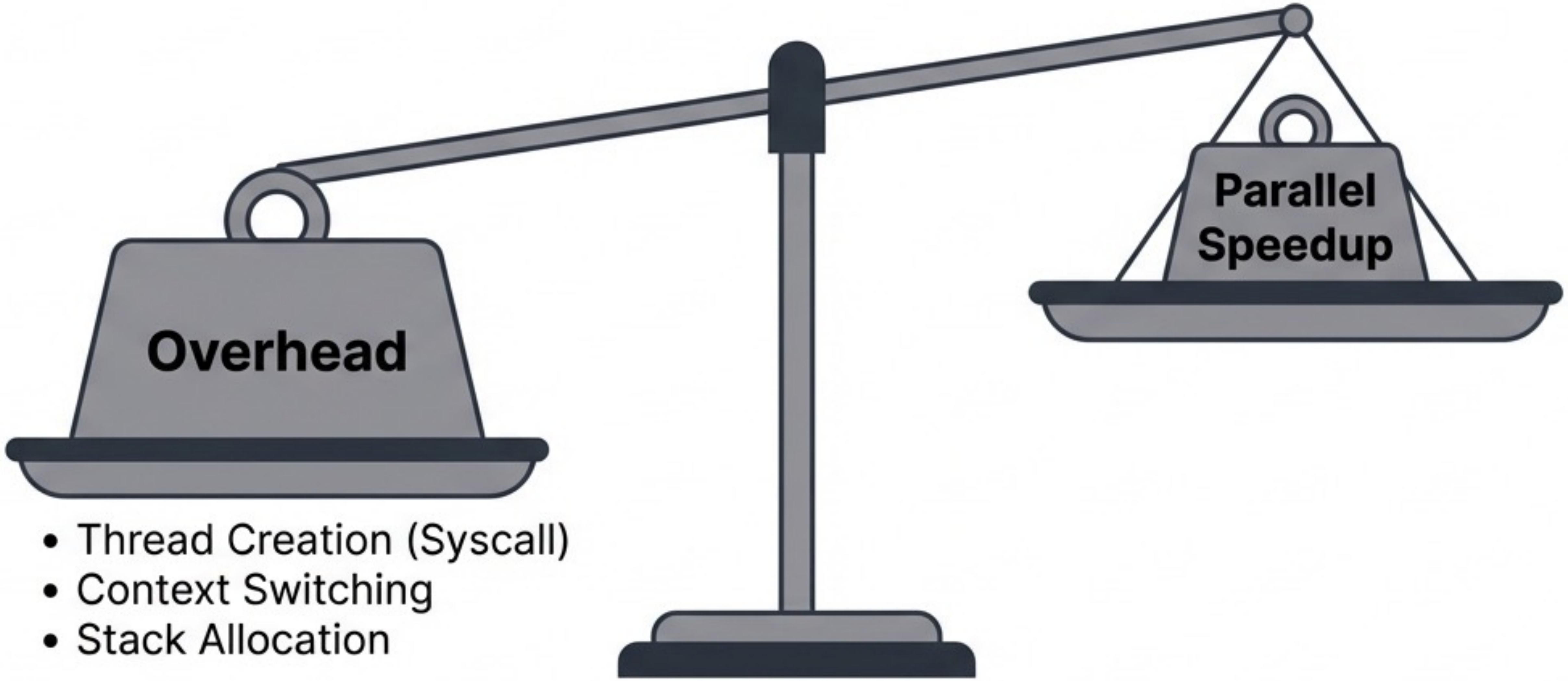


Lock Frequency Reduced by 10x

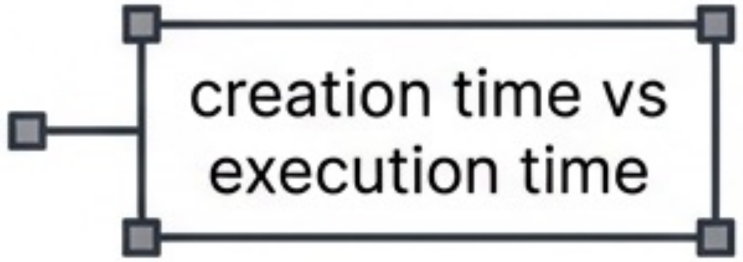
Task 2 Implementation Logic



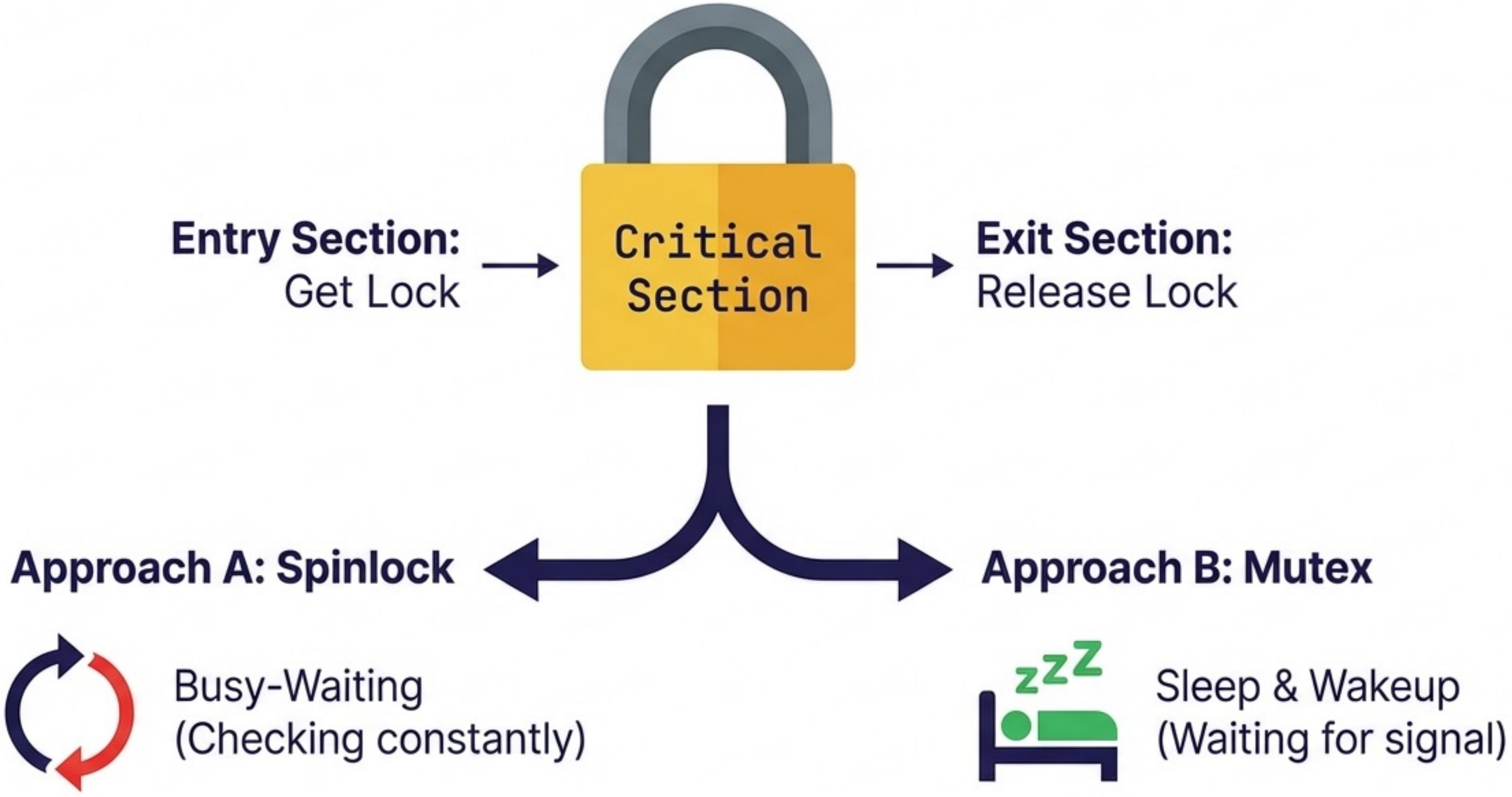
Task 3 Analysis: The Cost of Concurrency



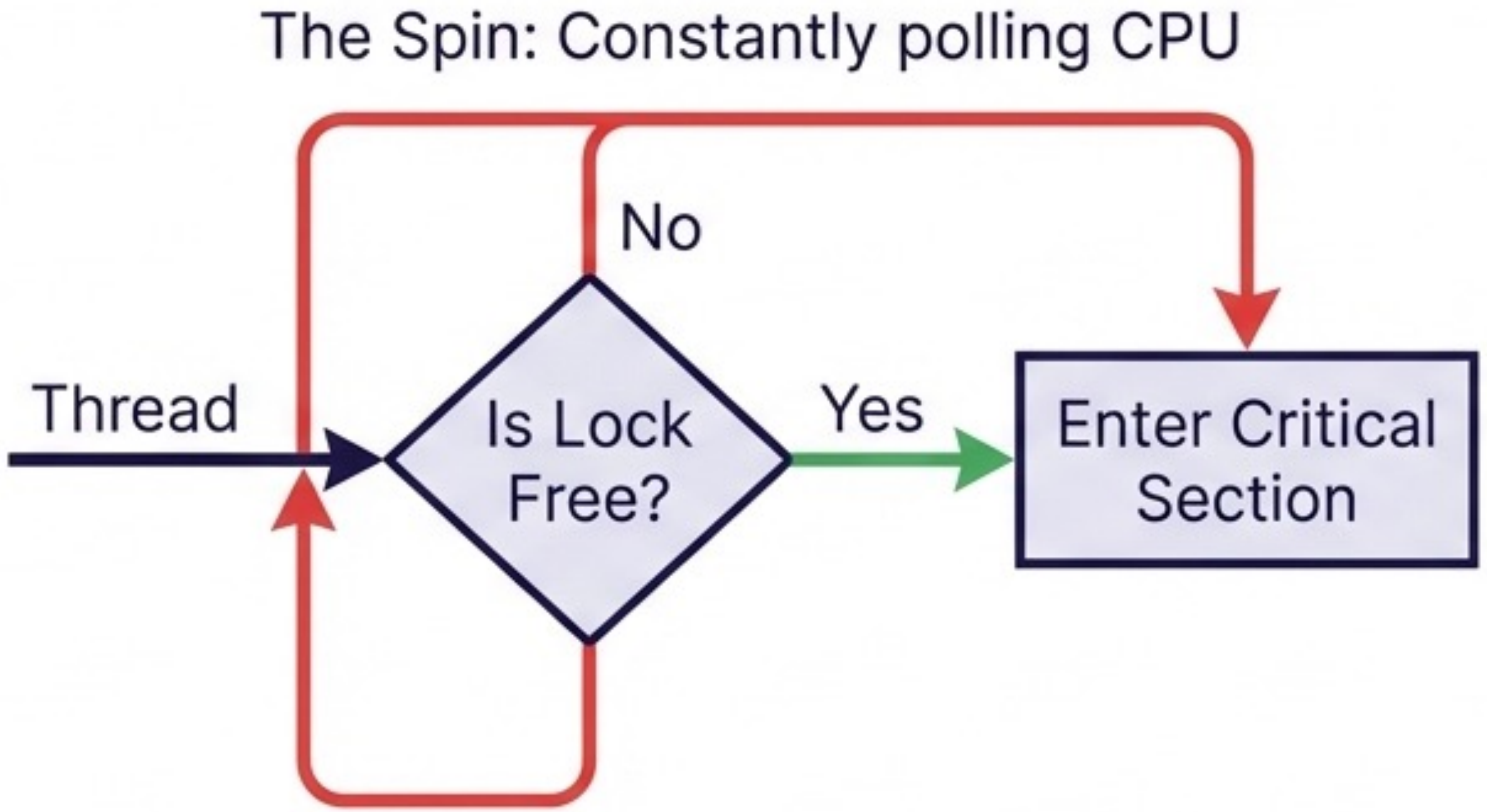
For very small workloads, the overhead of managing threads exceeds the computation time saved.



Restoring Order: Mutual Exclusion (Locks)



Strategy 1: The Spinlock (Busy-Waiting)

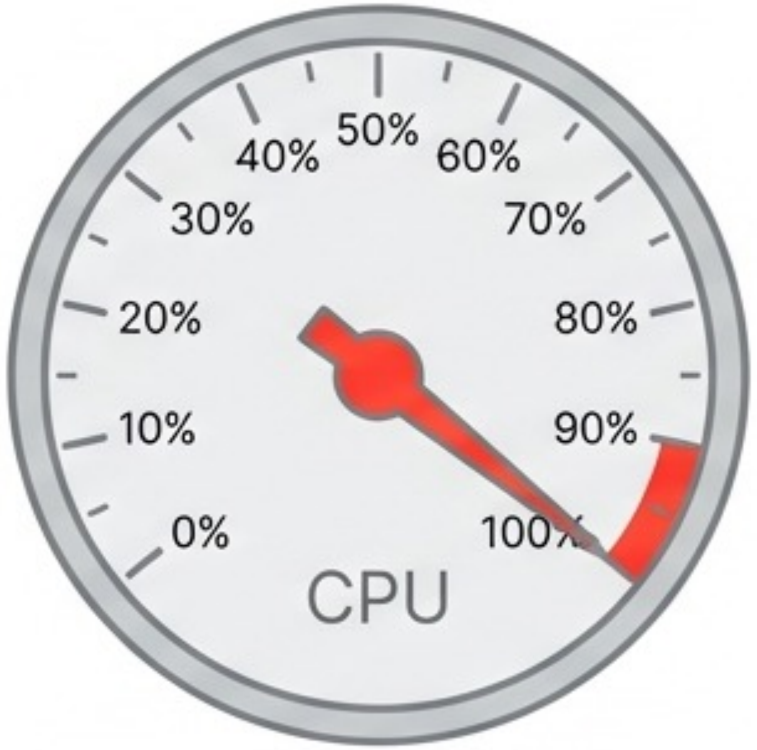



```
pthread_spin_lock(&splock);

// Critical Section
counter = counter + 1;
pthread_spin_unlock(&splock);
```

Mechanism: The thread never sleeps. It remains active on the CPU, repeatedly asking 'Are you ready?' until the lock opens.

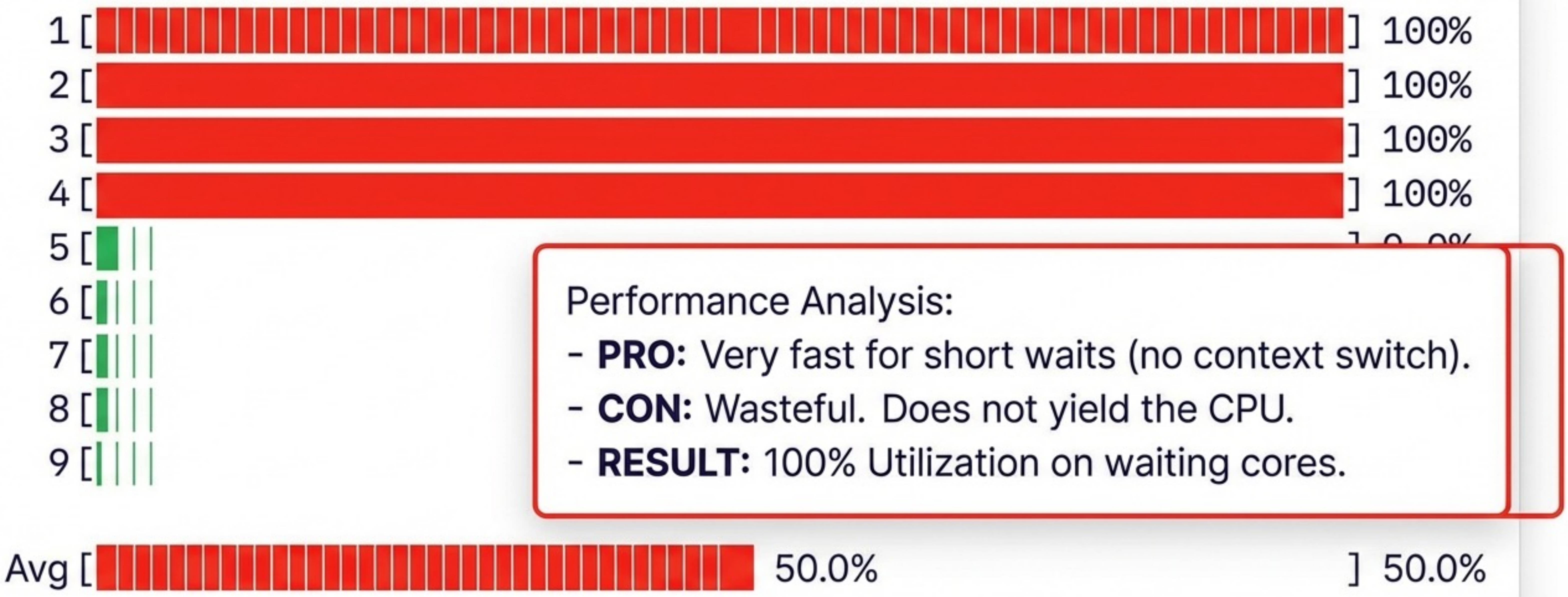
Deep Dive: Spinlocks

```
do {
   // Entry section: Check and acquire lock
   while (atomic_flag_test_and_set(&lock_flag)) {
      // Spin-wait: Busy-waiting
   }

   // Critical Section
   critical_section_code();

   // Exit section: Release lock
   atomic_flag_clear(&lock_flag);

} while (TRUE);
```



Efficiency Paradox: efficient only if the wait time is shorter than the time it takes to **context switch**. Ideal for the increment operations in Task 1.

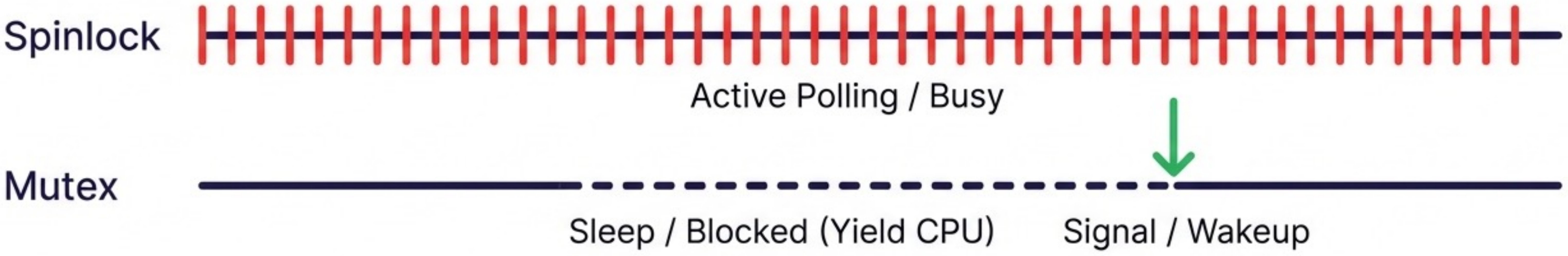
The Cost of Spinning: Burning CPU Cycles



Performance Analysis:

- **PRO:** Very fast for short waits (no context switch).
- **CON:** Wasteful. Does not yield the CPU.
- **RESULT:** 100% Utilization on waiting cores.

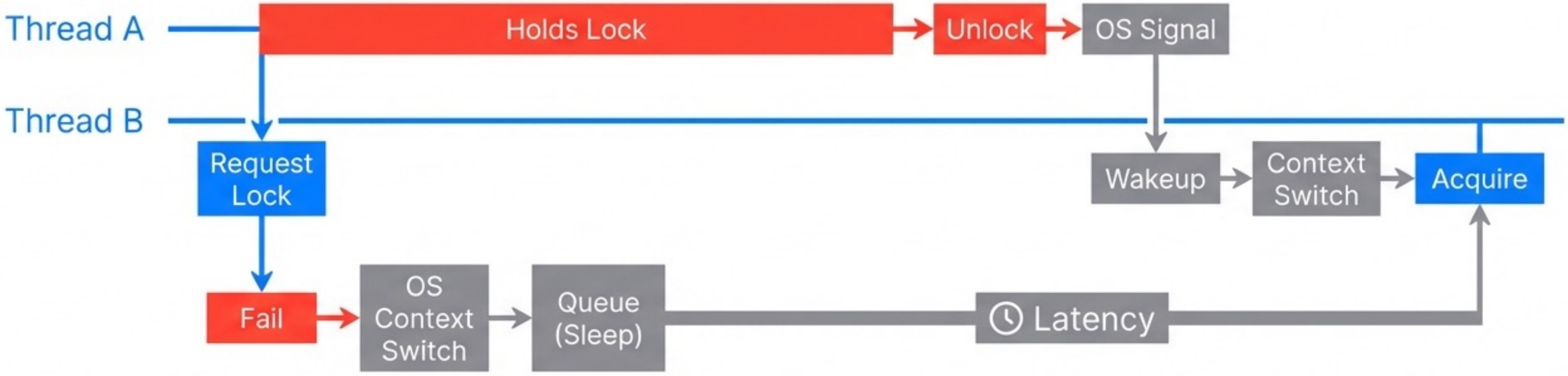
Strategy 2: The Mutex (Sleep & Wakeup)




```
Lock(mutex);
// Critical Section
Unlock(mutex);
```

A Mutex (Mutual Exclusion) allows the thread to sleep if the lock is **unavailable.** The OS wakes it up **only when** the resource is free.

Deep Dive: Mutex



The 'Sleep-and-Wakeup' cycle introduces significant OS overhead. If the Critical Section is just `i++`, the thread spends more time sleeping and waking than working.

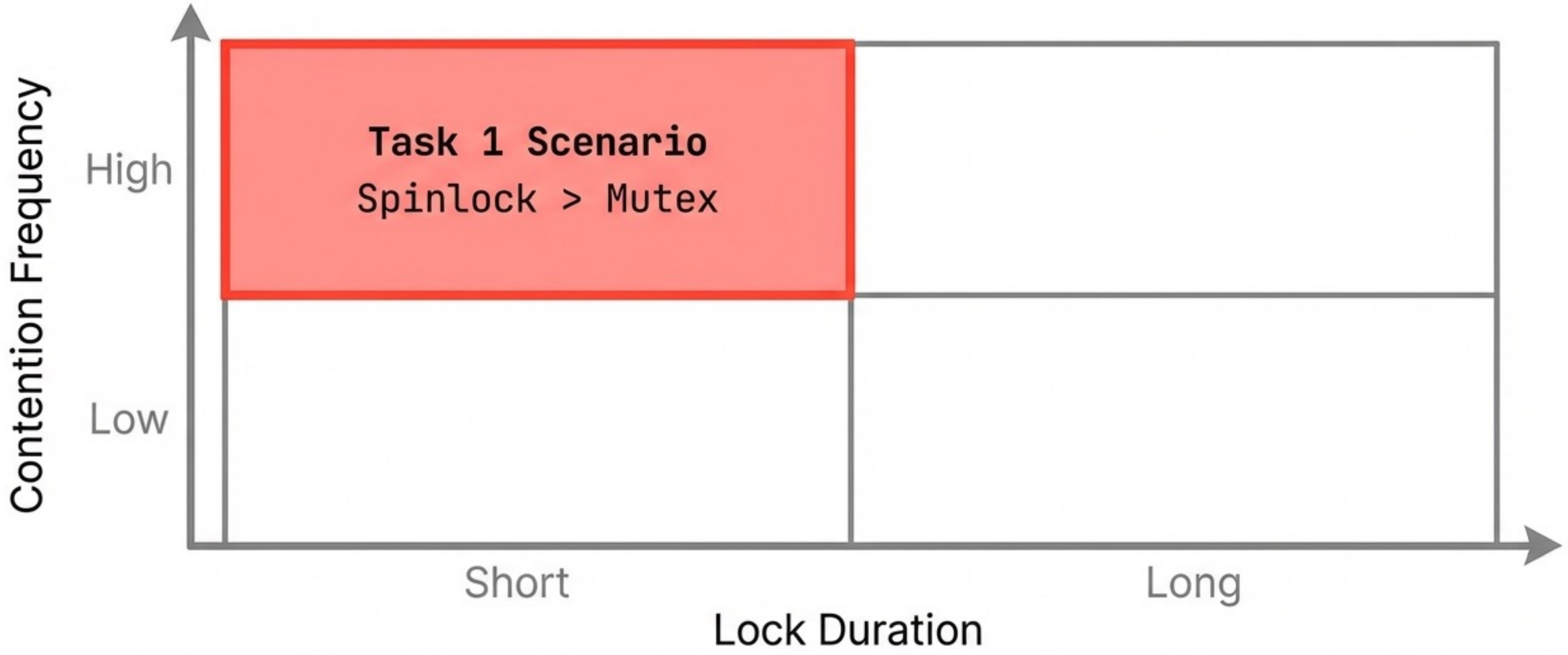
Locking Mechanisms: Spinlock vs. Mutex

Spinlock (Busy-Wait)	Mutex (Sleep-Wake)

Behavior
`do { check lock } while (locked)`
`} while (locked)`
	Behavior
	If locked -> Sleep -> Scheduler -> Wake

Pros No context switch, fast for short waits.	**Pros** Yields CPU to other tasks.
Cons Burns CPU cycles.	**Cons** High latency overhead.

High Contention Analysis



Because the integer addition is instant, putting a thread to sleep (Mutex) is wasteful. Spinning is preferred here.

Summary

Threads	Global Sum	Run Time (s)
1	10000000000	1.25
2	10000000000	0.85
3	10000000000	0.65
4	10000000000	0.55
5	10000000000	0.50
6	10000000000	0.48
7	10000000000	0.46
8	10000000000	0.45
9	10000000000	0.44
10	10000000000	0.44

- ☐ Correct Global Sum (Formula Validation)
- ☐ Task 1 (Naive) Performance Data
- ☐ Task 2 (Chunking) Performance Data
- ☐ Written Analysis (Single vs Multi, Spin vs Mutex)