ASSIGNMENT

By- Abhinandan singh slathia

Roll no -2022A1R025

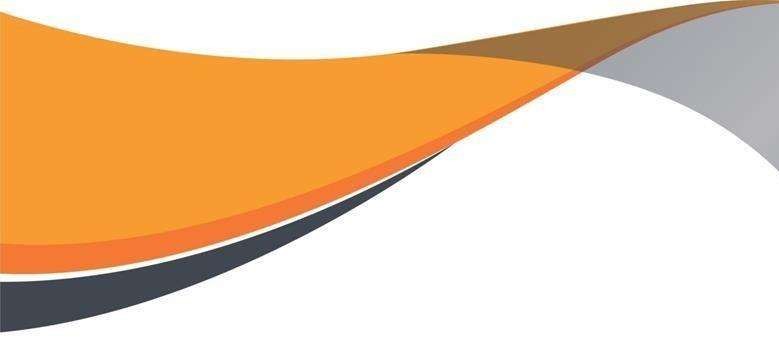
Semester-3rd

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: 4/12/23

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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.

A good synchronization process in an integrated system is important for managing shared resources and protecting society. Two tools commonly used for this purpose are mutexes and semaphores. Although they both share the goal of thread safety, they have unique properties that make them suitable for different situations.  
  
**Mutex lock:**

**Function:**  
  
**Binary state:** Mutex lock is a binary semaphore with two states, locked and unlocked.

**Ownership:** Normally mutex is taken and released by the same thread to provide exclusive ownership

during locked state.

**Exclusive access**: Mutex guarantees access to shared resources. If a thread holds the mutex, other threads must wait until it is released.

**Use Examples:**  
**Critical Sections:** Mutexes are ideal for protecting critical sections of code, allowing only one thread to  run on that section  
Protection Block Service: When there is an integration such as a file that needs to control access structure or file mutexes enable the operation.

**Death tolerance:** Mutexes can help prevent death because they are usually released when the thread they are running on encounters an error.

**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();

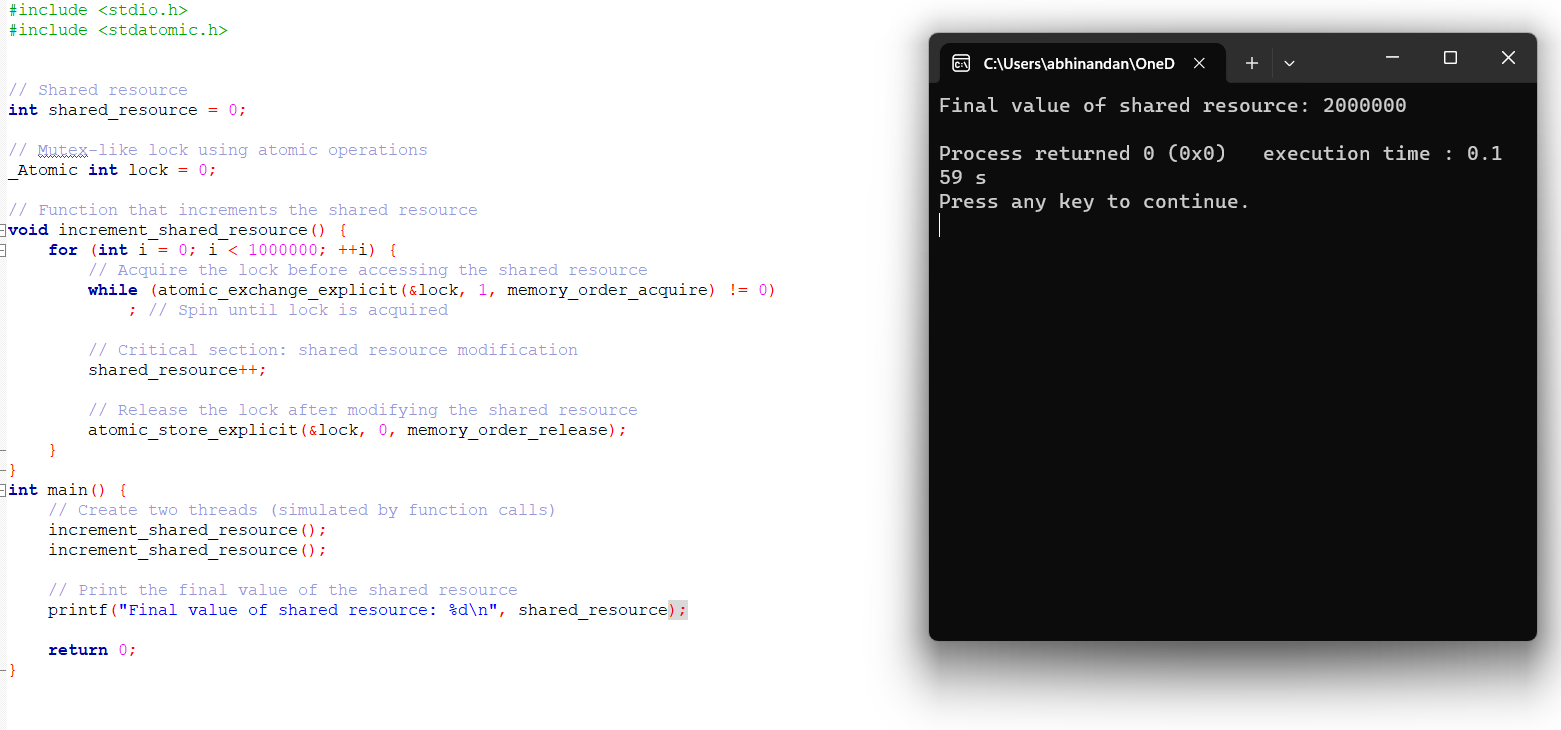
// Print the final value of the shared resource

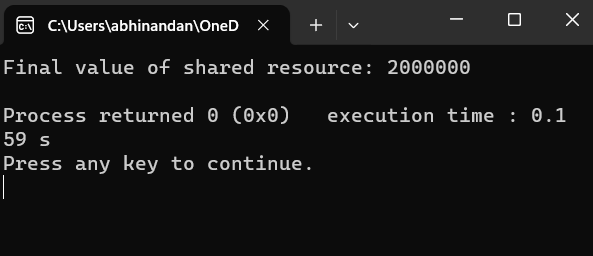
printf("Final value of shared resource: %d\n", shared\_resource);

return 0;

}

**Output-**

****

****

**Semaphore:**

**Function:**  
  
**Counting capacity**: The number of semaphore can be greater than 1; this allows multiple threads of execution to access the resource simultaneously.

**General:** Semaphores are widely used and are suitable for synchronization and counting events.

**Use Examples:**  
  
**Security Management:**Semaphores are useful for limiting the number of calls that can access resources simultaneously.

**Consumer Problem:** Semaphores solve synchronization problems such as controlling the interaction between producers and consumers.

**Task Synchronization:** Semaphores are useful for coordinating tasks that can run independently but need to be synchronized at some point.

**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

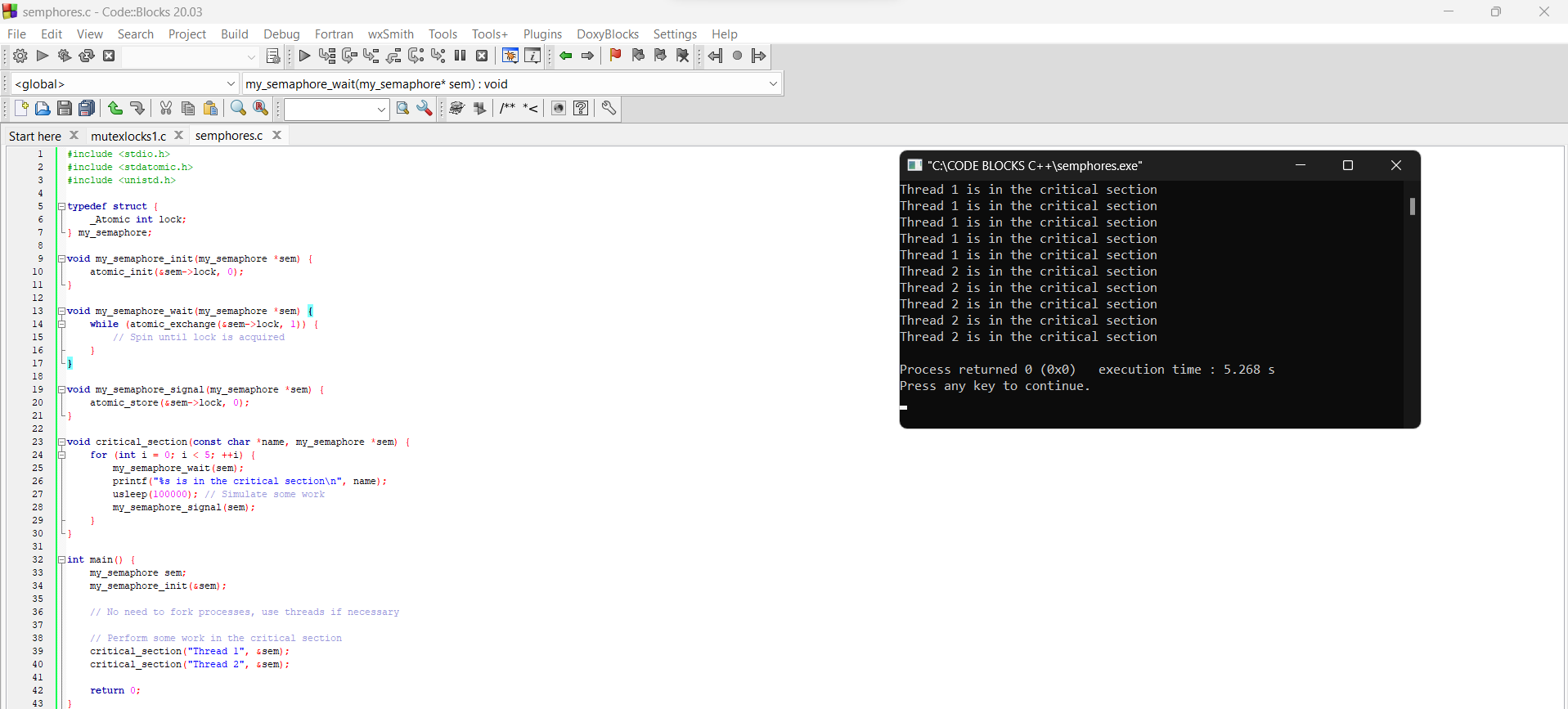
critical\_section("Thread 1", &sem);

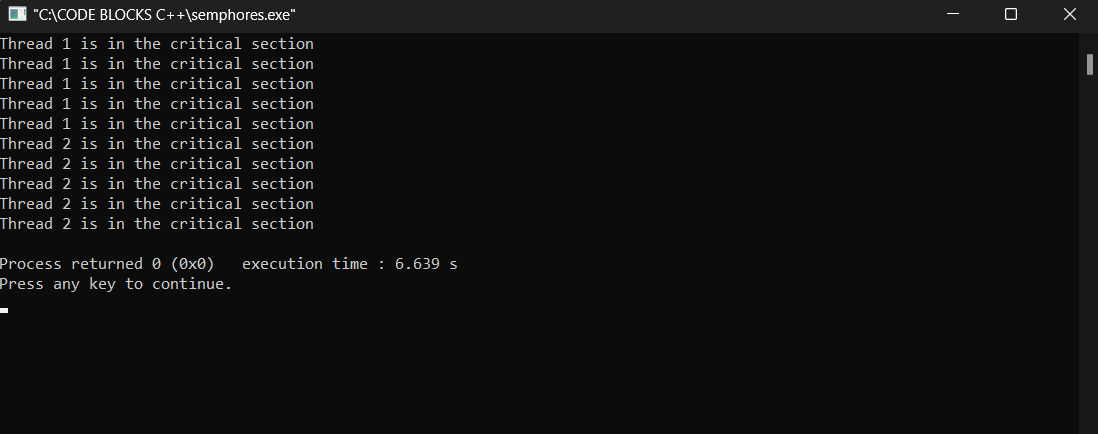
critical\_section("Thread 2", &sem);

return 0;

}

**OUTPUT:**





**Choose between mutexes and semaphores:**

**Mutex locks:**  
  
Good for situations that require access to specific resources.  
Best for critical sections where only one thread can be executed at a time. Best for events requiring net  members.

**Semaphore:**  
  
Suitable for situations where multiple threads can access resources simultaneously.  
Useful for large resources and situations where the number of successful accesses to resources must be limited.  
Effective when access is required or restricted.

In summary, although both mutexes and semaphores provide synchronization solutions, the choice depends on the specific needs of the program. Use mutex locks for special cases, select semaphores when multiple threads need to be controlled, and simplify computational semantics. Understanding the nuances of each system is crucial to creating strong and efficient systems.

**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 Edger 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. Follow the bank's algorithm to avoid death by simulating the request and release process. Below is a  simple C program that demonstrates the banker's algorithm.

Key Components and Principles:

1 The Banker's algorithm is a resource allocation and deadlock avoidance algorithm used in operating systems. Its key components and principles include:

# Key Components:

1. Processes:

- A set of processes that compete for resources.

2. Resources:

- A set of resource types, each with a certain number of instances. Examples of resources include CPU cycles, memory, printers, etc.

3. Maximum Claim Matrix (Max):

- A matrix indicating the maximum number of resources of each type that a process may need throughout its execution.

4. Allocation Matrix (Allocation):

A matrix indicating the number of resources of each type currently allocated to each process.

5. Need Matrix (Need):

A matrix indicating the remaining needs of each process, calculated as the difference between the maximum claim and the resources currently allocated.

6. Available Vector:

- A vector indicating the number of available resources of each type.

Principles:

1. Safety:

- The primary principle of the Banker's algorithm is to ensure the system's safety, which means that it should always be able to allocate resources in such a way that no deadlock will occur.

2. Request and Release:

- Processes must explicitly request resources before they begin their execution and release resources when they finish. This information is used to update the state of the system.

3. Resource Allocation Check:

- Before allocating resources to a process, the system checks whether the allocation will lead to a safe state. If not, the allocation is postponed until it can be guaranteed to be safe.

4. Safety Check:

- The safety check is performed using the concept of a safe sequence. A sequence of processes is considered safe if, for each process in the sequence, the resources it needs can be satisfied either by the currently available resources or by the resources held by processes earlier in the sequence.

5. Avoidance of Unsafe States:

- The Banker's algorithm ensures that the system avoids entering into an unsafe state, where deadlock becomes a possibility.

6. Deadlock Detection:

- The Banker's algorithm is proactive in preventing deadlock, as opposed to reactive strategies like deadlock detection and recovery. It does not allow the system to reach a state where deadlock is possible.

7. Dynamic Resource Allocation:

- The Banker's algorithm supports dynamic allocation of resources, allowing processes to request and release resources dynamically during their execution.

8. Conservative Resource Allocation:

- Resources are allocated conservatively based on the maximum needs of processes, ensuring that the system remains in a safe state.

9. Rollback Mechanism:

- If a resource allocation request would lead to an unsafe state, the system may need to roll back the allocation and try again later.

10. Information Requirement:

- The Banker's algorithm requires information about the maximum resource needs of each process in advance, which may not always be available or accurately estimated.

By adhering to these principles and utilizing the key components mentioned, the Banker's algorithm ensures the safe and efficient allocation of resources in a way that minimizes the risk of deadlocks.

**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");

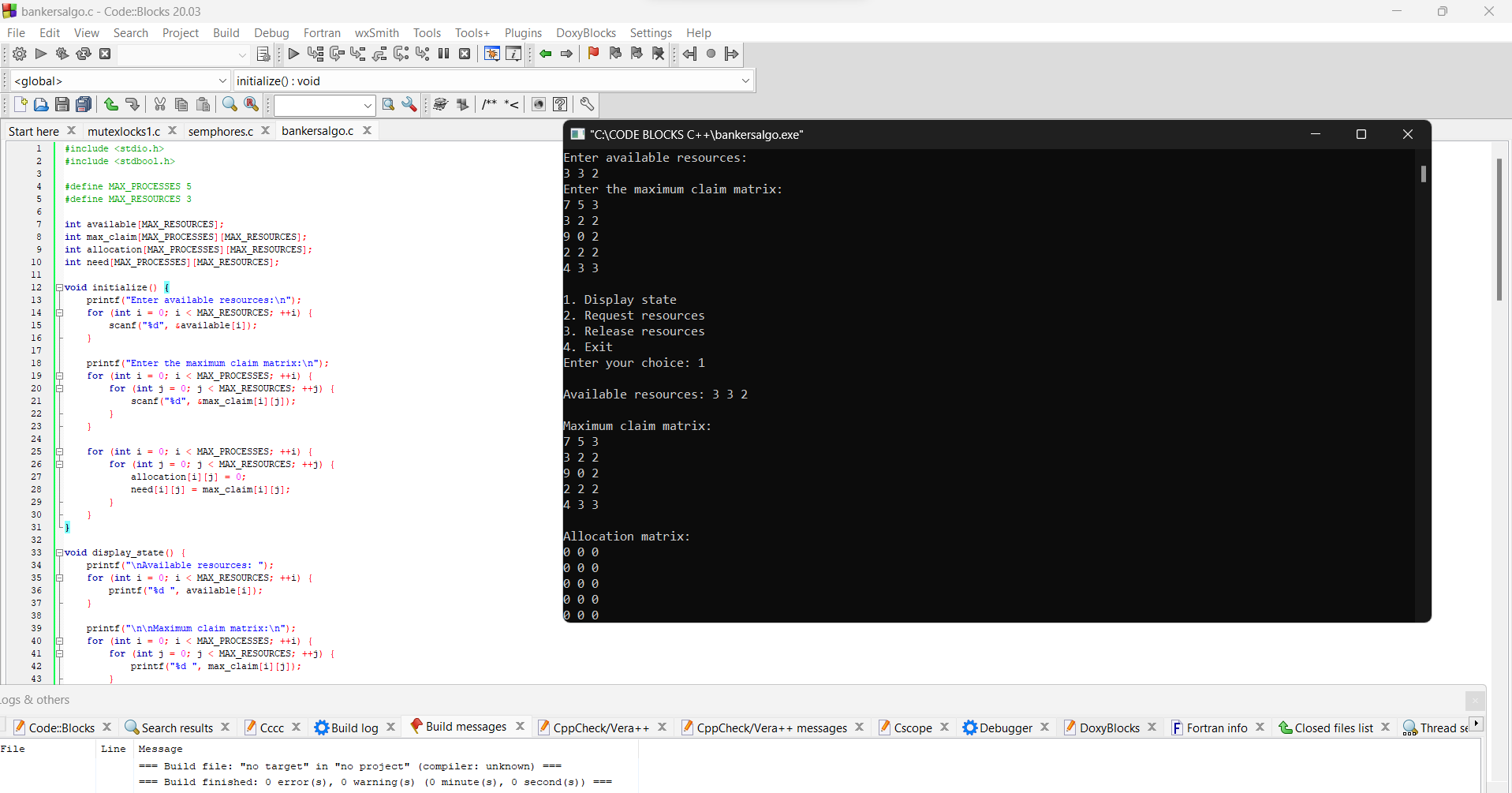
}

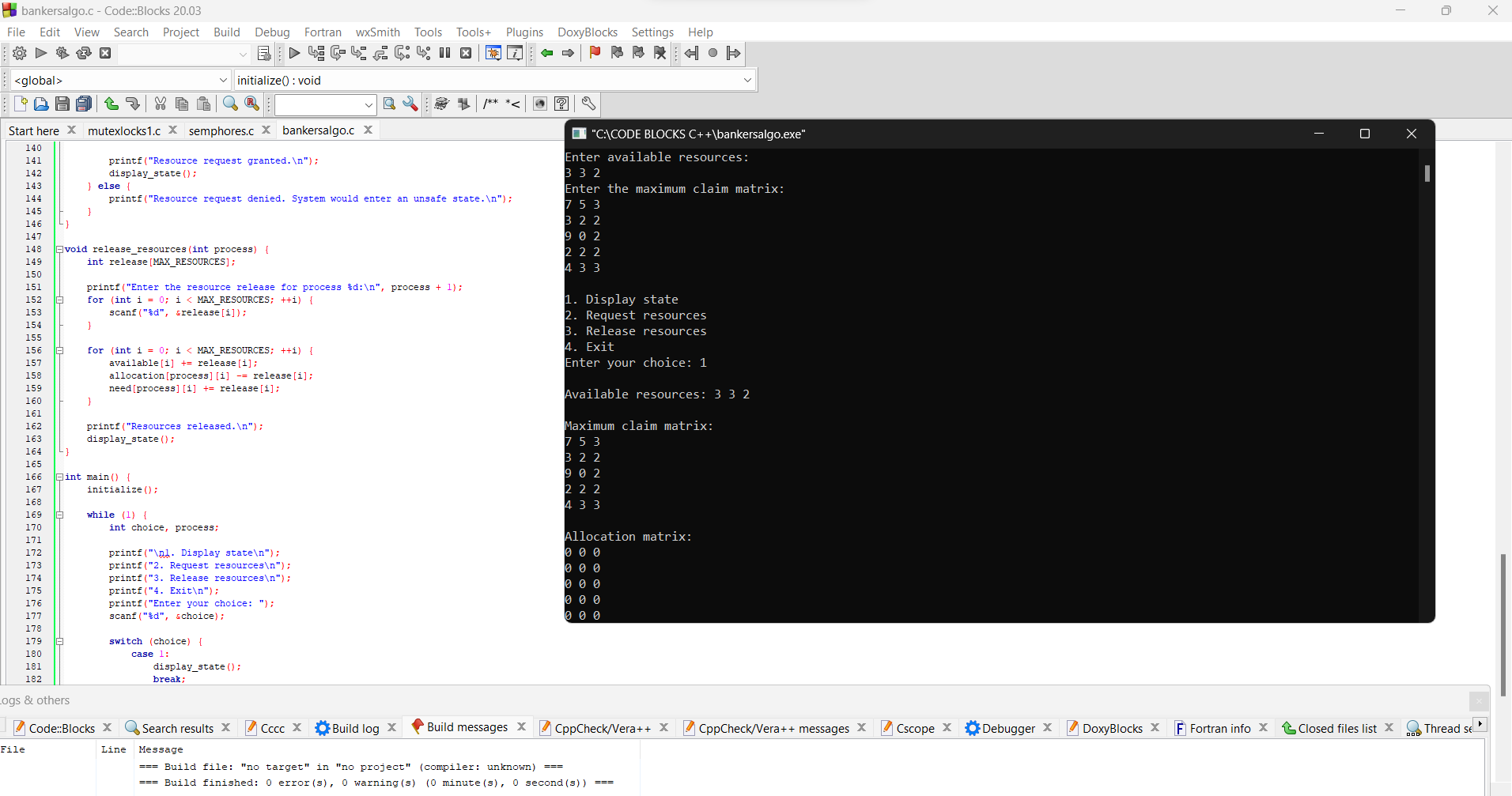
}

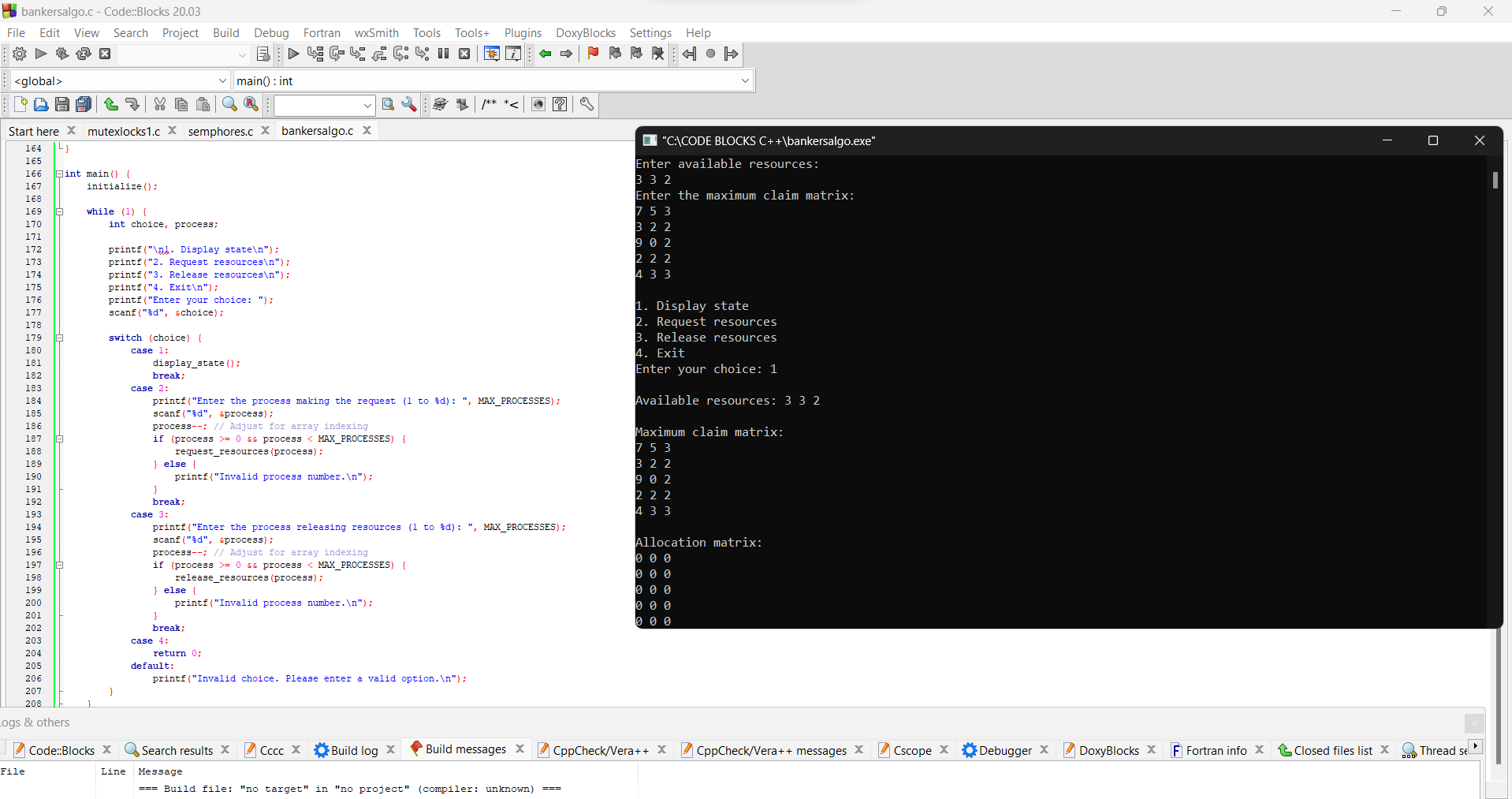
return 0;

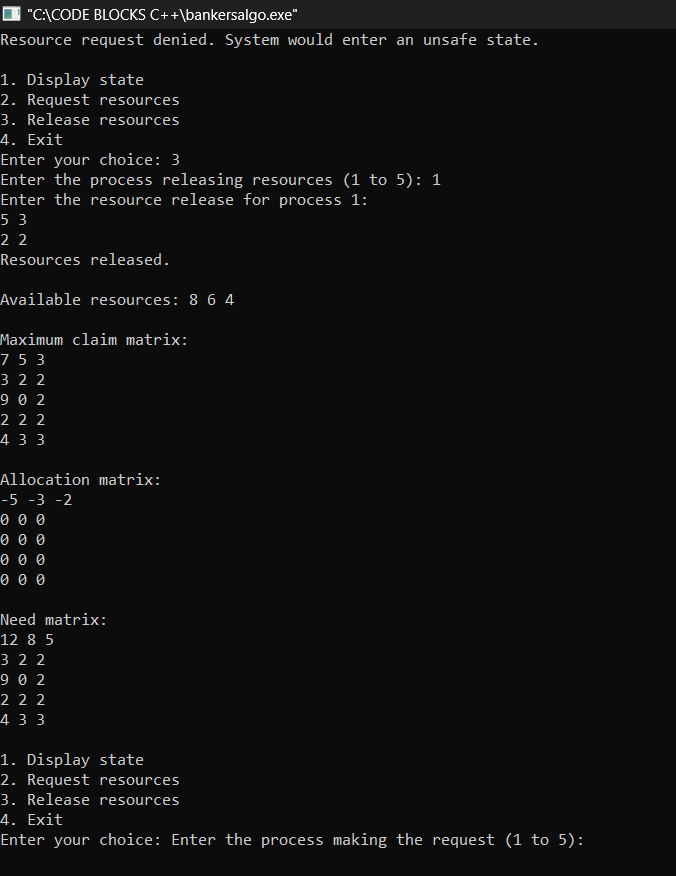
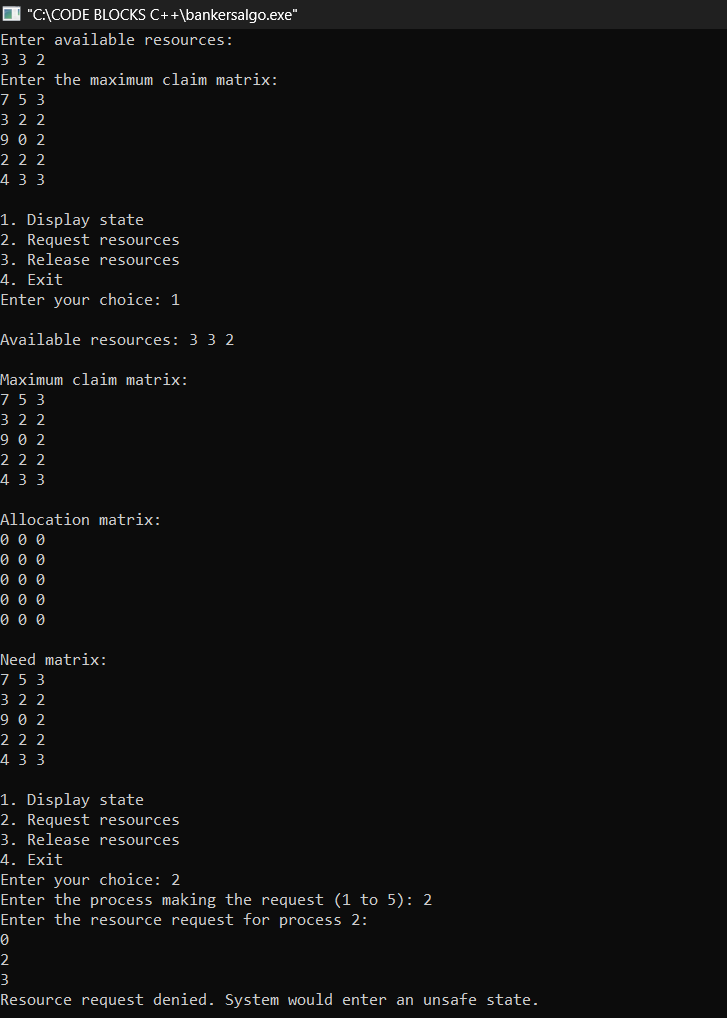
}

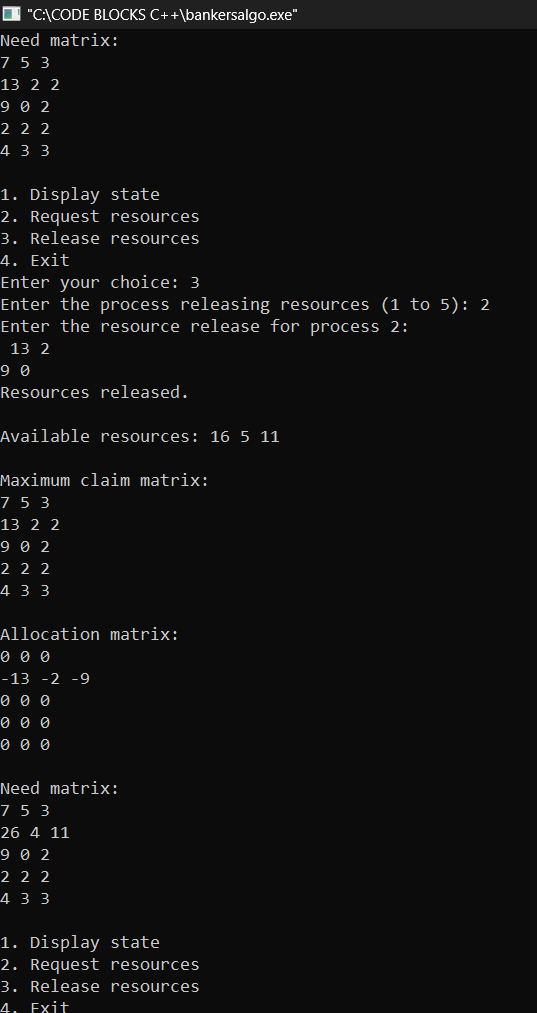
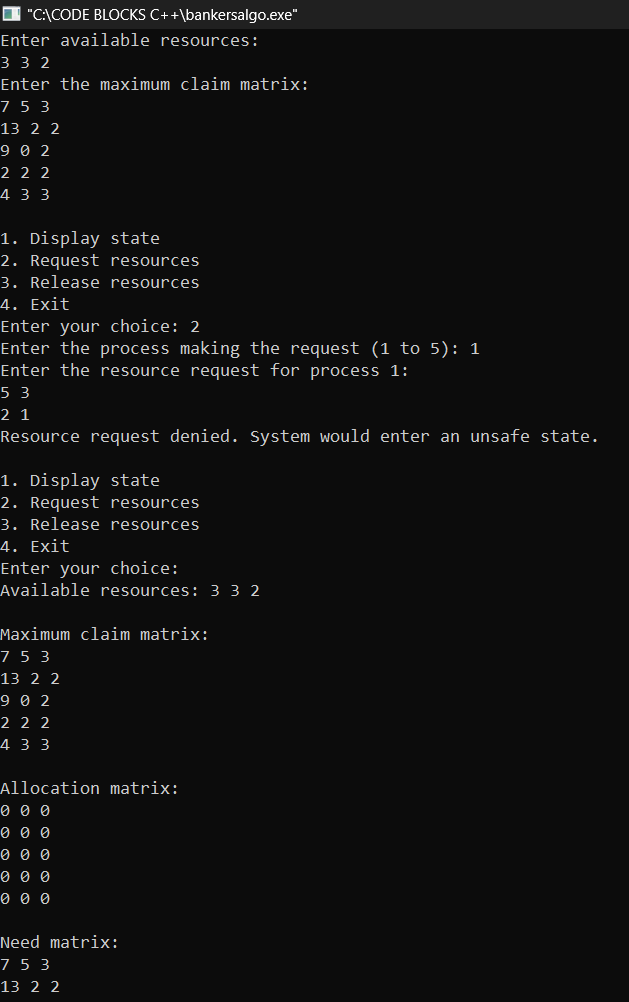
**OUTPUTS:**











**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.