**7.2**

（1）

**Mutual Exclusion:** At least one resource must be held in a non-shareable mode. In the dining philosophers problem, this condition is satisfied as each chopstick can only be used by one philosopher at a time.

**Hold and Wait:** A process must be holding at least one resource and waiting to acquire additional resources held by other processes. In the dining philosophers problem, each philosopher holds one chopstick and may wait for another one, leading to potential deadlock.

**No Preemption:** Resources cannot be preempted. Once a philosopher holds a chopstick, it cannot be taken away until the philosopher releases it voluntarily. This lack of preemption can contribute to deadlock if philosophers are holding onto their chopsticks without releasing them.

**Circular Wait:** A circular chain of processes, where each process is waiting for a resource held by the next process in the chain. In the dining philosophers problem, if each philosopher holds one chopstick and waits for the next one, a circular wait condition can arise.

（2）

**Mutual Exclusion:**

Avoidance: Allow multiple philosophers to share a chopstick. This eliminates the exclusivity of chopsticks and prevents deadlock.

**Hold and Wait:**

Avoidance: Introduce a protocol where philosophers must pick up both chopsticks simultaneously or none at all. If a philosopher cannot obtain both chopsticks, they must release the one they have. This ensures that a philosopher only holds resources when all needed resources are available.

**No Preemption:**

Avoidance: Introduce a mechanism for preemption, allowing a philosopher to release a chopstick forcibly if it is holding onto it for too long.

**Circular Wait:**

Avoidance: Establish a hierarchy for acquiring chopsticks. For example, require philosophers to always pick up the left chopstick before the right one. This prevents circular wait conditions.

**7.6**

If the system enters a deadlock, it means that each process holds one resource and is waiting for another. Since there are three processes and four resources in the system, one process must acquire two resources. As this process releases its resources upon completion and does not require additional resources, it ensures that the system does not remain in a deadlock state indefinitely.

**7.7**

Suppose there exists a situation of deadlock. In this scenario, if deadlock occurs, then sum(Allocate) = m. However, as per the given condition, sum(Max) < m + n. Additionally, sum(Max) = sum(Need) + sum(Allocate). Therefore, sum(Need) < n. Consequently, it is certain that there must be a processor with a need of 0. Therefore, this processor can execute and release resources. Consequently, the assumed condition is not valid.