Experiment no: 7

Name: Saloni Vishwakarma
Batch-Roll no: C1-13
Semester-Section: 4th-C
Subject: OS(Operating System) Lab
Date of execution: 21 June 2023

Aim: Write C programs to implement threads and semaphores for process synchronization.
A) To demonstrate multi- threading for implementing linear search in a C program.
B) To demonstrate Producer- Consumer Problem using semaphore.

Theory:
1) Multithreading: Multithreading is a concept in operating systems that allows for the execution of multiple threads within a single process. A thread is a sequence of instructions that can be scheduled and executed by the CPU independently. By enabling multiple threads, the operating system enables concurrent execution, where different parts of a program can run simultaneously.

Multithreading offers several benefits. It can improve the responsiveness of applications by allowing tasks to be executed concurrently. For example, in a web browser, one thread can handle user input while another thread loads a webpage. Multithreading can also enhance performance by utilizing multiple CPU cores.

To ensure proper behavior, synchronization mechanisms are used in multithreading. These mechanisms, such as locks, mutexes, and semaphores, coordinate access to shared resources among threads. They prevent race conditions, where multiple threads access shared data simultaneously and produce incorrect results. Synchronization also helps in enforcing mutual exclusion, where only one thread can access a resource at a time.

Threads can communicate with each other through shared memory or message passing. Shared memory allows threads to access common data structures, but synchronization is needed to avoid conflicts. Message passing involves threads exchanging messages to share information.

The operating system scheduler manages the execution of threads. It determines which thread to run and when, based on scheduling algorithms. These algorithms aim to provide fairness, efficiency, and responsiveness in allocating CPU time to threads.

2) Semaphores: Semaphores in operating systems are synchronization mechanisms that control access to shared resources and coordinate the execution of multiple processes or threads. They are represented as variables that can be accessed and modified by different entities concurrently.

Semaphores support two essential operations: "wait" (also known as "P" or "down") and "signal" (also known as "V" or "up"). The "wait" operation decrements the semaphore value and may cause the calling entity to block if the value becomes zero, indicating unavailability of the resource. The "signal" operation increments the semaphore value and notifies waiting entities that the resource is now available.

Semaphores can be categorized as binary semaphores or count-based semaphores. Binary semaphores have values of either 0 or 1 and are commonly used for mutual exclusion, allowing only one entity to access a resource at a time. Count-based semaphores have non-negative integer values and are useful for scenarios where multiple entities can access a resource simultaneously up to a specified limit.

Semaphores are utilized in various synchronization scenarios, such as coordinating access to shared data structures, preventing race conditions, implementing producer-consumer patterns, managing critical sections, and controlling resource allocation. They help ensure that processes or threads coordinate their actions properly and avoid conflicts when accessing shared resources.

Viva Questions:

1) What is multithreading?
→ Multithreading is a feature of operating systems that allows the execution of multiple threads within a single process concurrently.

2) What are the benefits of multithreading?
→ Multithreading offers advantages such as improved responsiveness, enhanced performance through parallel execution, and efficient utilization of system resources.

3) What are some challenges in multithreading?

→ Challenges in multithreading include proper synchronization to avoid race conditions and ensure data integrity, thread coordination and communication, and potential debugging complexities.

4) Define a semaphore.
→ A semaphore is a synchronization construct used in operating systems to control access to shared resources and coordinate the execution of multiple processes or threads.

5) What are the two fundamental operations supported by semaphores?
→ The two fundamental operations supported by semaphores are "wait" (P or down) and "signal" (V or up). The "wait" operation decreases the value of the semaphore and potentially blocks the process or thread if the value reaches zero. The "signal" operation increases the value of the semaphore and notifies waiting processes or threads if necessary.

6) What are the types of semaphores?
→ Semaphores can be classified as binary semaphores or count-based semaphores. Binary semaphores have values of either 0 or 1 and are used for mutual exclusion. Count-based semaphores have non-negative integer values and allow multiple entities to access a resource up to a specified limit.

7) What are the main applications of semaphores?
→ Semaphores find applications in coordinating access to shared data structures, preventing race conditions, implementing producer-consumer patterns, managing critical sections, and enforcing synchronization in concurrent systems.

8) How does synchronization play a role in multithreading and semaphore usage?
→ Synchronization mechanisms, such as locks, mutexes, and semaphores, are essential for ensuring correct and coordinated access to shared resources. They prevent race conditions, data corruption, and conflicts when multiple threads or processes access shared resources simultaneously.

9) How does the operating system scheduler manage multithreaded execution?

→ The operating system scheduler determines which thread or process to run and when, based on scheduling algorithms. It aims to provide fairness, efficiency, and responsiveness in allocating CPU time to threads or processes.

10) What are some potential issues in multithreading and semaphore usage?
→ Some potential issues include deadlocks (when threads or processes are stuck waiting for each other to release resources), starvation (continuous deprivation of necessary resources), livelocks (active but unproductive threads due to excessive coordination), and priority inversion (low-priority thread holding a resource needed by a high-priority one).

Code and Output:

A) Linear Search using multi-threading:

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

#define MAX_SIZE 100
#define NUM_THREADS 4

int arr[MAX_SIZE];
int size;
int target;
int found = 0;

pthread_mutex_t mutex;

void *linearSearch(void *arg) {
    int thread_id = *(int *)arg;
    int start = thread_id * (size / NUM_THREADS);
    int end = start + (size / NUM_THREADS);

    for (int i = start; i < end; i++) {
        if (arr[i] == target) {
            pthread_mutex_lock(&mutex);
            found = 1;
            pthread_mutex_unlock(&mutex);
            break;
        }
    }
```

```
    pthread_exit(NULL);
}

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

    pthread_mutex_init(&mutex, NULL);

    printf("\n Enter the size of the array: ");
    scanf("%d", &size);

    printf("\n Enter the array elements: ");
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }

    printf("\n Enter the element to be searched: ");
    scanf("%d", &target);

    // Create threads
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, linearSearch, &thread_ids[i]);
    }

    // Join threads
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    pthread_mutex_destroy(&mutex);

    // Check if target was found
    if (found) {
        printf("\n Element (%d) found in the array.\n",target);
    } else {
        printf("\n Element (%d) not found in the array.\n",target);
    }
    return 0;
```

}

```
Enter the size of the array: 6

Enter the array elements: 3 7 6 2 9 61

Enter the element to be searched: 6

Element (6) found in the array.


...Program finished with exit code 0
Press ENTER to exit console.
```

```
Enter the size of the array: 8

Enter the array elements: 67 54 83 92 78 1 11 29

Enter the element to be searched: 30

Element (30) not found in the array.


...Program finished with exit code 0
Press ENTER to exit console.
```

B) Producer-Consumer Problem (Code and Output):

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

int *buffer;
int buffer_size;
int buffer_index = 0;
int num_producers;
int num_consumers;
int num_items;

sem_t empty_slots;
sem_t filled_slots;
pthread_mutex_t buffer_mutex;

void *producer(void *arg) {
        int item;
        int producer_id = *(int *)arg;

        for (int i = 0; i < num_items; i++) {
        item = i;

        sem_wait(&empty_slots);
        pthread_mutex_lock(&buffer_mutex);

        // Check if the buffer is full
        if (buffer_index == buffer_size) {
        printf("\n Buffer is full. Producer %d waiting.\n", producer_id);
        }

        buffer[buffer_index] = item;
        buffer_index = (buffer_index + 1) % buffer_size;
        printf(" Producer %d produced item: %d\n", producer_id+1, item);

        pthread_mutex_unlock(&buffer_mutex);
        sem_post(&filled_slots);
        }
```

```
        pthread_exit(NULL);
}

void *consumer(void *arg) {
        int item;
        int consumer_id = *(int *)arg;

        for (int i = 0; i < num_items; i++) {
        sem_wait(&filled_slots);
        pthread_mutex_lock(&buffer_mutex);

        // Check if the buffer is empty
        if (buffer_index == 0) {
        printf("\n Buffer is full\n");
        }

        item = buffer[buffer_index - 1];
        buffer_index = (buffer_index - 1 + buffer_size) % buffer_size;
        printf(" Consumer %d consumed item: %d\n", consumer_id+1, item);

        // Check if the buffer was full and producers were waiting
        if (buffer_index == 0 ){
        printf("\n Buffer is empty.\n");
        }

        pthread_mutex_unlock(&buffer_mutex);
        sem_post(&empty_slots);
        }

        pthread_exit(NULL);
}

int main() {
        printf("\n Enter the number of producers: ");
        scanf("%d", &num_producers);

        printf("\n Enter the number of consumers: ");
        scanf("%d", &num_consumers);

        printf("\n Enter the buffer size: ");
```

```
scanf("%d", &buffer_size);

printf("\n Enter the number of items to produce/consume: ");
scanf("%d", &num_items);

// Allocate memory for the buffer
buffer = (int *)malloc(buffer_size * sizeof(int));

pthread_t producer_threads[num_producers];
pthread_t consumer_threads[num_consumers];

// Initialize semaphores and mutex
sem_init(&empty_slots, 0, buffer_size);
sem_init(&filled_slots, 0, 0);
pthread_mutex_init(&buffer_mutex, NULL);

// Create producer threads
for (int i = 0; i < num_producers; i++) {
int *producer_id = (int *)malloc(sizeof(int));
*producer_id = i;
pthread_create(&producer_threads[i], NULL, producer, producer_id);
}

// Create consumer threads
for (int i = 0; i < num_consumers; i++) {
int *consumer_id = (int *)malloc(sizeof(int));
*consumer_id = i;
pthread_create(&consumer_threads[i], NULL, consumer, consumer_id);
}

// Join producer threads
for (int i = 0; i < num_producers; i++) {
pthread_join(producer_threads[i], NULL);
}

// Join consumer threads
for (int i = 0; i < num_consumers; i++) {
pthread_join(consumer_threads[i], NULL);
}
```

```
    // Destroy semaphores and mutex
    sem_destroy(&empty_slots);
    sem_destroy(&filled_slots);
    pthread_mutex_destroy(&buffer_mutex);

    // Free memory
    free(buffer);

    return 0;
}
```

```
Enter the number of producers: 1

Enter the number of consumers: 1

Enter the buffer size: 4

Enter the number of items to produce/consume: 4
Producer 1 produced item: 0
Producer 1 produced item: 1
Consumer 1 consumed item: 1
Consumer 1 consumed item: 0

Buffer is empty.
Producer 1 produced item: 2
Producer 1 produced item: 3
Consumer 1 consumed item: 3
Consumer 1 consumed item: 2

Buffer is empty.


...Program finished with exit code 0
Press ENTER to exit console.
```

```
Enter the number of producers: 1

Enter the number of consumers: 1

Enter the buffer size: 3

Enter the number of items to produce/consume: 7
Producer 1 produced item: 0
Producer 1 produced item: 1
Producer 1 produced item: 2

Buffer is full
Consumer 1 consumed item: 0
Consumer 1 consumed item: 1
Consumer 1 consumed item: 0

Buffer is empty.
Producer 1 produced item: 3
Producer 1 produced item: 4
Producer 1 produced item: 5

Buffer is full
Consumer 1 consumed item: 0
Consumer 1 consumed item: 4
Consumer 1 consumed item: 3

Buffer is empty.
Producer 1 produced item: 6
Consumer 1 consumed item: 6

Buffer is empty.


...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusion: In conclusion, multithreading and semaphores are vital components of operating systems, enabling concurrent execution and resource synchronization. Multithreading allows for efficient utilization of modern hardware, while semaphores provide synchronization mechanisms to ensure thread-safe access to shared resources. Proper understanding and careful implementation of these concepts are crucial to building reliable and efficient concurrent systems.