1.

Busy waiting occurs when a process continuously checks for a condition to be true (usually in a loop), without releasing the CPU, thus "wasting" computational resources. It keeps the CPU busy while waiting for some event or condition. An example would be constantly checking if a lock is available rather than sleeping until the lock is free.

No, busy waiting is generally not beneficial for system performance because it consumes CPU cycles unnecessarily while waiting. The system could perform other tasks instead of just waiting in a loop, leading to inefficiency, especially in multi-core systems with limited CPU resources.

A semaphore can avoid busy waiting by putting the process to sleep until the required condition is met (i.e., when the semaphore value is incremented). Instead of a process continuously checking if the semaphore is available, the process is blocked, and the CPU is freed up for other processes. When the semaphore becomes available, the waiting process is awakened to continue.

2.

A race condition is possible when multiple processes access shared resources concurrently and the outcome depends on the non-deterministic order in which the processes execute. In the auction system example, if two users call bid(amount) at the same time with higher bids, both might read the highestBid as the current value before either can update it. This leads to an incorrect highest bid being recorded, because the bid logic doesn’t ensure mutual exclusion when updating highestBid.

We can prevent the race condition, by protecting the bid function for the synchronization mechanisms like locks or atomic operations. For example, a mutex or a semaphore can be used

to ensure that only one thread can access and update highestBid at a time, thus avoiding the condition where both bids are accepted at the same time.

3.

a. Lock Held for a Short Duration:

In this case, a spinlock is better because the lock is only held for a short time. The cost of the spinning process (checking the lock repeatedly) is minimal, so it’s more efficient to "spin" rather than putting the process to sleep, which would involve more overhead for waking up the process later.

b. Lock Held for a Long Duration:

In this case, a mutex lock is better because the lock will be held for a longer time. Using a spinlock would waste CPU resources while other processes are waiting. With a mutex, the waiting processes are put to sleep, and the CPU can be freed for other tasks.

4.

This results in a liveness failure because the process enters the critical section and then waits again on the same mutex before exiting. If another process is trying to acquire the mutex, it will be blocked because the process never releases the mutex (signal(mutex) is missing). This leads to deadlock, as the process is waiting for itself to release the lock, which will never happen.

5.

Deadlock in Dining Philosophers: Deadlock can occur in the Dining Philosophers problem if each philosopher picks up one fork and then waits for the other. If all philosophers

simultaneously pick up the fork on their right, they are all waiting for the fork on their left, which is being held by another philosopher. No philosopher can proceed, resulting in a deadlock where all processes are blocked and unable to make progress.

A possible solution to this problem involves using a lock hierarchy or using a timeout mechanism to avoid indefinite blocking.

6.

Signaled State:

In the signaled state, the dispatcher object indicates that a condition has been met or that the resource is available. A thread waiting for the object in a signaled state will continue execution.

Non-Signaled State:

In the non-signaled state, the condition has not been met, and the resource is not available. A thread that is waiting for the object in a non-signaled state will be blocked (i.e., put to sleep) until the state changes to signaled.

7.

1. atomic set(&val, 10);

   This sets val to 10. Now, val = 10.
2. atomic sub(8, &val);

   This subtracts 8 from val. Now, val = 10 - 8 = 2.
3. atomic inc(&val);

   This increments val by 1. Now, val = 2 + 1 = 3.

4. atomic inc(&val);

   This increments val by 1 again. Now, val = 3 + 1 = 4.

5. atomic add(6, &val);

   This adds 6 to val. Now, val = 4 + 6 = 10.

6. atomic sub(3, &val);

   This subtracts 3 from val. Now, val = 10 - 3 = 7.

Final Value of val: 7.