Quiz #4

June 9, 2025

Section: 1 or 2 **Student ID:** **Name:**

Q1. What is a race condition? (1point)

A situation, where several processes access and manipulate the same data concurrently. The outcome of execution depends on the particular order in which the access takes place.

Q2. What does fairness mean in the context of lock design, and why is it an important requirement? (1point)

- **Fairness ensures that threads acquire a lock in the same order they requested it, preventing any thread from being indefinitely delayed (or starved).**
- **Why important?**
  Without fairness, some threads might never get the lock if others keep acquiring it first. Fairness guarantees that every thread will eventually get its turn.
  Fair locks provide more deterministic and testable behavior, which is important for real-time or safety-critical systems.
  It prevents thread monopolization, which can degrade system performance or cause delays in critical tasks.

Q3. Implement the *lock()* and *unlock()* functions using the atomic *CompareAndSwap()* primitive. The *lock()* function should ensure mutual exclusion, but does not need to satisfy other requirements for lock implementation. (1point)

```
typedef struct __lock_t {
    int flag;
} lock_t;

int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;
    return actual;
}

void init(lock_t *lock) {
    // 0 indicates that lock is available, 1 that it is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
        ;
}

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

}
```

Q4. Complete the implementation of a linked list that supports concurrent access using a *mutex*. (1point)

```
// Linked list node structure
typedef struct __node_t {
    int value;
    struct __node_t *next;
} node_t;

// Basic list structure (one used per list)
typedef struct __list_t {
    node_t *head;
    pthread_mutex_t lock;
} list_t;

void List_Init(list_t *L) {
    L->head = NULL;
    pthread_mutex_init(&L->lock, NULL);
}

int List_Insert(list_t *L, int value) {
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) {
        perror("malloc");
        return -1;
    }
    new->value = value;
    pthread_mutex_lock(&L->lock);
    new->next = L->head;
    L->head = new;
    pthread_mutex_unlock(&L->lock);
    return 0;
}

int List_Lookup(list_t *L, int value) {
    int rv = -1;
    pthread_mutex_lock(&L->lock);
    node_t *curr = L->head;
    while (curr) {
        if (curr->value == value) {
            rv = 0;
            break;
        }
        curr = curr->next;
    }
    pthread_mutex_unlock(&L->lock);
    return rv;
}
```

Q5. The *main()* function creates two child threads. Each thread should print "`child 1`" and "`child 2`" respectively. However, "`child 1`" must always execute before "`child 2`", regardless of thread scheduling. After both child threads have finished, the parent should print "`parent: end`". Modify the following code to enforce this printing order using semaphores. Do not use *pthread_join()*. (1point)

*** Expected results**

```
parent: begin
child 1
child 2
parent: end
```

* Code

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

sem_t sem1;  // For child1 → child2
sem_t sem2;  // For child2 → parent

void* child1(void* arg) {
    printf("child 1\n");
    sem_post(&sem1);  // Signal to child2
    return NULL;
}

void* child2(void* arg) {
    sem_wait(&sem1);  // Wait for child1
    printf("child 2\n");
    sem_post(&sem2);  // Signal to parent
    return NULL;
}

int main() {
    pthread_t t1, t2;

    sem_init(&sem1, 0, 0);  // child2 waits initially
    sem_init(&sem2, 0, 0);  // parent waits initially

    printf("parent: begin\n");

    pthread_create(&t1, NULL, child1, NULL);
    pthread_create(&t2, NULL, child2, NULL);

    sem_wait(&sem2);     // Wait until child2 finishes

    printf("parent: end\n");

    sem_destroy(&sem1);  // can be skipped
    sem_destroy(&sem2);

    return 0;
}
```