IMPLEMENTATION DETAILS - AYUSH JAIN [jain207@purdue.edu]
All variable names & constants in bold.
Final Report is printed at the end of the simulation when the code is executed
Specific questions asked in lab1 for implementation at the end of Steps section.

Shared buffer has been implemented using circular queue where insertions happen at tail end and deletion happen at the head end. Since the queue is circular, head is not restricted to the zero index.

<u>Steps taken for the implementation</u>:

INITIALIZATION

- Initialization of shared buffer **- shared_queue** using a default meaningless value '-'.

- Reverse mapping of producer tags in **producer_inx** array where each cell value represents a producer and index of the array represents a hashed value of the corresponding producer tag (Hash function is (tag char-'0') to get the index location in the array). The reverse mapping is done to map the remaining producer tags to the respective producer process after the end of simulation for faster access.

- Initialization of 2-D array consumption_stat of size **NCONSUMERS** X **NPRODUCERS**. Thus, each cell in this array a(i,j) represents the number of tags consumed by consumer i produced by producer j.

- Initialization of all cells of **produced_count** to 0. This array contains count for the produced tags for each producer process. The count for each producer process is incremented whenever it adds a producer tag to the shared buffer.

- Initialization of all cells of **final_buffer_count** to 0. This array contains count of the remaining tags for each producer process after the end of simulation.

- In order to satisfy the concurrency requirements given for the producer-consumer problem in Lab1, 3 semaphores have been created : **use_queue** (initialized to 1) , **empty_count** (initialized to **BUFFERSIZE**) and **full_count** (initialized to 0).

- All the producer and consumer processes have been created with the same priority 20 and their specific index (as an identifier) sent as an argument to the respective **produce** and **consume** methods.

PRODUCER PROCESS

- The process calls the **produce** method which inserts the designated tags in a set and sleeps for the required time as given by **producer_sleep_times** in a while loop.

- The addition of tag is done by **insert_queue** method which adds the tag as per the concurrency requirements.

CONSUMER PROCESS

- The process calls the **consume** method which consumes the tags in a set and sleeps for the required time as given by **consumer_sleep_times** array in a while loop.
- The deletion of tag is done by **delete_queue** method which deletes the tag as per the concurrency requirements.

STOPPING THE SIMULATION

- The simulation is stopped by killing all the producer & consumer processes and freeing all the created semaphores.

Answers to Specific Questions asked in Lab1

- It is ensured that only process has access to the shared buffer using the **use_queue** semaphore . If the semaphore value is 1 and wait is called on it, then that process acquires full control over the queue until signal is called on it.
- When producer process acquires control over the buffer, it is checked whether buffer is empty using the **empty_count** semaphore for the producer to add tags only when buffer is not full. In case buffer is full, producer process waits(blocks) till consumer consumes some tag and signals the **use_queue** semaphore to give up control over the queue.
- When consumer process acquires control over the buffer, it is checked whether buffer is non empty using the **full_count** semaphore for the consumer to consume tags only when buffer is not empty. In case buffer is empty, consumer process waits(blocks) till producer produces some tag and signals the **use_queue** semaphore to give up control over the queue.
- After each addition or consumption, producer and consumer processes signal the **full_count** and **empty_count** respectively to unblock the producer and consumer process because of full and empty capacity of shared buffer respectively.

1) Thus, it is guaranteed that that only one process will attempt to delete an item at a given time.
2) When a producer needs to insert an item in the shared buffer, it gets the control over the buffer and consumers are excluded as well because it involves manipulation of head and tail pointers for the queue. If consumer is allowed access then it may consume wrong tag due to race condition with producer process.
3) When a consumer starts to extract items, it is guaranteed that the consumer will consume contiguous locations in the shared buffer as it has full control over the buffer

using the **use_queue** semaphore. So, deletion can happen continuously. In case the buffer is empty, it blocks until it gets more producer tags to consume and finishes its set.

<u>Extra Credit Answer</u>

Yes, changing the priorities of producers and consumers affect fair access to the shared buffer. Fair access means when the process scheduler schedules the process based on first come first serve basis. This is only achieved when all the processes have same priorities but not when their priorities are not same. This is proved by the following experiments.

When priority of all Producers(40) > Consumers (20) and count of each process is 1, we can see the producer keeps on getting the CPU processing time to add the tags even after completing their respective set and giving up control over the shared buffer. The scheduler keeps scheduling them till buffer is full and they are blocked. Then only, consumers get a chance to consume tags. Consumers don't get a chance before even when they are ready and producer has completed its set.

Producers - 3, Consumers - 3, Buffersize - 60

```
---------FINAL REPORT AT THE END OF SIMULATION----------
Size before addition--0----Producer Process 0 adding A
Producer Process 0 finished a set---Current Size  1---Head 0 ----Tail 0

Size before addition--1----Producer Process 0 adding A
Producer Process 0 finished a set---Current Size  2---Head 0 ----Tail 1

Size before addition--2----Producer Process 0 adding A
Producer Process 0 finished a set---Current Size  3---Head 0 ----Tail 2

Size before addition--3----Producer Process 0 adding A
Producer Process 0 finished a set---Current Size  4---Head 0 ----Tail 3
```

…… and so on till buffer is full as Producer processes keep on adding tags and consumer process don't get chance to consumer till producer processes block only when buffer is full.

```
---------------REPORT---------------
Producer A: created 63 items
Producer B: created 3 items
Producer C: created 3 items

Consumer a: deleted 4 items from producer A 0 items from producer B 0 items from producer C
Consumer b: deleted 3 items from producer A 0 items from producer B 0 items from producer C
Consumer c: deleted 3 items from producer A 0 items from producer B 0 items from producer C
```

```
The shared buffer contains: 53 items from producer A 3 items from producer B 3 items from
producer C
-------------------------------------
```

The deletion by consumer is very few with respect to the producer addition of tags.

Similarly when priority of all Producers (20) < Consumers (40) and count of each process is 1, we can see the consumer keeps on getting the CPU processing time to delete the tags even after consuming their respective set and giving up control over the shared buffer. The scheduler keeps scheduling them till buffer is empty and they are blocked. Then only, producers get a chance to produce tags.

When the producers have unequal priority among themselves and consumers have 20 priority, the producer with the higher priority keeps on getting time to produce tags

Producers - 3, Consumers - 3, Buffersize - 60
Counts of Producers - 1, 3, 1
Priorities of Producers - 30, 40, 30
Priorities of Consumers - 20, 20, 20

```
---------FINAL REPORT AT THE END OF SIMULATION----------
Size before addition--0----Producer Process 0 adding A
Producer Process 0 finished a set---Current Size  1---Head 0 ----Tail 0

Size before addition--1----Producer Process 1 adding A
Size before addition--2----Producer Process 1 adding A
Size before addition--3----Producer Process 1 adding A
Producer Process 1 finished a set---Current Size  4---Head 0 ----Tail 3

Size before addition--4----Producer Process 1 adding A
Size before addition--5----Producer Process 1 adding A
Size before addition--6----Producer Process 1 adding A
Producer Process 1 finished a set---Current Size  7---Head 0 ----Tail 6

Size before addition--7----Producer Process 2 adding A
Producer Process 2 finished a set---Current Size  8---Head 0 ----Tail 7
```

And so on…....
So, it can be seen producer 1 with highest priority keeps getting more CPU time.