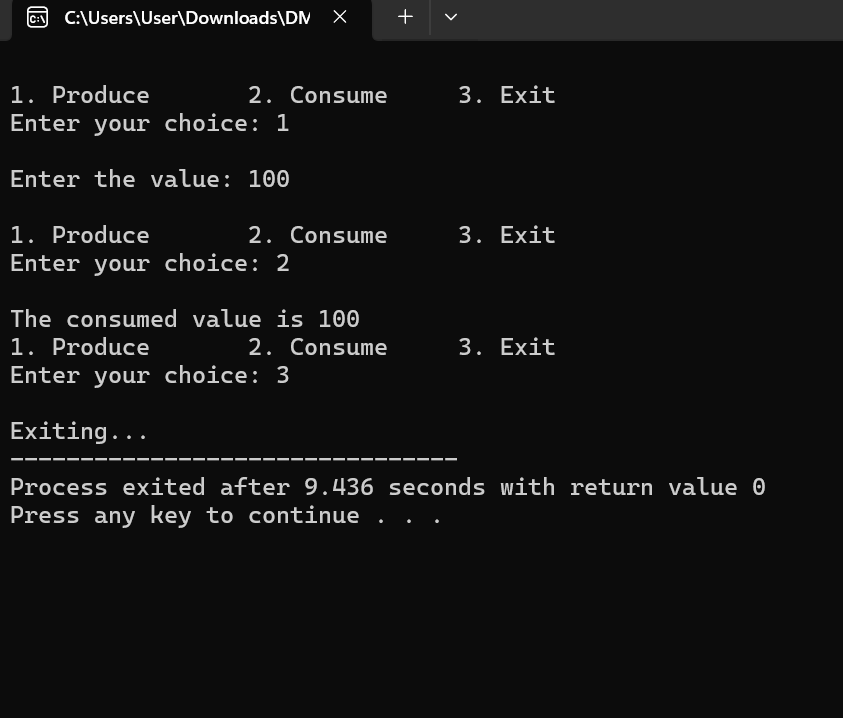
Lab-4

Q1:

|  |
| --- |
| #include <stdio.h>  int main() {  int buffer[10], bufsize = 10, in = 0, out = 0;  int produce, consume, choice = 0;  while (choice != 3) {  printf("\n1. Produce \t 2. Consume \t3. Exit");  printf("\nEnter your choice: ");  scanf("%d", &choice);  switch (choice) {  case 1:  if ((in + 1) % bufsize == out) {  printf("\nBuffer is Full");  } else {  printf("\nEnter the value: ");  scanf("%d", &produce);  buffer[in] = produce;  in = (in + 1) % bufsize;  }  break;  case 2:  if (in == out) {  printf("\nBuffer is Empty");  } else {  consume = buffer[out];  printf("\nThe consumed value is %d", consume);  out = (out + 1) % bufsize;  }  break;  case 3:  printf("\nExiting...");  break;  default:  printf("\nInvalid choice");  break;  }  }  return 0;  } |



Q:2

In producer-consumer problem what difference will it make if we utilize stack for the buffer rather than an array?

**1. Array/Queue as Buffer (Typical Case)**

* **FIFO (First In, First Out)** behavior
* The **oldest produced item** is consumed **first**
* This is ideal for many real-world scenarios where data should be processed in the order it arrives.

**2. Stack as Buffer**

* **LIFO (Last In, First Out)** behavior
* The **most recently produced item** is consumed **first**
* Useful in specific cases (e.g., undo operations), but often **undesirable** for producer-consumer problems

### Key Differences

| **Aspect** | **Queue/Array (FIFO)** | **Stack (LIFO)** |
| --- | --- | --- |
| Consumption order | Oldest item first | Newest item first |
| Use-case suitability | Real-time data processing | Backtracking, undo systems |
| Data loss risk | Lower (predictable order) | Higher in time-sensitive systems |
| Example use-case | Print jobs, packet processing | Undo functionality, recursion |

Q:3 Solve the producer-consumer problem using linked list. (You can perform this task using any programming language) Note: Keep the buffer size to 10 places.

|  |
| --- |
| import threading  import time  import random  # Node class for linked list  class Node:  def \_\_init\_\_(self, data):  self.data = data  self.next = None  # Buffer using a linked list  class LinkedListBuffer:  def \_\_init\_\_(self, max\_size=10):  self.head = None  self.tail = None  self.size = 0  self.max\_size = max\_size  self.lock = threading.Lock()  def insert(self, data):  if self.size >= self.max\_size:  return False # Buffer full  new\_node = Node(data)  if self.tail:  self.tail.next = new\_node  self.tail = new\_node  else:  self.head = self.tail = new\_node  self.size += 1  return True  def remove(self):  if self.size == 0:  return None # Buffer empty  removed\_data = self.head.data  self.head = self.head.next  self.size -= 1  if self.head is None:  self.tail = None  return removed\_data  buffer = LinkedListBuffer(10)  condition = threading.Condition()  # Producer thread  def producer():  while True:  item = random.randint(1, 100)  with condition:  while buffer.size >= buffer.max\_size:  condition.wait()  buffer.insert(item)  print(f"Produced: {item}")  condition.notify\_all()  time.sleep(random.uniform(0.5, 1.5))  # Consumer thread  def consumer():  while True:  with condition:  while buffer.size == 0:  condition.wait()  item = buffer.remove()  print(f"Consumed: {item}")  condition.notify\_all()  time.sleep(random.uniform(0.5, 2.0))  # Creating threads  producer\_thread = threading.Thread(target=producer)  consumer\_thread = threading.Thread(target=consumer)  # Starting threads  producer\_thread.start()  consumer\_thread.start() |