**EXPERIMENT 5**

**Aim:** To study and implement Peterson’s algorithm

**Lab Objective:** Students will be able to:

* Implement Peterson’s algorithm

# Theory:

Peterson's algorithm is a classic synchronization algorithm that provides a solution to the critical section problem, which is a fundamental problem in concurrent computing. The critical section problem involves multiple processes or threads trying to access a shared resource (the critical section) in a way that ensures mutual exclusion, meaning that only one process can access the critical section at a time. Peterson's algorithm is a software-based approach to achieve mutual exclusion and was developed by Gary L. Peterson in 1981.

Here's a brief overview of Peterson's algorithm and its key components:

1. Shared Variables:

Two integer flags, often denoted as flag [0] and flag [1], to indicate the intention of each process to enter the critical section.

An integer variable `turn` to determine whose turn it is to enter the critical section.

2. Initialization:

Initialize flag [0] and flag [1] to false, indicating that neither process intends to enter the critical section initially.

Initialize turn to an arbitrary value (0 or 1) to specify which process should get the chance first.

3. Process Logic:

Each process follows a protocol to enter the critical section:

Set its flag to indicate its intention to enter the critical section.

Set turn to the other process's index (e.g., if Process 0 wants to enter, it sets turn to 1).

- Check if the other process's flag is true and if it is their turn. If so, it enters a busy- wait loop until the conditions are met.

- Once it exits the busy-wait loop, it can access the critical section.

- After finishing its work in the critical section, it sets its flag to false, indicating it is done.

4. Mutual Exclusion:

The algorithm guarantees mutual exclusion because if one process is in the critical section, the other process's flag variable will prevent it from entering. Additionally, the turn variable ensures fairness by alternating the order in which processes get access to the critical section.

5. Progress:

The algorithm guarantees progress because if a process wants to enter the critical section and the other process is not interested (both flags are false), it can enter without any delay.

6. Bounded Waiting:

The algorithm ensures bounded waiting because if a process wants to enter the critical section and the other process also wants to enter, it will eventually get its turn when the other process resets the turn variable to indicate its intention.

# Advantages:

• Simplicity: Peterson's algorithm is relatively simple to understand and implement compared to more complex synchronization techniques like semaphores or mutexes. It is based on a straightforward idea of using flags and turn variables.

• Mutual Exclusion: Peterson's algorithm guarantees mutual exclusion, which means that only one process or thread can enter the critical section at a time. This prevents data races and ensures that shared resources are accessed safely.

• Fairness: The algorithm is designed to be fair, meaning that it ensures that processes or threads requesting access to the critical section are granted access in the order they requested it. This helps prevent starvation, where a process is indefinitely denied access to the critical section.

• Low Overhead: Peterson's algorithm has relatively low overhead in terms of memory usage and computational cost. It doesn't rely on busy-waiting or polling, which can consume CPU resources.

• Portability: The algorithm can be implemented in a platform-independent manner, making it suitable for a wide range of operating systems and programming languages.

# Disadvantages:

• Limited to two processes: Peterson's algorithm is designed to work only with two concurrent processes. Extending it to work with more processes requires complex modifications, making it less suitable for scenarios involving more than two processes.

• Busy-waiting: Peterson's algorithm uses busy-waiting or spinning, where processes repeatedly check a condition in a loop. This can be very inefficient as it consumes CPU resources even when processes are waiting to access the critical section.

• Priority inversion: The algorithm can suffer from priority inversion issues. If a high priority process is waiting for access to a critical section while a low-priority process holds it, the high-priority process will be blocked, leading to inefficient resource utilization.

• Starvation: Peterson's algorithm does not guarantee fairness or prevent starvation. A process might repeatedly be denied access to the critical section if other processes continually access it.

• Limited applicability: The algorithm is primarily designed for single-processor systems or scenarios with two cooperating processes. In modern multi-core and distributed systems, more advanced synchronization techniques like semaphores, mutexes, or atomic operations are preferred.

• Complexity with more processes: As mentioned earlier, extending Peterson's algorithm to handle more than two processes can be complex and error-prone. Other synchronization mechanisms are better suited for managing multiple processes or threads.

• Lack of scalability: Peterson's algorithm does not scale well with an increasing number of processes or threads. As the number of processes grows, the complexity and overhead of using Peterson's algorithm increase significantly.

• Hardware support: Peterson's algorithm relies on shared variables and atomic operations, which may not be efficiently supported by all hardware architectures, potentially leading to performance issues.

• Code complexity: The algorithm involves non-trivial code that must be carefully implemented to work correctly. Mistakes in the implementation can lead to subtle and hard-to-debug issues.

# Source Code:

import threading

N = 2

flag = [False] \* N

turn = 0

def producer(j):

    while True:

        flag[j] = True

        turn = 1 - j

        while flag[1 - j] and turn == 1 - j:

            pass

        flag[j] = False

def consumer(i):

    while True:

        flag[i] = True

        turn = i

        while flag[1 - i] and turn == i:

            pass

        flag[i] = False

producer\_thread = threading.Thread(target=producer, args=(0,))

consumer\_thread = threading.Thread(target=consumer, args=(1,))

producer\_thread.start()

consumer\_thread.start()

producer\_thread.join()

# consumer\_thread.join()

**Lab Outcome:** Students were able to:

• Understand the advantages and disadvantages of round robin scheduling algorithm