**LAB09**

**TOPIC: CLASSICAL PROBLEM OF SYNCHRONIZATION**

**Objectives**

* **Students will be able to use Mutex and Semaphore libraries to solve critical section problem.**
* **Students will be able to solve some classical synchronization problem using pthread, Mutex & Semaphore libraries.**

1. **Producer-consumer Problem**

The producer and consumer share a fixed-size buffer used as a queue. The producer’s job is to generate data and put this in the buffer. The consumer’s job is to consume the data from this buffer, one at a time.

**Problem Statement**

How do you make sure that producer doesn’t try to put data in buffer when the buffer is full and consumer doesn’t try to consumer data when the buffer is empty?

When producer tries to put data into the buffer when it is full, it wastes cpu cycles. The same is true for consumer it tries to consumer from an empty buffer. It’s better that they go on sleep in these cases so that the scheduler can schedule another process.

**Implementation using simple integer variables:**

See the attached code file: ProducerConsumerSimpleC.c

**Task-01: Understand and run the above program. Check the correctness of output. Submit the screenshot of output as directed by lab instructor.**

**Issues in above implementation?**

**Pseudocode Solution using Semaphore and Mutex**

**Initialization**

Mutex mutex; // Used to provide mutual exclusion for critical section

Semaphore empty = N; // Number of empty slots in buffer

Semaphore full = 0 // Number of slots filled

int in = 0; //index at which producer will put the next data

int out = 0; // index from which the consumer will consume next data

int buffer[N]

**Producer Code**

while(True) {

// produce an item

wait(empty); // wait/sleep when there are no empty slots

wait(mutex);

buffer[in] = item

in = (in+1)%buffersize;

signal(mutex);

signal(full); // Signal/wake to consumer that buffer has some data and they can consume now

}

**Consumer Code**

while(True) {

wait(full); // wait/sleep when there are no full slots

wait(mutex);

item = buffer[out];

out = (out+1)%buffersize;

signal(mutex);

signal(empty); // Signal/wake the producer that buffer slots are emptied and they can produce more

//consumer the item

}

**Solution in C using Semaphore and Mutex**

We will be converting the above Pseudocode to actual code in C language. Let’s first have a look at some important data structures we will be using in the code.

* One can use include header file and declare a semaphore of type sem\_t in c.
* Some important methods that can be used with semaphore in c
  1. **sem\_init** -> Initialise the semaphore to some initial value
  2. **sem\_wait** -> Same as wait() operation
  3. **sem\_post** -> Same as Signal() operation
  4. **sem\_destroy** -> Destroy the semaphore to avoid memory leak
* One can use include header file and declare a mutex of type pthread\_mutex\_t in c.
* Some important methods that can be used with semaphore in c
  1. **pthread\_mutex\_init** -> Initialise the mutex
  2. **pthread\_mutex\_lock()** -> Same as wait() operation
  3. **pthread\_mutex\_unlock()** -> Same as Signal() operation
  4. **pthread\_mutex\_destroy()** -> Destroy the mutex to avoid memory leak

**Implementation using mutex and semaphores in c:**

See the attached code file: ProducerConsumerUsingMutexandSemaphore.c

**Task-02: Understand and run the above program. Check the correctness of output. Submit the screenshot of output as directed by lab instructor.**

1. **Reader Writer Problem**

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as readers and to the latter as writers. Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the readers-writers problem.

The readers-writers problem has several variations, all involving priorities. The simplest one, referred to as the **first readers-writers problem**, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting.

| **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 |

**Pseudocode Solution using Semaphore and Mutex**

**Initialization**

Semaphore wrt = 1; // A binary semaphore that will be used both for mutual exclusion and signalling

Mutex mutex; // Provides mutual exclusion when readcount is being modified

int readcount = 0; // To keep count of the total readers

**Writer Process**

wait(wrt);

// Perform write operation

signal(wrt);

**Reader Process**

wait(mutex);

readcount++;

if(readcount == 1) {

wait(wrt);

}

signal(mutex);

// Perform read operation

wait(mutex);

readcount--;

if(readcount == 0) {

signal(wrt);

}

signal(mutex);

-------------------------------------------------------------------------------------------------------------------------------------

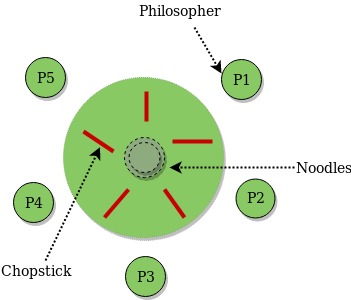
**Implementation in c:**

**See the attached code file:** FristReaderWriterProblemUsingMutexAndSemaphores.c

**Task-03: Understand and run the above program. Check the correctness of output. Submit the screenshot of output as directed by lab instructor.**

1. **The Dining Philosopher Problem**

The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pick up the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.



**The Solution of the Dining Philosophers Problem**

The solution to the process synchronization problem is Semaphores, A semaphore is an integer used in solving critical sections.

The critical section is a segment of the program that allows you to access the shared variables or resources. In a critical section, an atomic action (independently running process) is needed, which means that only single process can run in that section at a time.

Semaphore has 2 atomic operations: wait() and signal(). If the value of its input S is positive, the wait() operation decrements, it is used to acquire resource while entry. No operation is done if S is negative or zero. The value of the signal() operation's parameter S is increased, it used to release the reource once critical section is executed at exit.

**Pseudo Code:**

void Philosopher

{

while(1)

{

// Section where the philosopher is using chopstick

wait(use\_resource[x]);

wait(use\_resource[(x + 1) % 5]);

// Section where the philosopher is thinking

signal(free\_resource[x]);

signal(free\_resource[(x + 1) % 5]);

}

}

**Explanation:**

The wait() operation is implemented when the philosopher is using the resources while the others are thinking. Here, the threads use resource[x] and use resource[(x + 1) % 5] are being executed.

After using the resource, the signal() operation signifies the philosopher using no resources and thinking. Here, the threads free resource[x] and free resource[(x + 1) % 5] are being executed.

To model the Dining Philosophers Problem in a C program we will create an array of philosophers (processes) and an array of chopsticks (resources). We will initialize the array of chopsticks with locks to ensure mutual exclusion is satisfied inside the critical section.

We will run the array of philosophers in parallel to execute the critical section (dine ()), the critical section consists of thinking, acquiring two chopsticks, eating and then releasing the chopsticks.

**Task-04: Implement the dinning philosopher problem solution using mutex and semaphores as discussed above. Submit the source code and output screen-shot.**