Advanced C++ Multithreading Exam: Answers and Corrections

Instructions

This document provides detailed answers and explanations for the Advanced C++ Multithreading Exam. Each section is addressed with corrections, definitions, and code explanations.

—

Section 1: Multiple Choice Questions (Answers)

1. **What is the primary purpose of `std::mutex` in a multi-threaded program?**

   (a) To allow threads to wait for a condition

   (b) **To protect shared resources from concurrent access** (Correct)

   (c) To manage thread execution order

   (d) To signal between threads

   Explanation: `std::mutex` is used to ensure that only one thread can access a shared resource at a time, preventing race conditions.

2. **Which of the following is true about `std::condition_variable`?**

   (a) It can be used without a `std::mutex`

   (b) **It must always be used with a `std::mutex`** (Correct)

   (c) It is used to lock resources

(d) It replaces the need for `std::mutex`

Explanation: `std::condition_variable` requires a `std::mutex` to synchronize access to shared data and to avoid race conditions.

3. **What happens if a thread tries to lock a `std::mutex` that is already locked by another thread?**

   (a) The thread terminates immediately

   (b) The thread continues execution without locking

   (c) **The thread blocks until the mutex is unlocked** (Correct)

   (d) The thread throws an exception

   Explanation: A thread attempting to lock an already locked mutex will block (wait) until the mutex is unlocked by the owning thread.

4. **What is the purpose of `std::queue` in a producer-consumer scenario?**

   (a) To store threads

   (b) **To store shared data between threads** (Correct)

   (c) To lock resources

   (d) To signal threads

   Explanation: `std::queue` is used as a buffer to store data produced by one thread and consumed by another.

5. **Which of the following is true about `std::thread`?**

   (a) **A thread cannot be joined after it has been detached** (Correct)

   (b) A thread must always be joined

   (c) A thread can be both joined and detached

   (d) A thread cannot be detached

   Explanation: Once a thread is detached, it cannot be joined. A thread must be either joined or detached, but not both.

6. **What is the purpose of `std::condition_variable::wait()`?**

   (a) To lock a mutex

   (b) **To wait for a signal while releasing the mutex** (Correct)

   (c) To terminate a thread

   (d) To unlock a mutex

   Explanation: `wait()` releases the mutex and blocks the thread until it is notified by another thread.

7. **What happens if a `std::condition_variable` is signaled but no thread is waiting?**

   (a) **The signal is ignored** (Correct)

   (b) The program crashes

   (c) The signal is stored for the next waiting thread

   (d) An exception is thrown

   Explanation: If no thread is waiting, the signal is lost and has no effect.

8. **Which of the following is true about `std::unique_lock`?**

   (a) It cannot be used with `std::condition_variable`

   (b) **It automatically unlocks the mutex when it goes out of scope** (Correct)

   (c) It is less flexible than `std::lock_guard`

   (d) It cannot be used with `std::mutex`

   Explanation: `std::unique_lock` provides more flexibility than `std::lock_guard` and automatically unlocks the mutex when it goes out of scope.

9. **What is the purpose of `std::condition_variable::notify_all()`?**

   (a) To wake up a single waiting thread

   (b) **To wake up all waiting threads** (Correct)

   (c) To terminate all threads

(d) To lock all mutexes

Explanation: `notify_all()` wakes up all threads waiting on the condition variable.

10. **What is a deadlock in the context of multi-threading?**

    (a) A situation where a thread terminates unexpectedly

    (b) **A situation where two or more threads are blocked forever** (Correct)

    (c) A situation where a mutex is unlocked twice

    (d) A situation where a condition variable is signaled multiple times

    Explanation: Deadlock occurs when two or more threads are waiting for each other to release resources, causing all of them to be blocked indefinitely.

—

Section 2: Code Analysis (Answers)

1. **Analyze the following code and identify any potential issues:**

```
std::mutex mtx;
std::queue<int> queue;
std::condition_variable cv;

void producer() {
    for (int i = 0; i < 10; ++i) {
        std::unique_lock<std::mutex> lock(mtx);
        queue.push(i);
        cv.notify_one();
    }
}

void consumer() {
    while (true) {
        std::unique_lock<std::mutex> lock(mtx);
```

```
            cv.wait(lock);
            int value = queue.front();
            queue.pop();
            std::cout << value << std::endl;
        }
    }
```

Answer: The consumer does not check if the queue is empty before calling `queue.front()` and `queue.pop()`. This can lead to undefined behavior if the queue is empty. Additionally, the consumer runs in an infinite loop without a termination condition.

2. **What is the purpose of the `std::unique_lock` in the above code? Answer:** `std::unique_lock` is used to lock the mutex and ensure thread-safe access to the shared `std::queue`.

3. **What happens if the producer finishes before the consumer starts? Answer:** If the producer finishes before the consumer starts, the consumer will wait indefinitely because there will be no notifications to wake it up.

4. **How can you modify the consumer to handle the case where the queue is empty? Answer:** Modify the consumer to check if the queue is empty after waking up:

```
cv.wait(lock, [&]{ return !queue.empty(); });
```

5. **What is the purpose of `cv.notify_one()` in the producer? Answer:** `cv.notify_one()` wakes up one waiting thread (the consumer) to process the new data added to the queue.

6. **What happens if `cv.notify_all()` is used instead of `cv.notify_one()`? Answer:** If `cv.notify_all()` is used, all waiting threads will be woken up, which may lead to unnecessary contention if only one thread can process the data.

7. **What is the risk of not using `std::unique_lock` in the consumer? Answer:** Without `std::unique_lock`, the mutex will not be locked, leading to race conditions and undefined behavior when accessing the shared queue.

8. **How can you ensure the consumer thread terminates gracefully? Answer:** Add a termination condition, such as a boolean flag, and notify the consumer when the producer is done:

```
bool done = false;
// Producer sets done = true and calls cv.notify_one()
// Consumer checks for done and breaks the loop
```

9. **What is the purpose of the `std::queue` in this code? Answer:** `std::queue` is used as a buffer to store data produced by the producer and consumed by the consumer.

10. **What is the significance of the `std::mutex` in this code? Answer:** `std::mutex` ensures that only one thread can access the shared `std::queue` at a time, preventing race conditions.

—

Section 3: Code Implementation (Answers)

1. **Implement a producer-consumer model using `std::queue`, `std::mutex`, `std::condition_variable`, and `std::thread`. Answer:** See the corrected code in Section 2.

2. **Modify the above implementation to handle multiple producers and consumers. Answer:** Use multiple threads for producers and consumers, and ensure proper synchronization with `std::mutex` and `std::condition_variable`.

3. **Add a mechanism to gracefully shut down the consumer threads. Answer:** Use a boolean flag (e.g., `done`) and notify all consumers when the flag is set.

4. **Implement a thread-safe queue using `std::queue`, `std::mutex`, and `std::condition_variable`. Answer:** Wrap `std::queue` with `std::mutex` and `std::condition_variable` to ensure thread-safe operations.

5. **Write a program where two threads increment a shared counter using `std::mutex` for synchronization. Answer:** Use `std::mutex` to protect the shared counter and ensure atomic increments.

6. **Write a program where a thread waits for a signal from another thread using `std::condition_variable`. Answer:** Use `std::condition_variable` to synchronize the threads and signal when the condition is met.

7. **Implement a barrier synchronization mechanism using `std::mutex` and `std::condition_variable`. Answer:** Use a counter and `std::condition_variable` to block threads until all threads reach the barrier.

8. **Write a program to demonstrate a deadlock scenario involving two threads and two mutexes. Answer:** Create two threads that lock two mutexes in opposite orders, causing a deadlock.

9. **Fix the deadlock in the above program. Answer:** Ensure both threads lock the mutexes in the same order.

10. **Write a program to demonstrate the use of `std::async` with `std::mutex` and `std::condition_variable`. Answer:** Use `std::async` to launch tasks and synchronize them using `std::mutex` and `std::condition_variable`.

—

Section 4: Theoretical Questions (Answers)

1. **Explain the difference between `std::mutex` and `std::recursive_mutex`. Answer:** `std::mutex` cannot be locked multiple times by the same thread, while `std::recursive_mutex` can.

2. **What is spurious wakeup, and how can you handle it in `std::condition_variable`? Answer:** Spurious wakeup is when a thread wakes up without being notified. Handle it by using a predicate in `wait()`.

3. **Explain the difference between `std::lock_guard` and `std::unique_lock`.** **Answer:** `std::lock_guard` is simpler and cannot be unlocked manually, while `std::unique_lock` is more flexible and can be unlocked manually.

4. **What is the purpose of `std::condition_variable::wait_for()`?** **Answer:** `wait_for()` allows a thread to wait for a condition for a specified duration.

5. **Explain the concept of thread safety and how it applies to `std::queue`.** **Answer:** Thread safety ensures that shared data is accessed in a way that prevents race conditions. `std::queue` is not thread-safe by default and requires synchronization.

6. **What is the difference between `std::thread::join()` and `std::thread::detach()`?** **Answer:** `join()` waits for the thread to finish, while `detach()` allows the thread to run independently.

7. **Explain the concept of a race condition and how `std::mutex` prevents it.** **Answer:** A race condition occurs when multiple threads access shared data concurrently, leading to undefined behavior. `std::mutex` ensures only one thread accesses the data at a time.

8. **What is the purpose of `std::atomic` in multi-threading?** **Answer:** `std::atomic` ensures that operations on shared variables are performed atomically, without the need for a mutex.

9. **Explain the difference between `std::condition_variable` and `std::future`.** **Answer:** `std::condition_variable` is used for thread synchronization, while `std::future` is used to retrieve the result of an asynchronous operation.

10. **What is the role of the C++ memory model in multi-threading?** **Answer:** The C++ memory model defines how threads interact with memory, ensuring proper synchronization and visibility of shared data.