












COP4610 Assignment 4: Banker's Algorithm Implementation for Deadlock Avoidance

Table of Contents

1.  [TA Contact Information](#)
2.  [Assignment Overview](#)
3.  [Skills You'll Develop](#)
4.  [Learning Objectives](#)
5.  [Assignment Tasks](#)
6.  [Project Framework](#)
7.  [Development & Testing](#)
8.  [Assignment Submission Guidelines](#)
9.  [Deliverable Folder Structure](#)
10.  [Grading Rubric](#)

Important Note!

Please ensure that your implementation works on the course-designated ocelot server: **ocelot-bbhatkal.aul.fiu.edu** . Claims such as "it works on my machine" will not be considered valid, as all testing and grading will be conducted in that environment.

 **Important:** You are provided with an autograder to support consistent testing and maintain the highest level of grading transparency. The same autograder will be used by the instructor for final evaluation.

If your implementation passes the test case using the autograder, you are likely to receive full credit for your submission.

You must use these strict gcc compilation flags while building your application:

-std=c17

-D_POSIX_C_SOURCE=200809L


-Wall

-Wextra

-Werror

-g

-O0

 **There is only one submission - RESUBMISSION IS NOT ALLOWED.** You must test your implementation with the provided testing framework on the Ocelot server before the final submission.

1. TA Contact Information

For any questions or concerns related to assignment grading, please reach out to the TA listed below.

Communication Guidelines:

- Ensure that all communication is polite and respectful
- **You must contact the Graders first for any grading issues**
- If the matter remains unresolved, feel free to reach out to the instructor

Name: Instructor – bbhatkal@fiu.edu ( via the Canvas inbox only)

Section: COP 4610-U01 (84957)

Name: Angie Martinez – amart1070@fiu.edu ( via the Canvas inbox only)

Section: COP 4610-U02 (85031)

Name: Srushti Visweswaraiah – svisw003@fiu.edu ( via the Canvas inbox only)

Sections: COP 4610-UHA (85236)

2. Assignment Overview

This assignment provides **hands-on experience** with **deadlock avoidance**, **resource allocation management**, and **system safety analysis**. You will implement the **Banker's Algorithm**, a fundamental deadlock avoidance technique used in **modern operating systems** to ensure system safety by analyzing resource allocation patterns before granting requests.

Through this implementation, you will explore how operating systems make **critical decisions** about **resource allocation** to prevent **deadlock conditions** while maintaining **optimal system performance** and **resource utilization**. Your implementation must exhibit **deterministic behavior** – *for the same input, it must consistently produce the same output*, **correct safety state verification**, and **proper handling of resource allocation and release operations**—just as real operating system resource managers must.

The Banker's Algorithm is a **critical mechanism** enabling safe resource allocation, deadlock prevention, and efficient resource management in database systems, distributed computing platforms, and operating systems. Mastering its implementation will deepen your understanding of how operating systems achieve safety, efficiency, and reliability at the resource management level.

3. Skills You'll Develop

In this assignment, you'll **gain hands-on experience** with both the **technical skills** and **operating system concepts** that are essential for professional systems programming and OS-level development such as:

- **Implement** the Banker's Algorithm for deadlock avoidance in multi-threaded systems.
- **Calculate** need matrices and available resource vectors for system state analysis.
- **Implement** the safety algorithm to determine safe execution sequences for processes.

- **Validate** resource allocation and deallocation requests with comprehensive checks.
- **Handle** both resource allocation and release operations with state management.
- **Work** with complex matrix operations for resource tracking and dependency analysis.
- **Apply** state space analysis to evaluate system safety under various scenarios.
- **Understand** algorithms for correctness and performance in resource management.
- **Use** modular design principles for maintainable and reusable code.
- **Test** your code thoroughly against deadlock scenarios and edge cases.
- **Produce** professional-quality C code following industry standards.

4. Learning Objectives

- **Understand** and implement the Banker's Algorithm used in real operating systems.
- **Apply** matrix-based resource allocation tracking for multi-threaded systems.
- **Implement** need matrix calculation using matrix subtraction operations.
- **Design** available resource calculation from total and allocated resources.
- **Build** safety verification algorithm to find safe execution sequences.
- **Create** resource request processing with validation and rollback mechanisms.
- **Calculate** system state transitions for allocation and release operations.
- **Handle** edge cases including over-allocation attempts, resource release, and zero requests.

5. Assignment Tasks

Banker's Algorithm Implementation Requirements

You will implement **four core functions** of the Banker's Algorithm using a provided modular framework:

1). Calculate Need Matrix (`need.c`):

Calculate the need matrix representing remaining resource requirements for each process

- **Key Characteristics:** Matrix subtraction ($Max - Allocation$), foundation for safety analysis
- **Implementation Focus:** Accurate calculation for all threads and resources
- **Usage in OS:** Resource requirement tracking, process scheduling decisions
- **Critical Aspect:** Must be computed correctly as all other functions depend on it

2). Calculate Available Resources (`available.c`):

Determine currently available resource instances by analyzing total resources and allocations

- **Key Characteristics:** Vector subtraction from total resources, dynamic resource tracking
- **Implementation Focus:** Account for all allocated resources across all threads
- **Usage in OS:** Real-time resource availability monitoring, allocation decisions
- **Critical Aspect:** Essential for validating whether new requests can be satisfied

3). Safety Algorithm (`safety.c`):

Implement the core safety algorithm to determine if system is in a safe state

- **Key Characteristics:** Simulates process execution, finds safe sequences, prevents deadlock
- **Implementation Focus:** Iterative process completion simulation with available resources
- **Usage in OS:** Deadlock prevention, safe resource allocation decisions
- **Critical Aspect:** *Heart of deadlock avoidance* - determines if system can safely proceed

4). Resource Request Processing (`request.c`):

Process resource allocation and deallocation requests with comprehensive validation

- **Key Characteristics:** Request validation, temporary state changes, safety verification, rollback capability
- **Implementation Focus:** Complete request lifecycle with deadlock prevention guarantees
- **Usage in OS:** System calls for resource allocation, process resource management
- **Critical Aspect:** Implements both allocation and release with safety checks

Input/Output Specifications

The provided driver program reads system state from `SYSTEM_STATE.txt` and test cases from `INPUT.txt`, then executes the Banker's Algorithm with your implementations.

System State File Format (`SYSTEM_STATE.txt`):

```

1  Number of threads (T) = 5
2  Number of resources (R) = 3
3
4  Number of instances of resource R0 = 12
5  Number of instances of resource R1 = 8
6  Number of instances of resource R2 = 10
7
8  Allocation matrix
9  =====
10      R0  R1  R2
11  T0   1   1   2
12  T1   2   1   1
13  T2   2   1   1
14  T3   1   1   2
15  T4   3   1   0
16
17  Max matrix
18  =====
19      R0  R1  R2
20  T0   5   4   4
21  T1   4   3   3
22  T2   9   2   2
23  T3   3   2   2
24  T4   4   2   2

```

Test Cases File Format (INPUT . txt):

```

1  # Resource allocation request (positive values)
2  REQUEST 0 2 1 0
3
4  # Resource release (negative values)
5  REQUEST 2 -2 -1 -1
6
7  # Safety check command
8  SAFETY_CHECK

```

Expected Output Format:

```

1  INITIAL_SYSTEM_STATE:
2  Threads=5 Resources=3
3  Total: R0=12 R1=8 R2=10
4  Available: R0=3 R1=3 R2=4
5  Allocation: T0[1,1,2] T1[2,1,1] T2[2,1,1] T3[1,1,2] T4[3,1,0]
6  Max: T0[5,4,4] T1[4,3,3] T2[9,2,2] T3[3,2,2] T4[4,2,2]
7  Need: T0[4,3,2] T1[2,2,2] T2[7,1,1] T3[2,1,0] T4[1,1,2]
8  SAFETY_CHECK: SAFE <T1,T3,T4,T0,T2>
9
10 TEST_CASE_1:
11 REQUEST: T0 [2,1,0]
12 REQUEST_RESULT: GRANTED <T4,T0,T1,T2,T3>
13 Available: R0=1 R1=2 R2=4
14 Allocation: T0[3,2,2] T1[2,1,1] T2[2,1,1] T3[1,1,2] T4[3,1,0]
15 Max: T0[5,4,4] T1[4,3,3] T2[9,2,2] T3[3,2,2] T4[4,2,2]
16 Need: T0[2,2,2] T1[2,2,2] T2[7,1,1] T3[2,1,0] T4[1,1,2]
17
18 TEST_CASE_2:
19 REQUEST: T1 [0,2,0]
20 REQUEST_RESULT: DENIED
21 Available: R0=1 R1=2 R2=4
22 Allocation: T0[3,2,2] T1[2,1,1] T2[2,1,1] T3[1,1,2] T4[3,1,0]
23 Max: T0[5,4,4] T1[4,3,3] T2[9,2,2] T3[3,2,2] T4[4,2,2]
24 Need: T0[2,2,2] T1[2,2,2] T2[7,1,1] T3[2,1,0] T4[1,1,2]

```

6. Project Framework

This assignment provides a **complete development framework** including **skeleton files** (*templates for your implementation*), **driver program**, **Makefile**, **sample executable**, **test cases at THREE difficulty levels**, and **autograder scripts** to guide your implementation process.

1) Provided Files Structure

```

1  A4_PROVIDED_FILES/
2  └─ Framework/
3     └─ autograder_bankers.sh      # Automated testing script (single submission)
4     └─ batchgrader_bankers.sh    # Batch testing for multiple submissions
5     └─ banker_header.h           # Header file with structures and prototypes
6     └─ banker_driver.c           # Main driver program
7     └─ Makefile                   # Build configuration with strict flags
8     └─ SYSTEM_STATE.txt          # System configuration file
9     └─ INPUT.txt                  # Default test input file
10    └─ EXPECTED_OUTPUT.txt       # Default expected output file
11
12  └─ Sample_Executable/
13     └─ A4_sample.exe             # Reference implementation executable
14
15  └─ Skeleton_Codes/
16     └─ available_skeleton.c      # Calculate available resources skeleton
17     └─ need_skeleton.c           # Calculate need matrix skeleton
18     └─ request_skeleton.c        # Process requests skeleton
19     └─ safety_skeleton.c         # Safety algorithm skeleton
20
21  └─ Testcases/                  # Contains the three testcases you are required to PASS

```

2) Framework Files (🔒 Do NOT modify these files)

- `banker_header.h`: Complete header file with data structures, constants, and function prototypes
- `banker_driver.c`: Main program that orchestrates algorithm execution and manages test case processing
- `Makefile`: Professional build configuration with strict compiler flags (`-std=c17 -Wall -Wextra -Werror -g -O0`)
- `SYSTEM_STATE.txt`: System configuration defining threads, resources, allocation matrix, and max demand matrix
- `autograder_bankers.sh`: Single-submission automated testing script (identical to final grading mechanism)
- `batchgrader_bankers.sh`: Batch grading script for processing multiple student submissions

3) Skeleton Files (✅ You must implement these)

- `available_skeleton.c` → Rename to `available.c` after implementation
- `need_skeleton.c` → Rename to `need.c` after implementation
- `request_skeleton.c` → Rename to `request.c` after implementation
- `safety_skeleton.c` → Rename to `safety.c` after implementation

Each skeleton file contains:

- Function signature and parameters
- Detailed comments explaining requirements
- TODO sections marking where you write code
- Hints for implementation approach

4) Sample Executable

- `A4_sample.exe`: Reference implementation showing expected behavior
- Run this to understand correct output format and algorithm behavior
- Compare your output against this reference for validation

5) Execution Permissions Notice

If you **do not** have execute permissions for the instructor-provided files, you can manually grant the required permissions using the following command on a **Linux system**.

Note: When files are uploaded or transferred to Ocelot server (`ocelot-bbhatka1.aul.fiu.edu`), execute permissions **may be stripped** due to security restrictions. In such cases, you must explicitly grant **read (r)**, **write (w)**, and **execute (x)** permissions.

Granting Permissions to All Items in the Current Directory

```

1  # Check current permissions
2  ls -l
3
4  # Grant read, write, and execute permissions (safe and preferred – applies only what's missing)
5  chmod +rwx *
6
7  # Assign full permissions (read, write, execute) to all users for all files in the current
  # directory
8  # ⚠ Not preferred – use only as a last resort!
9  chmod 777 *
10
11 # Verify that permissions were applied
12 ls -l

```

 **Caution: Use `chmod 777` with care!**

Granting full permissions to everyone can:

- Expose sensitive data
- Allow unintended modifications or deletions
- Introduce security risks, especially in shared or multi-user environments

6) Test Cases at Three Difficulty Levels

You are provided with **three comprehensive test case sets** to validate your implementation incrementally:

1. **Level-0 (Simple):** Basic functionality testing
 - Simple allocation and release requests
 - Straightforward safe sequences
 - Basic validation scenarios
2. **Level-1 (Moderate):** Intermediate complexity scenarios
 - Multiple sequential requests
 - Mix of granted and denied requests

- Moderate resource contention
- 3. **Level-2 (Rigorous):** Complex edge cases and stress testing
 - Edge cases including zero requests
 - Resource over-allocation attempts
 - Complex state transitions
 - Maximum resource contention scenarios

⚠ **CRITICAL REQUIREMENT:** Your implementation **MUST** pass **ALL** three test case levels with 100% accuracy before submission.

7. 🛠 Development & Testing

Recommended Implementation Order

Follow this sequence for optimal development progression:

Step 1: Setup and Understand the Framework

```

1  # Extract the provided files
2  cd A4_PROVIDED_FILES/Framework/
3
4  # Examine the framework structure
5  cat banker_header.h      # Understand data structures
6  cat banker_driver.c      # See how driver calls your functions
7  cat SYSTEM_STATE.txt     # Study input format
8  cat EXPECTED_OUTPUT.txt  # Study output format
9
10 # Run sample executable to see expected behavior
11 cd ../Sample_Executable/
12 ./A4_sample.exe

```

Step 2: Implement Functions in Order

1. Start with `need.c` (Foundation)

```

1  # Copy skeleton to working file
2  cp ../Skeleton_Codes/need_skeleton.c need.c
3
4  # Implement need matrix calculation
5  # Need[i][j] = Max[i][j] - Allocation[i][j]

```

2. Then implement `available.c` (Resource Tracking)

```

1  cp ../Skeleton_Codes/available_skeleton.c available.c
2
3  # Implement available resources calculation
4  # Available[j] = Total[j] - Sum(Allocation[i][j]) for all threads i

```

3. Next implement `safety.c` (Core Algorithm)

```
1 cp ../Skeleton_Codes/safety_skeleton.c safety.c
2
3 # Implement safety algorithm
4 # Find safe sequence by simulating process completion
```

4. Finally implement `request.c` (Request Management)

```
1 cp ../Skeleton_Codes/request_skeleton.c request.c
2
3 # Implement request processing
4 # Validate → Tentatively allocate → Check safety → Commit or rollback
```

Step 3: Compile and Test Incrementally

```
1 # Compile with strict flags
2 make clean
3 make
4
5 # Test with Level-0 (Simple) test cases
6 cp Testcases/Level-0/INPUT.txt .
7 cp Testcases/Level-0/EXPECTED_OUTPUT.txt .
8 ./banker > STUDENT_OUTPUT.txt
9
10 # Compare outputs
11 diff STUDENT_OUTPUT.txt EXPECTED_OUTPUT.txt
```

Step 4: Progress Through Test Levels

```
1 # Test Level-1 (Moderate)
2 cp Testcases/Level-1/INPUT.txt .
3 cp Testcases/Level-1/EXPECTED_OUTPUT.txt .
4 ./banker > STUDENT_OUTPUT.txt
5 diff STUDENT_OUTPUT.txt EXPECTED_OUTPUT.txt
6
7 # Test Level-2 (Rigorous)
8 cp Testcases/Level-2/INPUT.txt .
9 cp Testcases/Level-2/EXPECTED_OUTPUT.txt .
10 ./banker > STUDENT_OUTPUT.txt
11 diff STUDENT_OUTPUT.txt EXPECTED_OUTPUT.txt
```

Step 5: Use Autograders for Final Validation

```

1 # Make autograders executable (if needed)
2 chmod +x autograder_bankers.sh
3 chmod +x batchgrader_bankers.sh
4
5 # Run the autograder (tests single submission)
6 ./autograder_bankers.sh
7
8 # For batch testing multiple submissions
9 ./batchgrader_bankers.sh

```

⚠ **Important:** The instructor will use the same autograder for final grading.

Autograder System

The autograder compares your program's output with pre-provided reference output to determine correctness.

Internal Process:

1. **Clean rebuild:** `make clean && make`
2. **Runs your implementation:** `./banker > STUDENT_OUTPUT.txt`
3. **Compares outputs:** Extracts safety checks, request results, and system states
4. **Calculates scores:** Awards points based on percentage of matching output
5. **Provides detailed feedback:** Shows exactly which components don't match

What Gets Compared:

- **Safety Check Results:** `SAFETY_CHECK: SAFE <T1,T3,T4,T0,T2>` format matching
- **Request Results:** `REQUEST_RESULT: GRANTED` vs `REQUEST_RESULT: DENIED` accuracy
- **System State Updates:** Matrix values after successful requests
- **Safe Sequence Generation:** Order of processes in safe execution sequences

Understanding Autograder Output







When the autograder detects mismatches, it shows:

```


1  ✓ Safety Algorithm: 100% match (30/30 points)
2
3  ✗ Valid Request Processing: 80% match (16/20 points)
4    Expected Output:
5      REQUEST_RESULT: GRANTED <T4,T0,T1,T2,T3>
6    Your Output:
7      REQUEST_RESULT: DENIED ← Incorrect decision
8
9  ⚠ Invalid Request Processing: 75% match (15/20 points)
10   Expected Output:
11     REQUEST_RESULT: DENIED
12   Your Output:
13     REQUEST_RESULT: GRANTED <T1,T2,T3> ← Should have been denied

```

8. Assignment Submission Guidelines

- All assignments **must be submitted via Canvas** by the specified deadline as a **single compressed (.ZIP) file**. Submissions by any other means **WILL NOT** be accepted and graded!
- The filename must follow the format: `A4_FirstName_LastName.zip` (e.g., **A4_Harry_Potter.zip**)
-  **Important:** Include the Firstname and Lastname of the **team's submitting member**, or your own name if **submitting solo**.
-  **Important:** For **group submissions**, each group must designate one **corresponding member**. This member will be responsible for submitting the assignment and will serve as the primary point of contact for all communications related to that assignment.
-  **Important Policy on Teamwork and Grading**
All members of a team will receive the **same grade** for collaborative assignments. It is the responsibility of the team to ensure that work is **distributed equitably** among members. Any disagreements or conflicts should be addressed and resolved within the team in a **professional manner**. **If a resolution cannot be reached internally, the instructor may be requested to intervene**.
-  **Important:** Your implementation **must be compiled and tested on the course designated Ocelot server** (`ocelot-bbhatka1.aul.fiu.edu`) **to ensure correctness and compatibility**. **Submissions that fail to compile or run as expected on Ocelot will receive a grade of ZERO.**
- Whether submitting as a **group or individually**, every submission must include a **README** file containing the following information:
 1. Full name(s) of all group member(s)
 2. **Prefix the name of the corresponding member with an asterisk** 
 3. Panther ID (PID) of each member
 4. FIU email address of each member
 5. Include any relevant notes about compilation, execution, or the programming environment needed to test your assignment. **Please keep it simple—avoid complex setup or testing requirements.**
-  **Important - Late Submission Policy:** For each day the submission is delayed beyond the **Due date** and until the **Until date** (as listed on Canvas), a **10% penalty** will be applied to the total score. **Submissions after the Until date will not be accepted or considered for grading.**
 - **Due Date:** ANNOUNCED ON CANVAS
 - **Until Date:** ANNOUNCED ON CANVAS
- Please refer to the course syllabus for any other details.

9. Deliverable Folder (ZIP file) Structure (**exact structure required - no additional files / subfolders**)

 **DO NOT submit any other files other than these required files. The batch autograder may treat any additional items in the ZIP file as invalid, which will result in a grade of zero:** `need.c`, `available.c`, `safety.c`, `request.c`, `README.txt`

📁 All other required framework files and the test cases will be supplied by the instructor to the autograder during final grading.

- ⚠️ You are required to submit your work as a **single ZIP archive file** to the Canvas.
- ⚠️ **Filename Format:** your `A4_Firstname_Lastname.zip` (exact format required).
- You will be held responsible for receiving a ZERO grade if the submission guidelines are not followed.**
- ⚠️ **For team submissions** – Include the first name and last name of the team's submitting member in the ZIP file name (e.g., `A4_Harry_Potter.zip`). **For solo submissions** – Use your own first and last name.

```

1 A4_FirstName_LastName.zip
2 |— need.c           # Calculate need matrix implementation
3 |— available.c      # Calculate available resources implementation
4 |— safety.c         # Safety algorithm implementation
5 |— request.c        # Resource request processing implementation
6 |— README.txt       # Team details (see submission guidelines)

```

10. ¹⁰⁰ Grading Rubric

Criteria	Points
⚠️ Program fails to run or crashes during testing	0
⚠️ Partial Implementation (Evaluated based on completeness and at the instructor's discretion)	0 - 70

Detailed Breakdown of 100 Points

Grading Component	Points
1. Safety Algorithm (<code>safety.c</code>) implementation	30
2. Valid Request Processing (<code>request.c</code>) - Granted requests	20
3. Invalid Request Processing (<code>request.c</code>) - Denied requests	20
4. Edge Case Handling - Zero requests, releases, over-allocation	20
5. Manual Grading - Clear code structure, Meaningful comments, well-written README, Overall readability	10
Total	100

⚠️ **Note:** The autograder awards 90 points automatically based on correctness. The remaining 10 points are manually graded for code quality, comments, structure, and README completeness.

🚀 Tips for Success:

- Run the sample executable (`A4_sample.exe`) to understand expected behavior
- Implement functions in recommended order: need → available → safety → request

- Test incrementally with all the test levels (Level-0, Level-1, Level-2)
- Ensure **exact output matching** including formatting and safe sequence order
- Verify safety algorithm finds valid safe sequences
- Test both allocation and release operations thoroughly
- Handle edge cases: zero requests, over-allocation, resource release
- Use autograder for validation before submission
- Maintain professional code quality throughout development

Good luck! 🙌
