✅ Q1. What is a lock? Why do we use locks in multithreaded programs?

✅ Answer:

A **lock** is a synchronization mechanism used to **prevent multiple threads** from accessing a **critical section (shared resource)** at the same time.

- It ensures **mutual exclusion** (only one thread holds the lock).
- Helps **prevent race conditions**.

✅ Q2. How do we build a lock using `Test-And-Set`? Explain with `acquire()` and `release()` functions.

✅ Answer:

🔹 `Test-And-Set` Pseudocode:

c

CopyEdit

```
int TestAndSet(int *ptr, int new) {
    int old = *ptr;
    *ptr = new;
    return old;
}
```

🔹 Lock Structure and Functions:

c

CopyEdit

```
typedef struct {
    int flag;  // 0 = free, 1 = locked
} lock_t;

void init(lock_t *lock) {
    lock->flag = 0;
}

void acquire(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1); // spin-wait
}

void release(lock_t *lock) {
    lock->flag = 0;
}
```

✅ Q3. Why do we need to initialize the lock?

✅ Answer:

Initialization sets `lock->flag = 0`, meaning the lock is **available**.
Without this, it may be in an unknown or busy state, preventing threads from entering the critical section.

✅ Q4. What is spinning or busy waiting?

✅ Answer:

Spinning means a thread keeps **looping (spinning)** while checking if the lock is available.
It **consumes CPU** but avoids thread switching overhead.
Example:

c

CopyEdit

```
while (TestAndSet(&lock->flag, 1) == 1); // spinning
```

✅ Q5. What is fairness in locking? How do ticket locks provide fairness?

✅ Answer:

Fairness ensures threads get the lock in **order of arrival** (like a ticket system).

Ticket Lock:

c

CopyEdit

```
typedef struct {
    int ticket;
    int turn;
} ticket_lock_t;

void acquire(ticket_lock_t *lock) {
```

```
    int my = FetchAndAdd(&lock->ticket, 1);
    while (lock->turn != my); // wait until your turn
}

void release(ticket_lock_t *lock) {
    lock->turn++;
}
```

- 🔹 Each thread **gets a ticket** and **waits** for its **turn**, ensuring fairness.

✅ Q6. What is the concept of `load-linked` (LL) and `store-conditional` (SC)?

✅ Answer:

LL/SC is an **alternative to Test-And-Set**, used in RISC architectures.

- `load-linked`: Reads value and **marks memory address**.
- `store-conditional`: Writes **only if** no other write has occurred at that address.

It allows atomicity **without a lock** using two instructions.

✅ Q7. What is Fetch-And-Add used for in locks?

✅ Answer:

It is used in **ticket locks** to assign **unique ticket numbers** atomically.
Each thread calls `FetchAndAdd(&ticket, 1)` and waits until `turn == my_ticket`.

✅ Q8. What is the difference between spinning and parking in locking?

✅ Answer:

Concept	Spinning	Parking
Behavior	CPU continuously checks lock	Thread **sleeps** (doesn't use CPU)
Efficiency	Wastes CPU	CPU efficient
Use Case	Short waits	Long waits
Wake-up	Thread checks repeatedly	Thread is **woken up** by signal

---

✅ Q9. How does `yield()` improve spinlocks?

✅ Answer:

Using `yield()` inside the spin loop:

c

CopyEdit

```
while (TestAndSet(&lock->flag, 1) == 1) {
    yield(); // Give up CPU to others
}
```

It prevents **CPU hogging** by letting other threads run — useful in **single-CPU systems**.

✅ Q10. How would an exam question look like?

> **Q:** You are given the `TestAndSet()` function. Write the full code to build a spin lock using it. Show initialization, `acquire()`, and `release()` functions. Also, explain the concept of fairness and how ticket locks solve starvation.

✅ Summary Table (Memory Aid)

Concept	Meaning
Lock	Prevents concurrent access
Init	Sets flag = 0
Acquire	Uses TestAndSet or ticket wait
Release	Sets flag = 0 or increases turn
Spinning	Looping to check lock
Parking	Sleep until lock is available
Fairness	Turn-based locking like ticket lock

TAS	Atomic test and update
LL/SC	Load memory and store if unchanged
Fetch-And-Add	Used in ticket locks for fairness