- What are a simple sequence of instructions that can cause a deadlock?

Thread1:

Get lock(m1)

Get lock(m2)

unlock(m2)

unlock(m1)

Thread2:

Get lock(m2)

Get lock(m1)

unlock(m1)

unlock(m2)

If these threads run concurrently and get through the first lines, then neither thread can continue because they will try to obtain locks that are already held by another thread

- What is *livelock*?

A livelock is where two threads can both repeatedly attempt a sequence while repeatedly failing to acquire both locks. This means that it is not a deadlock, because it keeps running but because it is not not making progress it is called a livelock.

- What is the difference between *partial ordering* and *total ordering*?

Total ordering relies on always ordering locks in certain ways to avoid deadlocks, partial ordering can be useful in more complex ways, when needing to place locks so as not to cause a deadlock, but not placing everylock when they aren't needed and other locks to make it more clear about what is happening where.

- What are more common, deadlock bugs or non-deadlock bugs?

Deadlock Bugs:

Mutual Exclusion: One thread holds control of a resource (e.g. via a mutex) and no other threads can use it while they have it.

No Preemption: If a thread is waiting, there's no external way to break it out of the wait. Or no way for the waiting thread to break itself out of the wait.

Circular Wait: Each thread is waiting on a resource the next thread has. And the last thread is waiting on a resource the first thread has. Classic Dining Philosopher deadlock.

Non-Deadlock Bugs:

Atomicity Violation: When you have sequence of instructions that cannot be interrupted in the middle, but they aren't wrapped in a mutex.

Order Violation: When two actions must occur in a specific order, but two threads execute them in the wrong order. This can be fixed with condition variables.

- Describe step-by-step how condition variables can be used to prevent order violations.
    1. Identify the shared resource
    2. Define a lock to ensure only one thread can access the resource at a time
    3. Identify the events that need to be syncronised, such as consumption and production events
    4. Define the condition variable for each event
    5. In a producing thread acquire a mutex lock and perform the production operation
    6. In a consuming thread acquired the lock and perform the consumption operation
    7. Once the production thread signals data is available, wake the consumer to re-acquire the mutex lock and consume the data.

- What are some methods of *deadlock avoidance*?

Instead of outright prevention sometime deadlock avoidence is preferable, avoidance requires some global knowledge of which locks threads might get during execution and subsequently schedules threads to guarantee no deadlock can occur. Static scheduling leads to a conservative approach where some threads run on the same processor, so the total time to complete jobs is lengthened considerably.

Another method is to allow deadlocks to occasionally occur, and then take some action when a deadlock is detected, this is apart of the detect and recover method. If deadlocks are rare, then we can just reset the process and continue, for example if there is a deadlock that can occur once a year, just resetting can be easier than working around the issue, such a non-solution can be practical in this instance.