Implementation of the Banker's Algorithm for Deadlock Avoidance in Resource Allocation

1. References for this Assignment Implementation

You are provided with a detailed **Assignment Specification Document**, an **Instructional Video**, and **Sample Executable(s)** to help you understand the implementation requirements for this assignment. Please follow these guidelines carefully:

1. Your implementation must strictly adhere to this **Assignment Specification Document**, which serves as the primary reference for this assignment.
2. Your implementation should precisely replicate the functionality, input handling, and output behavior demonstrated in the instructor-provided **Sample Executable(s)** and explained in the **Instructional Video**.
3. If you are unable to run the **Sample Executable(s)** for any reason, you must refer to this **Assignment Specification Document** and **Instructional Video** to fully understand the assignment requirements.

2. Skeleton Code (*For Reference Only*)

A **skeleton code** is provided as a reference to help you structure your implementation of the **Banker's Algorithm**. You may use this code as a starting point and complete the missing sections.

Important Notes:

- The skeleton code serves **only as a guideline** and is **not mandatory** for your implementation.
- You are encouraged to apply a **modular programming approach**, ensuring code clarity, reusability, and maintainability.
- If you choose to deviate from the skeleton code, ensure that your implementation adheres to the required functionality and produces correct results.

3. Overview

This assignment involves the implementation of the **Banker's Algorithm**, a deadlock avoidance technique designed for systems with **multiple instances of each resource type**. The algorithm evaluates system safety by analyzing resource allocation, maximum process demands, and available resources to ensure that the system remains in a **safe state** before granting resource requests. The implementation will read input from a structured file (`input.txt`), process allocation and need matrices, and dynamically handle user-initiated resource requests. Each request will be validated to determine whether granting it maintains system safety, ensuring deadlock prevention.

4. Objectives

By completing this assignment, you will:

- **Apply Deadlock Avoidance Principles:**
  Implement the **Banker's Algorithm** to manage resource allocation in a **multithreaded system**, ensuring that processes operate within a **safe state** to prevent deadlocks.
- **Analyze and Compute Resource Allocation Matrices:**
  Construct and manipulate key matrices—**Allocation, Maximum, Need, and Available Resources**—to dynamically evaluate system safety and determine the feasibility of granting resource requests.
- **Implement and Validate Safety Algorithm:**
  Develop a safety-checking mechanism to compute a **safe sequence** of process execution, ensuring that system stability is maintained before granting requested resources.
- **Simulate Dynamic Resource Request Handling:**
  Design an interactive system that processes real-time resource requests, validates them against process constraints, and executes **rollback mechanisms** if the request leads to an unsafe state.
- **Optimize System Performance:**
  Explore the implications of granting or denying resource requests in real time, and understand how small changes in resource allocation can affect the safety and performance of the overall system.

5. Implementation Details

Your implementation must adhere to the following guidelines:

1. **Input Handling:**
   - Your program must **read input from a text file (`input.txt`)** to demonstrate the application of the **Banker's Algorithm**.
   - The relevant functions for input parsing are already provided in the **Skeleton Code**.
   - If you choose to implement a different approach, ensure that your program still correctly reads and processes input from `input.txt`, as it will be tested using multiple test cases.
2. **Required Function Implementations:**
   You are required to implement the following **three key functions**, which are partially defined in the skeleton code:

   a) **Function to calculate available resources**

   - Computes the number of available instances for each resource type based on system allocations.

   b) **Function to check if the system is in a safe state**

   - Implements the **safety algorithm** to determine whether the system can allocate resources without leading to a deadlock.

   c) **Function to process a thread's resource request**

   - Validates a resource request, temporarily grants it, checks system safety, and rolls back if granting the request results in an unsafe state.
3. **Interactive User Input (Menu-Driven System):**

- Your implementation should be **menu-driven**, as demonstrated in the **instructor's sample executable**.
- The program should continuously prompt the user to enter:
  - A **thread number**
  - A **resource request**
- The option `q` should allow the user to **quit the program**.
- The provided `main()` function in the **Skeleton Code** already implements this logic. However, if you choose to modify `main()`, you must ensure that your implementation adheres to this interactive input requirement.

6. Testing

You must thoroughly test your implementation using various inputs. A good starting point is the **examples provided in the textbook and lecture slides**.

🔹 **Important:** Ensure that your implementation **strictly follows the specified format** of `input.txt`, as your program will be evaluated using test cases formatted exactly the same way.

6.1. Provided Test Cases

- The file `Test_Cases_STUDENTS` contains multiple test cases designed to verify the correctness of your program.
- Your implementation must produce the expected output for all provided test cases, as well as similar cases that will be used during evaluation.
- The provided test case includes **5 threads and 3 resources**, but your implementation must be **scalable** to handle **any number of threads and resources**, as implemented in the **instructor's sample executable**.

7. Deliverables

Submit a single `Bonus_Deliverables.zip` file containing the following:

1. A `CODE` Folder

The `CODE` folder must include:

- Source Code

  All `.c` files and any optional `.h` header files used in your project.
- Statically-Linked Executable
  - Provide statically-linked executable for your scheduler implementation that is ready to be run.
  - If you are using **non-Linux computer**, explore ways to generate a statically-linked executable by compiling your final source files on a Linux computer. For example:

    ```
    gcc bankers.c -static -o bankers
    ```

    - Alternatively, explore online C compilers capable of generating statically-linked executables.
- Make file
    - Provide a Makefile that supports the following commands:
        - `make`: Compiles the project and generates one executable file named `bankers` for your implementation.
        - `make clean`: Removes all object files, executable, and any temporary files generated during the build process.

2. A `README.txt` File

The `README.txt` file must include:

- Team Member Information
    - Section Name, Full Name, PID, FIU email of all team members.
- Compilation and Execution Instructions
    - Include any specific guidelines, notes, or considerations about the project implementation.

***** *Please refer to the course syllabus for additional assignment submission requirements and guidelines.*

7.1. Deliverables Folder Structure

You are required to strictly adhere to the following structure for the `Bonus_Deliverables.zip` file when submitting the assignment.

Bonus_Deliverables.zip
|
|-- **CODE/**
| |-- `*.c`   # All `.c` source files for your implementation
| |-- `*.h`   # Any required `.h` header files for your implementation
| |-- `makefile`   # Makefile with commands: `make` and `make clean`
| |-- `executable`   # `Statically linked executable file` of your implementation.
|-- **README.txt**   # Includes team details and any specific compilation instructions

8. Grading Rubric

Criteria	Marks
No Makefile	**0**
No Menu Driven Implementation	**0**
Cannot Generate/Run Executables	**0**
Cannot Read and Process Thread's Data from `input.txt` in the same format	**0**
Partial Implementation *(Based on completeness)*	**0 - 70**

8.1. Detailed Breakdown of 100 Marks

Category	Test Case	Marks
(1). Banker’s Algorithm Implementation	System is in Safe State	15
	A thread requested more than its need	10
	The requesting thread must wait, resources not available	10
	The thread request is denied as granting it will not leave the in a safe state	20
	A thread releasing all its resources	10
	A thread’s previously denied request now safe	15
	A thread exhausting resource(s)	10
	Total of (1)	**90**
(2). Code Quality, Documentation, README	**Total of (2)**	**10**

9. Guidelines and Important Notes

- Thoroughly refer to the **Assignment Specification Document**, **Instructional Video**, and **Sample Executables** to understand the required implementation details.
- Use debugging tools like `gdb` to effectively identify and troubleshoot issues.
- Consult **man pages** (`man <command>`) for detailed information and usage of library functions.