**Concurrent and Parallel Programming**

**Practical End Term (2023-24)**

**Note: Attempt all questions.**

1. Write a program to implement producer consumer problem ( Using POSIX semaphores)

**Description:** The producer-consumer problem (Also called the bounded-buffer problem.) illustrates the need for synchronization in systems where many processes share a resource. In the problem, two processes share a fixed-size buffer. One process (producer) produces information and puts it in the buffer, while the other process (consumer) consumes information from the buffer. These processes do not take turns accessing the buffer, they both work concurrently. Herein lies the problem. What happens if the producer tries to put an item into a full buffer? What happens if the consumer tries to take an item from an empty buffer? In order to synchronize these processes, we will block the producer when the buffer is full, and we will block the consumer when the buffer is empty. So the two processes, Producer and Consumer, should work as follows:

**Procedure for doing the experiment:**

1. Declare variable for producer & consumer as pthread-t-tid produce tid consume.

2 . Declare a structure to add items, semaphore variable set as struct.

3 .Read number of items to be produced and consumed.

4. Declare and define semaphore function for creation and destroy.

5 . Define producer function.

6 . Define consumer function.

7 . Call producer and consumer.

8. Stop the execution. **[7 marks]**

2. Write a thread program which demonstrates how to "wait" for thread completions by using the Pthread join routine. Since some implementations of Pthreads may not create threads in a joinable state, therefore explicitly created attribute in a joinable state so that they can be joined later. Created thread should perform the calculation of sum =sum + sin(i) + Tan(i), where i =0 to 10000. Print the out in the following manner **[6 marks]**

|  |
| --- |
| Main: creating thread 0 |
| Main: creating thread 1 |
| Thread 0 starting... |
| Main: creating thread 2 |
| Thread 1 starting... |
| Main: creating thread 3 |
| Thread 2 starting... |
| Thread 3 starting... |
| Thread 1 done. Result = -3.153838e+06 |
| Thread 0 done. Result = -3.153838e+06 |
| Main: completed join with thread 0 having a status of 0 |
| Main: completed join with thread 1 having a status of 1 |
| Thread 3 done. Result = -3.153838e+06 |
| Thread 2 done. Result = -3.153838e+06 |
| Main: completed join with thread 2 having a status of 2 |
| Main: completed join with thread 3 having a status of 3 |
| Main: program completed. Exiting. |

* + - 1. Implementation of Boss/Worker Model: Here the idea is to have a single boss thread that creates work and several worker threads that process the work. Typically the boss thread creates a certain number of workers right away -even before any work has arrived. The worker threads form a thread pool and are all programmed to block immediately. When the boss thread generates some work, it arranges to have one worker thread unblock to handle it. When all workers be busy the boss thread might do one of the following by taking request from the user

1. Queue up the work to be handled later as soon as a worker is free.

2. Create more worker threads.

3. Block until a worker is free to take the new work.

If no work has arrived recently and there are an excessive number of worker threads in the thread pool, the boss thread might terminate a few of them to recover resources. In any case, since creating and terminate threads is relatively expensive (compared to, say, blocking on a mutex) it is generally better to avoid creating a thread for each unit of work produced. [**7 marks]**