CS 3214 – Computer Systems

Test 2 Review Guide

Multithreading | Memory Management | Virtual Memory Spring 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 Dynamic Memory Allocation (malloc/free) . . . 5
2.9 Garbage Collection . . . 6
2.10 Virtual Memory . . . 6
2.11 Exercises and Project . . . 7
2.11.1 Exercise 3: MPSC Channel . . . 7
2.11.2 Project 2: Fork-Join Threadpool . . . 7

3 Key Concepts **9**

4 Self-Test Questions **13**
4.1 Multithreading and Synchronization . . . 13
4.2 Atomics and Locking Performance . . . 13
4.3 Dynamic Memory Allocation . . . 14
4.4 Garbage Collection . . . 14
4.5 Virtual Memory . . . 14
4.6 Exercises and Project . . . 15

5 Self-Test Answers **16**
5.1 Multithreading and Synchronization . . . 16
5.2 Atomics and Locking Performance . . . 17
5.3 Dynamic Memory Allocation . . . 17
5.4 Garbage Collection . . . 17
5.5 Virtual Memory . . . 18
5.6 Exercises and Project . . . 19

6 Study Strategy **20**

1 Exam Information

Date	April 13, 2026 (Monday)
Duration	75 minutes
Total	100 points, approximately 6 questions, ∼14 pages
Scope	Concurrency fundamentals, monitors, P2, Ex3, and memory management
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`
- Memory ordering: relaxed, acquire/release, sequential consistency
- Malloc: block header layout, coalescing rules, placement policies
- GC: mark-and-sweep algorithm, reachability graph, generational hypothesis
- Virtual memory: page table structure, page faults, page replacement algorithms

The exam covers 6 questions totaling 100 points.

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 ⇒ 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 Dynamic Memory Allocation (malloc/free)

☐ `malloc()`, `free()`, `realloc()`, `calloc()` API

☐ Heap organization: the allocator manages a region of memory between the program break (`sbrk()`/`brk()`) and the stack

☐ Implicit free list: each block has a header (size + allocated bit); the allocator traverses headers to find free blocks

☐ Explicit free list: free blocks are linked together for faster search; allocated blocks are not on the list

☐ Block header format: size (in bytes, including header) with the low bit used as an allocated flag (works because blocks are aligned)

☐ Alignment requirements: blocks must be aligned to 8 or 16 bytes (double-word or quad-word boundary)

☐ Internal fragmentation: wasted space inside an allocated block (due to padding, alignment, header overhead)

☐ External fragmentation: enough total free memory exists but no single contiguous block is large enough

☐ Placement policies: first-fit (fast, causes fragmentation at beginning), next-fit (avoids re-scanning), best-fit (minimizes waste, slow)

- ☐ Splitting: when a free block is larger than needed, split it into an allocated block and a smaller free block
- ☐ Coalescing: merging adjacent free blocks to reduce external fragmentation (e.g., freeing a block between two free blocks: all three merge into one large free block)
- ☐ Immediate vs. deferred coalescing: immediate merges on every `free()`; deferred batches merges
- ☐ Boundary tags (footers): store size at both the beginning and end of each block so the allocator can find the previous block's status in O(1) for coalescing

2.9 Garbage Collection

- ☐ Automatic memory management: the runtime system reclaims memory that is no longer reachable, eliminating explicit `free()` calls
- ☐ Reachability: an object is *live* (cannot be collected) if it is reachable from a root (stack variables, global variables, registers)
- ☐ Mark-and-sweep algorithm (CSAPP 9.10): (1) mark phase – start from roots, follow all pointers, mark reachable objects; (2) sweep phase – scan the heap, free all unmarked objects, then clear marks (e.g., roots {a, b}, a → X → Y, b → Z; after mark: X, Y, Z marked; sweep frees everything else)
- ☐ Mark-and-sweep is safe but causes stop-the-world pauses
- ☐ Generational GC hypothesis: "most objects die young" – the nursery (young generation) is collected frequently; survivors are promoted to the old generation, which is collected less often (e.g., `"User " + name` creates a temporary String that becomes garbage immediately – collected in the next minor GC)
- ☐ Minor vs. major collections: minor collects the young generation (fast); major collects the entire heap (slow)
- ☐ Memory leak in C: memory that is unreachable (no pointer to it) and was never freed – the allocator cannot reclaim it
- ☐ Memory leak in Java: memory that is reachable (references still exist) but unused by the program – the GC cannot collect it (e.g., a `HashMap` cache that grows without eviction)
- ☐ Bloat vs. churn: bloat is excessive retained live data; churn is high allocation/deallocation rate increasing GC frequency
- ☐ Stop-the-world vs. concurrent/incremental collection: concurrent collectors run alongside the application but are more complex

2.10 Virtual Memory

- ☐ Virtual memory is a *technique* combining address translation, paging, and protection – not just "using disk as RAM"
- ☐ Virtual address (VA) vs. physical address (PA): the CPU issues VAs; the MMU translates to PAs
- ☐ Paging: virtual address space divided into fixed-size *pages* (typically 4 KB); physical memory divided into *frames* of the same size
- ☐ Page table: per-process data structure mapping virtual page numbers (VPN) to physical frame numbers (PFN)
- ☐ Page table entry (PTE) bits: valid/present, read/write, user/supervisor, dirty, accessed/reference

- ☐ TLB (Translation Lookaside Buffer): hardware cache of recent VPN → PFN translations; TLB miss triggers a page table walk
- ☐ TLB flush on context switch between processes: part of the lecture's simplified model because each process has its own page table
- ☐ Multi-level page tables (x86-64): 4 levels (PML4, PDPT, PD, PT), each index is 9 bits, with a 12-bit offset – saves memory by not allocating page tables for unmapped regions
- ☐ Address translation: VPN = VA $\gg$ 12; PFN = PageTable[VPN]; PA = (PFN $\ll$ 12) | offset
- ☐ Page faults: (1) minor fault – page is in memory but not mapped, kernel allocates frame and updates PTE (e.g., first access to a heap page allocated by `malloc()` triggers a minor fault – the kernel allocates a physical frame and maps it); (2) major fault – page must be read from disk; (3) invalid – access violation → SIGSEGV
- ☐ Demand paging: pages are loaded into physical memory only when first accessed (lazy allocation)
- ☐ Copy-on-write (COW): after `fork()`, parent and child initially share physical frames marked read-only; a write triggers a page fault and copies just the written page
- ☐ Page replacement problem: when RAM is full, the OS must choose a victim page to evict
- ☐ Page replacement algorithms: OPT as benchmark, FIFO, LRU, and Clock
- ☐ Belady's anomaly: FIFO can have more page faults with more frames; stack algorithms such as OPT and LRU do not
- ☐ Working set and thrashing: if the working set does not fit in RAM, page faults dominate useful work
- ☐ Thrashing: when the working set exceeds physical memory, causing excessive page faults and near-zero useful work

2.11 Exercises and Project

2.11.1 Exercise 3: MPSC Channel

- ☐ Multi-producer, single-consumer bounded buffer: multiple sender threads, one receiver thread, fixed-capacity queue
- ☐ `mpsc_send`: blocks if the buffer is full; returns `SEND_ERROR` if the receiver has been dropped
- ☐ `mpsc_recv`: blocks if the buffer is empty; returns `RECV_ERROR` if all senders have been dropped
- ☐ Clone/drop semantics: `mpsc_sender_clone()` increments a reference count on the sender; `mpsc_sender_drop()` decrements it; when count reaches 0, the channel detects all senders are gone
- ☐ Synchronization: a mutex protects the buffer, and condition variables signal producers (buffer not full) and the consumer (buffer not empty)
- ☐ Detecting channel closure: the receiver uses the sender reference count to decide when to return `RECV_ERROR`; senders check whether the receiver has been dropped before sending

2.11.2 Project 2: Fork-Join Threadpool

- ☐ Work sharing vs. work stealing: work sharing uses a single global queue (high contention); work stealing uses per-worker deques (low contention, tasks stolen from other workers when idle)
- ☐ Future lifecycle: submitted → in-progress → 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.

Dynamic Memory Allocation

Implicit free list example:
Consider a heap with 4-byte headers. Each header stores the block size (including header) and an allocated bit in the low bit.

Heap state: `[16/1] [32/0] [16/1] [0/1]`
(16-byte allocated block, 32-byte free block, 16-byte allocated block, terminator)

`malloc(20)` with first-fit:

1. Search for a free block $\geq$ 24 bytes (20 payload + 4 header)
2. The 32-byte free block fits; split into 24-byte allocated + 8-byte free
3. Result: `[16/1] [24/1] [8/0] [16/1] [0/1]`

After `free()` of the first block (the 16-byte block):

1. Mark as free: `[16/0] [24/1] [8/0] [16/1] [0/1]`
2. With boundary tags, the allocator can check the previous block's footer and the next block's header
3. No adjacent free blocks in this case, so no coalescing occurs

After `free()` of the 24-byte block:

1. Mark as free: `[16/0] [24/0] [8/0] [16/1] [0/1]`
2. Coalesce with the preceding free block (16-byte) and the following free block (8-byte)
3. Result: `[48/0] [16/1] [0/1]` – one large 48-byte free block

Garbage Collection: Mark-and-Sweep

Java example – reachability tracing:

```
class Node { Node next; String name; }

Node a = new Node();   // a -> obj1
Node b = new Node();   // b -> obj2
Node c = new Node();   // c -> obj3
a.next = b;            // obj1.next -> obj2
b.next = c;            // obj2.next -> obj3
b = null;              // b no longer points to obj2
c = null;              // c no longer points to obj3
```

Reachability analysis:

- Roots: `a` (on stack)
- `a` → obj1 (reachable) → obj1.next → obj2 (reachable) → obj2.next → obj3 (reachable)
- All three objects are reachable through `a`, so mark-and-sweep collects **nothing**
- If we also set `a.next = null`, then obj2 and obj3 become unreachable and would be collected

Mark-and-sweep walkthrough on a small graph:
Suppose the heap contains objects {A, B, C, D, E} and roots point to {A, C}. Edges: A → B, C → D. Object E has no incoming edges from reachable objects.

1. **Mark phase:** start from roots. Mark A (root), follow A → B, mark B. Mark C (root), follow C → D, mark D. Marked set = {A, B, C, D}.
2. **Sweep phase:** scan heap. E is unmarked → free E. Clear all marks for next cycle.

Virtual Memory

Address translation example (simplified):
Given: 32-bit virtual addresses, 4 KB pages (12-bit offset), single-level page table.

- VPN = VA[31:12] = top 20 bits; offset = VA[11:0] = bottom 12 bits
- Page table has 2^{20} entries (one per virtual page)
- Each PTE contains: PFN + valid bit + permission bits

Example: translate VA = `0x00003A7F`

1. VPN = `0x00003` (top 20 bits), offset = `0xA7F` (bottom 12 bits)
2. Look up PTE at index 3 in the page table: PFN = `0x0007D`, valid = 1
3. PA = (PFN $\ll$ 12) | offset = `0x0007DA7F`

x86-64 multi-level page table:

- 48-bit virtual addresses (256 TB virtual address space)
- 4 levels: PML4[9 bits] → PDPT[9 bits] → PD[9 bits] → PT[9 bits] → offset[12 bits]

- Advantage: only allocates page table pages for regions that are actually mapped
- With 4 KB pages: each page table level has 512 entries (2^9), each entry is 8 bytes (one page per table)

TLB operation:

1. CPU issues a virtual address
2. TLB lookup: if hit → return PA immediately (fast, ∼1 cycle)
3. TLB miss → page table walk (slow, multiple memory accesses) → if PTE valid, install in TLB and retry; if PTE invalid, page fault
4. Page fault → kernel allocates frame (minor) or reads from disk (major) or sends SIGSEGV (invalid)

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 Dynamic Memory Allocation

13. What is the difference between internal and external fragmentation? Give one example of each.

14. A heap uses 4-byte headers with the allocated bit in the low bit. The heap state is: `[32/1] [64/0] [32/1] [16/0] [0/1]`. What block does first-fit choose for `malloc(50)`? What about best-fit?

15. Why does `free()` need boundary tags (footers) to coalesce in O(1)? What information does the footer provide that the header does not?

16. A program calls `malloc(1)` one million times. Each block needs at least a 4-byte header plus 1 byte of payload, but the allocator aligns to 16 bytes. How much memory is wasted due to internal fragmentation?

4.4 Garbage Collection

17. In mark-and-sweep GC, what are the "roots"? Name three sources of root pointers.

18. Given the following heap graph, which objects does mark-and-sweep collect?

    Roots: {R1, R2}. Objects: {A, B, C, D, E, F}. Edges: R1 → A, A → B, B → C, R2 → D, C → A (cycle). Objects E and F have no incoming edges from reachable objects.

19. True or false: A garbage collector can free an object that the program will never access again, even if a pointer to it still exists on the stack.

20. Explain the generational hypothesis. Why do programs that create many short-lived temporary objects put pressure on the garbage collector?

21. What is the difference between a memory leak in C and a memory leak in Java? Give a concrete example of each.

22. Why does GC need more memory to perform well vs. explicit memory management?

4.5 Virtual Memory

23. Explain the difference between a minor page fault, a major page fault, and an invalid page fault.

24. True or false: The TLB must be flushed on every context switch.

25. Why do modern systems use multi-level page tables instead of a single flat table? For x86-64 with 48-bit virtual addresses and 4 KB pages, how large would a flat page table be?

26. What happens when a process writes to a shared page after `fork()()` under copy-on-write?

27. Briefly compare FIFO, LRU, and Clock page replacement. Which one can exhibit Belady's anomaly?

28. What is a process working set, and why does exceeding physical memory lead to thrashing?

29. A process calls `malloc()` to allocate a 4 KB buffer but does not write to it immediately. What happens when the process first reads from that buffer? What type of page fault occurs?

4.6 Exercises and Project

30. In Exercise 3 (MPSC), what happens when a sender calls `mpsc_send` but the receiver has already been dropped?

31. In Exercise 3, why does `mpsc_sender_clone()` need a reference count? What goes wrong without it?

32. In Project 2, what is "inline helping" (work-assist) in `future_get()`, and why is it necessary for fork-join programs?

33. In Project 2, explain why a work-stealing threadpool scales better than a single shared queue (work-sharing) on many cores.

34. 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 Dynamic Memory Allocation

13. **Internal fragmentation**: wasted space inside an allocated block. Example: a `malloc(1)` call that returns a 16-byte block due to alignment requirements wastes 15 bytes.

    External fragmentation: enough total free memory exists, but it is split into non-contiguous blocks. Example: alternating allocated and free 32-byte blocks – there is plenty of total free memory, but `malloc(64)` cannot be satisfied because no single free block is large enough.

14. For `malloc(50)`: payload 50 + header 4 = 54, aligned up to 56 bytes (assuming 8-byte alignment).

    First-fit: scans from left, finds the 64-byte free block first. Allocates 56 bytes, splits off an 8-byte free block.

    Best-fit: checks all free blocks. The 64-byte block wastes $64 - 56 = 8$ bytes; the 16-byte block is too small ($16 < 56$). Best-fit chooses the 64-byte block (same as first-fit in this case, since the 16-byte block cannot satisfy the request).

15. Without boundary tags, the allocator examining a free block can easily check the *next* block's header (by advancing by the current block's size). But to check the *previous* block, it would need to scan from the beginning of the heap – $O(n)$. Boundary tags (footers) store the previous block's size and allocation status at the end of each block, allowing the allocator to find the previous block's header in O(1) by subtracting the footer's size value.

16. Each 1-byte allocation uses a 16-byte block (4-byte header + 1 byte payload, rounded up to 16 for alignment). Internal fragmentation per block: $16 - 4 - 1 = 11$ bytes wasted. Over one million allocations: $11 \times 10^6 = 11\,\text{MB}$ wasted. (The exact number depends on the allocator's minimum block size and alignment, but the key insight is that small allocations suffer high overhead from headers and alignment padding.)

5.4 Garbage Collection

17. Roots are pointers from which the GC begins its reachability search. Three sources: (1) local variables on thread stacks, (2) global/static variables, (3) CPU registers (which may hold pointers to heap objects).

18. Mark phase: start from roots R1 and R2. R1 → A (mark A), A → B (mark B), B → C (mark C), C → A (already marked, stop). R2 → D (mark D). Marked set = {A, B, C, D}. Sweep phase: E and F are unmarked → **E and F are collected**. Note that the cycle A → B → C → A does not prevent collection – mark-and-sweep correctly handles cycles because it traces from roots, not from individual objects.

19. False. A garbage collector can only free objects that are *unreachable* – i.e., no path from any root leads to them. If a pointer to the object exists on the stack, the object is reachable and cannot be freed, even if the program will never actually use that pointer again. (Some optimizing compilers may eliminate dead references, but the GC cannot know future intent.)

20. The generational hypothesis states that most objects die young – they are allocated, used briefly, and become unreachable. Programs that repeatedly create temporary objects cause GC pressure because those objects fill the young generation quickly. Even though most of them die almost immediately, the collector must still run minor collections to reclaim them. If allocation churn is high enough, the runtime spends a noticeable fraction of time allocating and collecting short-lived objects instead of doing useful work.

21. **C memory leak**: memory that is unreachable (no pointer to it exists anywhere in the program) and was never freed. Example: `char *p = malloc(100); p = NULL;` – the 100 bytes are lost forever because no pointer can reach them and `free()` was never called.

    Java memory leak: memory that is reachable (references still exist) but will never be used by the program. Example: a `HashMap<Integer, byte[]>` used as a cache that keeps growing because entries are never removed. The GC cannot collect the cached byte arrays because the map holds strong references to them.

22. GC benefits from heap headroom because collections happen less frequently when there is more free space. With a larger heap, each collection reclaims more garbage relative to its overhead, and the runtime spends less time pausing to collect. With tight memory, the collector runs much more often and can enter a thrashing regime where it reclaims only small amounts of memory each time.

5.5 Virtual Memory

23. **Minor page fault**: the page is valid but not yet in physical memory (or the PTE is not yet set up). The kernel allocates a physical frame, updates the PTE, and the process resumes – no disk I/O required. Example: first access to a demand-paged region.

    Major page fault: the page must be read from disk (e.g., swapped out or not yet loaded from a memory-mapped file). The process blocks while the page is read from disk.

    Invalid page fault: the virtual address is not mapped in the process's address space (e.g., null pointer dereference, accessing beyond allocated regions). The kernel sends SIGSEGV to the process.

24. In the lecture's simplified model, true. Different processes have different page tables, so switching between processes invalidates the old translations and the TLB must be flushed.

25. A flat (single-level) page table for 48-bit virtual addresses with 4 KB pages would have $2^{48}/2^{12} = 2^{36}$ entries. At 8 bytes per entry: $2^{36} \times 8 = 2^{39} = 512$ GB per process – clearly impractical. Multi-level page tables only allocate pages for regions that are actually mapped. Most of the 256 TB virtual address space is unused, so the intermediate levels do not need to exist, saving enormous amounts of memory.

26. The MMU raises a page fault because the shared page is marked read-only. The kernel recognizes it as a copy-on-write fault, allocates a new frame, copies the old contents, updates the writer's page table entry to point to the new writable frame, and retries the instruction. The other process continues using the original frame.

27. FIFO evicts the oldest page and can exhibit Belady's anomaly. LRU evicts the least recently used page and tracks recent history more accurately, but exact LRU is expensive. Clock approximates LRU using a reference bit and a circular scan, so it is much cheaper than exact LRU in practice.

28. A working set is the set of pages a process actively uses over some time window. If the working set does not fit in RAM, the OS keeps evicting pages that the process will need again soon, causing repeated faults and very little useful work; that is thrashing.

29. The `malloc()` call itself does not necessarily allocate physical memory immediately; the kernel uses demand paging. When the process first reads from the buffer, a **minor page fault** occurs: the virtual page is valid but has no physical frame mapped yet. The kernel allocates a physical frame (typically zero-filled for security), maps it into the process's page table, and the read proceeds. No disk I/O is involved, so it is a minor (not major) fault.

5.6 Exercises and Project

30. `mpsc_send` returns `SEND_ERROR`. The sender checks whether the receiver still exists (via a flag or reference) before attempting to enqueue. Since the receiver has been dropped, the channel is closed and no more messages can be sent.

31. The reference count tracks how many sender handles exist. When all senders are dropped (count reaches 0), the receiver needs to know so it can return `RECV_ERROR` instead of blocking forever waiting for messages that will never arrive. Without a reference count, the receiver has no way to detect that all senders are gone and would block indefinitely.

32. Inline helping means that when a thread calls `future_get()` on a task that has not yet started executing, the calling thread picks up and executes that task itself rather than waiting. This is essential for fork-join programs because a worker thread may submit child tasks and then call `future_get()` on them. If the worker simply blocks, it cannot make progress on the child tasks – and if all workers are blocking on their children, no one is left to execute the children, leading to deadlock. Inline helping ensures that waiting on a future makes forward progress.

33. 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.

34. 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 → 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. **Draw reachability graphs.** Given Java code with object references, draw the heap graph and determine what a GC would collect.
6. **Practice address translation and page replacement.** Given a page table and a virtual address, compute the physical address.
7. **Know copy-on-write.** Be able to explain why `fork()()` is efficient.
8. **Work through malloc examples.** Given a heap state, trace through `malloc()` and `free()` operations with splitting and coalescing.
9. **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
   - Malloc block header layout and coalescing rules
   - Mark-and-sweep algorithm steps
   - Page table address translation formula
   - Page fault/COW flowchart
   - C11 atomic operations summary