Operating System

Name：Yin Yuang	STU_NO：202383930027	Exp：4-5

Experiment4:Banker's Algorithm

Experimental Purpose

1.Gain a thorough understanding of the basic principles and implementation mechanisms of the Banker's Algorithm, and master the core ideas of resource allocation and deadlock avoidance.
2.Learn how to implement the Banker's Algorithm through programming, including key steps such as system initialization, resource request processing, and safety check.
3.Verify the effectiveness of the Banker's Algorithm in avoiding deadlocks during dynamic resource allocation, and observe the generation process of the safe sequence.
4.Cultivate students' system design capabilities and programming practice skills, and enhance their understanding of the resource management mechanism of operating systems.

Experimental Content

1.Algorithm Implementation Core

This experiment implements the Banker's Algorithm in C, with three modular core functions:

(1) is_safe(): The core function for safety check. It initializes the work vector with available resources, traverses processes to find executable ones (whose need ≤ work), releases their allocated resources after execution, and generates a safe sequence.
(2) init_system(): Reads user input (resource types, total resources, process count, Max and Allocation matrices), calculates the Need matrix (Need = Max - Allocation) and Available vector (Available = Total - sum of Allocation), and validates data legality.
(3) request_resources(): Handles dynamic resource requests with two legality checks, tentative allocation, safety recheck, and rollback mechanism for unsafe states.

2.Experimental Configuration

To focus on verifying the core mechanism of the Banker's Algorithm, the experiment uses a simplified but representative system configuration, with clear process and resource definitions as follows:

Configuration Item	Specific Content
Number of Processes	2 processes, numbered P_0 and P_1 respectively
Number of Resource Types	3 types, marked as A, B, C
Total Resources (A, B, C)	(10, 5, 7) — total amount of each resource in the system
Max Matrix (P_0 / P_1)	(7, 5, 3) / (3, 2, 2) — maximum resource demand of each process

Initial Allocation Matrix (P_0 / P_1)	(0, 1, 0) / (2, 0, 0) — initially allocated resources of each process
Calculated Need Matrix (P_0 / P_1)	(7, 4, 3) / (1, 2, 2) — remaining resource demand of each process

3.Key Experimental Steps & Results

The experiment follows the core process of "initialization → safety check → dynamic request simulation → edge case verification" to complete the test, and the key steps and corresponding results are as follows:

3.1 Initial Safety Check

Execute is_safe(): Work vector starts as (8,4,7). P_0 executes first (Need ≤ Work), Work updates to (8,5,7); then P_1 executes, Work updates to (10,5,7).

3.2 Dynamic Resource Request Simulation

Take the resource request initiated by process P_1 as the test object, and verify the complete processing flow of the algorithm for legal requests:

Step	Key Operation	Execution Details & Result
1	Initiate Request	P_1 submits a resource request (1, 0, 2)
2	Legality Check	1. Request ≤ Need(P_1): (1,0,2) ≤ (1,2,2) ; 2. Request ≤ Available: (1,0,2) ≤ (8,4,7) . The request is legal.
3	Tentative Allocation	Update system state: Available = (8-1, 4-0, 7-2) = (7, 4, 5); Allocation(P_1) = (2+1, 0+0, 0+2) = (3, 0, 2); Need(P_1) = (1-1, 2-0, 2-2) = (0, 2, 0).
4	Safety Recheck	Call is_safe() again: Work = (7,4,5). P_0's Need (7,4,3) ≤ Work, execute P_0 to update Work to (7,5,5); then execute P_1, Work returns to total resources. System remains safe.
5	Final Decision	Confirm the allocation, and the updated system state takes effect.

4.Edge Case Verification

To fully verify the robustness of the algorithm, three typical edge cases are designed for testing, and the processing results are as follows:

Edge Case	Test Scenario	Experimental Result
Request > Need	P_1 requests (2,0,2)	Request denied immediately
Request > Available	P_0 requests (9,0,0)	Process blocked
Post-allocation Unsafe State	Legal request leading to no safe sequence	System rolls back and denies request

Experimental Result

1.Initial System State

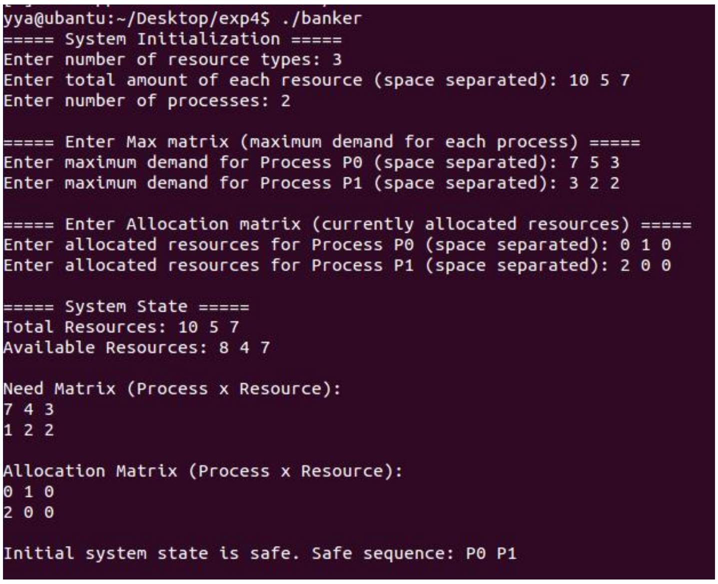


Figure 1. Screenshot of Initial System State

This screenshot records the initial state of the system after initialization. The physical memory is represented by resource matrices:

Resource Indicator	Specific Content
Total Resources	(10, 5, 7)
Available Resources	(8, 4, 7)
Need Matrix (P_0/P_1)	(7, 4, 3) / (1, 2, 2)
Allocation Matrix (P_0/P_1)	(0, 1, 0) / (2, 0, 0)

The initial safety check confirms the system is in a safe state with the safe sequence P0 P1, indicating no deadlock risk initially.

2.Dynamic Resource Request Simulation

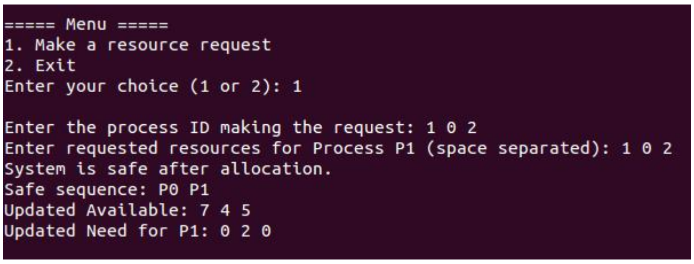


Figure 2. Screenshot of Legal Request Allocation

First, the algorithm completes two legality checks: the request does not exceed P_1's remaining need ($1 \leq 1$ for resource A), nor does it exceed the current available resources ($1 \leq 8$ for resource A). Then, the algorithm performs a tentative allocation (updating Available to (7, 4, 5)), and rechecks the safety state— the safe sequence (P_0 P_1) is still valid. Finally, the allocation is confirmed.

This process demonstrates the complete workflow of the Banker's Algorithm: "legality verification → tentative allocation → safety check → formal allocation", which ensures that resource allocation will not lead to system insecurity.

3.Illegal Request Handling

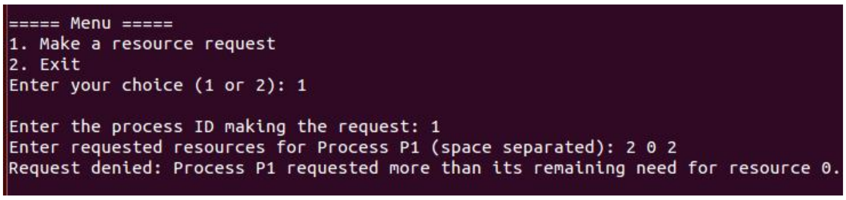


Figure 3. Screenshot of Request Exceeding Need

The screenshot shows Process P_1 initiating a request (2, 0, 2). The algorithm directly denies the request, with the prompt "Process P1 requested more than its remaining need for resource 0".

This result reflects the legality filter mechanism of the Banker's Algorithm: the request of a process cannot exceed its maximum remaining need (Need). This mechanism avoids logical conflicts in resource requests (e.g., a process requesting more resources than it actually requires), ensuring the rationality of the system's resource demand.

4 Resource Unavailability Handling

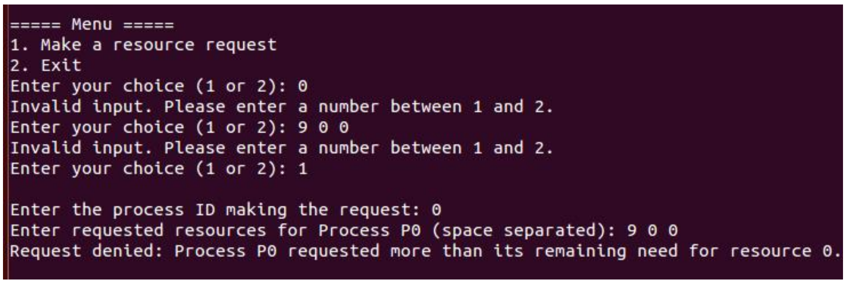


Figure 4. Screenshot of Request Exceeding Available Resources

The screenshot shows Process P0 initiating a request (9, 0, 0) after the previous allocation. The algorithm denies the request and blocks the process, with the prompt "Resources unavailable. Process P0 will be blocked".

This result reflects the resource supply-demand coordination mechanism of the Banker's Algorithm: when the available resources are insufficient to meet the request, the process is blocked (rather than forced to allocate) to avoid the exhaustion of system resources. This ensures that the system can maintain basic resource supply capacity for other processes.

Experiment5:Memory Management

Experimental Purpose

1.Understand the core principles of dynamic memory allocation and reclamation, and master the implementation logic of the First Fit algorithm.
2.Simulate the partition management mechanism of physical memory, including key operations such as free block merging, memory block splitting, resource allocation and reclamation.
3.Implement the memory status monitoring function, which can count the total free memory, the number of free blocks, the size of the largest free block and the external fragmentation rate.
4.Verify the effectiveness and stability of the memory allocation algorithm through random simulation experiments, and observe the generation and variation rules of memory fragments.

Experimental Content

1.Data Structures

Two core data structures are used in the simulator: Segment (logical memory unit) and MemoryBlock (physical memory unit). Their key attributes are summarized below.

Structure	Attributes	Description
Segment	segId	Unique logical segment identifier
	size	Requested memory size (KB)
MemoryBlock	segId	Segment bound to the block (−1 for free block)
	startAddr	Starting physical memory address
	size	Block size (KB)
	isAllocated	Allocation status (true = allocated, false = free)

2.Memory Initialization

At program startup, the simulator initializes the physical memory as a single free block of 1024 KB, representing an empty continuous memory space.
Additionally:
(1) A segment ID counter is prepared to generate unique IDs for each allocated segment.
(2) A random seed is initialized to support the randomized behavior used in the simulation stage.

This setup provides a clean and controlled environment for observing how memory allocation evolves over time.

3.Core Functionality

3.1 Memory Allocation(First-Fit Strategy)

When a memory request is issued, the First-Fit algorithm scans the memory blocks sequentially and selects the first free block large enough to satisfy the request:
(1) Exact match—If block size equals the request size, the block is allocated directly.
(2) Splitting—If the block is larger, it is split into one allocated block matching the requested size or a smaller remaining free block

(3) Failure—If no free block can satisfy the request, allocation fails.

This demonstrates how First-Fit prioritizes low search time but may lead to fragmentation.

3.2 Memory Deallocation and Block Merging

Memory is released by specifying its segment ID.

After a block is marked as free, the simulator invokes a **coalescing procedure** that merges all adjacent free blocks to reduce external fragmentation.This behavior mimics real operating systems, where coalescing is essential to maintain larger continuous free regions.

3.3 Memory Statistics and Visualization

The simulator provides information on total memory, free memory, number of free blocks, largest free block, and external fragmentation.

Metric	Description
Total memory	Fixed at 1024 KB
Free memory	Sum of all free block sizes
Allocated memory	Memory currently in use
Number of free blocks	Indicator of system fragmentation
Largest free block	Reflects availability of large contiguous space
External fragmentation	Measures scattering of free memory

External fragmentation is calculated as:

$$\text{FragRate}=\frac{\text{Total Free Memory}-\text{Largest Free Block}}{\text{Total Free Memory}}\times 100\%$$

The simulator also prints a memory block table, making the allocation status and fragmentation visually observable.

4.Random Simulation

A multi-round randomized simulation is performed, alternating between:

(1) random-sized memory allocations

(2) random deallocations

This process emulates a real operating environment, allowing us to observe:

(1) how memory fragments over time

(2) how often First-Fit succeeds or fails

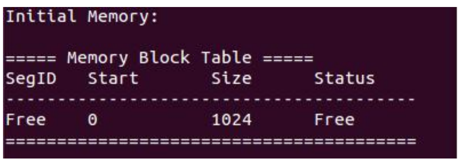
(3) how block merging mitigates fragmentation

The random simulation provides insight into the long-term performance and limitations of dynamic partition allocation strategies.

Experimental Results

This experiment simulates the dynamic partition allocation and reclamation process of a 1024KB physical memory based on the First-Fit algorithm. By conducting multiple rounds of random requests, the changes in memory status are observed, and the specific results are as follows:

1.Initial Memory Status



```
Initial Memory:

===== Memory Block Table =====
SegID   Start       Size      Status
----------------------------------------
Free    0           1024      Free
========================================
```

Figure1. Screenshot of Initial Memory Status

This screenshot records the memory block table at the start of the experiment: the physical memory is initialized as a continuous free block with SegID marked as "Free", starting address 0, size 1024KB, and status Free, presenting an initial state without fragmentation.

2.Multi-Round Random Allocation and Reclamation Process

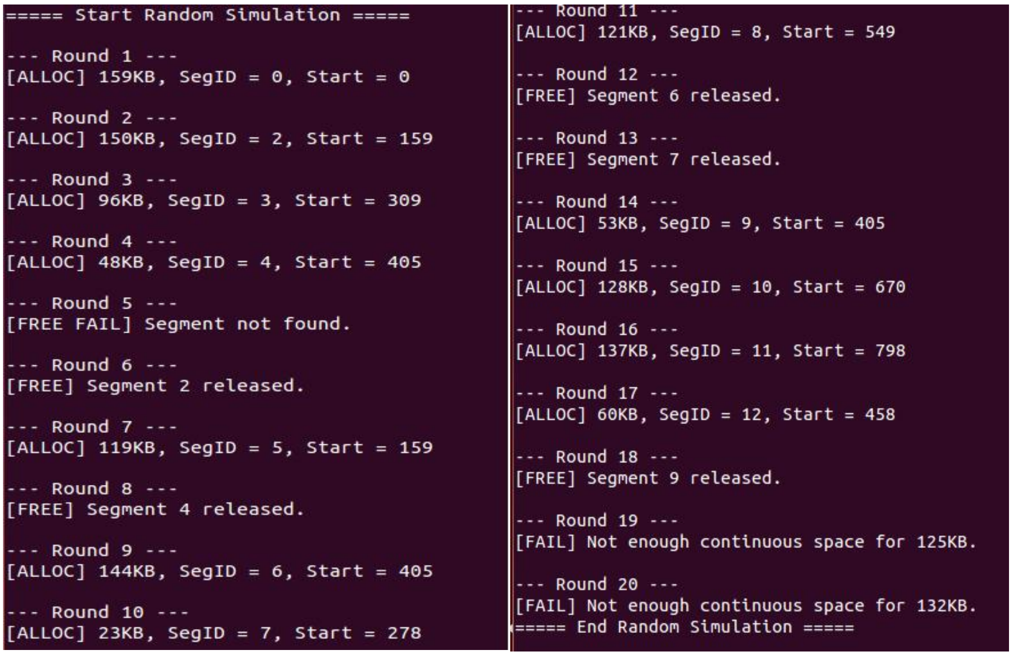


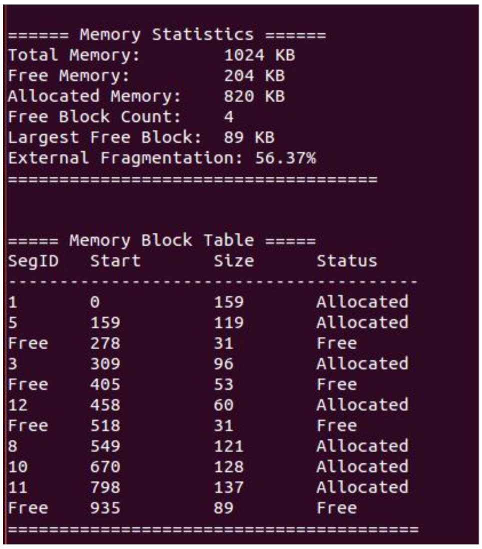
Figure2. Screenshot of Round 1-20 Simulation Process

This screenshot shows the execution process of all 20 rounds of random operations. The details of each round are presented in the table below:

Round	Operation Type	Specific Operation Content	Result Description
1	Allocation	Request 159KB	Success. First-Fit selects the first free block (the initial 1024KB block)
2	Allocation	Request 150KB	Success. First-Fit selects the remaining free block (the 865KB block)
3	Allocation	Request 96KB	Success. First-Fit selects the remaining free block (the 715KB block)
4	Allocation	Request 48KB	Success. First-Fit selects the remaining free block (the 619KB block)
5	Reclamation	Release a non-existent segment	Failure. Invalid segment ID
6	Reclamation	Release Segment 2 (150KB)	Success. A 150KB free block is formed
7	Allocation	Request 119KB	Success. The free block released by Segment 2 is split, with 31KB remaining
8	Reclamation	Release Segment 4 (48KB)	Success. A 48KB free block is formed

9	Allocation	Request 144KB	Success. First-Fit selects the corresponding free block
10	Allocation	Request 23KB	Success. First-Fit selects the corresponding free block
11	Allocation	Request 121KB	Success. First-Fit selects the corresponding free block
12	Reclamation	Release Segment 6	Success. A free block of corresponding size is formed
13	Reclamation	Release Segment 7	Success. A free block of corresponding size is formed
14	Allocation	Request 53KB	Success. First-Fit selects the corresponding free block
15	Allocation	Request 128KB	Success. First-Fit selects the corresponding free block
16	Allocation	Request 137KB	Success. First-Fit selects the corresponding free block
17	Allocation	Request 60KB	Success. First-Fit selects the corresponding free block
18	Reclamation	Release Segment 9	Success. A free block of corresponding size is formed
19	Allocation	Request 125KB	Failure. Insufficient continuous free space
20	Allocation	Request 132KB	Failure. Insufficient continuous free space

3.Memory Statistics and Final Block Table



```
====== Memory Statistics ======
Total Memory:        1024 KB
Free Memory:         204 KB
Allocated Memory:    820 KB
Free Block Count:    4
Largest Free Block:  89 KB
External Fragmentation: 56.37%
====================================


===== Memory Block Table =====
SegID   Start       Size      Status
----------------------------------------
1       0           159       Allocated
5       159         119       Allocated
Free    278         31        Free
3       309         96        Allocated
Free    405         53        Free
12      458         60        Allocated
Free    518         31        Free
8       549         121       Allocated
10      670         128       Allocated
11      798         137       Allocated
Free    935         89        Free
========================================
```

Figure3. Screenshot of Memory Statistics and Final Memory Block Table

This screenshot displays the memory statistics and the final memory block table upon the completion of the experiment. The statistical indicators show that the total memory is 1024KB, free memory is 204KB, allocated memory is 820KB, the number of free blocks is 4, the size of the largest free block is 89KB, and the external fragmentation rate reaches 56.37%. Meanwhile, the final memory block table exhibits obvious characteristics of fragmented distribution—allocated blocks and free blocks exist alternately, with the four free blocks of 31KB, 53KB, 31KB and 89KB scattered among the allocated blocks, failing to form a continuous large free space that can be used for large memory requests.

Appendix: Experimental Codes

Appendix A: Code for Experiment 4 (Banker's Algorithm)

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#define MAX_PROCESS 10
#define MAX_RESOURCE 10
#define BUFFER_SIZE 100
int process_num;
int resource_num;
int total_resources[MAX_RESOURCE];
int allocation[MAX_PROCESS][MAX_RESOURCE];
int max[MAX_PROCESS][MAX_RESOURCE];
int need[MAX_PROCESS][MAX_RESOURCE];
int available[MAX_RESOURCE];
// Function to check system safety
bool is_safe(int safe_seq[]) {
    int finish[MAX_PROCESS] = {0};
    int work[MAX_RESOURCE];

    // Initialize work with available resources
    for (int i = 0; i < resource_num; i++) {
        work[i] = available[i];
    }

    int count = 0;
    while (count < process_num) {
        bool found = false;
        for (int i = 0; i < process_num; i++) {
            if (!finish[i]) {
                bool can_allocate = true;
                // Check if all resources can be allocated
                for (int j = 0; j < resource_num; j++) {
                    if (need[i][j] > work[j]) {
                        can_allocate = false;
                        break;
                    }
                }

                if (can_allocate) {
                    // Allocate resources and mark process as finished
```

```
                    for (int j = 0; j < resource_num; j++) {
                        work[j] += allocation[i][j];
                    }
                    safe_seq[count++] = i;
                    finish[i] = 1;
                    found = true;
                }
            }
        }

        if (!found) {
            break; // No more processes can be allocated
        }
    }

    // Check if all processes are finished
    for (int i = 0; i < process_num; i++) {
        if (!finish[i]) {
            return false;
        }
    }
    return true;
}
// Function to read integer input safely
int read_int(const char *prompt, int min, int max) {
    char buffer[BUFFER_SIZE];
    int value;

    while (1) {
        printf("%s", prompt);
        if (!fgets(buffer, sizeof(buffer), stdin)) {
            continue;
        }
        if (sscanf(buffer, "%d", &value) == 1 && value >= min && value <= max) {
            return value;
        }
        printf("Invalid input. Please enter a number between %d and %d.\n", min, max);
    }
}
// Function to read array input safely
void read_int_array(const char *prompt, int arr[], int size, int min_val, int max_val) {
    char buffer[BUFFER_SIZE];

```

```c
    while (1) {
        printf("%s", prompt);
        if (!fgets(buffer, sizeof(buffer), stdin)) {
            continue;
        }

        int count = 0;
        char *token = strtok(buffer, " \t\n");
        bool valid = true;

        while (token && count < size) {
            int val;
            if (sscanf(token, "%d", &val) != 1 || val < min_val || (max_val != -1 && val > max_val)
) {
                valid = false;
                break;
            }
            arr[count++] = val;
            token = strtok(NULL, " \t\n");
        }

        if (valid && count == size) {
            return;
        }

        if (max_val == -1) {
            printf("Invalid input. Please enter %d non-negative integers.\n", size);
        } else {
            printf("Invalid input. Please enter %d integers between %d and %d.\n", size, min_val,
max_val);
        }
    }
}
// Initialize system
void init_system() {
    printf("===== System Initialization =====\n");

    // Read number of resource types
    resource_num = read_int("Enter number of resource types: ", 1, MAX_RESOURCE);

    // Read total amount of each resource
```

```c
    read_int_array("Enter total amount of each resource (space separated): ", total_resources, resource_num, 0, -1);

    // Read number of processes
    process_num = read_int("Enter number of processes: ", 1, MAX_PROCESS);

    // Read Max matrix
    printf("\n===== Enter Max matrix (maximum demand for each process) =====\n");
    for (int i = 0; i < process_num; i++) {
        char prompt[BUFFER_SIZE];
        snprintf(prompt, sizeof(prompt), "Enter maximum demand for Process P%d (space separated): ", i);
        read_int_array(prompt, max[i], resource_num, 0, -1);
    }

    // Read Allocation matrix
    printf("\n===== Enter Allocation matrix (currently allocated resources) =====\n");
    for (int i = 0; i < process_num; i++) {
        char prompt[BUFFER_SIZE];
        snprintf(prompt, sizeof(prompt), "Enter allocated resources for Process P%d (space separated): ", i);
        read_int_array(prompt, allocation[i], resource_num, 0, -1);
    }

    // Compute Need matrix and validate
    for (int i = 0; i < process_num; i++) {
        for (int j = 0; j < resource_num; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
            if (need[i][j] < 0) {
                printf("Error: Process P%d allocated more than its maximum demand for resource %d.\n", i, j);
                printf("Program will exit. Please restart and enter valid values.\n");
                exit(1);
            }
        }
    }

    // Compute Available vector and validate
    for (int j = 0; j < resource_num; j++) {
        int sum_alloc = 0;
```

```
        for (int i = 0; i < process_num; i++) {
            sum_alloc += allocation[i][j];
        }
        available[j] = total_resources[j] - sum_alloc;
        if (available[j] < 0) {
            printf("Error: Allocated resources exceed total resources for resource %d.\n", j);
            printf("Program will exit. Please restart and enter valid values.\n");
            exit(1);
        }
    }

    // Print system state
    printf("\n===== System State =====\n");
    printf("Total Resources: ");
    for (int j = 0; j < resource_num; j++) {
        printf("%d", total_resources[j]);
        if (j < resource_num - 1) {
            printf(" ");
        }
    }
    printf("\n");

    printf("Available Resources: ");
    for (int j = 0; j < resource_num; j++) {
        printf("%d", available[j]);
        if (j < resource_num - 1) {
            printf(" ");
        }
    }
    printf("\n");

    printf("\nNeed Matrix (Process x Resource):\n");
    for (int i = 0; i < process_num; i++) {
        for (int j = 0; j < resource_num; j++) {
            printf("%d", need[i][j]);
            if (j < resource_num - 1) {
                printf(" ");
            }
        }
        printf("\n");
    }

    printf("\nAllocation Matrix (Process x Resource):\n");
```

```
    for (int i = 0; i < process_num; i++) {
        for (int j = 0; j < resource_num; j++) {
            printf("%d", allocation[i][j]);
            if (j < resource_num - 1) {
                printf(" ");
            }
        }
        printf("\n");
    }
}
// Handle resource request
void request_resources() {
    int p_id;
    int request[MAX_RESOURCE];
    int safe_seq[MAX_PROCESS];

    // Read process ID
    p_id = read_int("\nEnter the process ID making the request: ", 0, process_num - 1);

    // Read requested resources
    char prompt[BUFFER_SIZE];
    snprintf(prompt, sizeof(prompt), "Enter requested resources for Process P%d (space sep
arated): ", p_id);
    read_int_array(prompt, request, resource_num, 0, -1);

    // Check if request exceeds Need
    for (int j = 0; j < resource_num; j++) {
        if (request[j] > need[p_id][j]) {
            printf("Request denied: Process P%d requested more than its remaining need for re
sour
ce %d.\n", p_id, j);
            return;
        }
    }

    // Check if request exceeds Available
    for (int j = 0; j < resource_num; j++) {
        if (request[j] > available[j]) {
            printf("Request denied: Resources unavailable. Process P%d will be blocked.\n", p
_id);
            return;
        }
    }
```

```c

    // Try allocation
    for (int j = 0; j < resource_num; j++) {
        available[j] -= request[j];
        allocation[p_id][j] += request[j];
        need[p_id][j] -= request[j];
    }

    // Check if system is safe after allocation
    if (is_safe(safe_seq)) {
        printf("System is safe after allocation.\n");
        printf("Safe sequence: ");
        for (int i = 0; i < process_num; i++) {
            printf("P%d", safe_seq[i]);
            if (i < process_num - 1) {
                printf(" ");
            }
        }
        printf("\n");

        printf("Updated Available: ");
        for (int j = 0; j < resource_num; j++) {
            printf("%d", available[j]);
            if (j < resource_num - 1) {
                printf(" ");
            }
        }
        printf("\n");

        printf("Updated Need for P%d: ", p_id);
        for (int j = 0; j < resource_num; j++) {
            printf("%d", need[p_id][j]);
            if (j < resource_num - 1) {
                printf(" ");
            }
        }
        printf("\n");
    } else {
        // Rollback allocation
        for (int j = 0; j < resource_num; j++) {
            available[j] += request[j];
            allocation[p_id][j] -= request[j];
            need[p_id][j] += request[j];
```

```c
        }
        printf("System would be unsafe! Request denied, state rolled back.\n");
    }
}
int main() {
    int choice;
    int safe_seq[MAX_PROCESS];

    // Initialize system
    init_system();

    // Check initial system safety
    if (is_safe(safe_seq)) {
        printf("\nInitial system state is safe. Safe sequence: ");
        for (int i = 0; i < process_num; i++) {
            printf("P%d", safe_seq[i]);
            if (i < process_num - 1) {
                printf(" ");
            }
        }
    } else {
        printf("\nInitial system state is not safe!");
    }
    printf("\n");

    // Menu loop
    while (1) {
        printf("\n===== Menu =====\n");
        printf("1. Make a resource request\n");
        printf("2. Exit\n");
        choice = read_int("Enter your choice (1 or 2): ", 1, 2);

        if (choice == 1) {
            request_resources();
        } else if (choice == 2) {
            printf("Program exiting.\n");
            break;
        }
    }

    return 0;
}
```

Appendix B: Code for Experiment 5 (Memory Management)

```cpp
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <iomanip>
#include <algorithm>
using namespace std;
/* ---------------- Segment and Memory Block Definitions ---------------- */
// Logical segment (just for demonstration)
struct Segment {
    int segId;
    int size;
};
// Physical memory block (free or allocated)
struct MemoryBlock {
    int segId;        // Segment ID (-1 means free)
    int startAddr;
    int size;         // Size in KB
    bool isAllocated;
};
/* ---------------- Global Variables ---------------- */
vector<MemoryBlock> memoryBlocks;    // physical memory
const int TOTAL_MEMORY = 1024;       // 1 MB
int nextSegId = 1;                   // auto-increment segment ID
int randomSeed = time(0);
/* ---------------- Initialization ---------------- */
void initMemory() {
    memoryBlocks.clear();
    MemoryBlock block;
    block.segId = -1;
    block.startAddr = 0;
    block.size = TOTAL_MEMORY;
    block.isAllocated = false;
    memoryBlocks.push_back(block);
    srand(randomSeed);
}
/* ---------------- Coalescing Free Blocks ---------------- */
void mergeFreeBlocks() {
    vector<MemoryBlock> newList;
    for (int i = 0; i < memoryBlocks.size(); i++) {
        MemoryBlock cur = memoryBlocks[i];
        if (!cur.isAllocated) {
```

```
            while (i + 1 < memoryBlocks.size() &&
                  !memoryBlocks[i+1].isAllocated) {
                cur.size += memoryBlocks[i+1].size;
                i++;
            }
        }
        newList.push_back(cur);
    }
    memoryBlocks = newList;
}
/* ---------------- First Fit Allocation ---------------- */
int allocate(int size) {
    if (size <= 0 || size > TOTAL_MEMORY) {
        cout << "Invalid allocation size.\n";
        return -1;
    }
    for (int i = 0; i < memoryBlocks.size(); i++) {
        MemoryBlock &blk = memoryBlocks[i];
        if (!blk.isAllocated && blk.size >= size) {
            if (blk.size == size) {
                blk.isAllocated = true;
                blk.segId = nextSegId++;
                cout << "[ALLOC] " << size << "KB, SegID = " << blk.segId
                     << ", Start = " << blk.startAddr << endl;
                return blk.segId;
            }
            // Split block
            MemoryBlock newFree;
            newFree.segId = -1;
            newFree.startAddr = blk.startAddr + size;
            newFree.size = blk.size - size;
            newFree.isAllocated = false;
            blk.size = size;
            blk.isAllocated = true;
            blk.segId = nextSegId++;
            memoryBlocks.insert(memoryBlocks.begin() + i + 1, newFree);
            cout << "[ALLOC] " << size << "KB, SegID = " << blk.segId
                 << ", Start = " << blk.startAddr << endl;
            return blk.segId;
        }
    }
    cout << "[FAIL] Not enough continuous space for " << size << "KB.\n";
    return -1;
```

```
}
/* --------------- Deallocate Segment --------------- */
bool deallocate(int segId) {
    for (auto &blk : memoryBlocks) {
        if (blk.segId == segId && blk.isAllocated) {
            blk.segId = -1;
            blk.isAllocated = false;
            cout << "[FREE] Segment " << segId << " released.\n";
            mergeFreeBlocks();
            return true;
        }
    }
    cout << "[FREE FAIL] Segment not found.\n";
    return false;
}
/* --------------- Memory Statistics --------------- */
void memoryStats() {
    int freeTotal = 0, freeBlocks = 0, maxFree = 0;
    for (auto &blk : memoryBlocks) {
        if (!blk.isAllocated) {
            freeBlocks++;
            freeTotal += blk.size;
            maxFree = max(maxFree, blk.size);
        }
    }
    double fragRate =
        (freeTotal > 0 ? (freeTotal - maxFree) * 100.0 / freeTotal : 0.0);
    cout << "\n====== Memory Statistics ======\n";
    cout << "Total Memory:        1024 KB\n";
    cout << "Free Memory:         " << freeTotal << " KB\n";
    cout << "Allocated Memory:    " << TOTAL_MEMORY - freeTotal << " KB\n";
    cout << "Free Block Count:    " << freeBlocks << "\n";
    cout << "Largest Free Block:  " << maxFree << " KB\n";
    cout << fixed << setprecision(2);
    cout << "External Fragmentation: " << fragRate << "%\n";
    cout << "=====================================\n\n";
}
/* --------------- Print Memory Blocks --------------- */
void printMemory() {
    cout << "\n===== Memory Block Table =====\n";
    cout << left << setw(8) << "SegID"
         << setw(12) << "Start"
         << setw(10) << "Size"
```

```cpp
         << setw(12) << "Status" << endl;
    cout << "----------------------------------------\n";
    for (auto &blk : memoryBlocks) {
        cout << left
             << setw(8)  << (blk.segId == -1 ? string("Free") : to_string(blk.segId))
             << setw(12) << blk.startAddr
             << setw(10) << blk.size
             << setw(12) << (blk.isAllocated ? "Allocated" : "Free")
             << endl;
    }
    cout << "==================================================\n\n";
}
/* ---------------- Random Simulation ---------------- */
void simulateRandom(int rounds) {
    vector<int> allocated;
    cout << "\n===== Start Random Simulation =====\n";
    for (int i = 1; i <= rounds; i++) {
        cout << "\n--- Round " << i << " ---\n";
        bool doAlloc = (allocated.empty() ? true : rand() % 2 == 0);
        if (doAlloc) {
            int size = 10 + rand() % 150;
            int segId = allocate(size);
            if (segId != -1) allocated.push_back(segId);
        }
        else {
            int idx = rand() % allocated.size();
            int segId = allocated[idx];
            if (deallocate(segId)) {
                allocated.erase(allocated.begin() + idx);
            }
        }
    }
    cout << "===== End Random Simulation =====\n\n";
}
/* ---------------- Main ---------------- */
int main() {
    initMemory();
    cout << "Initial Memory:\n";
    printMemory();
    simulateRandom(20);
    memoryStats();
    printMemory();
    return 0;
```

```cpp
}
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <iomanip>
#include <algorithm>
using namespace std;
/* ---------------- Segment and Memory Block Definitions ---------------- */
// Logical segment (just for demonstration)
struct Segment {
    int segId;
    int size;
};
// Physical memory block (free or allocated)
struct MemoryBlock {
    int segId;          // Segment ID (-1 means free)
    int startAddr;
    int size;           // Size in KB
    bool isAllocated;
};
/* ---------------- Global Variables ---------------- */
vector<MemoryBlock> memoryBlocks;     // physical memory
const int TOTAL_MEMORY = 1024;        // 1 MB
int nextSegId = 1;                    // auto-increment segment ID
int randomSeed = time(0);
/* ---------------- Initialization ---------------- */
void initMemory() {
    memoryBlocks.clear();
    MemoryBlock block;
    block.segId = -1;
    block.startAddr = 0;
    block.size = TOTAL_MEMORY;
    block.isAllocated = false;
    memoryBlocks.push_back(block);
    srand(randomSeed);
}
/* ---------------- Coalescing Free Blocks ---------------- */
void mergeFreeBlocks() {
    vector<MemoryBlock> newList;
    for (int i = 0; i < memoryBlocks.size(); i++) {
        MemoryBlock cur = memoryBlocks[i];
        if (!cur.isAllocated) {
```

```
            while (i + 1 < memoryBlocks.size() &&
                   !memoryBlocks[i+1].isAllocated) {
                cur.size += memoryBlocks[i+1].size;
                i++;
            }
        }
        newList.push_back(cur);
    }
    memoryBlocks = newList;
}
/* ---------------- First Fit Allocation ---------------- */
int allocate(int size) {
    if (size <= 0 || size > TOTAL_MEMORY) {
        cout << "Invalid allocation size.\n";
        return -1;
    }
    for (int i = 0; i < memoryBlocks.size(); i++) {
        MemoryBlock &blk = memoryBlocks[i];
        if (!blk.isAllocated && blk.size >= size) {
            if (blk.size == size) {
                blk.isAllocated = true;
                blk.segId = nextSegId++;
                cout << "[ALLOC] " << size << "KB, SegID = " << blk.segId
                     << ", Start = " << blk.startAddr << endl;
                return blk.segId;
            }
            // Split block
            MemoryBlock newFree;
            newFree.segId = -1;
            newFree.startAddr = blk.startAddr + size;
            newFree.size = blk.size - size;
            newFree.isAllocated = false;
            blk.size = size;
            blk.isAllocated = true;
            blk.segId = nextSegId++;
            memoryBlocks.insert(memoryBlocks.begin() + i + 1, newFree);
            cout << "[ALLOC] " << size << "KB, SegID = " << blk.segId
                 << ", Start = " << blk.startAddr << endl;
            return blk.segId;
        }
    }
    cout << "[FAIL] Not enough continuous space for " << size << "KB.\n";
    return -1;
```

```cpp
}
/* --------------- Deallocate Segment --------------- */
bool deallocate(int segId) {
    for (auto &blk : memoryBlocks) {
        if (blk.segId == segId && blk.isAllocated) {
            blk.segId = -1;
            blk.isAllocated = false;
            cout << "[FREE] Segment " << segId << " released.\n";
            mergeFreeBlocks();
            return true;
        }
    }
    cout << "[FREE FAIL] Segment not found.\n";
    return false;
}
/* --------------- Memory Statistics --------------- */
void memoryStats() {
    int freeTotal = 0, freeBlocks = 0, maxFree = 0;
    for (auto &blk : memoryBlocks) {
        if (!blk.isAllocated) {
            freeBlocks++;
            freeTotal += blk.size;
            maxFree = max(maxFree, blk.size);
        }
    }
    double fragRate =
        (freeTotal > 0 ? (freeTotal - maxFree) * 100.0 / freeTotal : 0.0);
    cout << "\n====== Memory Statistics ======\n";
    cout << "Total Memory:        1024 KB\n";
    cout << "Free Memory:         " << freeTotal << " KB\n";
    cout << "Allocated Memory:    " << TOTAL_MEMORY - freeTotal << " KB\n";
    cout << "Free Block Count:    " << freeBlocks << "\n";
    cout << "Largest Free Block:  " << maxFree << " KB\n";
    cout << fixed << setprecision(2);
    cout << "External Fragmentation: " << fragRate << "%\n";
    cout << "======================================\n\n";
}
/* --------------- Print Memory Blocks --------------- */
void printMemory() {
    cout << "\n===== Memory Block Table =====\n";
    cout << left << setw(8) << "SegID"
         << setw(12) << "Start"
         << setw(10) << "Size"
```

```cpp
         << setw(12) << "Status" << endl;
    cout << "----------------------------------------\n";
    for (auto &blk : memoryBlocks) {
        cout << left
             << setw(8)  << (blk.segId == -1 ? string("Free") : to_string(blk.segId))
             << setw(12) << blk.startAddr
             << setw(10) << blk.size
             << setw(12) << (blk.isAllocated ? "Allocated" : "Free")
             << endl;
    }
    cout << "============================================================\n\n";
}
/* --------------- Random Simulation --------------- */
void simulateRandom(int rounds) {
    vector<int> allocated;
    cout << "\n===== Start Random Simulation =====\n";
    for (int i = 1; i <= rounds; i++) {
        cout << "\n--- Round " << i << " ---\n";
        bool doAlloc = (allocated.empty() ? true : rand() % 2 == 0);
        if (doAlloc) {
            int size = 10 + rand() % 150;
            int segId = allocate(size);
            if (segId != -1) allocated.push_back(segId);
        }
        else {
            int idx = rand() % allocated.size();
            int segId = allocated[idx];
            if (deallocate(segId)) {
                allocated.erase(allocated.begin() + idx);
            }
        }
    }
    cout << "===== End Random Simulation =====\n\n";
}
/* --------------- Main --------------- */
int main() {
    initMemory();
    cout << "Initial Memory:\n";
    printMemory();
    simulateRandom(20);
    memoryStats();
    printMemory();
    return 0;
```

```
}
```