

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 \leq 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 ($\text{Need} \leq \text{Work}$), 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. $\text{Request} \leq \text{Need}(P_1)$: $(1,0,2) \leq (1,2,2)$; 2. $\text{Request} \leq \text{Available}$: $(1,0,2) \leq (8,4,7)$. 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 <code>is_safe()</code> again: Work = $(7,4,5)$. P_0 's Need $(7,4,3) \leq \text{Work}$, 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

```

ya@ubuntu:~/Desktop/exp4$ ./banker
===== System Initialization =====
Enter number of resource types: 3
Enter total amount of each resource (space separated): 10 5 7
Enter number of processes: 2

===== Enter Max matrix (maximum demand for each process) =====
Enter maximum demand for Process P0 (space separated): 7 5 3
Enter maximum demand for Process P1 (space separated): 3 2 2

===== Enter Allocation matrix (currently allocated resources) =====
Enter allocated resources for Process P0 (space separated): 0 1 0
Enter allocated resources for Process P1 (space separated): 2 0 0

===== System State =====
Total Resources: 10 5 7
Available Resources: 8 4 7

Need Matrix (Process x Resource):
7 4 3
1 2 2

Allocation Matrix (Process x Resource):
0 1 0
2 0 0

Initial system state is safe. Safe sequence: P0 P1

```

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

```

===== Menu =====
1. Make a resource request
2. Exit
Enter your choice (1 or 2): 1

Enter the process ID making the request: 1 0 2
Enter requested resources for Process P1 (space separated): 1 0 2
System is safe after allocation.
Safe sequence: P0 P1
Updated Available: 7 4 5
Updated Need for P1: 0 2 0

```

Figure 2. Screenshot of Legal Request Allocation

First, the algorithm completes two legality checks: the request does not exceed P₁'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 (P₀ P₁) 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

```

===== Menu =====
1. Make a resource request
2. Exit
Enter your choice (1 or 2): 1

Enter the process ID making the request: 1
Enter requested resources for Process P1 (space separated): 2 0 2
Request denied: Process P1 requested more than its remaining need for resource 0.

```

Figure 3. Screenshot of Request Exceeding Need

The screenshot shows Process P1 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

```

===== Menu =====
1. Make a resource request
2. Exit
Enter your choice (1 or 2): 0
Invalid input. Please enter a number between 1 and 2.
Enter your choice (1 or 2): 9 0 0
Invalid input. Please enter a number between 1 and 2.
Enter your choice (1 or 2): 1

Enter the process ID making the request: 0
Enter requested resources for Process P0 (space separated): 9 0 0
Request denied: Process P0 requested more than its remaining need for resource 0.

```

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

```
===== Start Random Simulation =====
--- Round 1 ---
[ALLOC] 159KB, SegID = 0, Start = 0
--- Round 2 ---
[ALLOC] 150KB, SegID = 2, Start = 159
--- Round 3 ---
[ALLOC] 96KB, SegID = 3, Start = 309
--- Round 4 ---
[ALLOC] 48KB, SegID = 4, Start = 405
--- Round 5 ---
[FREE FAIL] Segment not found.
--- Round 6 ---
[FREE] Segment 2 released.
--- Round 7 ---
[ALLOC] 119KB, SegID = 5, Start = 159
--- Round 8 ---
[FREE] Segment 4 released.
--- Round 9 ---
[ALLOC] 144KB, SegID = 6, Start = 405
--- Round 10 ---
[ALLOC] 23KB, SegID = 7, Start = 278
--- Round 11 ---
[ALLOC] 121KB, SegID = 8, Start = 549
--- Round 12 ---
[FREE] Segment 6 released.
--- Round 13 ---
[FREE] Segment 7 released.
--- Round 14 ---
[ALLOC] 53KB, SegID = 9, Start = 405
--- Round 15 ---
[ALLOC] 128KB, SegID = 10, Start = 670
--- Round 16 ---
[ALLOC] 137KB, SegID = 11, Start = 798
--- Round 17 ---
[ALLOC] 60KB, SegID = 12, Start = 458
--- Round 18 ---
[FREE] Segment 9 released.
--- Round 19 ---
[FAIL] Not enough continuous space for 125KB.
--- Round 20 ---
[FAIL] Not enough continuous space for 132KB.
===== End Random Simulation =====
```

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)

| | |
|----|--|
| 1 | <code>#include <stdio.h></code> |
| 2 | <code>#include <stdlib.h></code> |
| 3 | <code>#include <stdbool.h></code> |
| 4 | <code>#include <string.h></code> |
| 5 | <code>#define MAX_PROCESS 10</code> |
| 6 | <code>#define MAX_RESOURCE 10</code> |
| 7 | <code>#define BUFFER_SIZE 100</code> |
| 8 | <code>int process_num;</code> |
| 9 | <code>int resource_num;</code> |
| 10 | <code>int total_resources[MAX_RESOURCE];</code> |
| 11 | <code>int allocation[MAX_PROCESS][MAX_RESOURCE];</code> |
| 12 | <code>int max[MAX_PROCESS][MAX_RESOURCE];</code> |
| 13 | <code>int need[MAX_PROCESS][MAX_RESOURCE];</code> |
| 14 | <code>int available[MAX_RESOURCE];</code> |
| 15 | <i>// Function to check system safety</i> |
| 16 | <code>bool is_safe(int safe_seq[])</code> { |
| 17 | <code> int finish[MAX_PROCESS] = {0};</code> |
| 18 | <code> int work[MAX_RESOURCE];</code> |
| 19 | |
| 20 | <i>// Initialize work with available resources</i> |
| 21 | <code> for (int i = 0; i < resource_num; i++) {</code> |
| 22 | <code> work[i] = available[i];</code> |
| 23 | <code> }</code> |
| 24 | |
| 25 | <code> int count = 0;</code> |
| 26 | <code> while (count < process_num) {</code> |
| 27 | <code> bool found = false;</code> |
| 28 | <code> for (int i = 0; i < process_num; i++) {</code> |
| 29 | <code> if (!finish[i]) {</code> |
| 30 | <code> bool can_allocate = true;</code> |
| 31 | <i>// Check if all resources can be allocated</i> |
| 32 | <code> for (int j = 0; j < resource_num; j++) {</code> |
| 33 | <code> if (need[i][j] > work[j]) {</code> |
| 34 | <code> can_allocate = false;</code> |
| 35 | <code> break;</code> |
| 36 | <code> }</code> |
| 37 | <code> }</code> |
| 38 | |
| 39 | <code> if (can_allocate) {</code> |
| 40 | <i>// Allocate resources and mark process as finished</i> |

```

41     for (int j = 0; j < resource_num; j++) {
42         work[j] += allocation[i][j];
43     }
44     safe_seq[count++] = i;
45     finish[i] = 1;
46     found = true;
47 }
48 }
49 }
50
51 if (!found) {
52     break; // No more processes can be allocated
53 }
54 }
55
56 // Check if all processes are finished
57 for (int i = 0; i < process_num; i++) {
58     if (!finish[i]) {
59         return false;
60     }
61 }
62 return true;
63 }
64 // Function to read integer input safely
65 int read_int(const char *prompt, int min, int max) {
66     char buffer[BUFFER_SIZE];
67     int value;
68
69     while (1) {
70         printf("%s", prompt);
71         if (!fgets(buffer, sizeof(buffer), stdin)) {
72             continue;
73         }
74         if (sscanf(buffer, "%d", &value) == 1 && value >= min && value <= max) {
75             return value;
76         }
77         printf("Invalid input. Please enter a number between %d and %d.\n", min, max);
78     }
79 }
80 // Function to read array input safely
81 void read_int_array(const char *prompt, int arr[], int size, int min_val, int max_val) {
82     char buffer[BUFFER_SIZE];
83 }
```

```

84     while (1) {
85         printf("%s", prompt);
86         if (!fgets(buffer, sizeof(buffer), stdin)) {
87             continue;
88         }
89
90         int count = 0;
91         char *token = strtok(buffer, " \t\n");
92         bool valid = true;
93
94         while (token && count < size) {
95             int val;
96             if (sscanf(token, "%d", &val) != 1 || val < min_val || (max_val != -1 && val > max_val))
97             ) {
98                 valid = false;
99                 break;
100            }
101            arr[count++] = val;
102            token = strtok(NULL, " \t\n");
103        }
104
105        if (valid && count == size) {
106            return;
107        }
108
109        if (max_val == -1) {
110            printf("Invalid input. Please enter %d non-negative integers.\n", size);
111        } else {
112            printf("Invalid input. Please enter %d integers between %d and %d.\n", size, min_val,
113 max_val);
114        }
115    }
116 }
117 // Initialize system
118 void init_system() {
119     printf("===== System Initialization =====\n");
120
121     // Read number of resource types
122     resource_num = read_int("Enter number of resource types: ", 1, MAX_RESOURCE);
123
124     // Read total amount of each resource

```

```

125     read_int_array("Enter total amount of each resource (space separated): ", total_resource
      s, resource_num, 0, -1);
126
127     // Read number of processes
128     process_num = read_int("Enter number of processes: ", 1, MAX_PROCESS);
129
130     // Read Max matrix
131     printf("\n===== Enter Max matrix (maximum demand for each process) =====\n");
132     for (int i = 0; i < process_num; i++) {
133         char prompt[BUFFER_SIZE];
134         snprintf(prompt, sizeof(prompt), "Enter maximum demand for Process P%d (space se
      para
135         ted): ", i);
136         read_int_array(prompt, max[i], resource_num, 0, -1);
137     }
138
139     // Read Allocation matrix
140     printf("\n===== Enter Allocation matrix (currently allocated resources) =====\n");
141     for (int i = 0; i < process_num; i++) {
142         char prompt[BUFFER_SIZE];
143         snprintf(prompt, sizeof(prompt), "Enter allocated resources for Process P%d (space s
      epara
144         ted): ", i);
145         read_int_array(prompt, allocation[i], resource_num, 0, -1);
146     }
147
148     // Compute Need matrix and validate
149     for (int i = 0; i < process_num; i++) {
150         for (int j = 0; j < resource_num; j++) {
151             need[i][j] = max[i][j] - allocation[i][j];
152             if (need[i][j] < 0) {
153                 printf("Error: Process P%d allocated more than its maximum demand for resour
      ce %
154                 ce.\n", i, j);
155                 printf("Program will exit. Please restart and enter valid values.\n");
156                 exit(1);
157             }
158         }
159     }
160
161     // Compute Available vector and validate
162     for (int j = 0; j < resource_num; j++) {
163         int sum_alloc = 0;

```

```

164     for (int i = 0; i < process_num; i++) {
165         sum_alloc += allocation[i][j];
166     }
167     available[j] = total_resources[j] - sum_alloc;
168     if (available[j] < 0) {
169         printf("Error: Allocated resources exceed total resources for resource %d.\n", j);
170         printf("Program will exit. Please restart and enter valid values.\n");
171         exit(1);
172     }
173 }
174
175 // Print system state
176 printf("\n===== System State =====\n");
177 printf("Total Resources: ");
178 for (int j = 0; j < resource_num; j++) {
179     printf("%d", total_resources[j]);
180     if (j < resource_num - 1) {
181         printf(" ");
182     }
183 }
184 printf("\n");
185
186 printf("Available Resources: ");
187 for (int j = 0; j < resource_num; j++) {
188     printf("%d", available[j]);
189     if (j < resource_num - 1) {
190         printf(" ");
191     }
192 }
193 printf("\n");
194
195 printf("\nNeed Matrix (Process x Resource):\n");
196 for (int i = 0; i < process_num; i++) {
197     for (int j = 0; j < resource_num; j++) {
198         printf("%d", need[i][j]);
199         if (j < resource_num - 1) {
200             printf(" ");
201         }
202     }
203     printf("\n");
204 }
205
206 printf("\nAllocation Matrix (Process x Resource):\n");

```

```

207  for (int i = 0; i < process_num; i++) {
208      for (int j = 0; j < resource_num; j++) {
209          printf("%d", allocation[i][j]);
210          if (j < resource_num - 1) {
211              printf(" ");
212          }
213      }
214      printf("\n");
215  }
216 }
217 // Handle resource request
218 void request_resources() {
219     int p_id;
220     int request[MAX_RESOURCE];
221     int safe_seq[MAX_PROCESS];
222
223     // Read process ID
224     p_id = read_int("\nEnter the process ID making the request: ", 0, process_num - 1);
225
226     // Read requested resources
227     char prompt[BUFFER_SIZE];
228     snprintf(prompt, sizeof(prompt), "Enter requested resources for Process P%d (space separated): ", p_id);
229     read_int_array(prompt, request, resource_num, 0, -1);
230
231     // Check if request exceeds Need
232     for (int j = 0; j < resource_num; j++) {
233         if (request[j] > need[p_id][j]) {
234             printf("Request denied: Process P%d requested more than its remaining need for resource %d.\n", p_id, j);
235             return;
236         }
237     }
238 }
239
240 // Check if request exceeds Available
241 for (int j = 0; j < resource_num; j++) {
242     if (request[j] > available[j]) {
243         printf("Request denied: Resources unavailable. Process P%d will be blocked.\n", p_id);
244         return;
245     }
246 }

```

```

247
248 // Try allocation
249 for (int j = 0; j < resource_num; j++) {
250     available[j] -= request[j];
251     allocation[p_id][j] += request[j];
252     need[p_id][j] -= request[j];
253 }
254
255 // Check if system is safe after allocation
256 if (is_safe(safe_seq)) {
257     printf("System is safe after allocation.\n");
258     printf("Safe sequence: ");
259     for (int i = 0; i < process_num; i++) {
260         printf("P%d", safe_seq[i]);
261         if (i < process_num - 1) {
262             printf(" ");
263         }
264     }
265     printf("\n");
266
267     printf("Updated Available: ");
268     for (int j = 0; j < resource_num; j++) {
269         printf("%d", available[j]);
270         if (j < resource_num - 1) {
271             printf(" ");
272         }
273     }
274     printf("\n");
275
276     printf("Updated Need for P%d: ", p_id);
277     for (int j = 0; j < resource_num; j++) {
278         printf("%d", need[p_id][j]);
279         if (j < resource_num - 1) {
280             printf(" ");
281         }
282     }
283     printf("\n");
284 } else {
285 // Rollback allocation
286     for (int j = 0; j < resource_num; j++) {
287         available[j] += request[j];
288         allocation[p_id][j] -= request[j];
289         need[p_id][j] += request[j];

```

| | |
|-----|---|
| 290 | } |
| 291 | printf("System would be unsafe! Request denied, state rolled back.\n"); |
| 292 | } |
| 293 | } |
| 294 | int main() { |
| 295 | int choice; |
| 296 | int safe_seq[MAX_PROCESS]; |
| 297 | |
| 298 | <i>// Initialize system</i> |
| 299 | init_system(); |
| 300 | |
| 301 | <i>// Check initial system safety</i> |
| 302 | if(is_safe(safe_seq)) { |
| 303 | printf("\nInitial system state is safe. Safe sequence: "); |
| 304 | for (int i = 0; i < process_num; i++) { |
| 305 | printf("P%d", safe_seq[i]); |
| 306 | if (i < process_num - 1) { |
| 307 | printf(" "); |
| 308 | } |
| 309 | } |
| 310 | } else { |
| 311 | printf("\nInitial system state is not safe!"); |
| 312 | } |
| 313 | printf("\n"); |
| 314 | |
| 315 | <i>// Menu loop</i> |
| 316 | while (1) { |
| 317 | printf("\n===== Menu =====\n"); |
| 318 | printf("1. Make a resource request\n"); |
| 319 | printf("2. Exit\n"); |
| 320 | choice = read_int("Enter your choice (1 or 2): ", 1, 2); |
| 321 | |
| 322 | if (choice == 1) { |
| 323 | request_resources(); |
| 324 | } else if (choice == 2) { |
| 325 | printf("Program exiting.\n"); |
| 326 | break; |
| 327 | } |
| 328 | } |
| 329 | |
| 330 | return 0; |
| 331 | } |

Appendix B: Code for Experiment 5 (Memory Management)

```

1 #include <iostream>
2 #include <vector>
3 #include <cstdlib>
4 #include <ctime>
5 #include <iomanip>
6 #include <algorithm>
7 using namespace std;
8 /* ----- Segment and Memory Block Definitions ----- */
9 // Logical segment (just for demonstration)
10 struct Segment {
11     int segId;
12     int size;
13 };
14 // Physical memory block (free or allocated)
15 struct MemoryBlock {
16     int segId;      // Segment ID (-1 means free)
17     int startAddr;
18     int size;        // Size in KB
19     bool isAllocated;
20 };
21 /* ----- Global Variables ----- */
22 vector<MemoryBlock> memoryBlocks; // physical memory
23 const int TOTAL_MEMORY = 1024; // 1 MB
24 int nextSegId = 1;           // auto-increment segment ID
25 int randomSeed = time(0);
26 /* ----- Initialization ----- */
27 void initMemory() {
28     memoryBlocks.clear();
29     MemoryBlock block;
30     block(segId = -1;
31     block.startAddr = 0;
32     block.size = TOTAL_MEMORY;
33     block.isAllocated = false;
34     memoryBlocks.push_back(block);
35     srand(randomSeed);
36 }
37 /* ----- Coalescing Free Blocks ----- */
38 void mergeFreeBlocks() {
39     vector<MemoryBlock> newList;
40     for (int i = 0; i < memoryBlocks.size(); i++) {
41         MemoryBlock cur = memoryBlocks[i];
42         if (!cur.isAllocated) {

```

```

43     while (i + 1 < memoryBlocks.size() &&
44         !memoryBlocks[i+1].isAllocated) {
45         cur.size += memoryBlocks[i+1].size;
46         i++;
47     }
48 }
49 newList.push_back(cur);
50 }
51 memoryBlocks = newList;
52 }
53 /* ----- First Fit Allocation ----- */
54 int allocate(int size) {
55     if (size <= 0 || size > TOTAL_MEMORY) {
56         cout << "Invalid allocation size.\n";
57         return -1;
58     }
59     for (int i = 0; i < memoryBlocks.size(); i++) {
60         MemoryBlock &blk = memoryBlocks[i];
61         if (!blk.isAllocated && blk.size >= size) {
62             if (blk.size == size) {
63                 blk.isAllocated = true;
64                 blk(segId = nextSegId++);
65                 cout << "[ALLOC] " << size << "KB, SegID = " << blk.segId
66                     << ", Start = " << blk.startAddr << endl;
67                 return blk.segId;
68             }
69             // Split block
70             MemoryBlock newFree;
71             newFree.segId = -1;
72             newFree.startAddr = blk.startAddr + size;
73             newFree.size = blk.size - size;
74             newFree.isAllocated = false;
75             blk.size = size;
76             blk.isAllocated = true;
77             blk.segId = nextSegId++;
78             memoryBlocks.insert(memoryBlocks.begin() + i + 1, newFree);
79             cout << "[ALLOC] " << size << "KB, SegID = " << blk.segId
80                 << ", Start = " << blk.startAddr << endl;
81             return blk.segId;
82         }
83     }
84     cout << "[FAIL] Not enough continuous space for " << size << "KB.\n";
85     return -1;

```

```

86 }
87 /* ----- Deallocate Segment ----- */
88 bool deallocate(int segId) {
89     for (auto &blk : memoryBlocks) {
90         if (blk(segId == segId && blk.isAllocated) {
91             blk(segId = -1;
92             blk.isAllocated = false;
93             cout << "[FREE] Segment " << segId << " released.\n";
94             mergeFreeBlocks();
95             return true;
96         }
97     }
98     cout << "[FREE FAIL] Segment not found.\n";
99     return false;
100 }
101 /* ----- Memory Statistics ----- */
102 void memoryStats() {
103     int freeTotal = 0, freeBlocks = 0, maxFree = 0;
104     for (auto &blk : memoryBlocks) {
105         if (!blk.isAllocated) {
106             freeBlocks++;
107             freeTotal += blk.size;
108             maxFree = max(maxFree, blk.size);
109         }
110     }
111     double fragRate =
112         (freeTotal > 0 ? (freeTotal - maxFree) * 100.0 / freeTotal : 0.0);
113     cout << "\n===== Memory Statistics =====\n";
114     cout << "Total Memory:    1024 KB\n";
115     cout << "Free Memory:    " << freeTotal << " KB\n";
116     cout << "Allocated Memory: " << TOTAL_MEMORY - freeTotal << " KB\n";
117     cout << "Free Block Count: " << freeBlocks << "\n";
118     cout << "Largest Free Block: " << maxFree << " KB\n";
119     cout << fixed << setprecision(2);
120     cout << "External Fragmentation: " << fragRate << "%\n";
121     cout << "===== ======\n\n";
122 }
123 /* ----- Print Memory Blocks ----- */
124 void printMemory() {
125     cout << "\n===== Memory Block Table =====\n";
126     cout << left << setw(8) << "SegID"
127         << setw(12) << "Start"
128         << setw(10) << "Size"

```

```

129     << setw(12) << "Status" << endl;
130     cout << "-----\n";
131     for (auto &blk : memoryBlocks) {
132         cout << left
133             << setw(8) << (blk(segId == -1 ? string("Free") : to_string(blk(segId)))
134             << setw(12) << blk.startAddr
135             << setw(10) << blk.size
136             << setw(12) << (blk.isAllocated ? "Allocated" : "Free")
137             << endl;
138     }
139     cout << "=====\\n\\n";
140 }
141 /* ----- Random Simulation ----- */
142 void simulateRandom(int rounds) {
143     vector<int> allocated;
144     cout << "\\n===== Start Random Simulation =====\\n";
145     for (int i = 1; i <= rounds; i++) {
146         cout << "\\n--- Round " << i << " ---\\n";
147         bool doAlloc = (allocated.empty() ? true : rand() % 2 == 0);
148         if (doAlloc) {
149             int size = 10 + rand() % 150;
150             int segId = allocate(size);
151             if (segId != -1) allocated.push_back(segId);
152         } else {
153             int idx = rand() % allocated.size();
154             int segId = allocated[idx];
155             if (deallocate(segId)) {
156                 allocated.erase(allocated.begin() + idx);
157             }
158         }
159     }
160 }
161 cout << "===== End Random Simulation =====\\n\\n";
162 }
163 /* ----- Main ----- */
164 int main() {
165     initMemory();
166     cout << "Initial Memory:\\n";
167     printMemory();
168     simulateRandom(20);
169     memoryStats();
170     printMemory();
171     return 0;

```

```

172 }
173 #include <iostream>
174 #include <vector>
175 #include <cstdlib>
176 #include <ctime>
177 #include <iomanip>
178 #include <algorithm>
179 using namespace std;
180 /* ----- Segment and Memory Block Definitions ----- */
181 // Logical segment (just for demonstration)
182 struct Segment {
183     int segId;
184     int size;
185 };
186 // Physical memory block (free or allocated)
187 struct MemoryBlock {
188     int segId;      // Segment ID (-1 means free)
189     int startAddr;
190     int size;       // Size in KB
191     bool isAllocated;
192 };
193 /* ----- Global Variables ----- */
194 vector<MemoryBlock> memoryBlocks; // physical memory
195 const int TOTAL_MEMORY = 1024; // 1 MB
196 int nextSegId = 1;           // auto-increment segment ID
197 int randomSeed = time(0);
198 /* ----- Initialization ----- */
199 void initMemory() {
200     memoryBlocks.clear();
201     MemoryBlock block;
202     block(segId = -1;
203     block.startAddr = 0;
204     block.size = TOTAL_MEMORY;
205     block.isAllocated = false;
206     memoryBlocks.push_back(block);
207     srand(randomSeed);
208 }
209 /* ----- Coalescing Free Blocks ----- */
210 void mergeFreeBlocks() {
211     vector<MemoryBlock> newList;
212     for (int i = 0; i < memoryBlocks.size(); i++) {
213         MemoryBlock cur = memoryBlocks[i];
214         if (!cur.isAllocated) {

```

```

215     while (i + 1 < memoryBlocks.size() &&
216         !memoryBlocks[i+1].isAllocated) {
217         cur.size += memoryBlocks[i+1].size;
218         i++;
219     }
220 }
221 newList.push_back(cur);
222 }
223 memoryBlocks = newList;
224 }
225 /* ----- First Fit Allocation ----- */
226 int allocate(int size) {
227     if (size <= 0 || size > TOTAL_MEMORY) {
228         cout << "Invalid allocation size.\n";
229         return -1;
230     }
231     for (int i = 0; i < memoryBlocks.size(); i++) {
232         MemoryBlock &blk = memoryBlocks[i];
233         if (!blk.isAllocated && blk.size >= size) {
234             if (blk.size == size) {
235                 blk.isAllocated = true;
236                 blk(segId = nextSegId++);
237                 cout << "[ALLOC] " << size << "KB, SegID = " << blk.segId
238                     << ", Start = " << blk.startAddr << endl;
239                 return blk.segId;
240             }
241             // Split block
242             MemoryBlock newFree;
243             newFree.segId = -1;
244             newFree.startAddr = blk.startAddr + size;
245             newFree.size = blk.size - size;
246             newFree.isAllocated = false;
247             blk.size = size;
248             blk.isAllocated = true;
249             blk.segId = nextSegId++;
250             memoryBlocks.insert(memoryBlocks.begin() + i + 1, newFree);
251             cout << "[ALLOC] " << size << "KB, SegID = " << blk.segId
252                 << ", Start = " << blk.startAddr << endl;
253             return blk.segId;
254         }
255     }
256     cout << "[FAIL] Not enough continuous space for " << size << "KB.\n";
257     return -1;

```

```

258 }
259 /* ----- Deallocate Segment ----- */
260 bool deallocate(int segId) {
261     for (auto &blk : memoryBlocks) {
262         if (blk(segId == segId && blk.isAllocated) {
263             blk(segId = -1;
264             blk.isAllocated = false;
265             cout << "[FREE] Segment " << segId << " released.\n";
266             mergeFreeBlocks();
267             return true;
268         }
269     }
270     cout << "[FREE FAIL] Segment not found.\n";
271     return false;
272 }
273 /* ----- Memory Statistics ----- */
274 void memoryStats() {
275     int freeTotal = 0, freeBlocks = 0, maxFree = 0;
276     for (auto &blk : memoryBlocks) {
277         if (!blk.isAllocated) {
278             freeBlocks++;
279             freeTotal += blk.size;
280             maxFree = max(maxFree, blk.size);
281         }
282     }
283     double fragRate =
284     (freeTotal > 0 ? (freeTotal - maxFree) * 100.0 / freeTotal : 0.0);
285     cout << "\n===== Memory Statistics =====\n";
286     cout << "Total Memory:    1024 KB\n";
287     cout << "Free Memory:    " << freeTotal << " KB\n";
288     cout << "Allocated Memory: " << TOTAL_MEMORY - freeTotal << " KB\n";
289     cout << "Free Block Count: " << freeBlocks << "\n";
290     cout << "Largest Free Block: " << maxFree << " KB\n";
291     cout << fixed << setprecision(2);
292     cout << "External Fragmentation: " << fragRate << "%\n";
293     cout << "===== ======\n\n";
294 }
295 /* ----- Print Memory Blocks ----- */
296 void printMemory() {
297     cout << "\n===== Memory Block Table =====\n";
298     cout << left << setw(8) << "SegID"
299             << setw(12) << "Start"
300             << setw(10) << "Size"

```

```

301     << setw(12) << "Status" << endl;
302     cout << "-----\n";
303     for (auto &blk : memoryBlocks) {
304         cout << left
305             << setw(8) << (blk(segId == -1 ? string("Free") : to_string(blk(segId)))
306             << setw(12) << blk.startAddr
307             << setw(10) << blk.size
308             << setw(12) << (blk.isAllocated ? "Allocated" : "Free")
309             << endl;
310     }
311     cout << "=====\\n\\n";
312 }
313 /* ----- Random Simulation ----- */
314 void simulateRandom(int rounds) {
315     vector<int> allocated;
316     cout << "\\n===== Start Random Simulation =====\\n";
317     for (int i = 1; i <= rounds; i++) {
318         cout << "\\n--- Round " << i << " ---\\n";
319         bool doAlloc = (allocated.empty() ? true : rand() % 2 == 0);
320         if (doAlloc) {
321             int size = 10 + rand() % 150;
322             int segId = allocate(size);
323             if (segId != -1) allocated.push_back(segId);
324         } else {
325             int idx = rand() % allocated.size();
326             int segId = allocated[idx];
327             if (deallocate(segId)) {
328                 allocated.erase(allocated.begin() + idx);
329             }
330         }
331     }
332 }
333     cout << "===== End Random Simulation =====\\n\\n";
334 }
335 /* ----- Main ----- */
336 int main() {
337     initMemory();
338     cout << "Initial Memory:\\n";
339     printMemory();
340     simulateRandom(20);
341     memoryStats();
342     printMemory();
343     return 0;

```

344 }