ASSIGNMENT

By- Vinay Pratap Singh

Roll no 2022A1R024

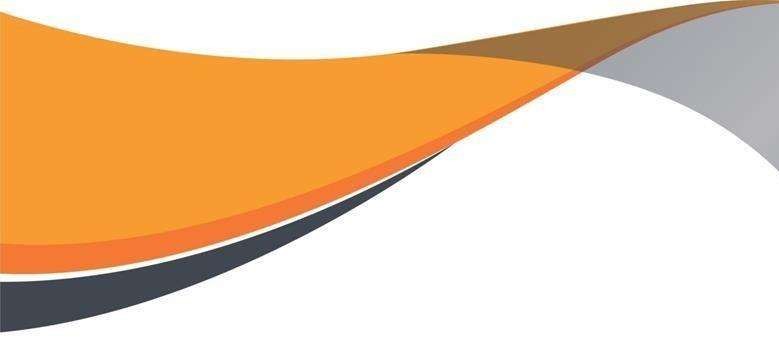
Semester-3rd

Department Name-CSE



Model Institute of Engineering & Technology (Autonomous)

(Permanently Affiliated to the University of Jammu, Accredited by NAAC with “A” Grade)

Jammu, India 2023

ASSIGNMENT

Subject Code: COM 302

Due Date: 30/11/23

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Question Number | Course Outcomes | Blooms’ Level | Maximum Marks | Marks Obtain |
| Q1 | CO 4 | 3-6 | 10 |  |
| Q2 | CO 5 | 3-6 | 10 |  |
|  | Total Marks |  | 20 |  |

Be

|  |
| --- |
| Faculty signature: |
| Email : Mekhla.cse@mietjammu.in |

**Task 1:** Analyze the differences between mutex locks and semaphores in terms of functionality and use cases for synchronization. Explain situations in which you would choose one over the other and provide specific examples to support your analysis.

Ans.- Mutex locks and semaphores serve as synchronization mechanisms in concurrent programming, aiming to control access to shared resources and prevent race conditions. Despite their common goal, they differ in functionality and use cases.

* **Mutex Locks: Functionality:**
  + A mutex, short for mutual exclusion, acts as a binary semaphore with ownership semantics, allowing exclusive access to a critical section by one thread at a time.
  + When a thread acquires a mutex, it gains sole access to the protected resource, and any other attempting thread is blocked until the mutex is released.
* **Uses:**
  + Mutex locks are ideal for safeguarding a single, non-shareable resource, such as a variable or a specific code section.
  + They are well-suited for scenarios where only one thread should execute a critical code section concurrently.
* **Example:**
  + Consider a scenario where multiple threads are updating a shared counter. Using a mutex ensures that only one thread can modify the counter at any given time.

**CODE:**

#include <stdio.h>

#include <stdatomic.h>

// Shared resource

int shared\_resource = 0;

// Mutex-like lock using atomic operations

\_Atomic int lock = 0;

// Function that increments the shared resource

void increment\_shared\_resource() {

for (int i = 0; i < 1000000; ++i) {

// Acquire the lock before accessing the shared resource

while (atomic\_exchange\_explicit(&lock, 1, memory\_order\_acquire) != 0)

; // Spin until lock is acquired

// Critical section: shared resource modification

shared\_resource++;

// Release the lock after modifying the shared resource

atomic\_store\_explicit(&lock, 0, memory\_order\_release);

}

}

int main() {

// Create two threads (simulated by function calls)

increment\_shared\_resource();

increment\_shared\_resource();

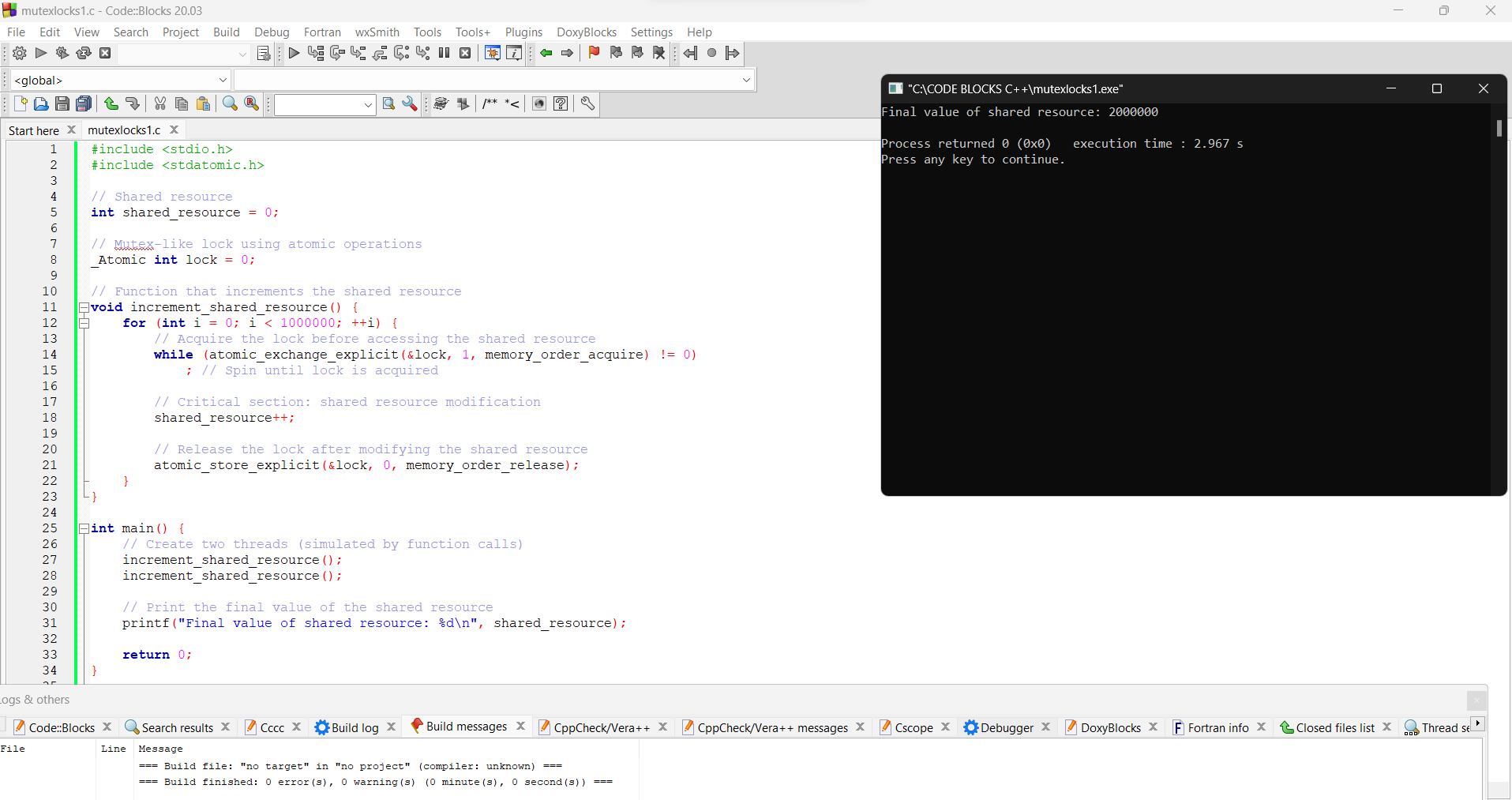
// Print the final value of the shared resource

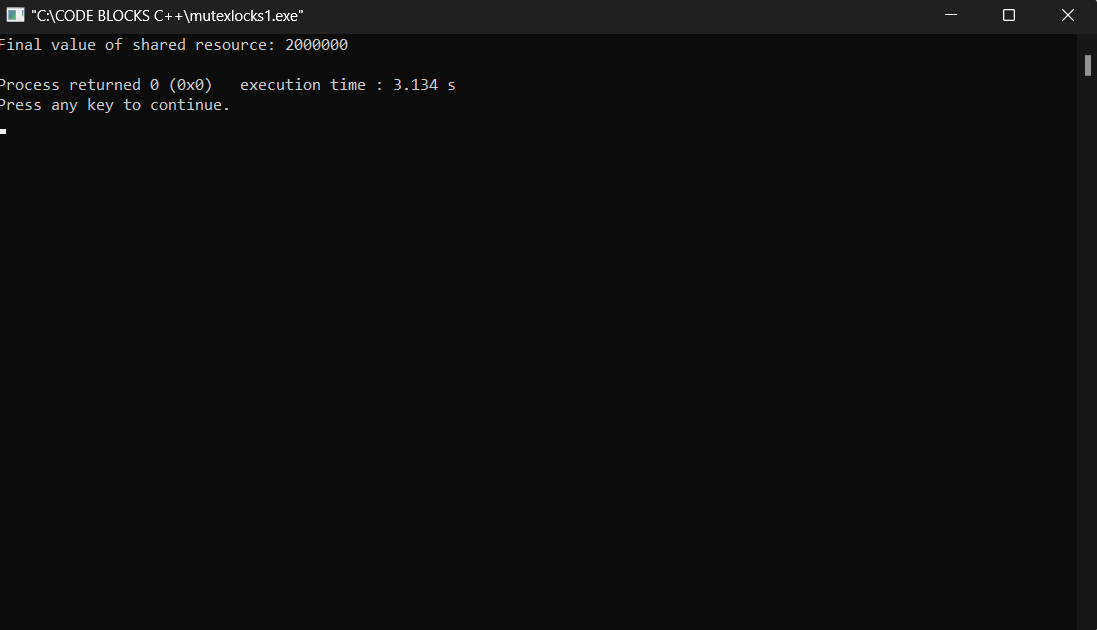
printf("Final value of shared resource: %d\n", shared\_resource);

return 0;

}

**OUTPUT:**





**Semaphores:** A semaphore is a synchronization primitive used in concurrent programming to control access to a shared resource or to manage the execution order of multiple threads. It was introduced by Edsger Dijkstra in 1965. Semaphores can be thought of as counters that help coordinate the activities of different threads.

* **Functionality:**
  + Semaphores are more versatile synchronization primitives that cater to both mutual exclusion and signaling between threads.
  + Unlike mutexes, semaphores can have values greater than 1, enabling more complex synchronization scenarios.
* **Use Cases:**
  + Semaphores are suitable when dealing with multiple instances of a resource, allowing a specified number of threads simultaneous access.
  + They find application in scenarios requiring signaling between threads, where one thread signals an event and another awaits the signal.
* **Example:**
  + Consider a situation where a limited number of resources, like database connections, are available. A semaphore can control access to these resources, permitting only a specified number of threads to use them concurrently.

**CODE:**

**#include <stdio.h>**

**#include <stdatomic.h>**

**#include <unistd.h>**

**typedef struct {**

**\_Atomic int lock;**

**} my\_semaphore;**

**void my\_semaphore\_init(my\_semaphore \*sem) {**

**atomic\_init(&sem->lock, 0);**

**}**

**void my\_semaphore\_wait(my\_semaphore \*sem) {**

**while (atomic\_exchange(&sem->lock, 1)) {**

**// Spin until lock is acquired**

**}**

**}**

**void my\_semaphore\_signal(my\_semaphore \*sem) {**

**atomic\_store(&sem->lock, 0);**

**}**

**void critical\_section(const char \*name, my\_semaphore \*sem) {**

**for (int i = 0; i < 5; ++i) {**

**my\_semaphore\_wait(sem);**

**printf("%s is in the critical section\n", name);**

**usleep(100000); // Simulate some work**

**my\_semaphore\_signal(sem);**

**}**

**}**

**int main() {**

**my\_semaphore sem;**

**my\_semaphore\_init(&sem);**

**// No need to fork processes, use threads if necessary**

**// Perform some work in the critical section**

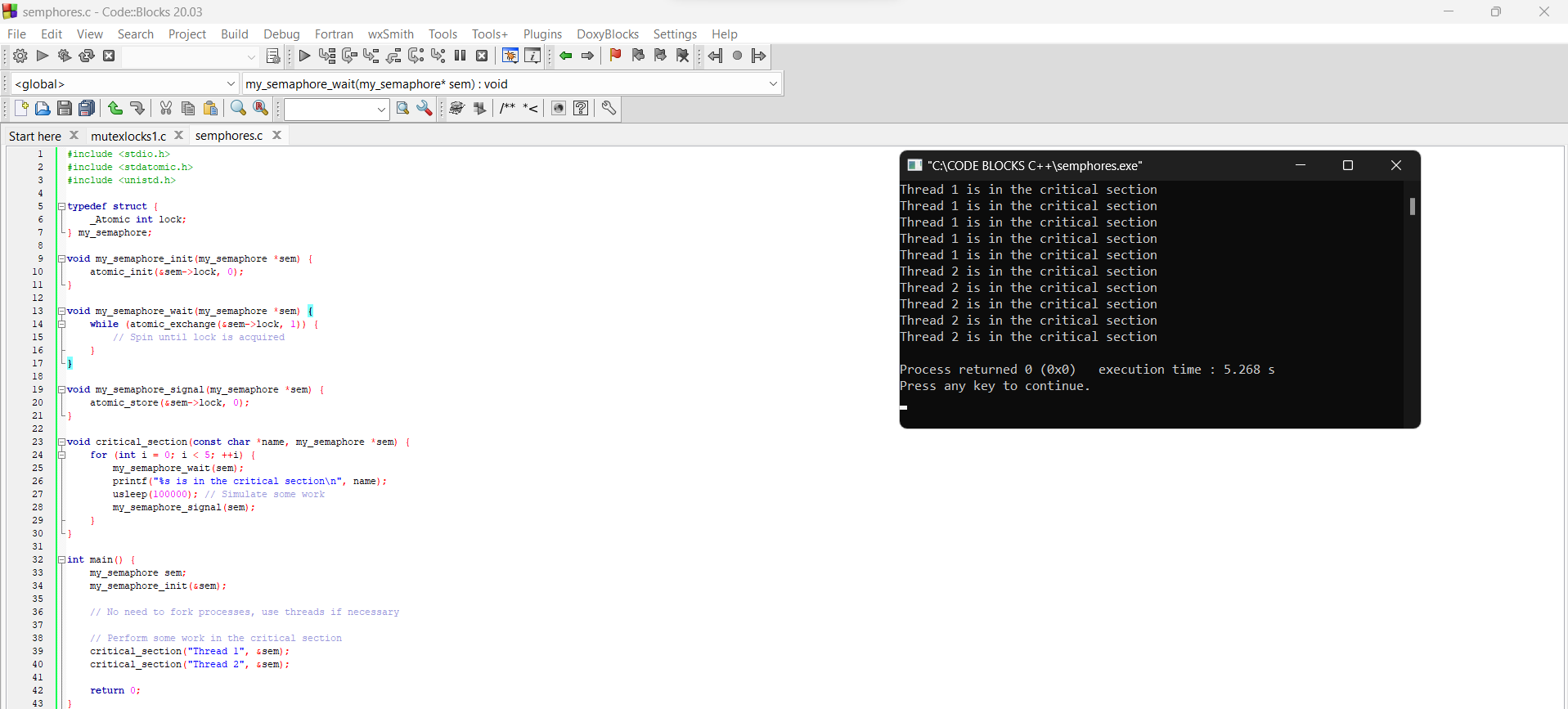
**critical\_section("Thread 1", &sem);**

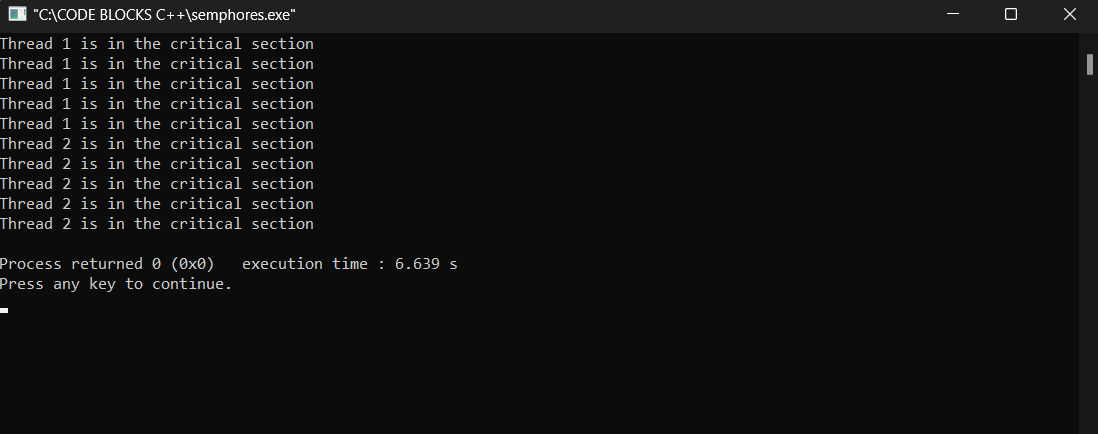
**critical\_section("Thread 2", &sem);**

**return 0;**

**}**

**OUTPUT:**





**Choosing Between Mutex Locks and Semaphores:**

* Use **mutex locks** when safeguarding access to a single resource, and only one thread should access it concurrently.
* Use **semaphores** when managing multiple instances of a resource or when signaling capabilities are needed.
* Note that if a semaphore's value is 1 and it is used for mutual exclusion, it essentially behaves like a mutex.
* The decision between mutex locks and semaphores hinges on the specific requirements of your concurrent programming scenario.

**TASK 2:** Write a program that implements the Banker's algorithm for deadlock avoidance. Simulate multiple processes making resource requests and releases. Demonstrate how the algorithm ensures safe states and prevents deadlocks. Discuss the advantages and limitations of the Banker's algorithm.

Ans- **Banker's algorithm:** The Banker's algorithm, introduced by Edsger Dijkstra in 1965, is a deadlock avoidance mechanism widely employed in operating systems. Its primary objective is to manage the allocation of resources to processes in a manner that minimizes the risk of deadlock occurrences.

Key Components and Principles:

1. Resource Management:

- The system encompasses various resource types, each having a specific number of available instances.

- Resources are categorized, such as printers, CPUs, or memory.

2. Process Interaction:

- Multiple processes coexist within the system, with the ability to request and release resources dynamically during their execution.

- Each process declares its maximum demand for each resource type.

3. Allocation Matrix:

- An allocation matrix is maintained, depicting the current allocation of resources for each process by resource type.

4. Maximum Claim Matrix:

- The system keeps track of a maximum claim matrix, representing the highest resource demand each process may make.

5. Need Matrix:

- Derived from the maximum claim and allocation matrices, the need matrix reflects the outstanding resource needs for each process and resource type.

6. Safety Algorithm:

- Prior to resource allocation, the safety algorithm assesses the system's safety status.

- A system is considered safe if there exists a process sequence wherein each process can attain its maximum resources and subsequently release them, allowing the next process to execute.

7. Resource Requests:

- Before allocating resources to a process, the Banker's algorithm evaluates whether the allocation preserves the system's safety.

- Resource allocation proceeds only if the system remains in a safe state; otherwise, the requesting process is required to wait.

8. Resource Releases:

- Upon process completion, the allocated resources are released.

- The available resources are augmented, potentially enabling other waiting processes to obtain the released resources.

The Banker's algorithm mitigates deadlock risks by ensuring resource allocation follows principles such as avoiding circular waiting, holding and waiting, and non-preemption. This approach facilitates dynamic resource allocation while minimizing the likelihood of unsafe states leading to deadlock situations.

The Banker's algorithm, a foundational theoretical model, has influenced the development of more practical deadlock avoidance strategies in contemporary operating systems.

**CODE:**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 5

#define MAX\_RESOURCES 3

int available[MAX\_RESOURCES];

int max\_claim[MAX\_PROCESSES][MAX\_RESOURCES];

int allocation[MAX\_PROCESSES][MAX\_RESOURCES];

int need[MAX\_PROCESSES][MAX\_RESOURCES];

void initialize() {

printf("Enter available resources:\n");

for (int i = 0; i < MAX\_RESOURCES; ++i) {

scanf("%d", &available[i]);

}

printf("Enter the maximum claim matrix:\n");

for (int i = 0; i < MAX\_PROCESSES; ++i) {

for (int j = 0; j < MAX\_RESOURCES; ++j) {

scanf("%d", &max\_claim[i][j]);

}

}

for (int i = 0; i < MAX\_PROCESSES; ++i) {

for (int j = 0; j < MAX\_RESOURCES; ++j) {

allocation[i][j] = 0;

need[i][j] = max\_claim[i][j];

}

}

}

void display\_state() {

printf("\nAvailable resources: ");

for (int i = 0; i < MAX\_RESOURCES; ++i) {

printf("%d ", available[i]);

}

printf("\n\nMaximum claim matrix:\n");

for (int i = 0; i < MAX\_PROCESSES; ++i) {

for (int j = 0; j < MAX\_RESOURCES; ++j) {

printf("%d ", max\_claim[i][j]);

}

printf("\n");

}

printf("\nAllocation matrix:\n");

for (int i = 0; i < MAX\_PROCESSES; ++i) {

for (int j = 0; j < MAX\_RESOURCES; ++j) {

printf("%d ", allocation[i][j]);

}

printf("\n");

}

printf("\nNeed matrix:\n");

for (int i = 0; i < MAX\_PROCESSES; ++i) {

for (int j = 0; j < MAX\_RESOURCES; ++j) {

printf("%d ", need[i][j]);

}

printf("\n");

}

}

bool is\_safe\_state(int process, int request[]) {

for (int i = 0; i < MAX\_RESOURCES; ++i) {

if (request[i] > need[process][i] || request[i] > available[i]) {

return false;

}

}

for (int i = 0; i < MAX\_RESOURCES; ++i) {

available[i] -= request[i];

allocation[process][i] += request[i];

need[process][i] -= request[i];

}

int work[MAX\_RESOURCES];

for (int i = 0; i < MAX\_RESOURCES; ++i) {

work[i] = available[i];

}

bool finish[MAX\_PROCESSES] = {false};

while (true) {

bool found = false;

for (int i = 0; i < MAX\_PROCESSES; ++i) {

if (!finish[i]) {

bool can\_allocate = true;

for (int j = 0; j < MAX\_RESOURCES; ++j) {

if (need[i][j] > work[j]) {

can\_allocate = false;

break;

}

}

if (can\_allocate) {

for (int j = 0; j < MAX\_RESOURCES; ++j) {

work[j] += allocation[i][j];

}

finish[i] = true;

found = true;

}

}

}

if (!found) {

break;

}

}

for (int i = 0; i < MAX\_RESOURCES; ++i) {

available[i] += request[i];

allocation[process][i] -= request[i];

need[process][i] += request[i];

}

for (int i = 0; i < MAX\_PROCESSES; ++i) {

if (!finish[i]) {

return false;

}

}

return true;

}

void request\_resources(int process) {

int request[MAX\_RESOURCES];

printf("Enter the resource request for process %d:\n", process + 1);

for (int i = 0; i < MAX\_RESOURCES; ++i) {

scanf("%d", &request[i]);

}

if (is\_safe\_state(process, request)) {

for (int i = 0; i < MAX\_RESOURCES; ++i) {

available[i] -= request[i];

allocation[process][i] += request[i];

need[process][i] -= request[i];

}

printf("Resource request granted.\n");

display\_state();

} else {

printf("Resource request denied. System would enter an unsafe state.\n");

}

}

void release\_resources(int process) {

int release[MAX\_RESOURCES];

printf("Enter the resource release for process %d:\n", process + 1);

for (int i = 0; i < MAX\_RESOURCES; ++i) {

scanf("%d", &release[i]);

}

for (int i = 0; i < MAX\_RESOURCES; ++i) {

available[i] += release[i];

allocation[process][i] -= release[i];

need[process][i] += release[i];

}

printf("Resources released.\n");

display\_state();

}

int main() {

initialize();

while (1) {

int choice, process;

printf("\n1. Display state\n");

printf("2. Request resources\n");

printf("3. Release resources\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

display\_state();

break;

case 2:

printf("Enter the process making the request (1 to %d): ", MAX\_PROCESSES);

scanf("%d", &process);

process--; // Adjust for array indexing

if (process >= 0 && process < MAX\_PROCESSES) {

request\_resources(process);

} else {

printf("Invalid process number.\n");

}

break;

case 3:

printf("Enter the process releasing resources (1 to %d): ", MAX\_PROCESSES);

scanf("%d", &process);

process--; // Adjust for array indexing

if (process >= 0 && process < MAX\_PROCESSES) {

release\_resources(process);

} else {

printf("Invalid process number.\n");

}

break;

case 4:

return 0;

default:

printf("Invalid choice. Please enter a valid option.\n");

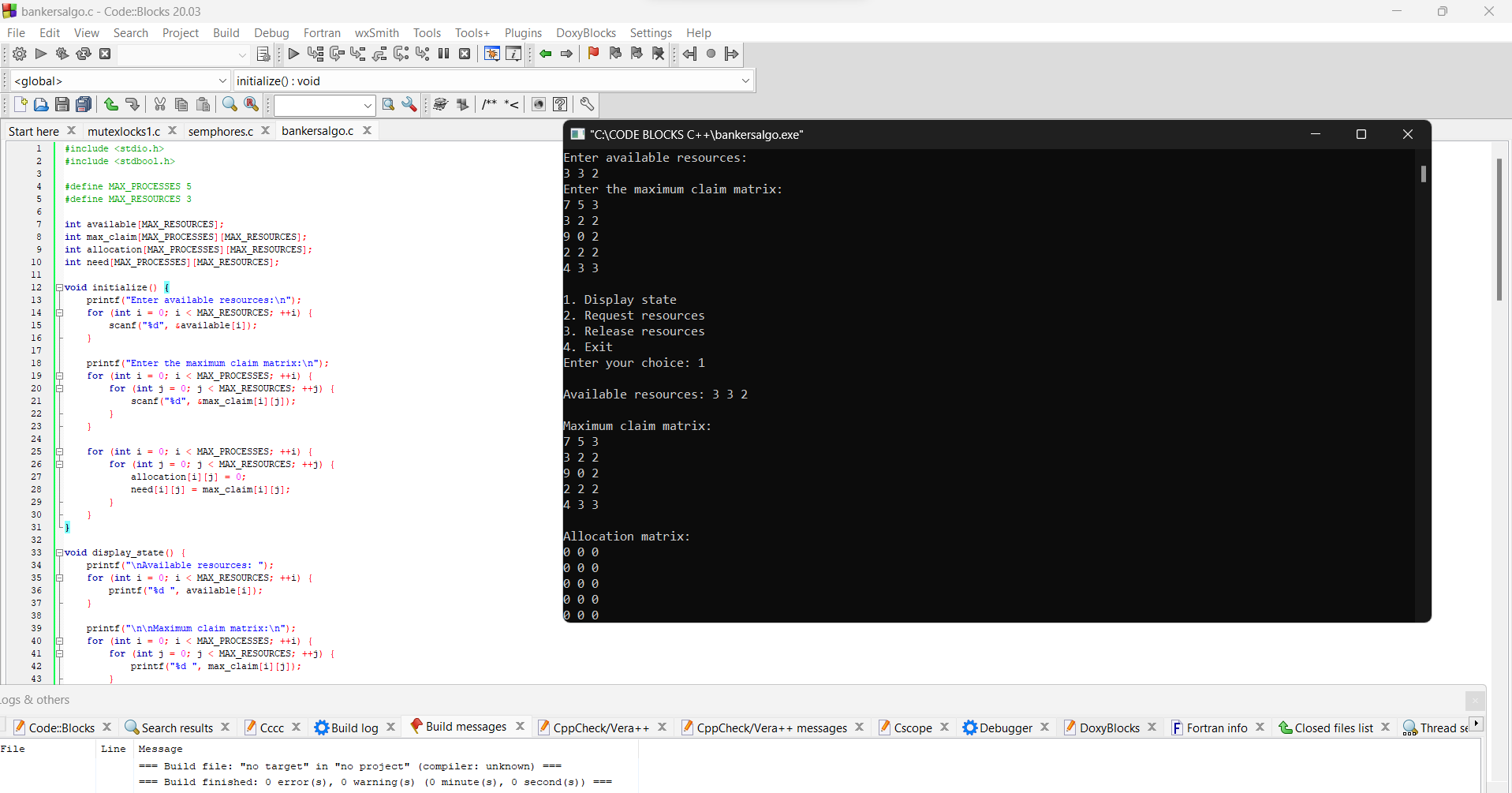
}

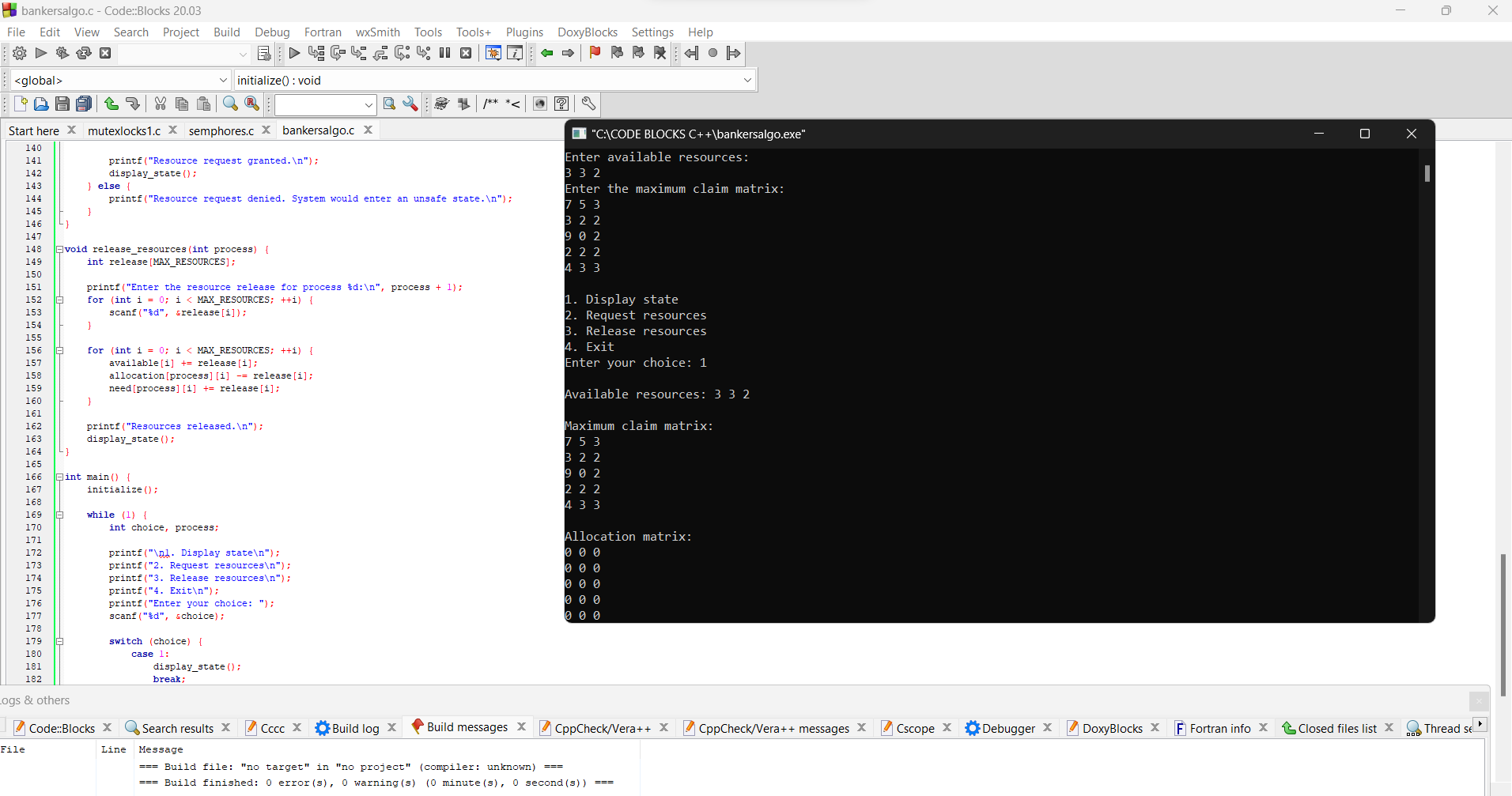
}

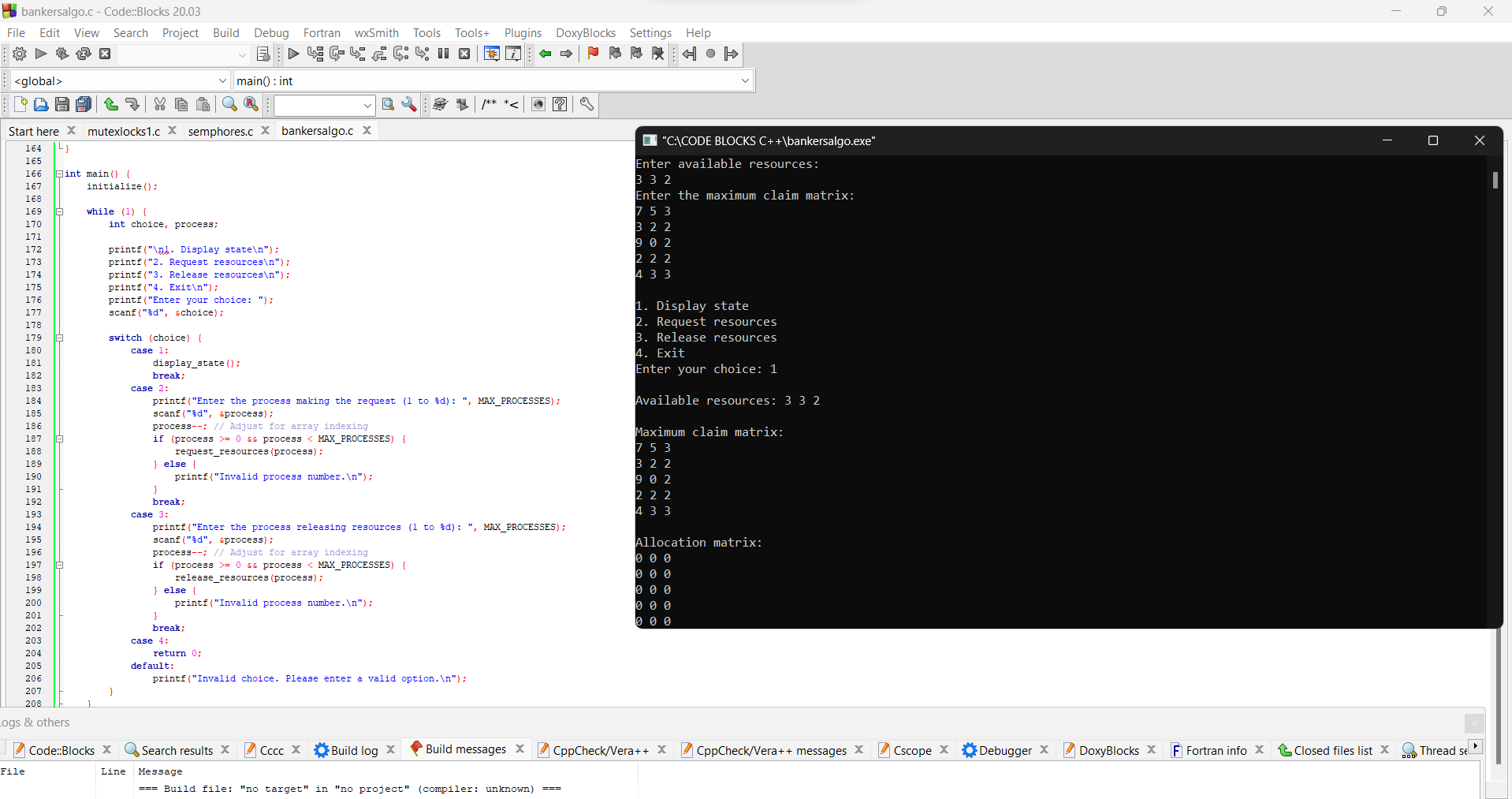
return 0;

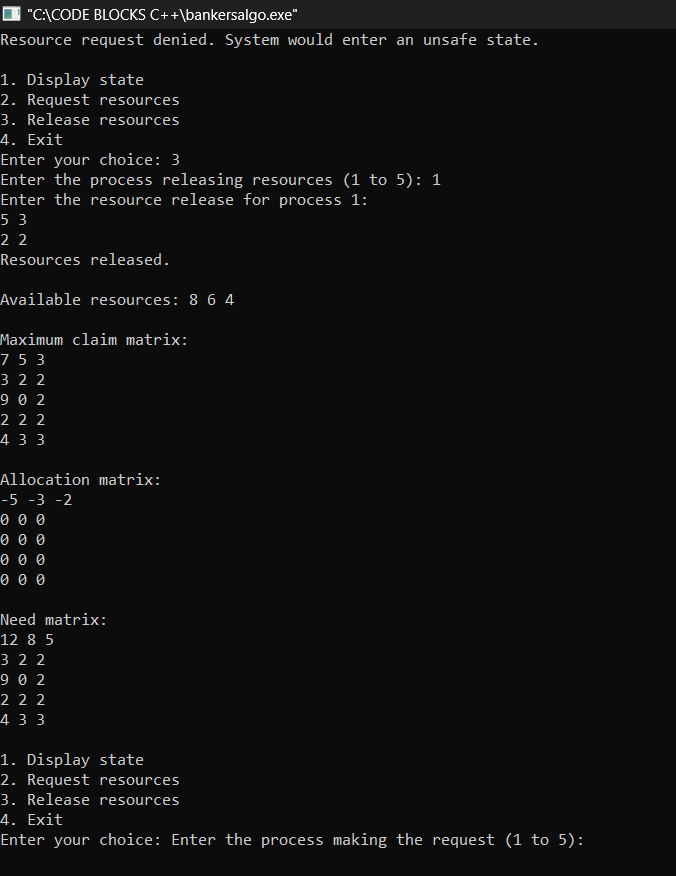
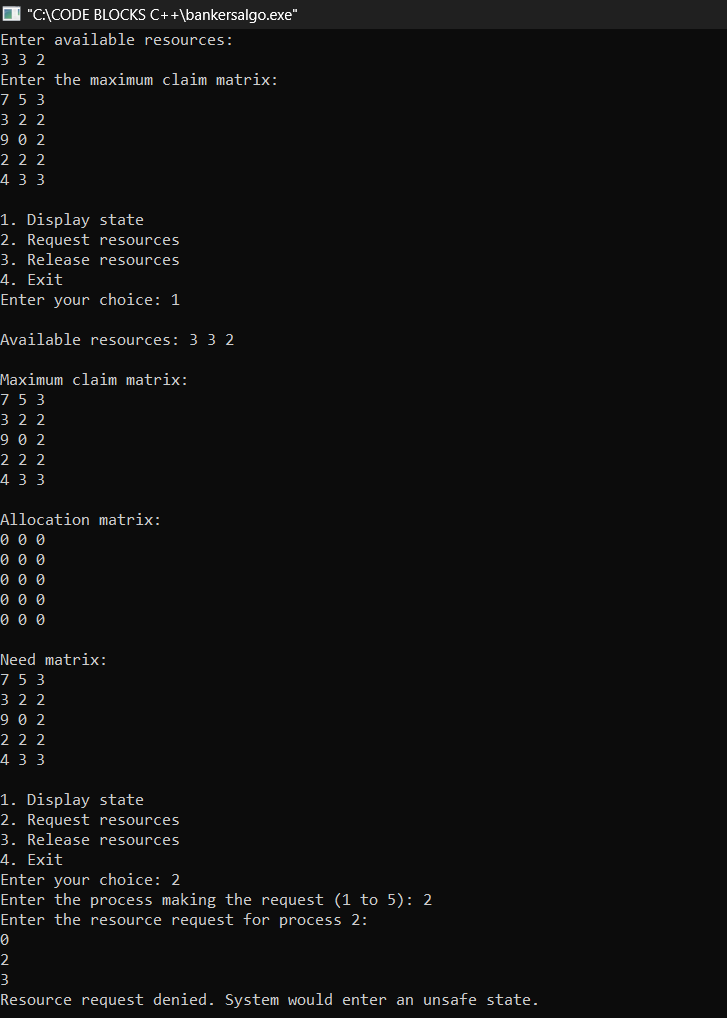
}

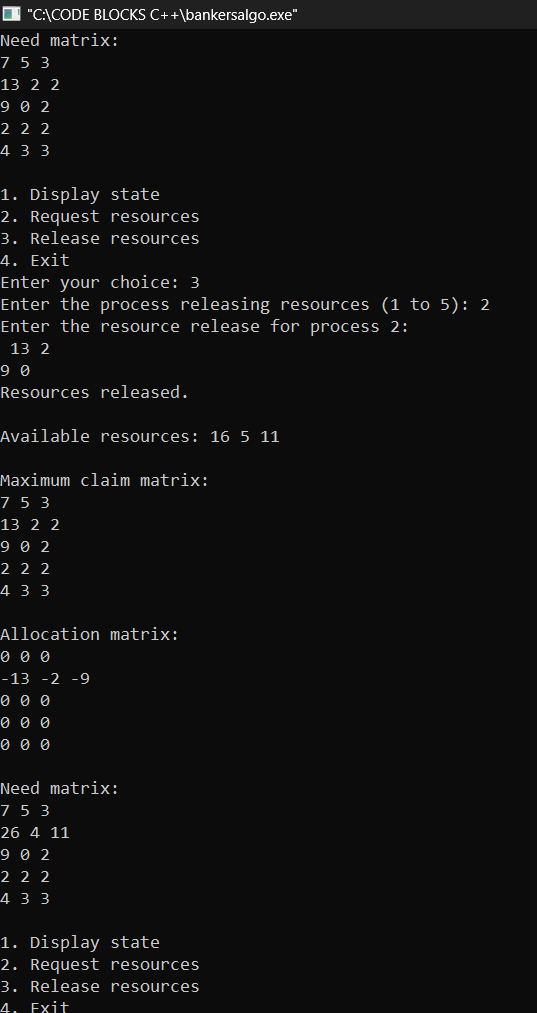
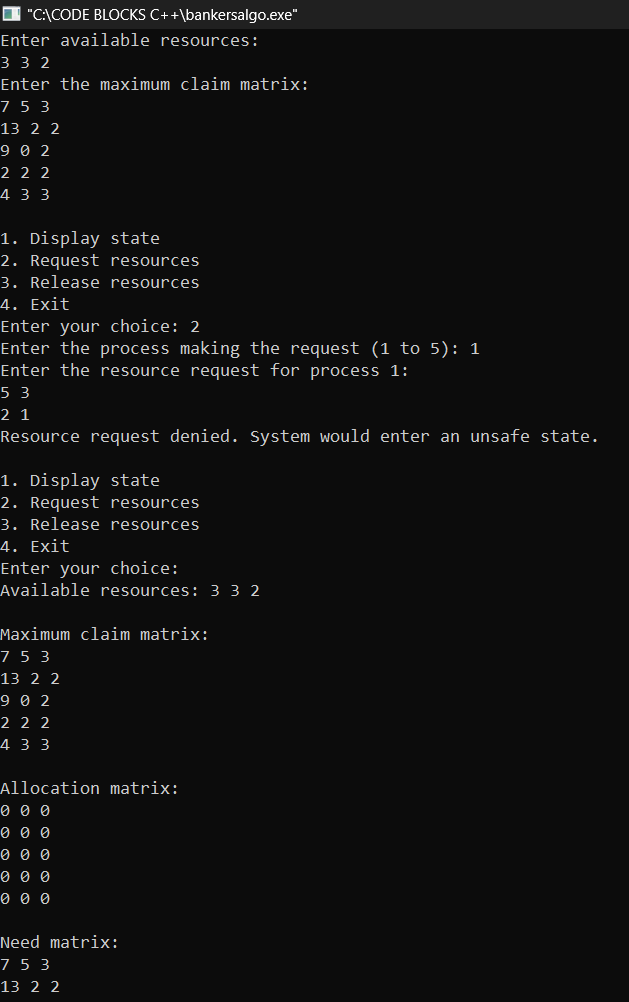
**OUTPUTS:**











**Advantages of the Banker's Algorithm:**

* **Deadlock Avoidance:** The primary advantage is its ability to avoid deadlock by checking whether the system will remain in a safe state after resource allocation.
* **Maximized Resource Utilization:** It attempts to maximize resource utilization by only granting requests that won't jeopardize the system's safety.
* **Dynamic Resource Allocation:** It allows for dynamic allocation of resources, responding to the changing resource needs of processes.

**Limitations of the Banker's Algorithm:**

* **Knowledge of Future Requests:** The algorithm requires knowledge of the maximum resource needs of each process, which may not always be known in advance.
* **Resource Hold-and-Wait:** The algorithm assumes that processes request all their required resources upfront, which may not be practical in all scenarios. This assumption leads to potential resource underutilization.
* **Static Allocation:** It may lead to underutilization of resources because the algorithm is conservative and does not grant resource requests unless it can guarantee a safe state.
* **Process Starvation:** There's a possibility of process starvation if a process's resource requests cannot be satisfied due to the conservative nature of the algorithm.
* **Sequential Resource Allocation:** Resources must be requested and released in a predetermined sequence, limiting the flexibility of process resource management.

The Banker's algorithm is a theoretical model that serves as a foundation for understanding deadlock avoidance strategies. While it has practical limitations, it provides a basis for more advanced deadlock avoidance techniques in real-world operating systems.