National University of Computer and Emerging Sciences

Laboratory Manual

for

Operating Systems Lab

Course Instructor	Mr. Zeeshan Ali Khan
Lab Instructor(s)	Usman Anwer M Aiss Shahid
Section	BCS-4E
Semester	Spring-2023

Department of Computer Science

FAST-NU, Lahore, Pakistan

Semaphores

Semaphores are synchronization primitives used in operating systems and concurrent programming to manage access to shared resources. A semaphore is a variable that can be used to control the access to a shared resource by multiple threads or processes.

There are two types of semaphores:

1. Binary Semaphore: Also known as a mutex (short for mutual exclusion), a binary semaphore can have only two values: 0 and 1. It is used to manage the access to a shared resource where only one thread or process can access the resource at a time. Binary semaphores are often used to protect critical sections of code or shared resources that can only be accessed by one thread or process at a time.
2. Counting Semaphore: A counting semaphore can have multiple values greater than or equal to 0. It is used to manage the access to a shared resource where multiple threads or processes can access the resource simultaneously up to a certain limit. Counting semaphores are often used to control the number of concurrent threads or processes that can access a shared resource, such as a pool of threads or a fixed-size buffer.

Semaphores provide synchronization mechanisms for threads or processes to coordinate their activities and avoid race conditions or other concurrency-related issues. They are used to ensure that threads or processes can access shared resources in a controlled and orderly manner, preventing conflicts and ensuring correctness in concurrent systems.

Examples of how semaphores can be used in C++ to implement synchronization between threads.

Example 1: Binary Semaphore (Mutex)

```
#include <iostream>

#include <mutex>

#include <thread>

std::mutex mtx; // Mutex to protect the shared resource

int sharedVar = 0; // Shared resource

void increment() {

   for (int i = 0; i < 1000000; ++i) {

      std::unique_lock<std::mutex> lock(mtx); // Acquire the mutex

      ++sharedVar; // Increment the shared variable

      lock.unlock(); // Release the mutex

   }
```

```
}

int main() {
    std::thread t1(increment);
    std::thread t2(increment);
    t1.join();
    t2.join();
    std::cout << "Final value of sharedVar: " << sharedVar << std::endl;
    return 0;
}
```

In this example, two threads (**t1** and **t2**) are created to increment a shared variable **sharedVar** in parallel. A mutex **mtx** is used to protect the access to the shared variable, ensuring that only one thread can access it at a time. The **std::unique_lock** is used to lock and unlock the mutex, ensuring that the shared variable is accessed safely without any race conditions.

some online references that provide more information on semaphores and their usage in C++:

1. C++ Reference - Semaphores: http://www.cplusplus.com/reference/mutex/unique_lock/
2. GeeksforGeeks - Semaphores in C++: https://www.geeksforgeeks.org/semaphores-in-c/
3. cppreference.com - std::mutex: https://en.cppreference.com/w/cpp/thread/mutex
4. cppreference.com - std::condition_variable: https://en.cppreference.com/w/cpp/thread/condition_variable
5. Synchronization Primitives in C++11 and C++14: https://www.modernescpp.com/index.php/synchronization-primitives-in-c-11-and-c-14

These references provide detailed explanations, examples, and usage guidelines for working with semaphores in C++. It's always a good practice to refer to official documentation and reputable online sources when learning and implementing synchronization primitives in your code.

Task: Implement a multi-threaded program that simulates a restaurant with a limited number of tables. The program should use semaphores to control access to the tables and ensure that only a limited number of threads (customers) can access the tables at a time. The program should have the following requirements:

1. The restaurant has a fixed number of tables, e.g., 5 tables.
2. Multiple customers (threads) arrive at the restaurant and try to occupy a table.
3. If there are available tables, customers can occupy a table and start eating.
4. If there are no available tables, customers should wait until a table becomes available.

5. When a customer finishes eating, they leave the table, and the table becomes available for other customers.
6. Use semaphores to control access to the tables, ensuring that only a limited number of threads (customers) can occupy tables at a time.
7. Print appropriate messages to indicate when customers arrive, occupy a table, finish eating, and leave the table.
8. Ensure that the output is properly synchronized to avoid race conditions and unexpected behavior.

This task will require you to implement semaphores and use them to synchronize access to shared resources (tables) in a multi-threaded environment. It will help you understand how semaphores can be used for managing concurrent access to limited resources and how to avoid race conditions in multi-threaded programs.