Computer Systems

Lecture 12: Managing Shared State

Huaicheng Li

March 2, 2026



CS 3214
Spring 2026

Code: Shared Counter, Is it Correct?

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

int global_shared;              // shared between threads!

static void *thread_func(void *_tn) {
    for (int i = 0; i < 1000000; i++)
        global_shared++;        // data race here
    return NULL;
}

int main() {
    pthread_t t[2];
    pthread_create(&t[0], NULL, thread_func, NULL);
    pthread_create(&t[1], NULL, thread_func, NULL);
    pthread_join(t[0], NULL);
    pthread_join(t[1], NULL);
    printf("global_shared = %d\n", global_shared);
}
```

The Lost Updates Mystery

Two threads each increment a shared counter 1 million times.

Expected final value: 2,000,000

Actual value: 1,372,485 (or some other random number)

The Lost Updates Mystery

Two threads each increment a shared counter 1 million times.

Expected final value: 2,000,000

Actual value: 1,372,485 (or some other random number)

Question: Where did 627,515 updates go?

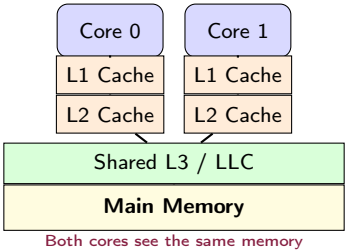
Answer: They were lost to *data races*.
Solving this is the heart of concurrent programming.

Today's Questions

1. **What are data races?**
   Why does `counter++` not work correctly?
2. **How do we fix them?**
   Mutual exclusion, critical sections, locks
3. **How do we use locks in C?**
   `pthread_mutex` API
4. **What can go wrong with locks?**
   Deadlock and how to avoid it
5. **What about performance?**
   Lock granularity, contention, alternatives

Introduction

- Multi-threaded programs share a virtual address space – they can directly access all objects: global variables, heap objects, etc.
- The non-determinism introduced by preemptive scheduling and/or simultaneous execution on multiple cores makes such sharing challenging
- Motivating example: `counter++`



Typical shared memory multicore architecture [3]

Key Point

Non-determinism means the same program can produce different results on each run – bugs may appear rarely and be hard to reproduce.

Data Races

Definition

Two or more threads access a shared variable, at least one access is a **write**, and the final result depends on the order of execution – specifically the order of their memory accesses.

- Such data races are one form of concurrency-related bugs (aka "race conditions")
- Can be intermittent and difficult to find ("Heisenbugs")
- Others include: ordering violations, atomicity violations
- Will discuss the following strategies:
  - Avoidance by duplication & partitioning
  - Use of mutual exclusion devices, e.g., locks
  - Use of atomic instructions

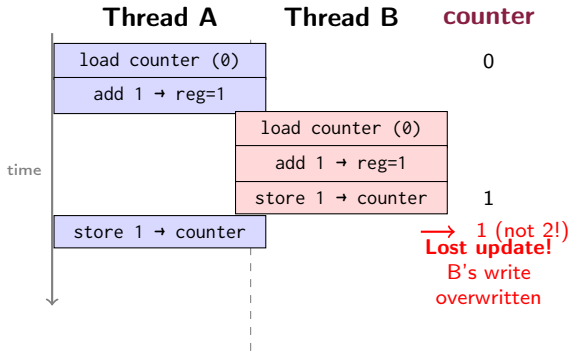
Data Race Example: counter++

```c
int counter = 0;  // Shared

void *thread_func(void *arg) {
    for (int i = 0; i < 1000000; i++)
        counter++;
    return NULL;
}
```

counter++ compiles to:

1. mov counter to register
2. add 1 to register
3. mov register to counter



The Problem

Between load and store, another thread can read/write counter – lost updates!

Avoiding Sharing

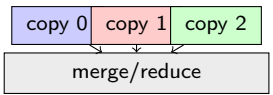
- Data races cannot occur where no data is shared
- Strategies to reformulate the problem:
  - **Partitioning:** separate data into disjoint parts operated on by different threads
  - **Duplicating:** provide separate copies that threads independently modify
- Many scalability breakthroughs came from redesigning data structures to avoid or reduce sharing:
  - Per-thread counters
  - Per-CPU ready queues
  - Region-based memory allocators
- However, these strategies incur management complexity and require additional space

Data Partitioning



No overlap ⇒ no races

Duplication



Ad-hoc Strategies

Many myths about what is safe and what isn't in data race free programs:

- Myth 1: "I'm just reading, not updating"
- Myth 2: "By cleverly arranging read/write order I can avoid a data race"
- Myth 3: "I can avoid races with `volatile`"

Is there a data race on x? [2]

```
1  bool done;
2  int x;
3
4  void thread1() {
5      x = rand() % 2;
6      done = true;
7  }
8
9  int thread2() {
10     while (!done) { }
11     return x;
12 }
```

Ad-hoc Strategies

Many myths about what is safe and what isn't in data race free programs:

- Myth 1: "I'm just reading, not updating"
- Myth 2: "By cleverly arranging read/write order I can avoid a data race"
- Myth 3: "I can avoid races with `volatile`"

Is there a data race on x? [2]

```
1  bool done;
2  int x;
3
4  void thread1() {
5      x = rand() % 2;
6      done = true;
7  }
8
9  int thread2() {
10     while (!done) { }
11     return x;
12 }
```

Yes!

- Compiler may reorder `x = ...` and `done = true`
- Thread 2 could see `done == true` before `x` is written
- `volatile` does NOT fix this – it only prevents register caching

Ad-hoc Strategies Fail

waitingonaflag.c

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

bool done;
int x;

void thread1() {
    x = rand() % 2;
    done = true;
}

int thread2() {
    while (!done) { }
    return x;
}
```

- compiler reorders statements
- compiler replaces loops with ifs

compiled with gcc 7.5

```
thread1:
    subq    $8, %rsp
    call    rand@PLT
    movl    %eax, %edx
    movb    $1, done(%rip)  # done = true
    shrl    $31, %edx
    addl    %edx, %eax
    andl    $1, %eax
    subl    %edx, %eax
    movl    %eax, x(%rip)   # x = ...
    addq    $8, %rsp
    ret

thread2:
    cmpb    $0, done(%rip)
    jne .L8     # if !done goto L8
.L7:            # else: loop forever
    jmp .L7
.L8:
    movl    x(%rip), %eax
    ret
```

Synchronizing Shared Access

The Problem

How can we ensure that a critical section executes as if it were a single atomic instruction?

A correct lock must provide three properties:

1. **Mutual exclusion:** Only one thread in the critical section at a time
2. **Progress (no deadlock):** If no thread is in the critical section and some thread wants in, it eventually gets in
3. **Bounded waiting (no starvation):** A waiting thread eventually enters

Synchronizing Shared Access

The Problem

How can we ensure that a critical section executes as if it were a single atomic instruction?

A correct lock must provide three properties:

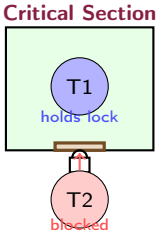
1. **Mutual exclusion:** Only one thread in the critical section at a time
2. **Progress (no deadlock):** If no thread is in the critical section and some thread wants in, it eventually gets in
3. **Bounded waiting (no starvation):** A waiting thread eventually enters

Key Point

You cannot build a correct lock with just loads and stores – hardware support (atomic instructions) is required. This is why we need `pthread_mutex_lock()`, not ad-hoc flag variables.

Mutual Exclusion

- Traditionally known as the "critical section" problem
- Idea:
  - Threads update shared data only after entering a critical section
  - Only 1 thread can be inside a critical section at a time
  - Threads enter critical sections only when accessing shared data
- Thus, all operations (reads and writes) performed within a critical section appear as one atomic (isolated) unit
- Provided by mutexes, commonly called "locks"
  - An *entry* operation "acquires," or locks the lock
  - The thread is then said to "hold" the lock
  - An *exit* operation "releases," or unlocks the lock



Only 1 thread may acquire a lock at a time

Key Point

Locks protect **data**, not code. The critical section is wherever the protected data is accessed – it may span multiple functions.

POSIX Mutex API

```c
#include <pthread.h>

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

// Or dynamically:
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);

// Critical section:
pthread_mutex_lock(&lock);
// ... access shared data ...
pthread_mutex_unlock(&lock);

// When done with mutex:
pthread_mutex_destroy(&lock);
```

Fixed counter++ Example

```c
int counter = 0;
pthread_mutex_t counter_lock = PTHREAD_MUTEX_INITIALIZER;

void *thread_func(void *arg) {
    for (int i = 0; i < 1000000; i++) {
        pthread_mutex_lock(&counter_lock);
        counter++;  // Now protected!
        pthread_mutex_unlock(&counter_lock);
    }
    return NULL;
}
```

Now counter = 2,000,000 every time!
But **slower** due to lock contention – acquiring and releasing a lock on every iteration is expensive.

How Do Locks Work? (Mutex Implementation, FYI)

Broken: Naive Approach

```
1  void lock(int *flag) {
2      while (*flag == 1)
3          ; // spin
4      *flag = 1;  // RACE!
5  }
6  void unlock(int *flag) {
7      *flag = 0;
8  }
```

Both threads can read 0 and both set 1!

```
1  void lock(int *flag) {
2      // test_and_set atomically:
3      //   old = *flag; *flag = 1;
4      //   return old;
5      while (test_and_set(flag, 1) == 1)
6          ; // spin until we get 0
7  }
8  void unlock(int *flag) {
9      *flag = 0;
10 }
```

Hardware guarantees atomicity!

Spinning vs. Blocking

Spinlock: Busy-waits (wastes CPU). Good for very short critical sections.

Blocking (sleep): OS puts thread to sleep, wakes on unlock. Better for long waits.

Spinning vs Blocking: The Cost of Waiting

When a thread can't acquire a lock, what should it do?

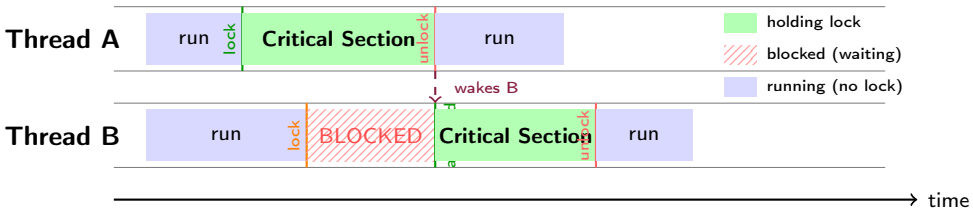
Option 1: Spin (busy-wait)

- Keep checking in a loop: "Is the lock free yet?"
- Pro: Very fast when the wait is short
- Con: Wastes CPU cycles doing nothing useful
- Used in: OS kernel (short critical sections)

Option 2: Sleep (block)

- Ask the OS to put you to sleep; wake me when it's free
- Pro: CPU is free for other threads
- Con: Sleep/wake overhead is expensive
- Used in: User-space mutexes (longer waits)

Critical Section Visualization



Key Insight

Thread B is **blocked** (sleeping) while A holds the lock – it is not wasting CPU cycles. When A unlocks, the OS wakes B. This is why mutexes are efficient for long critical sections.

Understanding Locks

- **Locks protect data, not code.** *Think of it as: "this lock guards this data structure."* No matter where in the code the data is accessed, that lock must be held.
- Each piece of shared data needs a lock that protects it:

```
1 struct list task_queue; // protected by task_queue_lock
2 pthread_mutex_t task_queue_lock; // protects task_queue
```

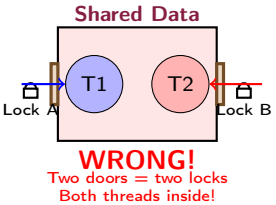
- Locks enable a **cooperative protocol** between threads – they don't protect data automatically
- Different pieces of data *may* be protected by the same lock (might not be a good strategy in practice)
  Rule of thumb: Start with one lock for all shared state. Split into finer-grained locks only when profiling shows contention.

Ignoring these rules accounts for most intermittent failures that involve data corruption in projects

Understanding Locks: Don'ts

- **Don't use different locks** for the same data
- Don't try to acquire a lock you already hold[a]
- Don't forget to unlock a lock on some path
  - Best practice: a lock is not held on entry and not held on exit; or held on entry and stays held on exit
- Don't hold a lock while blocking (sleeping, I/O, ...)
  - This makes threads needing the same lock wait for the same event

[a] unless recursive mutexes are used



Analogy

Using different locks for the same data is like having two doors to the same room – each with its own lock. Both threads get "in" simultaneously!

Deadlock: When Locks Fight Back

Definition

A situation where two or more threads are blocked forever, each waiting for a resource held by another.

Classic scenario:

Thread 1	Thread 2
lock(A)	lock(B)
lock(B) ← *waits*	lock(A) ← *waits*

Neither thread can proceed – both stuck forever.

Primary Defense: Lock Ordering

Always acquire locks in a **consistent global order**.
If all threads lock A before B, circular wait is impossible.

P2 Tip: If your threadpool hangs, use GDB:
`thread apply all bt`

Key Point

Full treatment (Coffman conditions, detection, prevention) in **Lecture 17**.

Sequential Consistency

- Consider each thread's program a series of "steps"
- A sequentially consistent execution is the result of some interleaving of these steps (without changing the order within each thread, and immediately propagating updates to other threads)
- By default, programming languages do not guarantee such an execution because
  - Compiler may reorder statements that do not logically depend on each other
  - Processors may reorder load/store instructions likewise
- See Adve & Boehm [2] for precise definition and discussion

Rule

Modern languages such as C, C++, Java guarantee sequentially consistent execution only for programs that are data race free. Achieving freedom from data races requires synchronization, specifically the proper use of critical sections (aka mutual exclusion, locks)

Understanding Locks: Do's

- Use race condition detection tools, e.g. Helgrind, DRD, Intel Thread Checker, TSan

```
$ valgrind --tool=helgrind ./race1       # detect data races
$ valgrind --tool=helgrind --history-level=full ./race1  # detailed trace
```

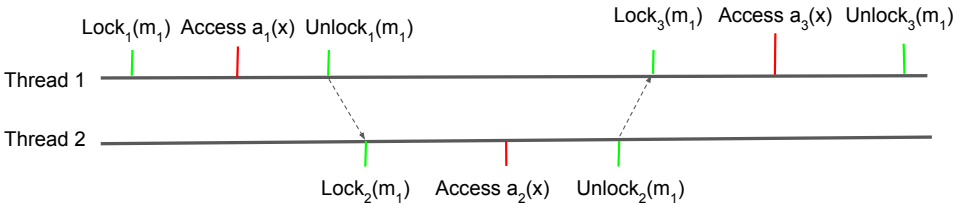


Figure: Proper locking establishes "happens-before" between accesses to x:
$a_1(x) \to U_1(m_1) \to L_2(m_1) \to a_2(x)$, etc. Unlock always happens-before the next successful lock.

Data Races Revisited

- The C11 Memory Model defines data races in terms of the happens-before relationship as follows:
- Different threads of execution are always allowed to access (read and modify) different memory locations concurrently, with no interference and no synchronization requirements.
- When an evaluation of an expression writes to a memory location and another evaluation reads or modifies the same memory location, the expressions are said to conflict. A program that has two conflicting evaluations has a data race unless either
  - both conflicting evaluations are atomic operations
  - one of the conflicting evaluations happens-before another
- If a data race occurs, the behavior of the program is undefined.

Other happens-before relationships

- Mutexes are not the only way to induce HB relationships across threads.
- Other methods include
  - Thread create: any operations done by the thread before it creates a new thread happen before any operations that new thread does. That means single writes (initializations) before a value is shared between threads will not conflict with simultaneous reads by those threads.
  - Thread join: any operations done by a thread happen before the thread joining it returns from the thread join call. This means data written by the former can be accessed by the latter without needing a lock.
  - Certain operations on semaphores and condition variables as well as constructs such as barriers; e.g. posting to a semaphore happens before returning from waiting on a semaphore.

Thread-Safety

Definition

Property of a function to yield correct result when called from multiple threads

- Attribute that must be documented as part of the API documentation of a library
  - Some traditional C functions are not thread-safe (`strtok`)
- Functions are not thread-safe if they
  1. Fail to protect shared variables
  2. Rely on persistent state across invocations
  3. Return a pointer to a static variable
  4. Call other functions that aren't thread safe

Relying on persistent state across invocations

- This function fails to protect shared state (`next`) and does not maintain deterministic order of pseudo-random numbers
- Fix: pass state to function, e.g. `rand_r()`
- C library provides `_r()` variants that are thread-safe
- The 'r' stands for **reentrant** – a stronger property than thread-safe. Reentrant functions can be safely reentered while a call is in progress (e.g., recursive functions)

Example: `rand()`

```c
static unsigned int next = 1;

/* rand -return pseudo-random integer
   in 0..32767 */
int rand(void) {
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) %
    ↪  32768;
}

/* srand-set seed for rand() */
void srand(unsigned intseed) {
    next = seed;
}
```

Fix: `int rand_r(unsigned int *seedp);`

Caller owns the seed – no shared state.

Lock Granularity: A Preview

Coarse-Grained (One Big Lock)

- One lock protects all shared state
- Simple to reason about
- **High contention:** threads serialize
- Good starting point

Fine-Grained (Many Locks)

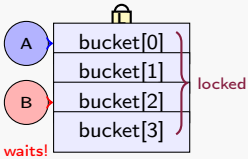
- Separate locks per data structure
- More parallelism possible
- **Complex:** deadlock risk increases
- Only when profiling demands it

Key Point

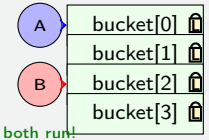
Start coarse. Measure. Then refine. "Premature optimization is the root of all evil" applies doubly to lock granularity.

Lock Granularity in Practice: Hash Table

Coarse: One Lock for Entire Table



Thread B waits even for a different bucket.



Threads A and B access different buckets concurrently.

When Fine-Grained Pays Off

- Hash tables, databases, file systems – **independent partitions**
- Throughput scales with number of buckets (up to N threads in parallel)
- But: more locks = more deadlock risk, more memory, more complexity

EXAM FOCUS: Data Races and Locks

You should be able to:

1. Define what a data race is
2. Identify data races in code
3. Explain why `counter++` is not atomic
4. Write correct code using `pthread_mutex`
5. Explain why locks protect data, not code

Two threads execute this code: `if (x == 0) x = 1;`
Is there a data race? Explain.

EXAM FOCUS: Data Races and Locks

You should be able to:

1. Define what a data race is
2. Identify data races in code
3. Explain why `counter++` is not atomic
4. Write correct code using `pthread_mutex`
5. Explain why locks protect data, not code

Two threads execute this code: `if (x == 0) x = 1;`
Is there a data race? Explain.

Yes! Both read and write to `x` without synchronization.

Debugging Data Races: ThreadSanitizer

Step 1: Compile with TSan

```
$ gcc -fsanitize=thread -g race1.c -o race1 -lpthread
```

Step 2: Run normally

```
$ ./race1
```

Step 3: Read the report

```
WARNING: ThreadSanitizer: data race
  Read of size 4 at 0x... by thread T1:
    #0 thread_func main.c:15
  Previous write of size 4 at 0x... by thread T2:
    #0 thread_func main.c:15
```

- Shows **exact line numbers** of conflicting accesses
- About 10x slowdown, but catches races you'd never find otherwise

Run your threadpool under TSan before submission! See: `demos/08-locking/HOWTO.md` for full walkthrough

EXAM FOCUS: Common Locking Mistakes

Watch out for these patterns on exams and in P2:

Mistakes That Cause Races

- Using **different locks** for the same shared data (two-door problem)
- Assuming single-statement operations like x++ are atomic
- Checking a condition outside the lock, then acting inside (TOCTOU)

Mistakes That Cause Deadlock or Stalls

- Acquiring more locks $\neq$ more safety – can cause deadlock!
- Forgetting to unlock on error paths (every `return` needs `unlock`)
- Holding a lock while sleeping, doing I/O, or calling unknown code

Lock acquisition should be brief, and critical sections should be short. If you can't see both `lock()` and `unlock()` on one screen, reconsider.

EXAM FOCUS: Locking Questions

```
1  int balance = 1000;
2
3  void deposit(int amount)  { balance += amount; }
4  void withdraw(int amount) { balance -= amount; }
```

Thread A calls `deposit(100)`, Thread B calls `withdraw(50)` concurrently.

Questions:

1. Is there a data race? On what variable?
2. What are the possible final values of `balance`?
3. How would you fix this using `pthread_mutex`?

EXAM FOCUS: Locking Questions

```
1  int balance = 1000;
2
3  void deposit(int amount)  { balance += amount; }
4  void withdraw(int amount) { balance -= amount; }
```

Thread A calls `deposit(100)`, Thread B calls `withdraw(50)` concurrently.

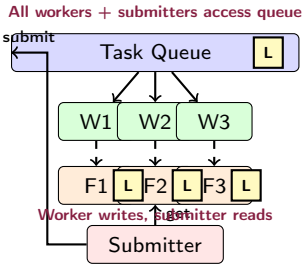
Questions:

1. Is there a data race? On what variable?
2. What are the possible final values of `balance`?
3. How would you fix this using `pthread_mutex`?

Answer

1. **Yes** – both threads read/write `balance` without synchronization.
2. Possible values: 1050 (correct), 1100 (withdraw lost), 950 (deposit lost).
3. Add a single `pthread_mutex_t` and lock/unlock around both `balance += ` and `balance -=`

P2 Connection: Locking in Your Threadpool



Common P2 Mistakes

1. Accessing queue without the lock
2. **TOCTOU:** checking "queue empty?" outside lock, dequeuing inside
3. Forgetting to lock when writing future result
4. Holding pool lock while running user's task function (deadlock if task submits!)

Compile with `-fsanitize=thread` early. TSan finds races that won't show until the autograder's stress tests.

Summary

Data Races

- Threads access shared data without synchronization
- Result: undefined behavior, lost updates
- Avoid: partition, duplicate, or lock
- Detect: ThreadSanitizer

Locks (Mutexes)

- Provide mutual exclusion for critical sections
- Protect **data**, not code
- Use consistently for *all* accesses
- Start coarse, refine later

Thread Safety

- Functions must be safe for concurrent calls
- Watch for static state, global variables
- Use `_r()` variants
- Deadlock: consistent lock ordering

Next: Atomic Variables and Operations

Coming up in Lecture 13:

- Why compilers and CPUs reorder instructions
- C11 `_Atomic` types and operations
- Compare-and-swap: the universal atomic primitive
- When to use atomics vs. locks

Later in the synchronization arc:

- Lecture 14: Performance in Multi-Threaded Programs
- Lecture 15: Semaphores
- Lecture 16: Condition Variables
- Lecture 17: Deadlock (detailed treatment)

References

- OSTEP Ch. 28: Locks [1]
- CS:APP Chapter 12.4–12.5 – Bryant & O'Hallaron
- `man pthread_mutex_init`, `man pthread_mutex_lock`
- The Linux Programming Interface, Ch. 30 – Michael Kerrisk

[1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau.
Operating Systems: Three Easy Pieces.
Arpaci-Dusseau Books, 1.00 edition, August 2018.

[2] Hans-J. Boehm and Sarita V. Adve.
You don't know jack about shared variables or memory models.
Commun. ACM, 55(2):48–54, February 2012.

[3] Daniel J. Sorin, Mark D. Hill, and David A. Wood.
A Primer on Memory Consistency and Cache Coherence.
Morgan & Claypool Publishers, 1st edition, 2011.