

CS 3214 – Computer Systems

Test 2 Review Guide

Multithreading

Fall 2026

Contents

1	Exam Information	2
2	Topic Checklist	3
2.1	Introduction to Multithreading	3
2.2	Basic Locking (Managing Shared State)	3
2.3	Atomic Variables and Operations	3
2.4	Locking Performance	4
2.5	Semaphores	4
2.6	Condition Variables and Monitors	4
2.7	Deadlock	5
2.8	Exercises and Project	5
2.8.1	Exercise 3: Outfoxed	5
2.8.2	Project 2: Fork-Join Threadpool	5
3	Key Concepts	7
4	Self-Test Questions	9
4.1	Multithreading and Synchronization	9
4.2	Atomics and Locking Performance	9
4.3	Exercises and Project	10
5	Self-Test Answers	11
5.1	Multithreading and Synchronization	11
5.2	Atomics and Locking Performance	12
5.3	Exercises and Project	12
6	Study Strategy	13

1 Exam Information

Date	Nov 2/3, 2026 (Monday/Tuesday)
Duration	50 minutes
Total	100 points, approximately 4-5 questions
Scope	Concurrency fundamentals, monitors, P2, Ex3
Format	Closed-book; one page of prepared notes (both sides) allowed

What to Put on Your Notes Sheet

Good candidates for your one-page cheat sheet:

- pthread API signatures: pthread_create, pthread_join, pthread_mutex_lock/unlock
- Semaphore API: sem_init, sem_wait, sem_post
- Condition variable API: pthread_cond_wait, pthread_cond_signal, pthread_cond_broadcast
- The monitor pattern template (mutex + CV + while-loop)
- Coffman's four conditions for deadlock
- C11 atomics: atomic_load, atomic_store, atomic_compare_exchange_strong

2 Topic Checklist

Use this checklist to track your review progress. Check off each topic as you feel confident about it.

2.1 Introduction to Multithreading

- ☐ Threads vs. processes: threads share the same address space (heap, globals, code); each thread has its own stack and registers
- ☐ pthread API: `pthread_create()`, `pthread_join()`, `pthread_detach()`, `pthread_self()`
- ☐ Thread memory model: shared heap and global variables, private stack (local variables)
- ☐ Why threads are cheaper than processes: no TLB flush, shared page tables, lighter-weight context switch
- ☐ Thread safety: a function is thread-safe if it can be safely called from multiple threads concurrently
- ☐ Passing arguments to threads: use `void *arg` parameter of `pthread_create()`
- ☐ Joining vs. detaching: `pthread_join()` blocks until the thread terminates and retrieves its return value; `pthread_detach()` lets the thread clean up automatically on exit

2.2 Basic Locking (Managing Shared State)

- ☐ Data race definition (C11 memory model): two evaluations conflict if they access the same memory location, at least one is a write, and they are not ordered by a happens-before relationship (e.g., two threads doing `counter++` without a lock – the load-add-store sequence can interleave, losing an update)
- ☐ Undefined behavior: a program with a data race has undefined behavior in C/C++
- ☐ Mutual exclusion with `pthread_mutex_t`: `pthread_mutex_lock()` and `pthread_mutex_unlock()` (e.g., `pthread_mutex_lock(&lock); shared_counter++; pthread_mutex_unlock(&lock);` ensures only one thread modifies the counter at a time)
- ☐ Critical sections: identifying what shared state to protect and choosing lock granularity
- ☐ Happens-before relationships: established by `pthread_create()`, `pthread_join()`, lock acquire/release pairs
- ☐ Atomicity violations vs. data races: an atomicity violation is a higher-level correctness bug (a sequence of operations should have been atomic); a data race is a specific memory-model violation
- ☐ TOCTTOU (Time-of-Check-to-Time-of-Use) bugs: checking a condition and acting on it non-atomically

2.3 Atomic Variables and Operations

- ☐ C11 `_Atomic` qualifier and `<stdatomic.h>` operations
- ☐ `atomic_load()` and `atomic_store()`: atomic reads and writes
- ☐ `atomic_fetch_add()`, `atomic_fetch_sub()`: atomic read-modify-write operations
- ☐ `atomic_compare_exchange_strong()`: compare-and-swap (CAS)
- ☐ Memory ordering: `memory_order_relaxed` (no ordering guarantees), `memory_order_acquire/release` (synchronization), `memory_order_seq_cst` (total order, default)

- Why atomics alone are insufficient for complex invariants (e.g., transferring money between accounts requires a lock to maintain the sum invariant)

2.4 Locking Performance

- Lock splitting/breaking: dividing a single lock into multiple locks to reduce contention (e.g., per-bucket locks in a hash table instead of one lock for the entire table)
- Measuring and profiling lock contention: `perf` tool, lock hold times, contention ratios
- Coarse-grained vs. fine-grained locking trade-offs: coarse is simpler but limits parallelism; fine is complex but allows more concurrency
- Scalability bottlenecks: identifying the critical path and serial sections
- Per-CPU or per-thread data structures to eliminate sharing

2.5 Semaphores

- Counting semaphore abstraction: an integer counter with atomic `sem_wait()` (decrement, block if zero) and `sem_post()` (increment, wake a waiter)
- `sem_init()`: initialize with a count (e.g., 0 for synchronization, 1 for mutex, N for bounded buffer)
- Binary semaphore for mutual exclusion: initialized to 1, acts like a mutex
- Producer-consumer with bounded buffer: two semaphores (empty initialized to N , full initialized to 0) plus a mutex (e.g., producer does `sem_wait(&empty)`, `lock(&mutex)`, enqueue item, `unlock(&mutex)`, `sem_post(&full)`; consumer does the reverse)
- Semaphore vs. mutex: a semaphore can be posted by any thread (not just the one that waited); a mutex must be unlocked by the thread that locked it
- Using semaphores for signaling/ordering: one thread waits, another posts to indicate an event occurred

2.6 Condition Variables and Monitors

- The Monitor Pattern: mutex + condition variable + while-loop predicate check
- `pthread_cond_wait()`: atomically releases the mutex and puts the thread to sleep; re-acquires the mutex upon waking up
- Why while (not if): spurious wakeups can occur; multiple waiters may be woken but only one should proceed; the condition may have changed between signal and wake (e.g., Thread A wakes from `pthread_cond_wait()`, but Thread B already consumed the item – without the while-loop, Thread A would proceed on stale state)
- `pthread_cond_signal()` vs. `pthread_cond_broadcast()`: signal wakes one waiter, broadcast wakes all
- When to use signal: when exactly one waiter can proceed (e.g., one item added to a bounded buffer)
- When to use broadcast: shutdown, or when different waiters may be waiting on different predicates
- Common bug #1: forgetting the condition variable entirely (busy-waiting on the predicate)
- Common bug #2: using a different mutex for `pthread_cond_wait()` than the one protecting the shared state

- ☐ Common bug #3: using `if` instead of `while` around the predicate check
- ☐ Common bug #4: lost wakeup – signaling before the other thread is waiting, with no state variable to record the event

2.7 Deadlock

- ☐ Deadlock definition: a set of threads are permanently blocked, each waiting for a resource held by another thread in the set (e.g., Thread 1 holds lock A, waits for lock B; Thread 2 holds lock B, waits for lock A)
- ☐ Coffman’s four necessary conditions (all must hold simultaneously):
 1. **Mutual exclusion**: at least one resource is held in a non-sharable mode
 2. **Hold and wait**: a thread holds one resource while waiting for another
 3. **No preemption**: resources cannot be forcibly taken from a thread
 4. **Circular wait**: a circular chain of threads, each waiting for a resource held by the next
- ☐ Prevention by breaking circular wait: impose a total ordering on locks and always acquire in order
- ☐ Prevention by breaking hold-and-wait: acquire all locks at once (e.g., `pthread_mutex_trylock()` and back off)
- ☐ Detection: resource allocation graphs (directed graph; cycle $\Rightarrow$ deadlock with single-instance resources)
- ☐ Dining philosophers problem: 5 philosophers, 5 forks, classic circular-wait scenario
- ☐ Dining philosophers solutions: lock ordering (pick up lower-numbered fork first), limit concurrent diners, use a single global lock

2.8 Exercises and Project

2.8.1 Exercise 3: Outfoxed

- ☐ Understand function of chickens as servers processing work
- ☐ Understand function of clues and suspects as “new work”
- ☐ Know how to solve the clue-to-clue signaling with semaphores
- ☐ Know how to solve suspect-to-chicken and clue-to-chicken signaling with condition variables
- ☐ Know how to solve chicken-to-chicken signaling with condition variables

2.8.2 Project 2: Fork-Join Threadpool

- ☐ Work sharing vs. work stealing: work sharing uses a single global queue (high contention); work stealing uses per-worker dequeues (low contention, tasks stolen from other workers when idle)
- ☐ Future lifecycle: submitted $\rightarrow$ in-progress $\rightarrow$ completed
- ☐ `future_get`: blocks until the future is completed and returns the result
- ☐ Inline helping (work-assist): when `future_get()` is called and the task has not started, the calling thread executes it directly – this prevents deadlock in fork-join patterns where a parent waits on children

- ☐ Thread-local variables: used to distinguish worker threads from external threads (workers can steal tasks; external threads cannot)
- ☐ Lock design: a single global lock eliminates hold-and-wait (no deadlock) but limits scalability; per-worker locks reduce contention but require careful lock ordering
- ☐ Deadlock prevention: if workers simply block on `future_get()` without inline helping, all workers can be blocked waiting for tasks that no one is executing
- ☐ Performance considerations: minimizing lock hold time and choosing task granularity carefully (too fine → overhead, too coarse → load imbalance)

3 Key Concepts

Data Races and Happens-Before

C11 data race definition: Two memory operations *conflict* if they access the same memory location and at least one is a write. A data race occurs when two conflicting operations from different threads are not ordered by a happens-before relationship. A program with a data race has **undefined behavior**.

Happens-before is established by:

- Sequencing within a single thread (program order)
- `pthread_create()`: the creating thread's operations before the call happen-before all operations in the new thread
- `pthread_join()`: all operations in the joined thread happen-before operations after the join call
- Lock release → lock acquire: an unlock happens-before any subsequent lock of the same mutex

Example:

```
int shared = 0; // global

void *thread_A(void *arg) { shared = 1; return NULL; }
void *thread_B(void *arg) { printf("%d\n", shared); return NULL; }
```

If threads A and B run concurrently with no synchronization, this is a data race (one write, one read, no ordering). Adding a mutex around both accesses eliminates the race.

The Monitor Pattern

The correct pattern for waiting on a condition with pthreads:

```
pthread_mutex_lock(&mutex);
while (!condition) { // MUST be while, not if
    pthread_cond_wait(&cv, &mutex); // atomically: release mutex + sleep
}
// ... condition is true, mutex is held, safe to act ...
pthread_mutex_unlock(&mutex);
```

Why while and not if:

1. **Spurious wakeups:** the OS may wake the thread even if no one signaled
2. **Multiple waiters:** another thread may have consumed the resource between signal and wake
3. **Broadcast:** all waiters are woken, but only one can proceed

Three common bugs:

1. **Busy-waiting** – checking the condition in a loop without a CV (wastes CPU, may be a data race)
2. **Different mutexes** – protecting the shared state with one mutex but passing a different mutex to `pthread_cond_wait()` (undefined behavior, lost wakeups)

3. **if instead of while** – the condition may no longer hold when the thread wakes up

Deadlock

Dining philosophers: Five philosophers sit around a table, each with a fork on either side. To eat, a philosopher needs both forks. If each philosopher picks up the left fork and waits for the right fork, all five are deadlocked.

Map to Coffman's conditions:

1. **Mutual exclusion:** each fork can be held by only one philosopher
2. **Hold and wait:** each philosopher holds one fork while waiting for another
3. **No preemption:** a fork cannot be forcibly taken from a philosopher
4. **Circular wait:** philosopher 0 waits for philosopher 1's fork, ..., philosopher 4 waits for philosopher 0's fork

Lock ordering solution: Number the forks 0–4. Each philosopher picks up the lower-numbered fork first. Philosopher 4 (who sits between forks 4 and 0) picks up fork 0 first, breaking the cycle.

4 Self-Test Questions

Try to answer each question without looking at your notes. Answers are provided in the next section.

4.1 Multithreading and Synchronization

1. True or false: After `pthread_create()`, the new thread shares the same heap as the creating thread but has its own stack.
2. Two threads execute the following code concurrently, where `counter` is a shared global initialized to 0:

```
counter++; // in thread A
counter++; // in thread B
```

What are the possible final values of `counter`? Explain why.

3. What is the difference between a data race and an atomicity violation? Give an example where there is an atomicity violation but no data race.
4. A bounded buffer uses a mutex and two condition variables (`not_full` and `not_empty`). Write the pseudocode for `produce(item)` using the monitor pattern.
5. True or false: Calling `pthread_cond_signal()` while not holding the associated mutex is always a bug.
6. A program uses `sem_init(&s, 0, 0)`. If thread A calls `sem_wait(&s)` and thread B later calls `sem_post(&s)`, what happens?
7. In a bounded buffer, one producer adds exactly one item and there are multiple consumers waiting on `not_empty`. Should the producer use `pthread_cond_signal()` or `pthread_cond_broadcast()`? Why?
8. What are Coffman's four necessary conditions for deadlock? If you break any one of them, is deadlock impossible?
9. Two threads each need to acquire locks A and B. Thread 1 acquires A then B; Thread 2 acquires B then A. Identify the Coffman condition that is violated by a lock-ordering fix, and describe the fix.
10. True or false: A deadlocked program consumes 100% CPU.

4.2 Atomics and Locking Performance

11. Why is an atomic instruction so much more expensive than a regular memory access?
12. A hash table uses a single global lock for all operations. Describe two techniques to improve its throughput under heavy concurrent access.

4.3 Exercises and Project

13. In Project 2, what is “inline helping” (work-assist) in `future_get()`, and why is it necessary for fork-join programs?
14. In Project 2, explain why a work-stealing threadpool scales better than a single shared queue (work-sharing) on many cores.
15. In Project 2, what happens if `future_get()` simply blocks (sleeps) until the task is done, without inline helping? Describe a scenario where this leads to deadlock.

5 Self-Test Answers

5.1 Multithreading and Synchronization

1. True. Threads within the same process share the heap, global variables, and code. Each thread has its own stack for local variables and function call state.
2. The possible values are 1 or 2. `counter++` is not atomic – it involves a load, increment, and store. If both threads load 0 before either stores, both store 1 (final value = 1). If one completes before the other starts, the final value is 2. This is a classic data race.
3. A data race is a memory-model-level concept: two unsynchronized, conflicting accesses to the same location. An atomicity violation is a higher-level correctness issue: a sequence of operations that should be indivisible is interleaved with other operations. Example: using atomic operations to check a condition and then act on it:

```
if (atomic_load(&flag) == 0)    // check
    atomic_store(&flag, 1);    // act
```

There is no data race (both operations are atomic), but there is an atomicity violation – another thread can change `flag` between the check and the store.

```
4. void produce(item_t item) {
    pthread_mutex_lock(&mutex);
    while (count == BUFFER_SIZE) {
        pthread_cond_wait(&not_full, &mutex);
    }
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    count++;
    pthread_cond_signal(&not_empty);
    pthread_mutex_unlock(&mutex);
}
```

5. False. It is technically allowed by the POSIX specification to signal without holding the mutex. However, it is a common source of bugs (lost wakeups) and is generally considered bad practice. The safe rule is: always hold the mutex when signaling, because you typically just modified the shared state.
6. Thread A blocks in `sem_wait` because the semaphore value is 0. When thread B calls `sem_post`, the value is incremented to 1 and thread A is unblocked, decrementing it back to 0. This pattern implements a rendezvous/synchronization point.
7. Use `pthread_cond_signal()`. Exactly one new item became available, so at most one consumer can make progress. Using `pthread_cond_broadcast()` would also be correct, but it would wake extra consumers that would immediately go back to sleep after rechecking the predicate.
8. The four conditions are: (1) mutual exclusion, (2) hold and wait, (3) no preemption, (4) circular wait. Yes – these are *necessary* conditions. If any one is broken, deadlock cannot occur.

9. The violated condition is **circular wait**. Fix: impose a total ordering on locks (e.g., always acquire A before B). Thread 2 must be changed to acquire A first, then B. This eliminates the possibility of a circular dependency.
10. False. Deadlocked threads are blocked (sleeping), waiting for a lock or resource that will never be released. They consume essentially no CPU – they are not spinning. (If threads are busy-waiting on a spinlock, they consume CPU, but traditional mutex-based deadlock does not.)

5.2 Atomics and Locking Performance

11. An atomic instruction requires the CPU to coordinate with the cache coherence protocol across all cores. It must obtain exclusive ownership of the cache line, ensure no other core has a stale copy, and maintain the total order of operations. This cross-core communication takes much longer than a simple L1 cache hit, which is a purely local operation.
12. Two techniques: (1) **Lock splitting** – use per-bucket locks instead of a single global lock, so operations on different buckets proceed in parallel. (2) Reduce the time spent while holding the lock so each operation serializes less of the workload.

5.3 Exercises and Project

13. With a single shared queue, every thread contends for the same lock on every enqueue and dequeue operation. As the number of cores increases, lock contention grows linearly, and the queue becomes a serial bottleneck. With work stealing, each worker has its own local deque and primarily operates on it without contention. Stealing only occurs when a worker's deque is empty, so contention is proportional to the load imbalance, not the total number of operations. This dramatically reduces synchronization overhead on many cores.
14. Consider a fork-join program with 4 worker threads. A task T is submitted and picked up by worker W_1 . T forks 4 subtasks (T_1, T_2, T_3, T_4) and then calls `future_get()` on each. Workers W_2, W_3, W_4 each pick up one subtask. But T_4 is still in the queue, and all 4 workers are now blocked: W_1 is blocked on `future_get()`(T_1), and W_2, W_3, W_4 are executing T_1, T_2, T_3 but *not* T_4 . Actually, in this scenario W_1 blocks on T_1 while T_4 sits unexecuted. Even simpler: with 1 worker, the worker submits a child task, then blocks on `future_get()` – but the child is in the queue and the only worker is blocked. No one can execute the child $\rightarrow$ deadlock.

6 Study Strategy

Recommended Approach

1. **Review lecture slides** on multithreading, synchronization, memory allocation, GC, and virtual memory from both sections. Focus on diagrams, code examples, and “key point” boxes. Pay special attention to synchronization patterns.
2. **Redo Exercise 3 (MPSC)**. Re-implement the bounded buffer from scratch. Make sure you can write correct producer-consumer code with condition variables without looking at your solution.
3. **Understand your P2 code**. Be able to explain how your threadpool handles task submission, work stealing, and `future_get()` with inline helping. Know your locking strategy and why it avoids deadlock.
4. **Practice synchronization code tracing**. Given a program with threads, locks, condition variables, and semaphores, trace through the execution and determine possible outputs or identify bugs.
5. **Prepare your notes sheet**. Organize it by topic. Good candidates:
 - Monitor pattern template (mutex + CV + while)
 - Semaphore API and producer-consumer template
 - Coffman’s four conditions and prevention strategies
 - C11 atomic operations summary