1.Discuss in detail the usage of semaphores. Write a program in C and explain binary semaphores along with its methods.

Semaphores are synchronization tools that are used to coordinate the activities of multiple processes or threads in a computer system. [They are used to enforce mutual exclusion, avoid race conditions and implement synchronization between processes or threads that share resources, such as memory, files, devices, etc1](https://www.geeksforgeeks.org/semaphores-in-process-synchronization/)[2](https://www.baeldung.com/cs/semaphore).

**Semaphore Operations**

Semaphores are integer variables that can be manipulated by two atomic operations: wait and signal. The wait operation decrements the value of the semaphore if it is positive, or blocks the process or thread if it is zero or negative. The signal operation increments the value of the semaphore and wakes up a blocked process or thread if any. [The wait and signal operations are sometimes also called P and V, or down and up, respectively1](https://www.geeksforgeeks.org/semaphores-in-process-synchronization/)[2](https://www.baeldung.com/cs/semaphore).

**Semaphore Types**

There are two types of semaphores: binary and counting. A binary semaphore can have only two values: 0 or 1. It is used to implement mutual exclusion between two processes or threads that access a shared resource. A counting semaphore can have any non-negative value. It is used to control access to a resource that has multiple instances or a limited capacity[1](https://www.geeksforgeeks.org/semaphores-in-process-synchronization/)[2](https://www.baeldung.com/cs/semaphore).

**Binary Semaphore**

A binary semaphore is also known as a mutex lock. It can have only two values: 0 or 1. Its value is initialized to 1. It is used to implement mutual exclusion between two processes or threads that access a shared resource. A process or thread that wants to enter the critical section (the region of code that accesses the shared resource) must first perform a wait operation on the binary semaphore. If the value of the semaphore is 1, it is decremented to 0 and the process or thread can enter the critical section. If the value of the semaphore is 0, it means that another process or thread is already in the critical section, and the current process or thread must wait until the semaphore becomes 1 again. After exiting the critical section, the process or thread must perform a signal operation on the binary semaphore to increment its value to 1 and allow another process or thread to enter the critical section[1](https://www.geeksforgeeks.org/semaphores-in-process-synchronization/)[2](https://www.baeldung.com/cs/semaphore).

**Binary Semaphore Program in C**

Here is an example of how to use a binary semaphore in C to implement mutual exclusion between two threads that increment a shared counter variable:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define N 100000 // number of iterations

#define M 2 // number of threads

int counter = 0; // shared counter variable

sem\_t mutex; // binary semaphore

// function for each thread

void \*thread\_func(void \*arg) {

int i;

for (i = 0; i < N; i++) {

// wait on mutex

sem\_wait(&mutex);

// enter critical section

counter++;

// exit critical section

// signal on mutex

sem\_post(&mutex);

}

return NULL;

}

int main() {

pthread\_t threads[M]; // array of threads

int i;

// initialize mutex with value 1

sem\_init(&mutex, 0, 1);

// create M threads

for (i = 0; i < M; i++) {

pthread\_create(&threads[i], NULL, thread\_func, NULL);

}

// join M threads

for (i = 0; i < M; i++) {

pthread\_join(threads[i], NULL);

}

// destroy mutex

sem\_destroy(&mutex);

// print final counter value

printf("Counter = %d\n", counter);

return 0;

}

The program creates two threads that each increment the counter variable N times. The counter variable is protected by a binary semaphore called mutex, which is initialized to 1. Before incrementing the counter, each thread performs a wait operation on the mutex. If the mutex is 1, it is decremented to 0 and the thread can enter the critical section. If the mutex is 0, it means that another thread is already in the critical section, and the current thread must wait until the mutex becomes 1 again. After incrementing the counter, each thread performs a signal operation on the mutex to increment its value to 1 and allow another thread to enter the critical section.

The expected output of this program is:

Counter = 200000

This means that both threads have successfully incremented the counter N times without interfering with each other, thanks to the binary semaphore.

2.With the help of diagram, explain the Race Condition.

A race condition is a situation where the outcome or behavior of a program depends on the relative timing or interleaving of multiple threads or processes that access or modify a shared resource. [A race condition may result in a bug, an error, a security vulnerability, or an unexpected result1](https://www.baeldung.com/cs/race-conditions)[2](https://www.geeksforgeeks.org/race-condition-vulnerability/).

**Race Condition Example**

One of the common examples of a race condition is the increment operation on a shared variable. Suppose we have a variable x that is initialized to 0, and two threads that each increment x by 1. The increment operation can be broken down into three steps: read x, add 1 to x, and write x. Ideally, we would expect the final value of x to be 2 after both threads finish their execution. However, depending on the order and timing of these steps, we may get different outcomes. For example:

Thread 1 Thread 2 x

read x (0) 0

read x (0) 0

add 1 to x (1) 0

add 1 to x (1) 0

write x (1) 1

write x (1) 1

Copy

In this scenario, both threads read the initial value of x as 0, and then add 1 to it. However, when they write back the new value of x, they overwrite each other’s result. Therefore, the final value of x is 1 instead of 2. This is an incorrect and undesired outcome caused by a race condition.

**Race Condition Diagram**

A race condition can be illustrated by a diagram that shows the possible interleavings of the steps involved in accessing or modifying a shared resource. For example, the diagram below shows the possible interleavings of two threads incrementing a shared variable x:

The diagram shows that there are four possible outcomes for the final value of x: 0, 1, 2, or 3. The outcome depends on which thread executes first and how much time each thread spends on each step. The correct outcome is x = 2, but due to the race condition, we may get any of the other outcomes as well.

**Race Condition Prevention**

To prevent race conditions, we need to ensure that any operation on a shared resource is executed atomically, meaning that it cannot be interrupted or interleaved by another operation. One way to achieve atomicity is by using synchronization mechanisms, such as locks, semaphores, monitors, etc., that enforce mutual exclusion between threads or processes that access a shared resource. [Another way is to use atomic operations that are supported by the hardware or the programming language to ensure indivisibility1](https://www.baeldung.com/cs/race-conditions)[2](https://www.geeksforgeeks.org/race-condition-vulnerability/).

3. Explain hardware approach in mutual exclusion.

Hardware approach in mutual exclusion is a method of implementing mutual exclusion between processes or threads that access a shared resource by using special-purpose machine instructions that are supported by the hardware. [Hardware approach in mutual exclusion is faster and simpler than software approach, but it cannot provide a complete solution for all synchronization problems1](https://www.geeksforgeeks.org/mutual-exclusion-in-synchronization/)[2](https://eng.libretexts.org/Courses/Delta_College/Operating_System%3A_The_Basics/05%3A_Process_Synchronization/5.3%3A_Mutual_Exclusion).

## Hardware Approach in Mutual Exclusion

Some of the common hardware approaches in mutual exclusion are:

* Disabling interrupts: One way to achieve mutual exclusion is to disable interrupts when a process or thread enters a critical section (the region of code that accesses the shared resource). This will prevent any context switches or preemptions that may interfere with the execution of the critical section. However, this approach has several drawbacks, such as:
  + It is not applicable to multiprocessor systems, where multiple processors can access the shared resource concurrently.
  + It affects the responsiveness and performance of the system, as it prevents the handling of other interrupts, such as timers, I/O devices, etc.
  + It requires privileged access to the hardware, which may not be available to user-level processes or threads[1](https://www.geeksforgeeks.org/mutual-exclusion-in-synchronization/)[2](https://eng.libretexts.org/Courses/Delta_College/Operating_System%3A_The_Basics/05%3A_Process_Synchronization/5.3%3A_Mutual_Exclusion)[3](https://stackoverflow.com/questions/49547034/hardware-supported-mutual-exclusion).
* Test-and-set instruction: Another way to achieve mutual exclusion is to use a special machine instruction called test-and-set, which atomically reads and writes a memory location. The test-and-set instruction can be used to implement a lock variable that controls the access to the shared resource. The lock variable is initialized to 0 (unlocked) and can have only two values: 0 or 1 (locked). A process or thread that wants to enter the critical section must first perform a test-and-set operation on the lock variable. If the lock variable is 0, it is set to 1 and the process or thread can enter the critical section. If the lock variable is 1, it means that another process or thread is already in the critical section, and the current process or thread must wait until the lock variable becomes 0 again. After exiting the critical section, the process or thread must reset the lock variable to 0. However, this approach also has some drawbacks, such as:
  + It requires busy waiting, which wastes CPU cycles and may cause starvation or deadlock.
  + It may not be available on all hardware platforms or programming languages[1](https://www.geeksforgeeks.org/mutual-exclusion-in-synchronization/)[2](https://eng.libretexts.org/Courses/Delta_College/Operating_System%3A_The_Basics/05%3A_Process_Synchronization/5.3%3A_Mutual_Exclusion) .
* Compare-and-swap instruction: A similar way to achieve mutual exclusion is to use another special machine instruction called compare-and-swap, which atomically compares and exchanges two memory locations. The compare-and-swap instruction can also be used to implement a lock variable that controls the access to the shared resource. The lock variable is initialized to 0 (unlocked) and can have any value. A process or thread that wants to enter the critical section must first perform a compare-and-swap operation on the lock variable and another variable that stores its own identifier. If the lock variable is 0, it is swapped with the identifier of the process or thread and the process or thread can enter the critical section. If the lock variable is not 0, it means that another process or thread is already in the critical section, and the current process or thread must wait until the lock variable becomes 0 again. After exiting the critical section, the process or thread must swap back its identifier with 0. This approach has some advantages over test-and-set, such as:
  + It allows multiple lock variables to be implemented with different values.
  + [It allows a process or thread to check if it already owns the lock without performing another compare-and-swap operation](https://www.geeksforgeeks.org/mutual-exclusion-in-synchronization/)

4.Discuss Peterson’s solution.

Peterson’s solution is a software-based solution to the critical section problem, which is the problem of ensuring mutual exclusion between processes or threads that access a shared resource. [Peterson’s solution allows two processes or threads to share a single resource without conflict, using only shared memory for communication1](https://www.geeksforgeeks.org/petersons-algorithm-in-process-synchronization/)[2](https://www.scaler.com/topics/petersons-solution/)[3](https://en.wikipedia.org/wiki/Peterson%27s_algorithm).

**Peterson’s Solution Algorithm**

Peterson’s solution uses two shared variables: a boolean array flag[2](https://www.scaler.com/topics/petersons-solution/) and an integer variable turn. The flag array indicates whether a process wants to enter the critical section or not. The turn variable indicates whose turn is it to enter the critical section. The algorithm for process P0 and P1 is as follows:

// Code for process P0

flag[0] = true; // P0 wants to enter the critical section

turn = 1; // P0 gives turn to P1

while (flag[1] && turn == 1) {

// busy wait

}

// critical section

...

// end of critical section

flag[0] = false; // P0 is done with the critical section

// Code for process P1

flag[1] = true; // P1 wants to enter the critical section

turn = 0; // P1 gives turn to P0

while (flag[0] && turn == 0) {

// busy wait

}

// critical section

...

// end of critical section

flag[1] = false; // P1 is done with the critical section

Copy

The algorithm works as follows:

* When a process wants to enter the critical section, it sets its flag to true and gives turn to the other process.
* Then it checks if the other process also wants to enter the critical section and if it is its turn. If both conditions are true, it waits in a busy loop until either condition becomes false.
* If either condition is false, it means that either the other process does not want to enter the critical section or it has given up its turn. In either case, the current process can enter the critical section safely.
* After exiting the critical section, the current process sets its flag to false, indicating that it does not want to enter the critical section anymore.

**Peterson’s Solution Properties**

Peterson’s solution satisfies the following properties:

* Mutual exclusion: Only one process can be in the critical section at a time. This is ensured by the flag and turn variables, which prevent both processes from entering the critical section simultaneously.
* Progress: If no process is in the critical section and some processes want to enter it, then one of them must eventually be able to do so. This is ensured by the turn variable, which gives priority to the process that has been waiting longer.
* Bounded waiting: There is a limit on how long a process has to wait before it can enter the critical section. This is ensured by the turn variable, which prevents a process from being starved by another process that repeatedly enters the critical section.