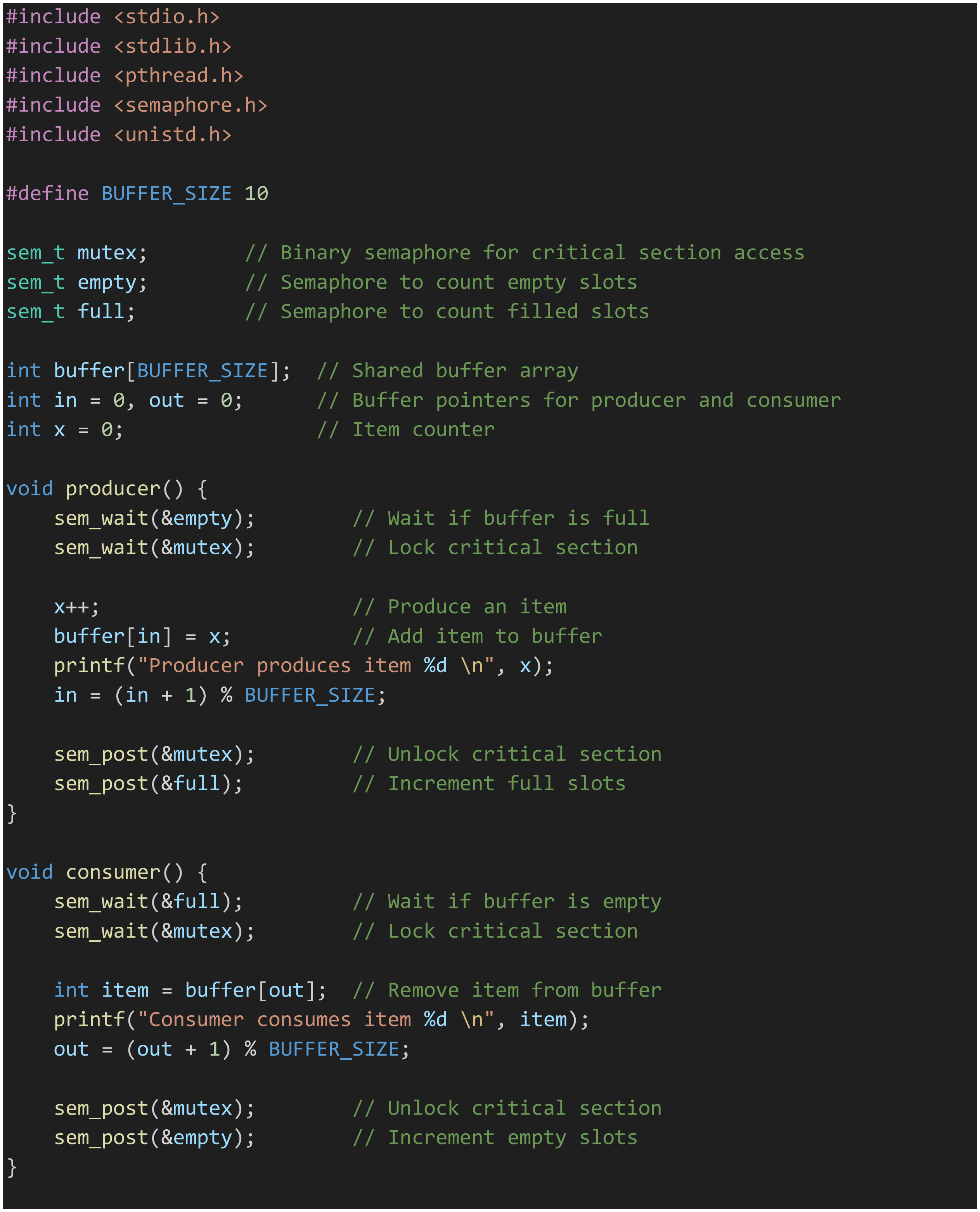
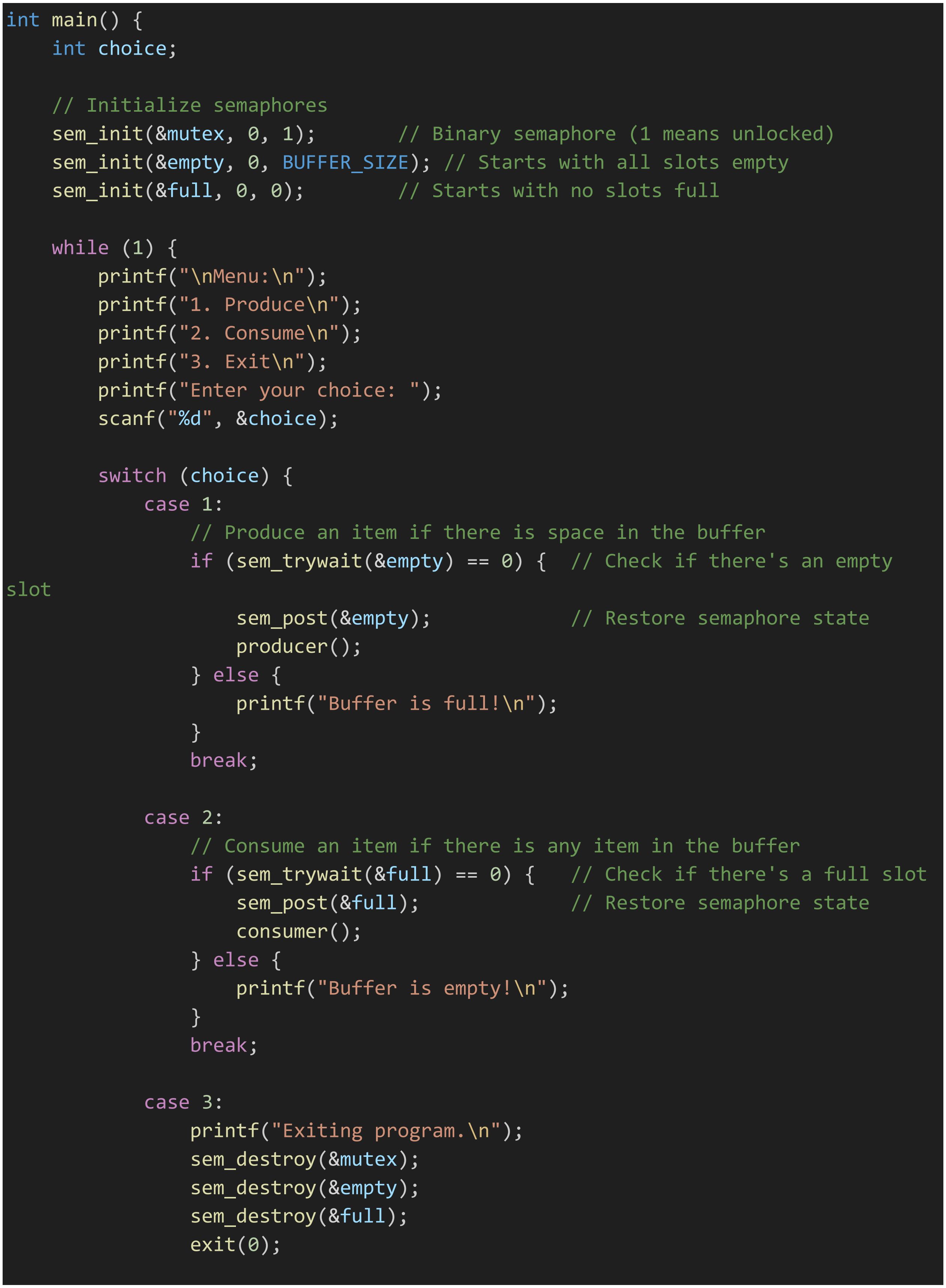
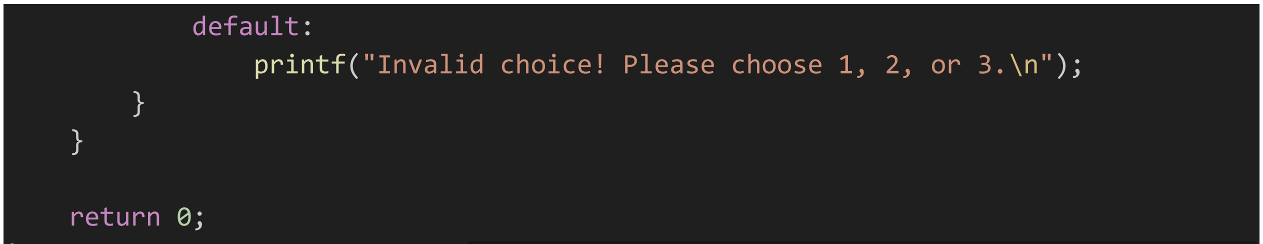
Que 1. Write a C program to implement the Producer & consumer Problem using Semaphore.



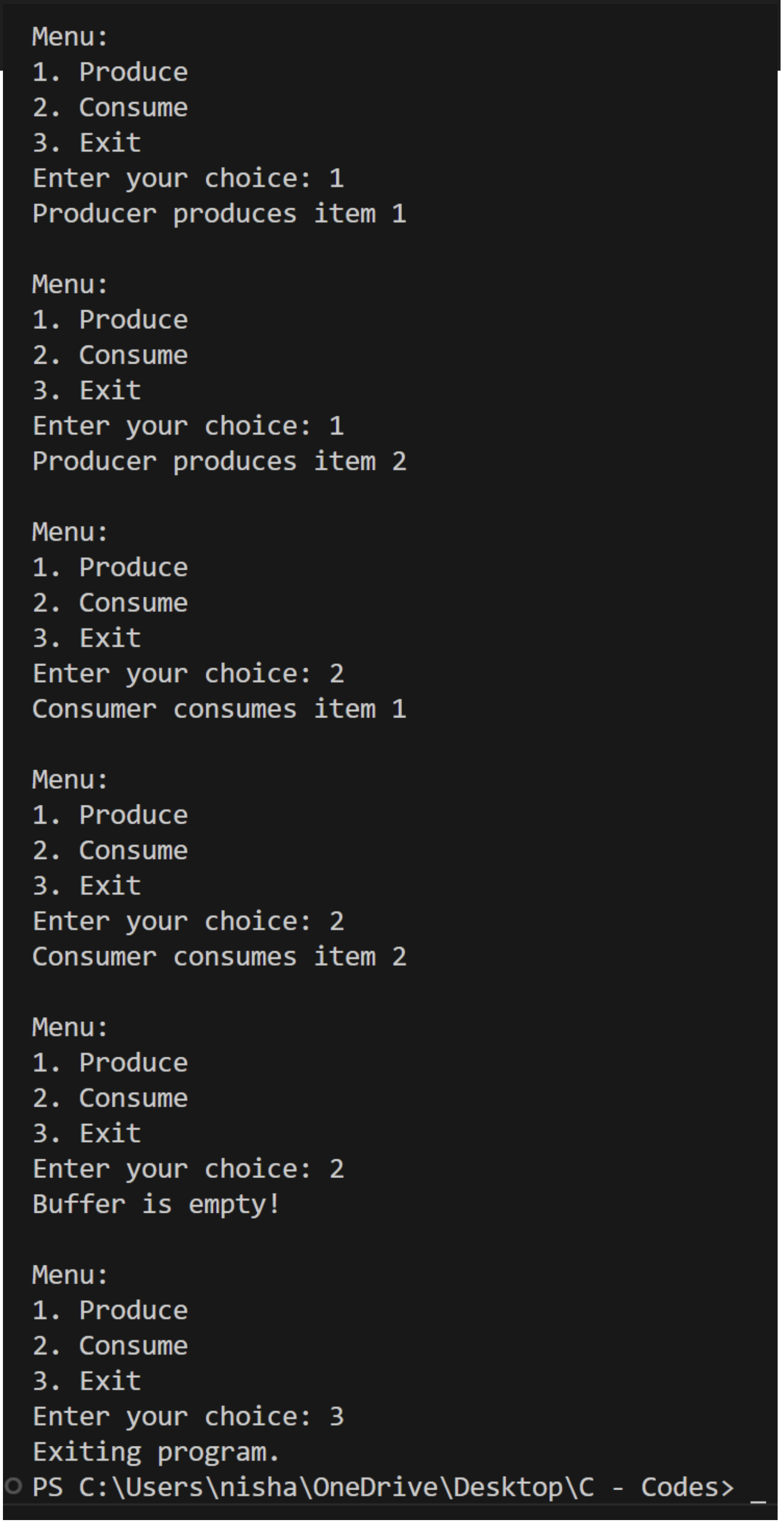
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define BUFFER_SIZE 10

sem_t mutex;        // Binary semaphore for critical section access
sem_t empty;        // Semaphore to count empty slots
sem_t full;         // Semaphore to count filled slots

int buffer[BUFFER_SIZE];  // Shared buffer array
int in = 0, out = 0;      // Buffer pointers for producer and consumer
int x = 0;                // Item counter

void producer() {
    sem_wait(&empty);        // Wait if buffer is full
    sem_wait(&mutex);        // Lock critical section

    x++;                     // Produce an item
    buffer[in] = x;          // Add item to buffer
    printf("Producer produces item %d \n", x);
    in = (in + 1) % BUFFER_SIZE;

    sem_post(&mutex);        // Unlock critical section
    sem_post(&full);         // Increment full slots
}

void consumer() {
    sem_wait(&full);         // Wait if buffer is empty
    sem_wait(&mutex);        // Lock critical section

    int item = buffer[out];  // Remove item from buffer
    printf("Consumer consumes item %d \n", item);
    out = (out + 1) % BUFFER_SIZE;

    sem_post(&mutex);        // Unlock critical section
    sem_post(&empty);        // Increment empty slots
}
```



```
int main() {
    int choice;

    // Initialize semaphores
    sem_init(&mutex, 0, 1);       // Binary semaphore (1 means unlocked)
    sem_init(&empty, 0, BUFFER_SIZE); // Starts with all slots empty
    sem_init(&full, 0, 0);        // Starts with no slots full

    while (1) {
        printf("\nMenu:\n");
        printf("1. Produce\n");
        printf("2. Consume\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                // Produce an item if there is space in the buffer
                if (sem_trywait(&empty) == 0) {  // Check if there's an empty
slot
                    sem_post(&empty);            // Restore semaphore state
                    producer();
                } else {
                    printf("Buffer is full!\n");
                }
                break;

            case 2:
                // Consume an item if there is any item in the buffer
                if (sem_trywait(&full) == 0) {   // Check if there's a full slot
                    sem_post(&full);             // Restore semaphore state
                    consumer();
                } else {
                    printf("Buffer is empty!\n");
                }
                break;

            case 3:
                printf("Exiting program.\n");
                sem_destroy(&mutex);
                sem_destroy(&empty);
                sem_destroy(&full);
                exit(0);
```

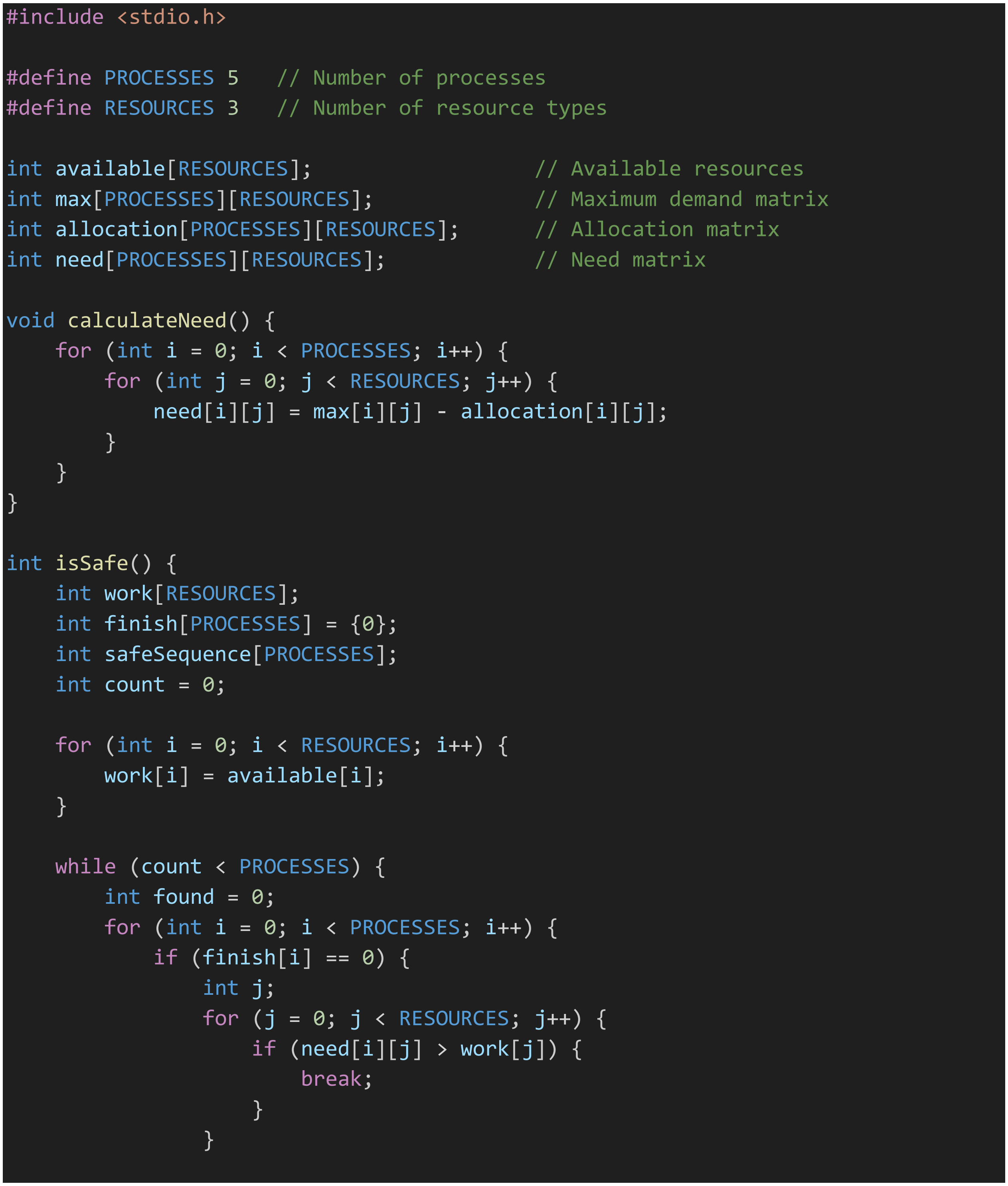


```
            default:
                printf("Invalid choice! Please choose 1, 2, or 3.\n");
        }
    }

    return 0;
}
```

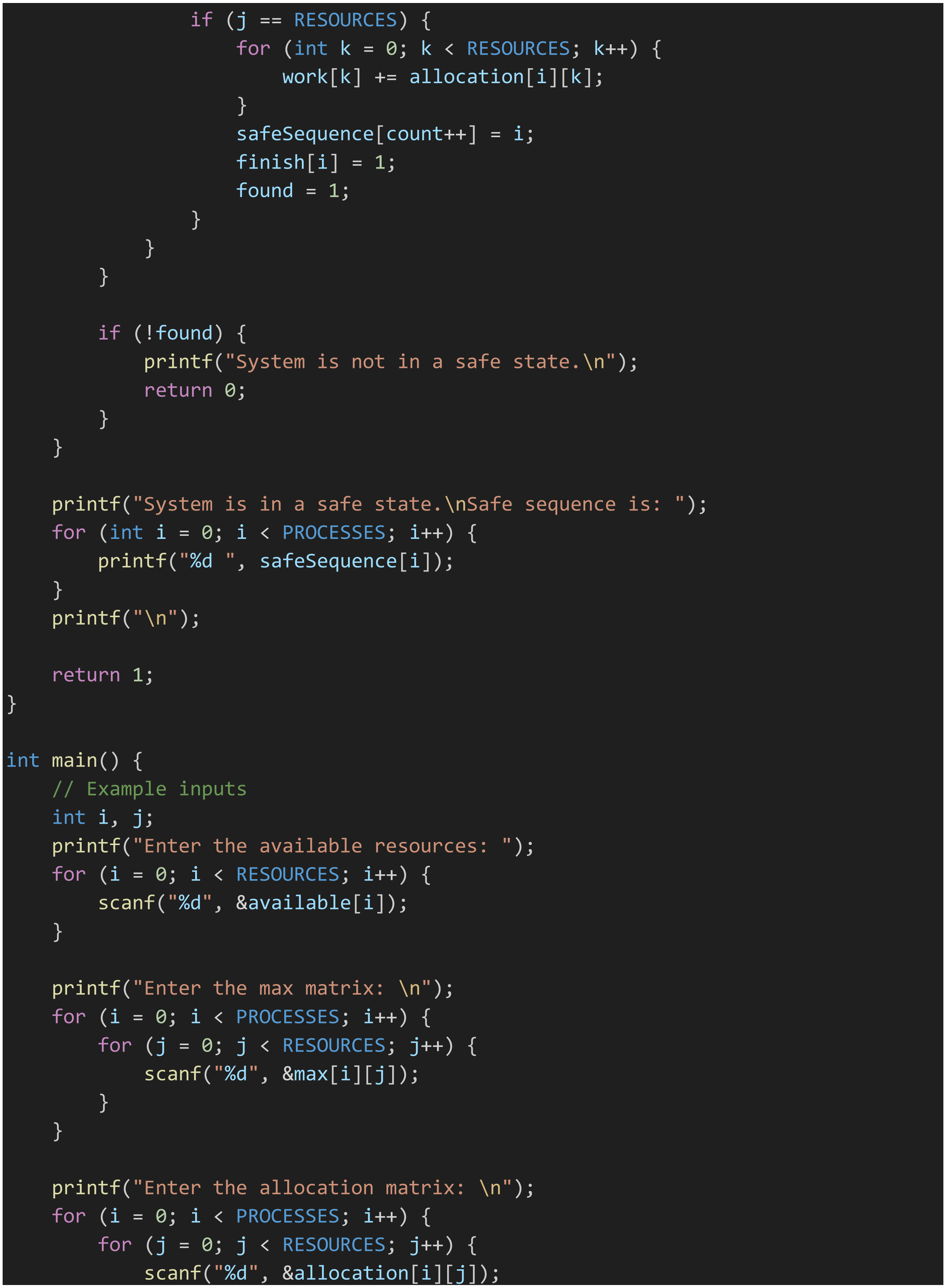


```
Menu:
1. Produce
2. Consume
3. Exit
Enter your choice: 1
Producer produces item 1

Menu:
1. Produce
2. Consume
3. Exit
Enter your choice: 1
Producer produces item 2

Menu:
1. Produce
2. Consume
3. Exit
Enter your choice: 2
Consumer consumes item 1

Menu:
1. Produce
2. Consume
3. Exit
Enter your choice: 2
Consumer consumes item 2

Menu:
1. Produce
2. Consume
3. Exit
Enter your choice: 2
Buffer is empty!

Menu:
1. Produce
2. Consume
3. Exit
Enter your choice: 3
Exiting program.
PS C:\Users\nisha\OneDrive\Desktop\C - Codes> _
```

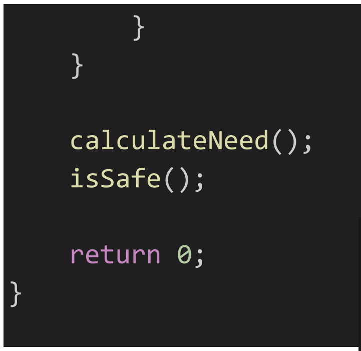
Que 2. Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.



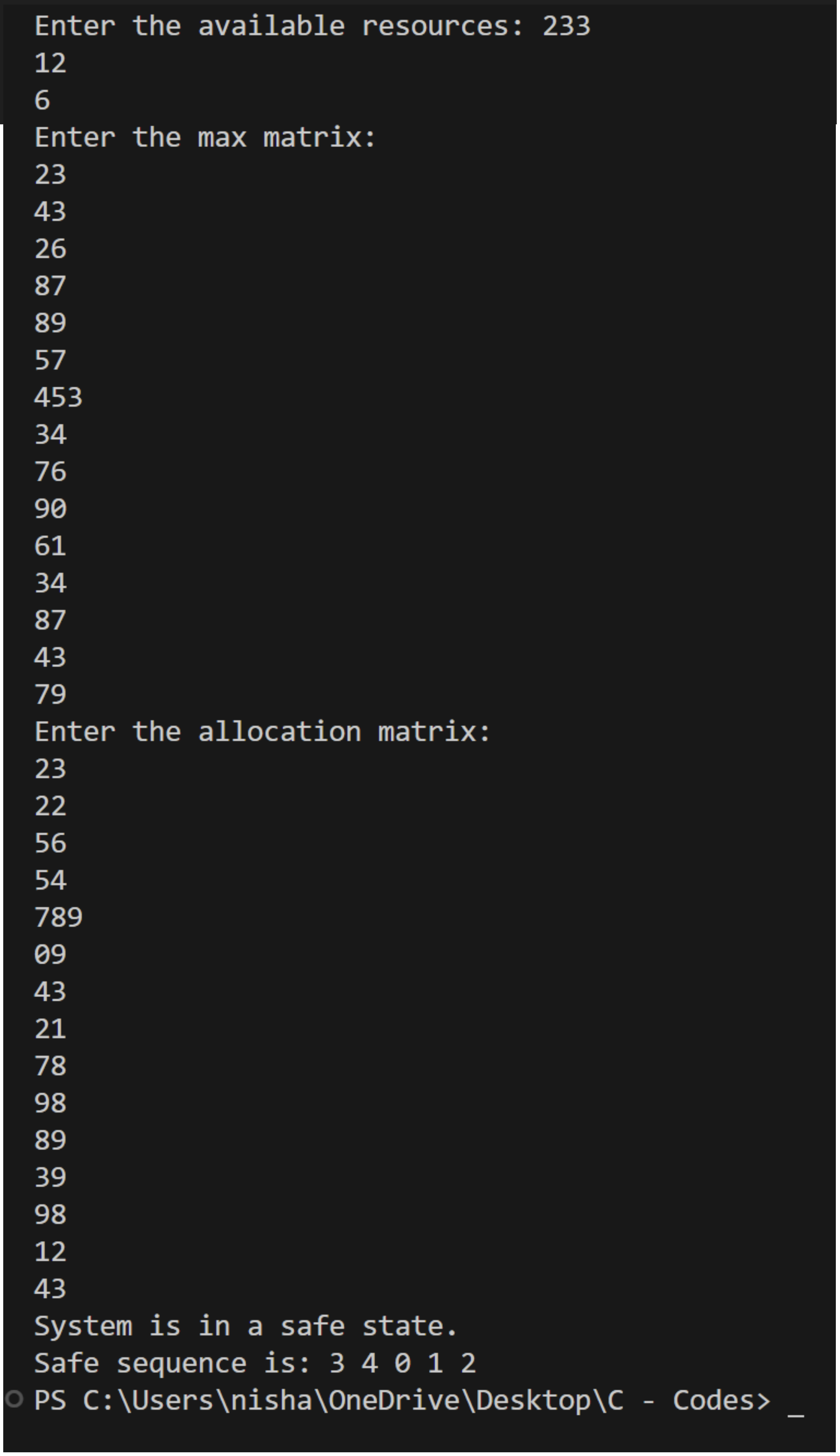
```
#include <stdio.h>

#define PROCESSES 5   // Number of processes
#define RESOURCES 3   // Number of resource types

int available[RESOURCES];                  // Available resources
int max[PROCESSES][RESOURCES];             // Maximum demand matrix
int allocation[PROCESSES][RESOURCES];      // Allocation matrix
int need[PROCESSES][RESOURCES];            // Need matrix

void calculateNeed() {
    for (int i = 0; i < PROCESSES; i++) {
        for (int j = 0; j < RESOURCES; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }
}

int isSafe() {
    int work[RESOURCES];
    int finish[PROCESSES] = {0};
    int safeSequence[PROCESSES];
    int count = 0;

    for (int i = 0; i < RESOURCES; i++) {
        work[i] = available[i];
    }

    while (count < PROCESSES) {
        int found = 0;
        for (int i = 0; i < PROCESSES; i++) {
            if (finish[i] == 0) {
                int j;
                for (j = 0; j < RESOURCES; j++) {
                    if (need[i][j] > work[j]) {
                        break;
                    }
                }
```



```
                if (j == RESOURCES) {
                    for (int k = 0; k < RESOURCES; k++) {
                        work[k] += allocation[i][k];
                    }
                    safeSequence[count++] = i;
                    finish[i] = 1;
                    found = 1;
                }
            }
        }

        if (!found) {
            printf("System is not in a safe state.\n");
            return 0;
        }
    }

    printf("System is in a safe state.\nSafe sequence is: ");
    for (int i = 0; i < PROCESSES; i++) {
        printf("%d ", safeSequence[i]);
    }
    printf("\n");

    return 1;
}

int main() {
    // Example inputs
    int i, j;
    printf("Enter the available resources: ");
    for (i = 0; i < RESOURCES; i++) {
        scanf("%d", &available[i]);
    }

    printf("Enter the max matrix: \n");
    for (i = 0; i < PROCESSES; i++) {
        for (j = 0; j < RESOURCES; j++) {
            scanf("%d", &max[i][j]);
        }
    }

    printf("Enter the allocation matrix: \n");
    for (i = 0; i < PROCESSES; i++) {
        for (j = 0; j < RESOURCES; j++) {
            scanf("%d", &allocation[i][j]);
```



```
        }
    }

    calculateNeed();
    isSafe();

    return 0;
}
```



```
Enter the available resources: 233
12
6
Enter the max matrix:
23
43
26
87
89
57
453
34
76
90
61
34
87
43
79
Enter the allocation matrix:
23
22
56
54
789
09
43
21
78
98
89
39
98
12
43
System is in a safe state.
Safe sequence is: 3 4 0 1 2
○ PS C:\Users\nisha\OneDrive\Desktop\C - Codes> _
```

Observations :-

Producer-Consumer (Semaphore-based) Algorithm

- Keeps Things in Sync: Semaphores are used to make sure that only one producer or consumer accesses the buffer at a time, which prevents errors.
- Buffer Control: Keeps track of buffer slots using `full` and `empty` semaphores, so producers know when the buffer is full, and consumers know when it's empty.
- Handles Waiting: If the buffer is full, the producer waits; if it's empty, the consumer waits. This way, neither side overflows or underflows the buffer.
- Efficient Use: Producers and consumers only operate when conditions are met, saving resources and processing time.

Banker's Algorithm (Deadlock Avoidance)

- Avoids Deadlocks: The algorithm checks if it's safe to grant resources to processes to prevent the system from getting stuck.
- Makes Safe Choices: For each resource request, it calculates whether granting it will keep the system in a "safe" state, meaning that every process can still complete.

- Simulates First: It pretends to allocate resources before actually doing it, which makes sure that there are always enough resources available for other processes.
- Works Best with Known Limits: The algorithm needs to know in advance how many resources each process could request, which is practical only when resource needs are predictable.

AP22110010245 | Nishant.