Execution Order 1: Thread A completes, followed by B, then C

1. **0.0, A, 4**: Thread `A` locks the mutex.
2. **0.0, A, 6 [count → 1]**: Thread `A` increments `count` to 1.
3. **0.0, A, 7**: `A` checks `count` and waits since `count < 3`. Thread `A` releases the mutex and waits on condition `C`.
4. **1.0, B, 4**: Thread `B` locks the mutex.
5. **1.0, B, 6 [count → 2]**: Thread `B` increments `count` to 2.
6. **1.0, B, 7**: `B` checks `count` and waits since `count < 3`. Thread `B` releases the mutex and waits on condition `C`.
7. **1.0, C, 4**: Thread `C` locks the mutex.
8. **1.0, C, 6 [count → 3]**: Thread `C` increments `count` to 3.
9. **1.0, C, 7.2**: Since `count == 3`, thread `C` signals condition `C` to wake up `A` and `B`.
10. **1.0, C, 8 [count → 0]**: Thread `C` resets `count` to 0.
11. **1.0, C, 9**: Thread `C` unlocks the mutex and exits.
12. **1.0, A, 8 [count → 0]**: Thread `A` wakes up, resets `count` to 0 (it was already reset by `C`).
13. **1.0, A, 9**: Thread `A` unlocks the mutex and exits.
14. **1.0, B, 8 [count → 0]**: Thread `B` wakes up, resets `count` to 0 (it was already reset by `C`).
15. **1.0, B, 9**: Thread `B` unlocks the mutex and exits.

Execution Order 2: Thread B completes first, followed by A, then C

1. **0.0, A, 4**: Thread `A` locks the mutex.
2. **0.0, A, 6 [count → 1]**: Thread `A` increments `count` to 1.
3. **0.0, A, 7**: `A` checks `count` and waits since `count < 3`. Thread `A` releases the mutex and waits on condition `C`.
4. **1.0, B, 4**: Thread `B` locks the mutex.
5. **1.0, B, 6 [count → 2]**: Thread `B` increments `count` to 2.
6. **1.0, B, 7**: `B` checks `count` and waits since `count < 3`. Thread `B` releases the mutex and waits on condition `C`.
7. **1.0, C, 4**: Thread `C` locks the mutex.
8. **1.0, C, 6 [count → 3]**: Thread `C` increments `count` to 3.
9. **1.0, C, 7.2**: Since `count == 3`, thread `C` signals condition `C` to wake up `A` and `B`.
10. **1.0, B, 8 [count → 0]**: Thread `B` wakes up and resets `count` to 0.
11. **1.0, B, 9**: Thread `B` unlocks the mutex and exits.

12. **1.0, A, 8 [count → 0]**: Thread `A` wakes up and resets `count` to 0 (it was already reset by `B`).
13. **1.0, A, 9**: Thread `A` unlocks the mutex and exits.
14. **1.0, C, 8 [count → 0]**: Thread `C` resets `count` to 0 (it was already reset by `B`).
15. **1.0, C, 9**: Thread `C` unlocks the mutex and exits.

Execution Order 3: Thread C completes first, followed by B, then A

1. **0.0, A, 4**: Thread `A` locks the mutex.
2. **0.0, A, 6 [count → 1]**: Thread `A` increments `count` to 1.
3. **0.0, A, 7**: `A` checks `count` and waits since `count < 3`. Thread `A` releases the mutex and waits on condition `C`.
4. **1.0, B, 4**: Thread `B` locks the mutex.
5. **1.0, B, 6 [count → 2]**: Thread `B` increments `count` to 2.
6. **1.0, B, 7**: `B` checks `count` and waits since `count < 3`. Thread `B` releases the mutex and waits on condition `C`.
7. **1.0, C, 4**: Thread `C` locks the mutex.
8. **1.0, C, 6 [count → 3]**: Thread `C` increments `count` to 3.
9. **1.0, C, 7.2**: Since `count == 3`, thread `C` signals condition `C` to wake up `A` and `B`.
10. **1.0, C, 8 [count → 0]**: Thread `C` resets `count` to 0.
11. **1.0, C, 9**: Thread `C` unlocks the mutex and exits.
12. **1.0, B, 8 [count → 0]**: Thread `B` wakes up and resets `count` to 0.
13. **1.0, B, 9**: Thread `B` unlocks the mutex and exits.
14. **1.0, A, 8 [count → 0]**: Thread `A` wakes up and resets `count` to 0 (it was already reset by `B`).
15. **1.0, A, 9**: Thread `A` unlocks the mutex and exits.

Summary of Possible Execution Orders:

1. **Order 1**: Thread A finishes first, followed by B and C.
2. **Order 2**: Thread B finishes first, followed by A and C.
3. **Order 3**: Thread C finishes first, followed by B and A.

In all cases:

- The shared `count` variable is correctly incremented to 3 and reset to 0.
- The mutex ensures that the threads operate on `count` in a consistent manner, and the condition variable manages the waking up of threads.