7 Experiment 7: Process Synchronization using Semaphore/-Mutex

7.1 Objective

The objective of this lab experiment is to acquaint students with the concept of process synchronization in concurrent programming using semaphore or mutex mechanisms.

7.2 Reading Material[1][3][2]

7.2.1 Synchronization

Process or thread synchronization refers to the coordination and orderly execution of concurrent processes or threads in a multi-threaded or multi-process system.

7.2.2 Race Condition

A race condition is a situation in concurrent programming where the outcome of the program depends on the order of execution of threads or processes. It arises when multiple threads or processes access shared resources or critical sections without proper synchronization or coordination, leading to unpredictable or incorrect behavior.

7.2.3 Semaphore

Definition	A semaphore is an abstract data type used for process synchronization in concurrent programming. It controls access to shared resources among multiple processes or threads by maintaining a counter that can be incremented or decremented.
Functionality	Semaphores manage access to shared resources, prevent race conditions, and ensure synchronization. They offer operations like initialization (`sem_init`), waiting (`sem_wait`), signaling (`sem_post`), and destruction (`sem_destroy`).
Types	Common types include Binary Semaphores (with values 0 and 1) and Counting Semaphores (with values greater than 1).

Function	Description	Programming Syntax
`sem_init`	Initialize a semaphore	`int sem_init(sem_t *sem, int pshared, unsigned int value);`
`sem_destroy`	Destroy a semaphore	`int sem_destroy(sem_t *sem);`
`sem_post`	Increment (signal) a semaphore	`int sem_post(sem_t *sem);`
`sem_wait`	Decrement (wait/block) a semaphore	`int sem_wait(sem_t *sem);`

Table 6: Library Functions Related to Semaphore

7.2.4 Mutex

Definition	A mutex (Mutual Exclusion) is a synchronization primitive used in multi-threaded programming to control access to shared resources. It allows only one thread at a time to access the resource, preventing concurrent access.
Functionality	Mutexes ensure mutual exclusion, preventing race conditions and maintaining data integrity. Operations include initialization (`pthread_mutex_init`), locking (`pthread_mutex_lock`), unlocking (`pthread_mutex_unlock`), and destruction (`pthread_mutex_destroy`).
Types	Mutexes can be recursive (allows the same thread to lock it multiple times) or non-recursive (deadlocks if the same thread tries to lock it multiple times).

Function	Description	Programming Syntax
`pthread_mutex_init`	Initialize a mutex	`int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);`
`pthread_mutex_destroy`	Destroy a mutex	`int pthread_mutex_destroy(pthread_mutex_t *mutex);`
`pthread_mutex_lock`	Lock a mutex	`int pthread_mutex_lock(pthread_mutex_t *mutex);`
`pthread_mutex_unlock`	Unlock a mutex	`int pthread_mutex_unlock(pthread_mutex_t *mutex);`

Table 7: Library Functions Related to Mutex

7.3 Sample Programs

1. C Program Simulating Race Condition.

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

#define NUM_THREADS 5
#define MAX_COUNT 1000000

int shared_variable = 0;

void *increment_variable(void *thread_id) {
    for (int i = 0; i < MAX_COUNT; i++) {
        shared_variable++;
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
```

```
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, increment_variable, (void *)i);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Value of shared variable after race condition: %d\n", shared_variable);

    return 0;
}
```

2. C Program with Semaphore to Avoid Race Condition.

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

#define NUM_THREADS 5
#define MAX_COUNT 1000000

int shared_variable = 0;
sem_t semaphore;

void *increment_variable(void *thread_id) {
    for (int i = 0; i < MAX_COUNT; i++) {
        sem_wait(&semaphore);
        shared_variable++;
        sem_post(&semaphore);
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    sem_init(&semaphore, 0, 1); // Initializing semaphore with value 1

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, increment_variable, (void *)i);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
```

```
    sem_destroy(&semaphore); // Destroying semaphore

    printf("Value of shared variable after synchronization: %d\n", shared_variable);

    return 0;
}
```

3. C Program with Mutex to Prevent Race Condition

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

#define NUM_THREADS 5
#define MAX_COUNT 1000000

int shared_variable = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *increment_variable(void *thread_id) {
    for (int i = 0; i < MAX_COUNT; i++) {
        pthread_mutex_lock(&mutex);
        shared_variable++;
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    pthread_mutex_init(&mutex, NULL); // Initializing mutex

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, increment_variable, (void *)i);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_mutex_destroy(&mutex); // Destroying mutex

    printf("Value of shared variable after synchronization: %d\n", shared_variable);

    return 0;
}
```

Video Reference:

https://youtube.com/playlist?list=PLWjmN065fOfGdAZrlP6316HVHh8jlve_

References

[1] Greg Gagne Abraham Silberschatz, Peter B. Galvin. *Operating System Concepts.* Wiley, 10 edition, 2018.

[2] Sumitabha Das. *Unix Concepts And Applications.* Wiley, 4 edition, 2006.

[3] Richard Stones Neil Matthew. *Beginning Linux Programming.* Wiley, 4 edition, 2007.

7.4 Lab Exercises

Exercise 1: Implement the producer-consumer problem using pthreads and mutex operations.
Constraints:

(a) A producer only produces if the buffer is empty, and the consumer only consumes if some content is in the buffer.

(b) A producer writes an item into the buffer, and the consumer deletes the last produced item in the buffer.

(c) A producer writes on the last consumed index of the buffer.

Exercise 2: Implement the reader-writer problem using semaphore and mutex operations to synchronize n readers active in the reader section at the same time and one writer active at a time.
Constraints:

(a) If n readers are active, no writer is allowed to write.

(b) If one writer is writing, no other writer should be allowed to read or write on the shared variable.

7.5 Sample Viva Questions

Question 1: What is a semaphore, and how does it facilitate synchronization among processes or threads?

Question 2: Discuss the functions `sem_init`, `sem_wait`, and `sem_post` in semaphore usage.

Question 3: Compare and contrast mutexes with semaphores in terms of functionality and usage.