Name: Purvam Sheth
Enrolment No. AU1940151
EndSem-OS

PART-C

1. **Producer**
   It is used to create data elements and able to store it in the buffer given buffer is not full.
2. **Consumer**
   If buffer is not empty and has some items a consumer process which executes statements which consume the data present in a buffer, sometime as a parameter.

Main problem arises when producer process and consumer process is not sync with each other

A fixed-size buffer serves as a queue for both the producer and the consumer. It is the producer's responsibility to generate data and store it in the buffer. The consumer's role is to consume this buffer's data one by one.

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
}
```

```
purvam@purvam-VirtualBox:~/Desktop/CSE332/End-Prac/AU1940151_EndSem_OS_PurvamSheth$ ./a.out
Producer 1: Insert Item 1804289383 at 0
Producer 2: Insert Item 846930886 at 1
Producer 3: Insert Item 1681692777 at 2
Producer 4: Insert Item 1714636915 at 3
Producer 5: Insert Item 1957747793 at 4
Consumer 1: Remove Item 1804289383 from 0
Consumer 1: Remove Item 846930886 from 1
Consumer 1: Remove Item 1681692777 from 2
Consumer 1: Remove Item 1714636915 from 3
Producer 4: Insert Item 596516649 at 0
Consumer 2: Remove Item 1957747793 from 4
Consumer 2: Remove Item 596516649 from 0
Producer 3: Insert Item 1649760492 at 1
Producer 3: Insert Item 1350490027 at 2
Producer 1: Insert Item 424238335 at 3
Producer 2: Insert Item 719885386 at 4
Producer 5: Insert Item 1189641421 at 0
Consumer 1: Remove Item 1649760492 from 1
Producer 3: Insert Item 783368690 at 1
Consumer 3: Remove Item 1350490027 from 2
Consumer 3: Remove Item 424238335 from 3
Consumer 3: Remove Item 719885386 from 4
Consumer 3: Remove Item 1189641421 from 0
Producer 4: Insert Item 1025202362 at 2
Producer 4: Insert Item 1540383426 at 3
Producer 1: Insert Item 1102520059 at 4
Producer 2: Insert Item 2044897763 at 0
Consumer 2: Remove Item 783368690 from 1
Consumer 2: Remove Item 1025202362 from 2
Consumer 2: Remove Item 1540383426 from 3
Producer 3: Insert Item 1365180540 at 1
Producer 5: Insert Item 1967513926 at 2
Producer 4: Insert Item 304089172 at 3
Consumer 3: Remove Item 1102520059 from 4
Producer 1: Insert Item 1303455736 at 4
Consumer 4: Remove Item 2044897763 from 0
Consumer 4: Remove Item 1365180540 from 1
Consumer 4: Remove Item 1967513926 from 2
Consumer 4: Remove Item 304089172 from 3
Producer 5: Insert Item 521595368 at 0
Producer 5: Insert Item 1726956429 at 1
```

PART-B

1) The round-robin (RR) scheduling algorithm was created with time-sharing systems in mind. Preemption is provided to allow the system to transition between processes, comparable to FCFS scheduling. A time quantum, also known as a time slice, is a tiny unit of time. The length of a time quantum is usually between 10 and 100 milliseconds. A circular queue is used to treat the ready queue. The CPU scheduler iterates over the ready queue, allocating CPU time to each task for up to one time quantum. We retain the ready queue as a FIFO queue o£ processes to perform RR scheduling. New processes are added to the ready queue's tail. The CPU scheduler selects the first process from the ready queue, sets a timer to interrupt the process after one time quantum, and dispatches it.

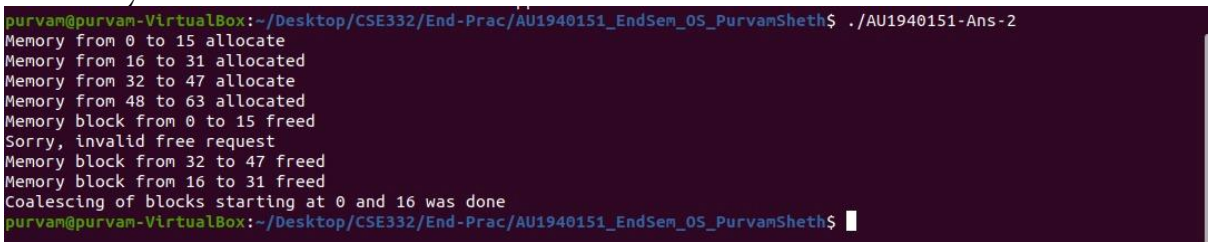
Then one of two things will occur. A CPU burst of less than one time quantum may occur during the procedure. In this instance, the process will freely surrender the CPU. Following that, the scheduler will move on to the next process in the ready queue. Otherwise, the timer will go off and trigger an interrupt to the operating system if the CPU burst of the presently executing process is longer than 1 time quantum. The process will be

moved to the back of the ready queue after a context transition. After that, the CPU scheduler will choose the next task in the ready queue.

Under the RR policy, the average waiting time is frequently long. Consider the following set of processes, which arrive at time 0 and have a CPU burst length of milliseconds

2) The buddy memory allocation approach is a memory allocation mechanism that splits memory into divisions in order to provide the best possible response to a memory request. This system divides memory into half in order to find the optimal match.
There are several types of buddy systems; the simplest and most prevalent are those in which each block is split into two smaller blocks. In this system, every memory block has an order, which is a number ranging from 0 to a set upper limit. A block of order n is proportionate to 2n in size, therefore the blocks are precisely twice the size of blocks of lower order. Because all pals are aligned on memory address boundaries that are powers of two, address calculation is straightforward with power-of-two block sizes. When a bigger block is split, it generates two smaller blocks, each of which becomes a unique friend for the other.A split block can only be combined with its one-of-a-kind buddy block, which reassembles the bigger block from which it was separated.

To begin, the smallest feasible block, i.e. the smallest memory block that may be allocated, is determined. There would be a lot of memory and computational overhead for the system to keep track of which sections of the memory are allocated and unallocated if there was no lower limit at all (e.g., bit-sized allocations were feasible). However, a low limit may be desired in order to reduce average memory waste per allocation (for allocations that are not multiples of the smallest block in size).
The lower limit is often minimal enough to reduce average wasted space per allocation while being large enough to avoid undue overhead. All higher orders are represented as power-of-two multiples of the lowest block size, which is subsequently used as the size of an order-0 block.
The programmer must then choose, or develop code to achieve, the greatest possible order that will fit in the remaining memory space. The biggest block size may not cover the whole memory of a computer system if the total accessible memory is not a power-of-two multiple of the minimum block size. The top restriction on the order would be 8 if the system had 2000 K of physical memory and the order-0 block size was 4 K, because an order-8 block (256 order-0 blocks, 1024 K) is the largest block that will fit in memory. As a result, allocating the whole physical memory in a single chunk is impracticable; the remaining 976 K of memory must be allocated in smaller chunks.



PART-A

Multithreads
For this programme, I've used a common piece of data called a flag to help select which item to print for the pattern. It also took three more variables to keep track of the upper case,

lower case, and numerical pattern. I generated three distinct threads to print the character for this program's execution.