Computer Systems

Lecture 14: Performance Considerations in Multi-Threaded Programs

Huaicheng Li

March 25, 2026



CS 3214
Spring 2026

Recap: What We Know About Concurrency So Far

From Previous Lectures

- **Threads**: Shared address space, lightweight processes, pthreads API
- **Locking**: Mutexes protect critical sections from data races
- **Semaphores**: Counting-based coordination (signaling, bounded buffer)
- **Condition Variables**: Wait-until-condition patterns (producer-consumer)
- **Deadlock**: Circular wait, Coffman conditions, lock ordering

We know *how* to synchronize threads. But at *what cost*?

Every mutex lock/unlock, every atomic operation, every cache-line bounce has a performance price. Today we learn to measure, predict, and minimize that price.

THE HOOK: The Lock That Killed Performance

A True Story from Database Engineering

The Scenario

A database system with 64 cores processes 100,000 transactions/second.
After adding a "simple" mutex to protect a counter, throughput drops to 8,000 transactions/second.

THE HOOK: The Lock That Killed Performance

A True Story from Database Engineering

The Scenario

A database system with 64 cores processes 100,000 transactions/second.
After adding a "simple" mutex to protect a counter, throughput drops to 8,000 transactions/second.

12.5× slowdown from a single lock protecting one integer!

THE HOOK: The Lock That Killed Performance

A True Story from Database Engineering

The Scenario

A database system with 64 cores processes 100,000 transactions/second.
After adding a "simple" mutex to protect a counter, throughput drops to 8,000 transactions/second.

12.5× slowdown from a single lock protecting one integer!

- Why do locks hurt performance so dramatically?
- How do we measure and predict locking costs?
- What strategies minimize lock overhead?

Performance Considerations

Upfront Note – Correctness First!

Correctness cannot be traded for performance. No one cares about the performance of code that contains data races, atomicity violations, ordering violations, or is prone to deadlocks.

Performance Considerations

Upfront Note – Correctness First!

Correctness cannot be traded for performance. No one cares about the performance of code that contains data races, atomicity violations, ordering violations, or is prone to deadlocks.

- That said, let's examine the cost of locking in particular
- **Indirect cost**: Loss of performance due to serialization
  - Threads waiting instead of doing useful work
  - Reduces parallelism, increases latency
- **Direct cost**: Overhead of the locking mechanism itself
  - CPU cycles for atomic operations
  - System calls for blocking/waking
  - Cache invalidation traffic

Understanding Indirect Cost: Simulation Setup

Simulation Parameters

- 5 CPU-bound threads contending for L locks
- Each thread: holds lock for duration D, then runs for duration U without lock
- Thread chooses which lock randomly

Color Key:

- Light green: Running without lock
- Other colors: Holding a lock
- White gaps: Blocked waiting for lock

Percentage = CPU Utilization

- 100% = Perfect parallelism
- Lower = More serialization

The ratio $U : D$ (time outside lock vs inside lock) determines contention

Indirect Cost: Single Lock – High Contention

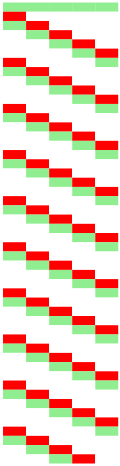


Figure: Fixed: U=2/D=2/L=1/40.8%

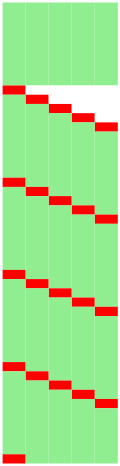


Figure: Fixed: U=18/D=2/L=1/96%



Figure: Poisson: U=2/D=2/L=1/41.6%

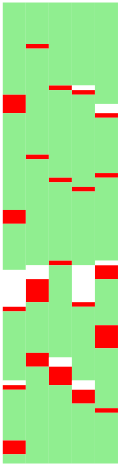


Figure: Poisson U=18/D=2/L=1/94.4%

Indirect Cost: Two Locks – Reduced Contention

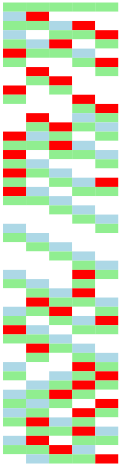


Figure: Fixed: U=2/D=2/L=2/65.2%

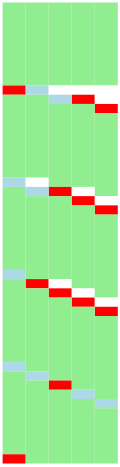


Figure: Fixed: U=18/D=2/L=2/96%

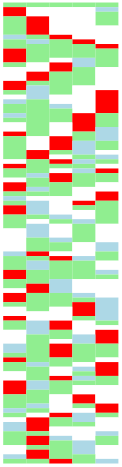


Figure: Poisson: U=2/D=2/L=2/72.2%

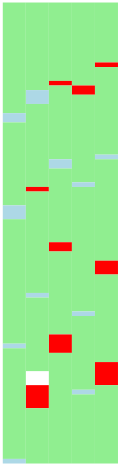


Figure: Poisson U=18/D=2/L=2/99.4%

Indirect Cost: Four Locks – Even Better



Figure: Fixed: U=2/D=2/L=4/89.6%

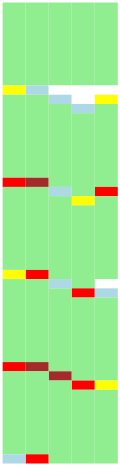


Figure: Fixed: U=18/D=2/L=4/98%

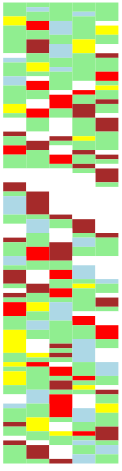


Figure: Poisson: U=2/D=2/L=4/79.2%

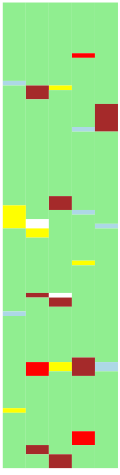


Figure: Poisson U=18/D=2/L=4/99.4%

Indirect Cost: Key Insights

- Serialization due to locks diminishes CPU utilization and increases latency
  - For parallel, CPU-bound applications: directly reduces speedup
  - Particularly bad when locks are **contended** (threads frequently blocked)
  - Even worse if nothing else to run during blocking

Indirect Cost: Key Insights

- Serialization due to locks diminishes CPU utilization and increases latency
  - For parallel, CPU-bound applications: directly reduces speedup
  - Particularly bad when locks are **contended** (threads frequently blocked)
  - Even worse if nothing else to run during blocking

Critical sections should **never** call functions that may block:

- I/O operations (read, write, network)
- `sleep()`, `sem_wait()`, `pthread_join()`

Otherwise, critical section becomes inaccessible to other threads!

Don't Block While Holding a Lock – Examples

BAD: Blocking I/O in Critical Section

```
pthread_mutex_lock(&lock);
// Critical section starts
update_shared_data();
// This can BLOCK for seconds!
write(log_fd, buffer, len);
pthread_mutex_unlock(&lock);
```

GOOD: I/O Outside Critical Section

```
pthread_mutex_lock(&lock);
update_shared_data();
// Copy data we need to log
memcpy(local_buf, buffer, len);
pthread_mutex_unlock(&lock);
// I/O outside critical section
write(log_fd, local_buf, len);
```

The Pattern

(1) Enter critical section (2) Do minimal work, copy what you need (3) Exit critical section
(4) Do slow operations (I/O, network) outside

Amdahl's Law and Lock Contention

Amdahl's Law

If fraction f of work must be serial, maximum speedup with n processors:

$$\text{Speedup} = \frac{1}{f + \frac{1-f}{n}}$$

Amdahl's Law and Lock Contention

Amdahl's Law

If fraction f of work must be serial, maximum speedup with n processors:

$$\text{Speedup} = \frac{1}{f + \frac{1-f}{n}}$$

Cores	Max Speedup	Efficiency
4	3.48×	87%
16	9.14×	57%
64	16.0×	25%
256	18.6×	7%

Amdahl's Law and Lock Contention

Amdahl's Law

If fraction f of work must be serial, maximum speedup with n processors:

$$\text{Speedup} = \frac{1}{f + \frac{1-f}{n}}$$

Cores	Max Speedup	Efficiency
4	3.48×	87%
16	9.14×	57%
64	16.0×	25%
256	18.6×	7%

Even 5% serial work limits 64-core speedup to 16× (not 64×)!

Solution: Breaking Up Locks (Fine-Grained Locking)

Historical Context

Several large systems started with "big lock" approaches:

- Linux kernel (Big Kernel Lock until 2011)
- Python's GIL (Global Interpreter Lock) – still exists!
- GTK GUI lock

Solution: Breaking Up Locks (Fine-Grained Locking)

Historical Context

Several large systems started with "big lock" approaches:

- Linux kernel (Big Kernel Lock until 2011)
- Python's GIL (Global Interpreter Lock) – still exists!
- GTK GUI lock

Fine-Grained Locking Strategy

Instead of lock L protecting data (A, B, C):
Introduce locks L_A, L_B, L_C to protect A, B, C separately.

Benefit: Updates to A don't block updates to B

Fine-Grained Locking: The Risks

1. **Higher risk of atomicity violations**
   - If A and B must be updated atomically
   - Must hold *both* locks – negates benefit of splitting
   - Forgetting to hold both $\rightarrow$ atomicity violation

Fine-Grained Locking: The Risks

1. **Higher risk of atomicity violations**
   - If A and B must be updated atomically
   - Must hold *both* locks – negates benefit of splitting
   - Forgetting to hold both $\rightarrow$ atomicity violation
2. **Higher risk of deadlocks**
   - More locks $\rightarrow$ more lock ordering constraints
   - Must establish and enforce consistent ordering
   - Easy to make mistakes as code grows

Fine-Grained Locking: The Risks

1. **Higher risk of atomicity violations**
   - If A and B must be updated atomically
   - Must hold *both* locks – negates benefit of splitting
   - Forgetting to hold both $\rightarrow$ atomicity violation
2. **Higher risk of deadlocks**
   - More locks $\rightarrow$ more lock ordering constraints
   - Must establish and enforce consistent ordering
   - Easy to make mistakes as code grows
3. **Increased direct cost (locking overhead)**
   - More lock/unlock calls
   - More memory for lock objects
   - More cache coherence traffic

Fine-Grained Locking: The Risks

1. **Higher risk of atomicity violations**
   - If A and B must be updated atomically
   - Must hold *both* locks – negates benefit of splitting
   - Forgetting to hold both $\rightarrow$ atomicity violation
2. **Higher risk of deadlocks**
   - More locks $\rightarrow$ more lock ordering constraints
   - Must establish and enforce consistent ordering
   - Easy to make mistakes as code grows
3. **Increased direct cost (locking overhead)**
   - More lock/unlock calls
   - More memory for lock objects
   - More cache coherence traffic

Fine-grained locking reduces **indirect** cost (serialization)
but increases **direct** cost (overhead) and complexity

Example: Coarse vs Fine-Grained Locking

Coarse-Grained (Simple, More Contention)

```c
struct database {
    pthread_mutex_t lock;
    struct user *users;
    struct order *orders;
    struct product *products;
};

void add_user(struct database *db,
              struct user *u) {
    pthread_mutex_lock(&db->lock);
    // Blocks ALL database ops
    insert_user(db->users, u);
    pthread_mutex_unlock(&db->lock);
}
```

Fine-Grained (Complex, Less Contention)

```c
struct database {
    pthread_mutex_t users_lock;
    pthread_mutex_t orders_lock;
    pthread_mutex_t products_lock;
    struct user *users;
    struct order *orders;
    struct product *products;
};
void add_user(struct database *db,
              struct user *u) {
    pthread_mutex_lock(&db->users_lock);
    // Only blocks user ops
    insert_user(db->users, u);
    pthread_mutex_unlock(&db->users_lock);
}
```

Direct Cost of Locking: What Happens Inside

`pthread_mutex_lock()` Under the Hood

Fast Path Atomic instruction tries to acquire lock (if available)

- E.g., `cmpxchg %rax, (%rbx)` – compare-and-swap
- No system call, no context switch
- Cost: $\sim$17ns (about 4$\times$ L2 cache access)

Slow Path If lock is held by another thread:

- System call: `futex(FUTEX_WAIT)`
- Thread blocks, context switch to other thread
- Cost: 1000s of nanoseconds (mode switch + context switch)

Direct Cost: Unlock and Wake

`pthread_mutex_unlock()` Under the Hood

Fast Path No one waiting: just set lock to unlocked state

- Single atomic write
- Cost: $\sim$10ns

Slow Path Someone is waiting:

- System call: `futex(FUTEX_WAKE)`
- Kernel wakes blocked thread(s)
- Woken threads added to ready queue
- Eventually scheduled – another context switch

Direct Cost: Unlock and Wake

`pthread_mutex_unlock()` Under the Hood

Fast Path No one waiting: just set lock to unlocked state

- Single atomic write
- Cost: $\sim$10ns

Slow Path Someone is waiting:

- System call: `futex(FUTEX_WAKE)`
- Kernel wakes blocked thread(s)
- Woken threads added to ready queue
- Eventually scheduled – another context switch

Why context switches are expensive: Pipeline flush, TLB flush, cache pollution (new thread has different working set), branch predictor pollution

Numbers Every Programmer Should Know

Latency Comparison (circa 2020)

Operation	Time
L1 cache reference	1 ns
L2 cache reference	4 ns
Mutex lock/unlock (uncontended)	**17 ns**
Main memory reference	100 ns
Context switch	**1,000-10,000 ns**
System call	**500-1,000 ns**
SSD random read	16,000 ns
Network round trip (same datacenter)	500,000 ns

Source: Jeff Dean / Peter Norvig / Colin Scott
`https://colin-scott.github.io/personal_website/research/interactive_latency.html`

Spinlocks vs Mutexes

Spinlock

```
void spin_lock(int *lock) {
    while (atomic_exchange(lock, 1) == 1)
        ; // Busy wait (spin)
}
void spin_unlock(int *lock) {
    atomic_store(lock, 0);
}
```

Pro: No context switch

Con: Wastes CPU while spinning

Mutex (pthread_mutex)

```
void mutex_lock(mutex_t *m) {
    if (!try_lock(m))
        futex_wait(m); // Block
}
void mutex_unlock(mutex_t *m) {
    unlock(m);
    if (has_waiters(m))
        futex_wake(m);
}
```

Pro: CPU free for other work

Con: Context switch overhead

Spinlocks vs Mutexes

Spinlock

```
1  void spin_lock(int *lock) {
2      while (atomic_exchange(lock, 1) == 1)
3          ; // Busy wait (spin)
4  }
5  void spin_unlock(int *lock) {
6      atomic_store(lock, 0);
7  }
```

Pro: No context switch

Con: Wastes CPU while spinning

Mutex (pthread_mutex)

```
1  void mutex_lock(mutex_t *m) {
2      if (!try_lock(m))
3          futex_wait(m); // Block
4  }
5  void mutex_unlock(mutex_t *m) {
6      unlock(m);
7      if (has_waiters(m))
8          futex_wake(m);
9  }
```

Pro: CPU free for other work

Con: Context switch overhead

Spinlock: Very short critical sections briefly, then block

Mutex: Longer sections, oversubscribed

Adaptive: Spin

Lock Contention Profiling

How to Find Lock Bottlenecks

```
# Linux perf to find lock contention
perf lock record ./my_program
perf lock report

# Valgrind DRD for lock statistics
valgrind --tool=drd --exclusive-threshold=100 ./my_program

# Mutex contention in gdb
(gdb) info threads  # See which threads are blocked
```

Lock Contention Profiling

How to Find Lock Bottlenecks

```
# Linux perf to find lock contention
perf lock record ./my_program
perf lock report

# Valgrind DRD for lock statistics
valgrind --tool=drd --exclusive-threshold=100 ./my_program

# Mutex contention in gdb
(gdb) info threads  # See which threads are blocked
```

What to Look For

Wait time: How long threads spend blocked **Hold time**: How long critical sections take
Acquisition count: How often each lock is taken **Contention rate**: % of acquisitions that block

Optimization Strategies Summary

1. **Reduce critical section duration**
   - Move non-critical work outside the lock
   - Pre-compute values before acquiring lock
   - Copy data out, release lock, then process

Optimization Strategies Summary

1. **Reduce critical section duration**
   - Move non-critical work outside the lock
   - Pre-compute values before acquiring lock
   - Copy data out, release lock, then process
2. **Reduce lock granularity**
   - Split one lock into multiple independent locks
   - Per-bucket locks for hash tables
   - Per-element locks for linked lists (careful!)

Optimization Strategies Summary

1. **Reduce critical section duration**
   - Move non-critical work outside the lock
   - Pre-compute values before acquiring lock
   - Copy data out, release lock, then process
2. **Reduce lock granularity**
   - Split one lock into multiple independent locks
   - Per-bucket locks for hash tables
   - Per-element locks for linked lists (careful!)
3. **Use lock-free data structures**
   - Atomic operations instead of locks
   - More complex but avoids blocking entirely
   - Good for simple structures (counters, queues)

Optimization Strategies Summary

1. **Reduce critical section duration**
   - Move non-critical work outside the lock
   - Pre-compute values before acquiring lock
   - Copy data out, release lock, then process
2. **Reduce lock granularity**
   - Split one lock into multiple independent locks
   - Per-bucket locks for hash tables
   - Per-element locks for linked lists (careful!)
3. **Use lock-free data structures**
   - Atomic operations instead of locks
   - More complex but avoids blocking entirely
   - Good for simple structures (counters, queues)
4. **Use read-write locks**
   - `pthread_rwlock_t`
   - Multiple readers OR single writer
   - Good when reads dominate writes

Read-Write Locks

When Reads Dominate

```c
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
void reader(void) {
    pthread_rwlock_rdlock(&rwlock);  // Multiple readers OK
    int val = read_shared_data();
    pthread_rwlock_unlock(&rwlock);
}
void writer(void) {
    pthread_rwlock_wrlock(&rwlock);  // Exclusive access
    write_shared_data(new_val);
    pthread_rwlock_unlock(&rwlock);
}
```

Trade-offs

Pro: Readers don't block each other **Con**: Writer starvation possible; more overhead than simple mutex

Best for: Read-heavy workloads (90%+ reads)

Thread-Local Storage: Eliminating Sharing

The Ultimate Lock Optimization: Don't Share at All

Scenario: a function needs to know which worker thread is calling it. Rather than looking up a shared table (requiring locks), give each thread its own private copy of the variable.

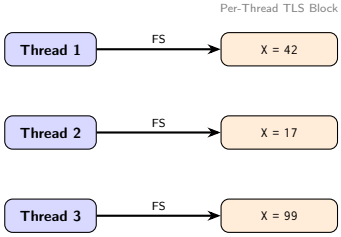
```c
_Thread_local struct worker *current_worker = NULL;  // C11
// __thread struct worker *current_worker = NULL;    // legacy GCC

void worker_start(void *arg) {
    current_worker = (struct worker *)arg;
    // ... all code in this thread sees this worker
}

struct worker *get_current_worker(void) {
    return current_worker;  // NULL if not a worker thread
}
```

No locks needed – each thread has its own copy. No data races, no contention.

TLS Under the Hood



Compiler-Generated Access via FS Segment Register

```
1  movl %fs:x@tpoff, %eax    # load thread-local x
2  addl $1, %eax
3  movl %eax, %fs:x@tpoff    # store back
```

Efficient: Access cost ≈ global variable. Each thread's FS register points to a per-thread storage area allocated at thread creation.

When to Use Thread-Local Storage

Use Cases

1. **Per-thread worker identity** (P2 threadpool: which worker am I?)
2. **Per-thread statistics** – count locally, aggregate at end (avoids false sharing!)
3. `errno` – the C library uses TLS so each thread has its own `errno`
4. **Per-thread caches/buffers** – avoid lock contention on shared pool

Limitations

- Only for global/static variables (not heap-allocated)
- Must be initialized per-thread (default: zero/NULL)
- Memory overhead: N threads $\times$ `sizeof(variable)`

Connection to false sharing fix: instead of `partial_sum[id]++`, use `_Thread_local long my_sum;` then write to the shared array once at the end.

TLS: POSIX Dynamic API and Java

Dynamic TLS: pthread_key_* API

For libraries that need TLS without compile-time _Thread_local declarations:

```c
static pthread_key_t color_key;
void init(void) { pthread_key_create(&color_key, free); }
void *worker(void *arg) {
    char *color = strdup((char *)arg);
    pthread_setspecific(color_key, color);    // set per-thread value
    char *my_color = pthread_getspecific(color_key);  // retrieve it
    printf("I am %s\n", my_color);
    return NULL;
}
```

Java Equivalent: ThreadLocal<T>

```
ThreadLocal<String> color = new ThreadLocal<>();   color.set("red");   color.get();
```

When to use which: _Thread_local for compile-time (faster, simpler); pthread_key_* for dynamic/library use cases.

Atomicity Violations: The Insidious Bug

Definition

A concurrency bug that can occur **even if proper locking discipline is followed**. Each individual lock/unlock is correct – but the *combination* is not atomic.

The Pattern

1. `Lock(S)` – acquire lock on shared state S
2. Read information I from S
3. `Unlock(S)` – release lock
4. ... commit to action based on I ...
5. `Lock(S')` – acquire lock again (same or different)
6. Act on S' using **stale** information I
7. `Unlock(S')` – release lock

Race detectors (TSan, Helgrind) will NOT catch this – locking discipline is correct!

Atomicity Violation: String Copy Example

```
1  char *p = ...;  /* shared string, protected by lp */
2
3  int getplen() {
4      pthread_mutex_lock(&lp);
5      int len = strlen(p);
6      pthread_mutex_unlock(&lp);
7      return len;
8  }
9
10 // Thread A:
11 int nchars = getplen();          // lock, strlen, unlock -- safe by itself
12 char *copy = malloc(nchars + 1); // commit to size based on stale info
13 pthread_mutex_lock(&lp);
14 strcpy(copy, p);                 // BUG: p may have changed! buffer overflow
15 pthread_mutex_unlock(&lp);
```

Between `getplen()` and `strcpy()`, another thread may reallocate `p` to point to a longer string → **buffer overflow**.

Found in JDK's `StringBuffer.append()` – Flanagan and Freund (POPL 2004)

Atomicity Violation: P2 Threadpool Race

Worker Thread

```
// Step 1: get task from queue
pthread_mutex_lock(&queue_lock);
task = dequeue(queue);
pthread_mutex_unlock(&queue_lock);

// Step 2: mark as running
pthread_mutex_lock(&task->lock);
task->state = RUNNING;
// ... execute task ...
pthread_mutex_unlock(&task->lock);
```

future_get() Thread

```
// Step 1: check task state
pthread_mutex_lock(&task->lock);
if (task->state == NEW) {
    // "I'll run it myself"
    pthread_mutex_lock(&queue_lock);
    remove_from_queue(queue, task);
    pthread_mutex_unlock(&queue_lock);
    task->state = RUNNING;
    execute(task);  // BUG!
}
pthread_mutex_unlock(&task->lock);
```

Task executed twice! State change + queue removal must be **one atomic operation**. Worker dequeued the task, but `future_get()` saw `state == NEW` before the worker set `RUNNING`.

Fixing Atomicity Violations

Three Strategies

1. **Hold lock across entire operation** – simplest, more contention

Fixing Atomicity Violations

Three Strategies

1. **Hold lock across entire operation** – simplest, more contention
2. **Recheck after acquiring second lock** – optimistic; abort and retry if state changed

Fixing Atomicity Violations

Three Strategies

1. **Hold lock across entire operation** – simplest, more contention
2. **Recheck after acquiring second lock** – optimistic; abort and retry if state changed
3. **Use trylock for opposite lock order** – avoids deadlock (see below)

Normal order

```
1 A.lock();
2 ... find B;
3 B.lock();
4 // holding A & B
```

Opposite order (safe)

```
1 retry:
2   B.lock();
3   ... find A;
4   if (!A.trylock())
5     { B.unlock(); goto retry; }
6   // holding A & B
```

EXAM FOCUS: Atomicity Violations

Identify the atomicity violation in this cache implementation:

```
struct cache_entry *lookup_or_create(struct cache *c, const char *key) {
    pthread_mutex_lock(&c->lock);
    struct cache_entry *e = find_entry(c, key);
    pthread_mutex_unlock(&c->lock);
    if (e != NULL) return e;    // found it
    e = create_entry(key);      // expensive allocation + init
    pthread_mutex_lock(&c->lock);
    insert_entry(c, key, e);    // BUG: another thread may have inserted same key!
    pthread_mutex_unlock(&c->lock);
    return e;
}
```

EXAM FOCUS: Atomicity Violations

Identify the atomicity violation in this cache implementation:

```
struct cache_entry *lookup_or_create(struct cache *c, const char *key) {
    pthread_mutex_lock(&c->lock);
    struct cache_entry *e = find_entry(c, key);
    pthread_mutex_unlock(&c->lock);
    if (e != NULL) return e;    // found it
    e = create_entry(key);      // expensive allocation + init
    pthread_mutex_lock(&c->lock);
    insert_entry(c, key, e);    // BUG: another thread may have inserted same key!
    pthread_mutex_unlock(&c->lock);
    return e;
}
```

Atomicity violation: Between lookup (miss) and insert, another thread can insert the same key → duplicate entries. **Fix**: Recheck after acquiring lock: `if (find_entry(c, key)) { free(e); }`

EXAM FOCUS: Lock Performance

Common Exam Questions

1. **Explain the difference between direct and indirect locking costs**
   - Direct: overhead of lock operations themselves
   - Indirect: loss of parallelism due to serialization
2. **Why shouldn't you block while holding a lock?**
   - Other threads waiting for lock also blocked
   - Can cause cascading delays or effective deadlock
3. **What are the trade-offs of fine-grained locking?**
   - Pro: Less contention, better parallelism
   - Con: More complexity, deadlock risk, overhead
4. **When would you use a spinlock vs mutex?**
   - Spinlock: very short critical sections
   - Mutex: longer sections, need to yield CPU

Conclusion: The Art of Lock Optimization

Key Takeaways

1. **Correctness first** – never sacrifice it for performance
2. **Measure before optimizing** – profile to find real bottlenecks
3. **Start simple** – coarse-grained locking is easier to get right
4. **Optimize incrementally** – fine-grained only where needed
5. **Know your costs** – both direct overhead and indirect serialization

Conclusion: The Art of Lock Optimization

Key Takeaways

1. **Correctness first** – never sacrifice it for performance
2. **Measure before optimizing** – profile to find real bottlenecks
3. **Start simple** – coarse-grained locking is easier to get right
4. **Optimize incrementally** – fine-grained only where needed
5. **Know your costs** – both direct overhead and indirect serialization

1. Get correct program with coarse-grained locks
2. Profile to identify lock bottlenecks
3. Reduce critical section size first
4. Consider finer granularity if still bottlenecked
5. Consider lock-free only for proven hot spots

Connection to Your Projects

Threadpool (P2)

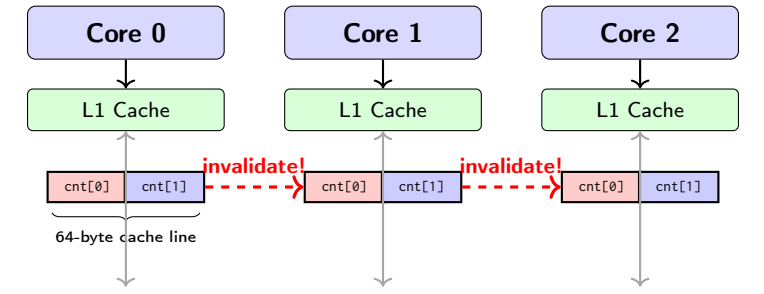
- Your thread pool manages a work queue with locks
- Balance between: single queue lock vs per-worker queues
- Work stealing needs careful locking strategy

Personal Server (P4)

- Shared state between request handlers
- Connection pools, session caches need synchronization
- Read-write locks good for session lookup

Start with simple, correct locking. Optimize only after measuring!

False Sharing: The Invisible Bottleneck



Core 0 writes `cnt[0]` → invalidates `cnt[1]` in Core 1's cache – **even though they are different variables!**

False Sharing: Code Example

BAD: False Sharing

```
// Adjacent = same cache line (64B)
struct counters {
    long count[NUM_THREADS];
};
struct counters c;
void *worker(void *arg) {
    int id = (int)(long)arg;
    for (int i = 0; i < N; i++)
        c.count[id]++;  // BOUNCE!
    return NULL;
}
```

GOOD: Padded to Separate Lines

```
struct padded_counter {
    long count;
    char pad[56]; // 64 - 8 = 56
};
struct padded_counter c[NUM_THREADS];
void *worker(void *arg) {
    int id = (int)(long)arg;
    for (int i = 0; i < N; i++)
        c[id].count++;  // No bounce
    return NULL;
}
```

False sharing can cause 10-100× slowdown with zero logical data races! Demo: `demos/12-atomics/falsesharing.c` vs `demos/12-atomics/truesharing.c`

Cache Coherence: Why False Sharing Hurts

The MESI Protocol (Simplified)

Each cache line is in one of four states:

Modified (M) – Only valid copy (dirty)

Exclusive (E) – Only copy (clean)

Shared (S) – Multiple caches have valid copies (read-only)

Invalid (I) – This cache's copy is stale

What Happens on False Sharing

1. Core 0 writes `cnt[0]` $\rightarrow$ line goes to **M** in Core 0
2. Core 1 writes `cnt[1]` $\rightarrow$ must fetch line from Core 0's cache
3. Core 0's copy goes to **I**, Core 1's goes to **M**
4. Core 0 writes again $\rightarrow$ fetch back from Core 1
5. **Ping-pong!** Each write costs $\sim$100ns instead of $\sim$1ns

Benchmarking Multi-Threaded Code

Common Pitfalls

1. **Not warming up caches** – first iteration is always slow
2. **Measuring wall time instead of CPU time** – includes scheduling noise
3. **Compiler optimizing away work** – use `volatile` or check results
4. **Not controlling for frequency scaling** – CPU may throttle under load
5. **Ignoring variance** – run multiple trials, report median + std dev

Measuring Speedup

```
# Measure with varying thread counts
for n in 1 2 4 8 16; do
    echo -n "threads=$n: "
    ./benchmark --threads=$n --trials=10
done
# Report: speedup = T(1) / T(n)
```

EXAM FOCUS: Performance Analysis

Given this code, identify the performance problem:

```c
struct shared_state {
    pthread_mutex_t lock;
    int total_requests;    // written by all threads
    int active_threads;    // written by all threads
    char log_buffer[4096]; // rarely used
};
void handle_request(struct shared_state *s) {
    pthread_mutex_lock(&s->lock);
    s->total_requests++;
    process_request();     // Takes 50ms!
    update_log(s->log_buffer);
    pthread_mutex_unlock(&s->lock);
}
```

EXAM FOCUS: Performance Analysis

Given this code, identify the performance problem:

```c
struct shared_state {
    pthread_mutex_t lock;
    int total_requests;    // written by all threads
    int active_threads;    // written by all threads
    char log_buffer[4096]; // rarely used
};
void handle_request(struct shared_state *s) {
    pthread_mutex_lock(&s->lock);
    s->total_requests++;
    process_request();     // Takes 50ms!
    update_log(s->log_buffer);
    pthread_mutex_unlock(&s->lock);
}
```

Three Problems: (1) `process_request()` takes 50ms inside critical section – all other threads blocked! (2) `total_requests` and `log_buffer` don't need same lock. **Fix**: Move `process_request()` outside lock; split into separate locks.

EXAM FOCUS: False Sharing Identification

Why does this parallel sum run slower with more threads?

```c
long partial_sum[MAX_THREADS]; // Global array

void *sum_worker(void *arg) {
    int id = (int)(long)arg;
    for (long i = id * chunk; i < (id+1) * chunk; i++)
        partial_sum[id] += data[i];  // Each thread writes its own slot
    return NULL;
}
```

No locks, no data races. So what's wrong?

EXAM FOCUS: False Sharing Identification

Why does this parallel sum run slower with more threads?

```c
long partial_sum[MAX_THREADS]; // Global array

void *sum_worker(void *arg) {
    int id = (int)(long)arg;
    for (long i = id * chunk; i < (id+1) * chunk; i++)
        partial_sum[id] += data[i];  // Each thread writes its own slot
    return NULL;
}
```

No locks, no data races. So what's wrong?

`partial_sum[0]`, `partial_sum[1]`, etc. are adjacent `long`s (8 bytes each). Multiple fit in one 64-byte cache line. Each thread's write invalidates other threads' cache lines.

Fix: Pad each element to 64 bytes, or use thread-local variable and write to array only at the end.

EXAM FOCUS: Key Concepts

You should be able to:

1. Explain **direct vs indirect** costs of locking
2. Apply **Amdahl's Law** to predict speedup limits
3. Identify **false sharing** in code and fix it with padding
4. Explain why **blocking while holding a lock** is catastrophic
5. Compare **coarse vs fine-grained** locking trade-offs
6. Explain **spinlock vs mutex** trade-offs
7. Describe the **MESI protocol** and cache-line bouncing
8. Know when **read-write locks** are beneficial (90%+ reads)
9. Identify **atomicity violations** in code (lock-release-relock patterns)
10. Know when to use **thread-local storage** vs locks vs atomics

Performance questions on the exam will often give you code and ask you to identify the bottleneck and propose a fix. Practice reading code with "performance glasses" on.

Summary

Costs of Locking

- Direct: atomic ops, syscalls
- Indirect: serialization
- Amdahl's Law limits speedup
- Never block in critical section

Optimization

- Minimize critical section
- Fine-grained locking
- Lock-free where proven
- Read-write locks
- Atomicity violations

Hidden Costs

- False sharing
- Cache-line bouncing
- MESI invalidation
- Measure, don't guess

Further Reading

- OSTEP Ch. 29: Lock-based Concurrent Data Structures
- CS:APP Chapter 12 (Concurrent Programming)
- Ulrich Drepper, "What Every Programmer Should Know About Memory" (2007)
- `man perf-lock`, `man pthread_rwlock_init`

References

- Computer Systems: A Programmer's Perspective, 3rd Ed. – Bryant & O'Hallaron
- Operating Systems: Three Easy Pieces – Arpaci-Dusseau
- The Linux Programming Interface – Michael Kerrisk
- Advanced Programming in the UNIX Environment – Stevens & Rago