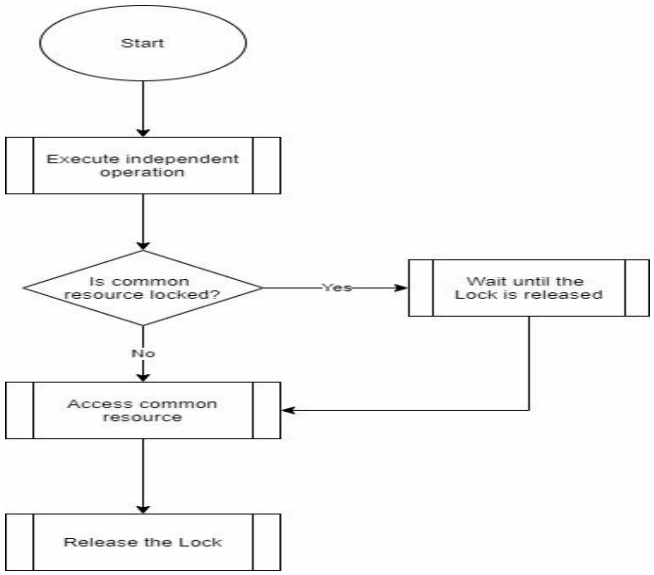
Assignment No. 5: Write a program to solve Classical Problems of Synchronization usingMutex and Semaphore.

Theory:

Mutex

- ✓ Mutex is used to ensure only one thread at a time can access the resource protected bythe mutex. The process that look the mutex must be the one that unlock it. Mutex is good only for managing mutual exclusion to some shared resource.
- ✓ Mutex is easy and efficient for implement.
- ✓ Mutex can be in one of two states: locked or unlocked.
- ✓ Mutex is represented by one bit. Zero (0) means unlock and other value represents locked. It uses two procedures.
- ✓ When a process need access to a critical section, it checks the condition of mutex_locks. If the mutex is currently unlocked then calling process enters into critical section.
- ✓ If the mutex is locked, the calling process is entered into blocked state and wait until the process in the critical section finishes its execution.
- ✓ Mutex variable have only two states so they are simple implement. Their use is limited to guarding entries to critical resigns.
- ✓ Mutex variable is like a binary semaphore. But both are not same.

Algorithm:



Semaphore :

- ✓ Semaphore is described by Dijkstra. Semaphore is a non-negative integer variable thatis used as a flag. Semaphore is an operating system abstract data type. If takes only integer value. It is used to solve critical section problem.
- ✓ Dijkstra introduces two operations (p and v) to operate on semaphore to solve process synchronization problem. A process calls the p operation when it wants to enter its critical section and calls v operation when it when to exit its critical section. The p operation and is called as signal operation.
- ✓ A wait operation on a semaphore decreases its value by one.

  Waits : < 0
  Do loops;
  S: = S – 1;
- ✓ A signal operation increments its value:
  Signal:
  - S := S + 1;
- ✓ A proper semaphore implementation requires that p and v indivisible operations. A semaphore operation is atomic. This may be possible by taking hardware support. The operation p and v are executed by the operating system in response to calls issued by any one process naming a semaphore as parameter.
- ✓ There is no guarantee that no two processes can execute wait and signal operations onthe same semaphore at the same time.

Properties of semaphore :

1. Semaphores are machine independent.
2. Semaphores are simple to implement.
3. Correctness is easy to determine.
4. Semaphore acquire many resources simultaneously.

Types of Semaphores :

There are mainly two types of Semaphores, or two types of signaling integer variables:

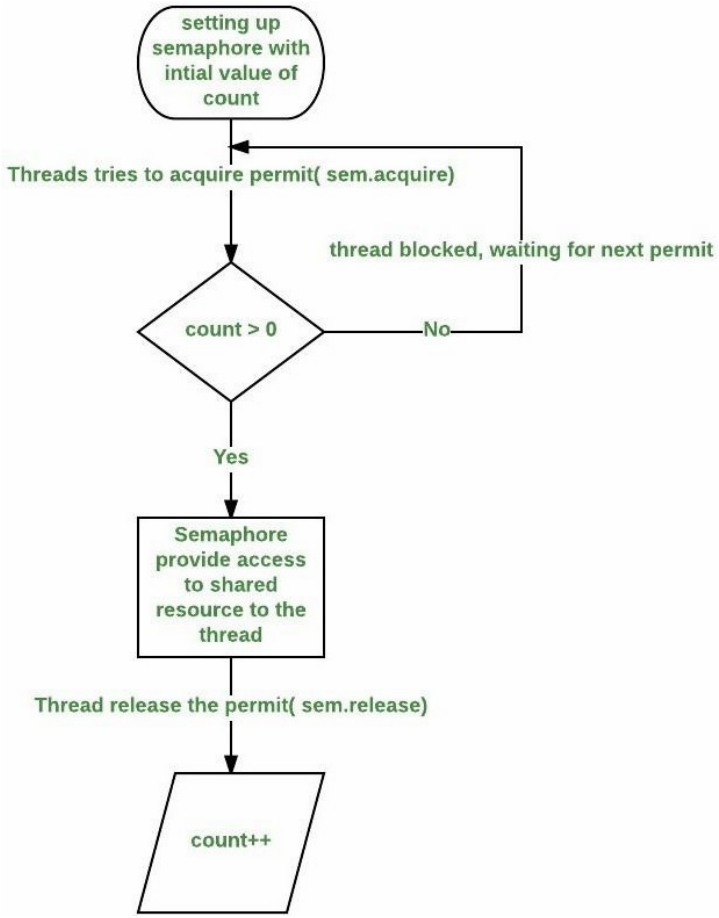
Binary Semaphores :

In this type of Semaphores the integer value of the semaphore can only be either 0 or 1. If the value of the Semaphore is 1, it means that the process can proceed to the critical section (the common section that the processes need to access). However, if the value of binary semaphore is 0, then the process cannot continue to the critical section of the code. When a process is using the critical section of the code, we change the Semaphore value to 0, and when a process is not using it, or we can allow a process to access the critical section, we change the value of semaphore to 1. Binary semaphore is also called mutex lock.

Counting Semaphores :

Counting semaphores are signaling integers that can take on any integer value. Using these Semaphores we can coordinate access to resources and here the Semaphore count is the number of resources available. If the value of the Semaphore is anywhere above 0, processes can access the critical section, or the shared resources. The number of processes that can access the resources / code is the value of the semaphore.
However, if the value is 0, it means that there aren't any resources that are available or the critical section is already being accessed by a number of processes and cannot be accessedby more processes.

Algorithm:



Process Synchronization Problems are as follows:

1. **Producer Consumer Problem**
2. **Reader-Writer Problem**
3. **Dining Philosopher Problem**

1. **Producer Consumer Problem:**

Consider a fixed-size buffer shared between a producer and a consumer.

- The producer generates an item and places it in the buffer.
- The consumer removes an item from the buffer.

The buffer is the critical section. At any moment:

- A producer cannot place an item if the buffer is full.
- A consumer cannot remove an item if the buffer is empty.

To manage this, we use three semaphores:

- mutex – ensures mutual exclusion when accessing the buffer.
- full – counts the number of filled slots in the buffer.
- empty – counts the number of empty slots in the buffer.

Semaphore Initialization

```
mutex = 1; // binary semaphore for mutual exclusion
full = 0; // initially no filled slots
empty = n; // buffer size
```

Producer

```
do {
// Produce an item
wait(empty); // Check for empty slot
wait(mutex); // Enter critical section
// Place item in buffer
signal(mutex); // Exit critical section
signal(full); // Increase number of full slots
} while (true);
```

Consumer

```
do {
wait(full); // Check for filled slot
wait(mutex); // Enter critical section
// Remove item from buffer
signal(mutex); // Exit critical section
signal(empty); // Increase number of empty slots
} while (true);
```

Explanation:

- Empty ensures that producers don't overfill the buffer.
- Full ensures that consumers don't consume from an empty buffer.
- Mutex ensures mutual exclusion, so only one process accesses the buffer at a time.

2. **Reader-Writer Problem**

The Readers-Writers Problem is a classic synchronization issue in operating systems. It deals with coordinating access to shared data (e.g., database, file) by multiple processes or threads.

- **Readers:** Multiple readers can read the shared data simultaneously without causing inconsistency (since they don't modify data).
- **Writers:** Only one writer can access the data at a time, and no readers are allowed while writing (to prevent data corruption).

The challenge is to design a synchronization scheme that ensures:

1. Multiple readers can access data together if no writer is writing.
1. Writers have exclusive access no other reader or writer can enter during writing.

Variants of the Problem

Readers Preference

- Readers are given priority.
- No reader waits if the resource is available for reading, even if a writer is waiting.
- Writers may suffer from *starvation*.

Writers Preference

- Writers are prioritized over readers.
- Ensures writers won't starve, but readers may wait longer.

Solution When Reader Has the Priority Over Writer

Here priority means, no reader should wait if the shared resource is currently open for reading. There are four types of cases that could happen here.

Case	Process 1	Process 2	Allowed/Not Allowed
Case 1	Writing	Writing	Not Allowed
Case 2	Writing	Reading	Not Allowed
Case 3	Reading	Writing	Not Allowed
Case 4	Reading	Reading	Allowed

Reader Process (Reader Preference)

1. Reader requests the entry to critical section.
2. If allowed:

- it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the **wrt** semaphore to restrict the entry of writers if any reader is inside.
- It then, signals mutex as any other reader is allowed to enter while others are already reading.
- After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore "wrt" as now, writer can enter the critical section.

3. If not allowed, it keeps on waiting.

```
do {
wait(mutex); // Lock before updating readcnt
readcnt++;
if (readcnt == 1)
wait(wrt); // First reader blocks writers
signal(mutex); // Allow other readers in
// ---- Critical Section (Reading) ----
wait(mutex);
readcnt--;
if (readcnt == 0)
signal(wrt); // Last reader allows writers
signal(mutex); // Unlock
} while(true);
```

Explanation:

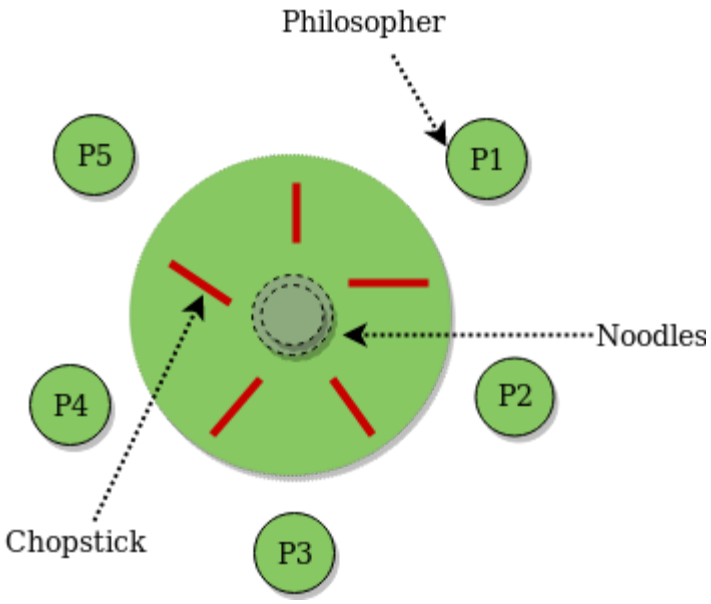
- When a reader enters, it locks mutex to update readcnt.
- If it's the first reader, it locks wrt so writers are blocked.
- Multiple readers can now read the data simultaneously.
- When a reader exits, it decrements readcnt.
- If it's the last reader, it unlocks wrt so writers can proceed.
  The first reader blocks writers, the last reader allows writers, and all readers in between share the resource. This gives preference to readers, but writers may starve.
  Writer's Process
  do {
  wait(wrt); // Lock resource

  // ---- Critical Section (Writing) ----

  signal(wrt); // Release resource
  } while(true);
  Explanation:
- **wait(wrt):** Writer locks the resource to get exclusive access.
- **Critical Section:** Writer performs writing (no other reader or writer can enter).
- **signal(wrt):** Writer releases the resource after finishing.
  Only one writer can write at a time, and no readers are allowed while writing. This ensures data integrity.
  The Readers-Writers Problem highlights the need for proper synchronization when multiple processes access shared data.
- Readers Preference solution allows many readers to access simultaneously while blocking writers, improving read efficiency.
- However, writers may starve if readers keep arriving continuously.
- To avoid this, Writers Preference or a Fair solution (using queues or advanced semaphores) can be used to ensure no starvation and balanced access.

3. Dining Philosopher Problem

The Dining Philosopher Problem is a classic synchronization problem introduced by Edsger Dijkstra in 1965. It illustrates the challenges of resource sharing in concurrent programming, such as deadlock, starvation, and mutual exclusion.



Problem Statement

- K philosophers sit around a circular table.
- Each philosopher alternates between thinking and eating.
- There is one chopstick between each philosopher (total K chopsticks).
- A philosopher must pick up two chopsticks (left and right) to eat.
- Only one philosopher can use a chopstick at a time.

The challenge: Design a synchronization mechanism so that philosophers can eat without causing **deadlock** (all waiting forever) or **starvation** (some never get a chance to eat).

Issues in the Problem

1. **Deadlock:** If every philosopher picks up their left chopstick first, no one can pick up the right one circular wait.
1. **Starvation:** Some philosophers may never get a chance to eat if others keep eating.
1. **Concurrency Control:** Must ensure no two adjacent philosophers eat simultaneously.

Semaphore Solution to Dining Philosopher

We use **semaphores** to manage chopsticks and avoid deadlock.

Algorithm

- Each chopstick is represented as a binary semaphore (mutex).
- Philosopher must acquire both left and right semaphores before eating.
- After eating, the philosopher releases both semaphores.

Pseudocode

```
semaphore chopstick[5] = {1,1,1,1,1};
Philosopher(i):
while(true) {
think();
wait(chopstick[i]); // pick left chopstick
wait(chopstick[(i+1)%5]); // pick right chopstick
eat();
signal(chopstick[i]); // put left chopstick
signal(chopstick[(i+1)%5]); // put right chopstick
}
```

Explanation:

- **semaphore chopstick[5] = {1,1,1,1,1};** Each chopstick is a binary semaphore initialized to

1 (available).

- **think();** Philosopher spends time thinking.
- **wait(chopstick[i]);** Tries to pick the left chopstick. If it's free, philosopher takes it; otherwise waits.
- **wait(chopstick[(i+1)%5]);** Tries to pick the right chopstick (using modulo for circular table).
- **eat();** Philosopher eats once both chopsticks are acquired.
- **signal(chopstick[i]);** and **signal(chopstick[(i+1)%5]); P**uts down both chopsticks, making them available for neighbors.

Conclusion: In this practical we successfully solve classical problems of synchronization using Mutex and Semaphore.