CT-353

Name	Aelia Taskeen Bibi
Roll No	Dt-22021
Course	Operating System
Department	CSIT(Data Science)

DT-021

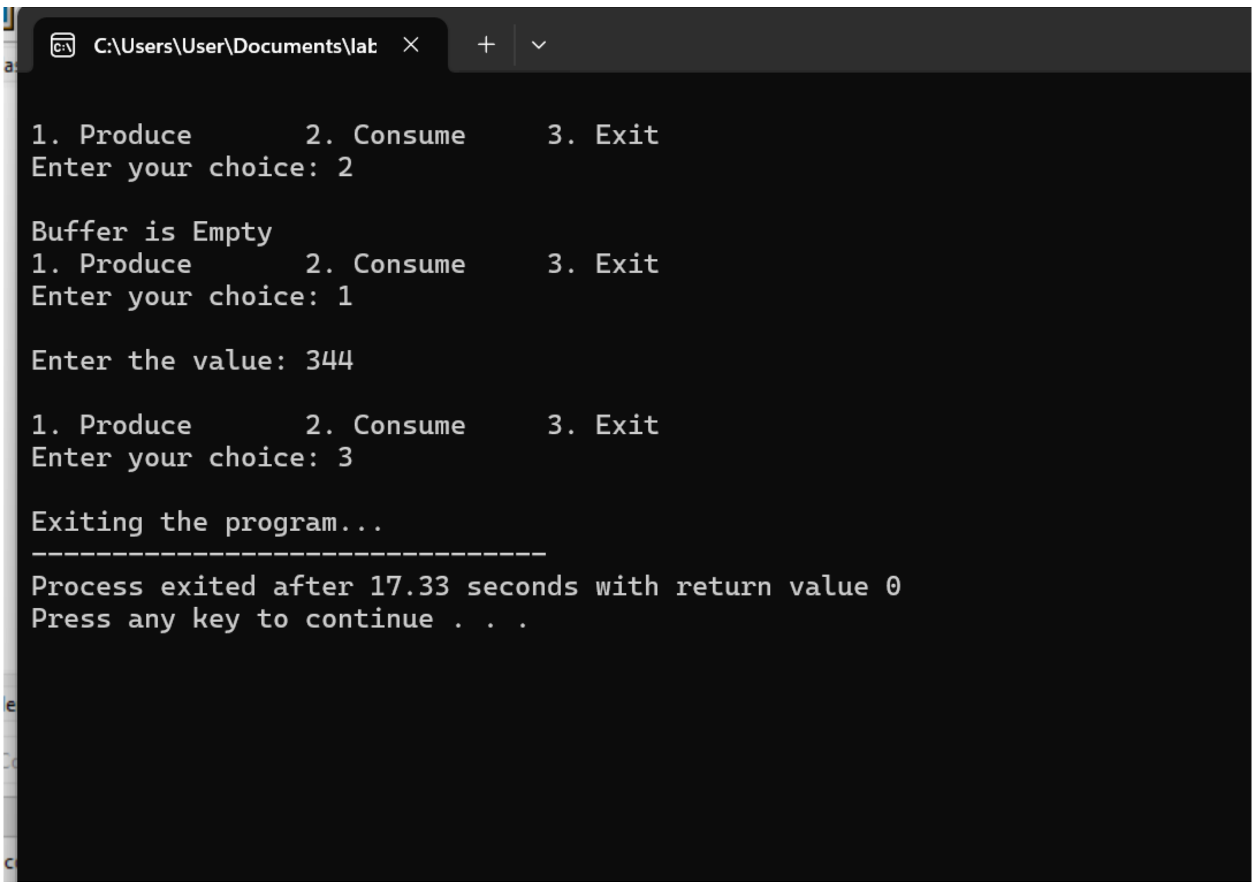
LAB:4

(Producer & Consumer Problem)

```
#include <stdio.h>


int main() {

    int buffer[10], bufsize, in, out, produce, consume, choice = 0;

    in = 0;

    out = 0;

    bufsize = 10;


    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 the program...");
    break;

default:
```

```
            printf("\nInvalid choice! Please enter 1, 2, or 3.");
        }
    }

    return 0;
}
```



2. 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.

```
#include <stdio.h>
```

```
#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>


// Define buffer size

#define BUFFER_SIZE 10


// Node structure for linked list

typedef struct Node {

    int data;

    struct Node* next;

} Node;


// Shared buffer (linked list head)

Node* head = NULL;


// Synchronization variables

pthread_mutex_t mutex;

sem_t full, empty;
```

```
// Function to add item to linked list (Producer)
void addItem(int item) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = item;
    newNode->next = head;
    head = newNode;
}

// Function to remove item from linked list (Consumer)
int removeItem() {
    if (head == NULL) return -1;

    Node* temp = head;
    int item = temp->data;
    head = head->next;
    free(temp);
    return item;
}

// Producer function
void* producer(void* arg) {
```

```
    int item;
    for (int i = 0; i < 15; i++) {  // Produce 15 items
        item = rand() % 100;  // Random number

        sem_wait(&empty);  // Wait if buffer is full
        pthread_mutex_lock(&mutex);  // Lock

        addItem(item);
        printf("Produced: %d\n", item);

        pthread_mutex_unlock(&mutex);  // Unlock
        sem_post(&full);  // Increase full count
        sleep(1);  // Simulate delay
    }
    return NULL;
}

// Consumer function
void* consumer(void* arg) {
    int item;
    for (int i = 0; i < 15; i++) {  // Consume 15 items
```

```
        sem_wait(&full);  // Wait if buffer is empty
        pthread_mutex_lock(&mutex);  // Lock

        item = removeItem();
        printf("Consumed: %d\n", item);

        pthread_mutex_unlock(&mutex);  // Unlock
        sem_post(&empty);  // Increase empty count
        sleep(2);  // Simulate delay
    }
    return NULL;
}

// Main function
int main() {
    pthread_t prod, cons;

    // Initialize mutex and semaphores
    pthread_mutex_init(&mutex, NULL);
    sem_init(&full, 0, 0);
    sem_init(&empty, 0, BUFFER_SIZE);
```

```
    // Create producer and consumer threads
    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    // Wait for threads to finish
    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    // Destroy mutex and semaphores
    pthread_mutex_destroy(&mutex);
    sem_destroy(&full);
    sem_destroy(&empty);

    return 0;
}
```

```
C:\Users\User\Documents\lab
Produced: 41
Consumed: 41
Produced: 67
Consumed: 67
Produced: 34
Produced: 0
Consumed: 0
Produced: 69
Produced: 24
Consumed: 24
Produced: 78
Produced: 58
Consumed: 58
Produced: 62
Produced: 64
Consumed: 64
Produced: 5
Produced: 45
Consumed: 45
Produced: 81
Produced: 27
Consumed: 27
Produced: 61
Consumed: 61
Consumed: 81
Consumed: 5
Consumed: 62
Consumed: 78
Consumed: 69
Consumed: 34
```

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

In the **Producer-Consumer Problem**, the buffer is typically implemented as a **queue (FIFO - First In, First Out)** to ensure that items are consumed in the order they were produced. However, if we use a **stack (LIFO - Last In, First Out)** instead of a queue, the following differences will occur:

1. **Order of Consumption**

  - **Queue (FIFO):** The oldest item is consumed first, ensuring that all items are processed in the order they arrive.
  - **Stack (LIFO):** The newest item is consumed first, which may lead to older items being left unprocessed if the buffer is full.
2. **Potential Data Loss**
   - With a **stack**, if the producer keeps adding new items rapidly, older items might **never be consumed**, leading to data loss.
   - A **queue ensures fairness**, where every item gets processed in the order it was added.
3. **Use Cases**
   - A **stack is useful** in scenarios like undo operations (where the last action is reversed first) or depth-first search algorithms.
   - A **queue is better** for real-world producer-consumer systems like job scheduling, order processing, and messaging systems, where maintaining order is important.