System Software

EECS 111

Project #3: ZotBank (Banker's Algorithm)

Calla Chen

37301155

June 06, 2025

1. Objective

The goal of this project is to simulate a dynamic resource allocation system using **Banker's Algorithm**, ensuring safe management of concurrent customer requests across multiple resource types. Each customer specifies a maximum resource demand, and the system must determine whether granting a request will leave the system in a safe state. If not, the request is denied preventing potential deadlock conditions.

The system supports a fully interactive command-line interface, along with scripted test execution, live state updates, and robust logging. Development was performed using **C++98**, following a modular and test-driven approach. Testing was conducted across three platforms: **Windows** (CLion 2025.1.2 IDE), **Ubuntu 24.04.2** (via VirtualBox), and **UCI's EECS Linux Servers** (I used laguna.eecs.uci.edu) to ensure compatibility and correctness.

For the final version, both manual and automated test cases are included, with output validation via structured logs and CSV summaries. Several **bonus features** have been implemented to enhance the simulator including:

- Savepoints and rollback
- System state comparison
- Verbose and color modes
- Per-customer statistics
- Deadlock tracking and analysis
- Python-based session visualization

These features significantly extend the core functionality and enable deeper insight into system behavior.

1.1. Project Overview

This project simulates a classic concurrency problem: ensuring safe access to limited shared resources across multiple clients. It implements **Banker's Algorithm** to manage requests and releases while avoiding deadlock, allowing real-time interaction and reproducible automated testing.

The simulation supports:

- Detection and prevention of unsafe system states

- Enforcement of maximum declared resource demands per customer
- Interactive command-based control (RQ, RL, *, report, etc.)
- Scripted .txt test cases for automated validation
- Comprehensive logging of state transitions, outcomes, and command usage
- System-level metrics, including denial breakdowns, usage summaries, and visual heatmaps

Additionally, the simulator includes **bonus features** such as named savepoints, state comparisons, per-customer logs, verbose output, and Python-powered CSV analysis, all designed to enhance the observability and grading convenience.

1.2. System Module & File Breakdown

- Core Banking Module
  - Files: *banker.cpp* and *banker.h*
  - Role: Implements logic for the Banker's Algorithm, including checking need, available, allocation, and safety conditions. Handles requests, releases, and maintains internal state matrices.
- Command Handler
  - Files: *command_handler.cpp* and *command_handler.h*
  - Role: Parses user/test commands and routes them to appropriate modules. Manages the **test** mode file input and logs command outcomes.
- Logging System
  - Files: *logger.cpp, logger.h, log_global.cpp,* and *log_global.h*
  - Role: Unified logging system that records all terminal-visible messages, command results, and detailed system events to log files such as *full_session.txt* and *events.log*. Color-coded output improves terminal clarity.
- Validator Module
  - Files: *validator.cpp* and *validator.h*
  - Role: Verifies the format and validity of user inputs before processing, helping catch out-of-range values or malformed commands.
- Main Module
  - Files: *main.cpp*
  - Role: Initializes resources, loads *maximum.txt*, processes input (manual or test file), manages program state, and summarizes the session.
- Test Directory
  - Files: *test_basic.txt, test_safe_sequence.txt, test_unsafe.txt,* etc.
  - Role: Provides auto-test scenarios such as .txt files, allowing repeatable simulations, includes safe and unsafe requests, invalid input tests, and special cases.
- Analysis Script (bonus)
  - Files: *analysis.py*
  - Role: Bonus feature that parses CSV logs (*report.csv*, *per_customer_log.csv*) and generates visualizations of system metrics, including request activity and resource usage trends. Requires Python 3 with matplotlib and pandas.
- Resource Demand Matrix
  - Files: *maximum.txt*
  - Role: Defines the declared maximum resource claim for each customer across all four resource types. This file is loaded at the start of execution to initialize the system's internal matrices
    - NOTE: I used a custom *maximum.txt* rather than the default provided in the assignment to ensure compatibility with all my automated and manual tests.

This file supports full verification coverage across safe, unsafe, and denial scenarios.

- Build Automation
  - File: *Makefile*
  - Role: Automates compilation of the project into ./zotbank. Supports full clean/build flow with C++98 compatibility flags.
- Project Instructions
  - File: *README.md*
  - Role: Provide build/run instructions, project layout, command list, test automation guide, and documentation for all bonus features. Final version includes grader notes and test coverage summary

1.3. Execution Environment

- Language: C++98 (Required for the course)
- IDE Used: CLion 2025.1.2 on Windows
- Platforms tested:
  - Ubuntu 24.04.2 (via VirtualBox)
  - EECS Linus Server – laguna.eecs.uci.edu
- Compiler: g++ -std=c++98 -Wall -Westra
- Testing interfaces: Manual CLI input and automated test execution via .txt files in the tests/ directory
- Log outputs:
  - *full_session.txt*: Complete transcript of all commands and system responses
  - *events.txt*: Concise, human-readable event log for granted/denied requests and summaries
  - *history.txt*: Persistent history of all entered commands (manual or test file)
  - *per_customer_log.csv*: Detailed stats per customer
    - Wait time
    - Retries
    - Turnaround time
    - Etc.
  - *log_summary.csv*: Session-wide command usage breakdown (RQ, RL, unsafe, etc.)
  - *request_heatmap.csv*: Matrix of request/release frequencies by customer
  - *deadlock_log.csv*: Logs each deadlock event with affected customers and missing resources
  - *session_summary.txt*: Final summary printed upon exit, capturing totals and safe/unsafe stats
  - *customer_P*.txt*: Log file per customer capturing individual actions and outcomes

NOTE: At the end of each simulation, a separate *customer_PX.txt* file is generated for each active customer (e.g., *customer_P0.txt*, *customer_P1.txt*, etc.). These files contain per-customer request and release logs, including timestamps, granted/denied status, and turnaround time. The number of log files matches the number of customers defined in maximum.txt.

1.4. Block Diagram of System Design

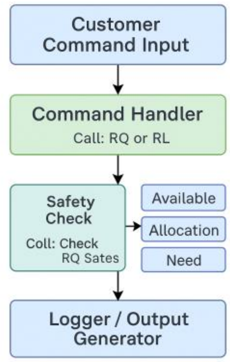


Fig. 1.4.1. Block diagram of ZotBank system architecture, showing flow of command input through the command handler, Banker's Algorithm logic, safety validation, and logging/output modules.

2. Procedure

This project implements the Banker's Algorithm for deadlock avoidance using a command-line simulation that supports file-based input, interactive commands, and detailed logging of system states and decision processes. All logic was implemented using C++98 and developed in a Linux-compatible environment, and extended with features such as savepoints, state comparison, verbose mode, and CSV-based performance summaries.

2.1. System Initialization and Input Handling

The simulator begins by reading initial configuration data, either from standard input or a provided file, including available resource quantities and the system's maximum claim matrix.

2.1.1. Input and Configuration – main.cpp, banker.cpp

- **Initial Input**: At startup, the system reads the total number of resources and customer count from command-line arguments. It then loads the available resource vector and customer maximum claims from the *maximum.txt* matrix file.
    - **Note**: A custom *maximum.txt* was used (not the default from the assignment) to support thorough testing across safe, unsafe, and edge-case scenarios
- **Banker Setup**: The Banker class encapsulates the core data structures:
    - *maximum[i][j]*: Max resources *i* may request
    - *allocation[i][j]*: Resources currently allocated
    - *need[i][j]*: Remaining resources required to fulfill the request
    - *available[i][j]*: Resources still available in the system

2.1.2. Logging Setup – log_global.cpp, log_global.h

- **Logging Streams**: The system logs both human-readable session output and structures logs via a shared logger interface. Outputs include:"
    - *logs/full_session.txt*: Full terminal output
    - *logs/events.txt*: Command result summary
    - *logs/report.csv*, *per_customer_log.csv*, *log_summary.csv*: CSV logs for system state, per-customer stats, and session summaries.
    - *logs/deadlock_log.csv*: Records of detected deadlocks and blocked customers
    - *logs/customer_P*.txt*: Per-customer generated at the end of the simulation
- **Highlight**: ANSI color-coded terminal output (green = granted, red = denied, yellow release) improves interactive feedback
- **New in final version**: Verbose mode logs add more detail for advanced debugging; session heatmap is recorded in *request_heatmap.csv*

2.2. Interactive Session and Command Handling

Once initialized, the program enters a loop where users may encounter commands directly or execute from tests/ directory.

2.2.1. Supported Commands – command_handler.cpp/h

- RQ i R0 R1 R2 R3: Request resources for customer *i*

- RL i R0 R1 R2 R3: Release resources for customer *i*
- *** (Asterisk) ***: Print full system state
- report: Generate a CSV summary of current allocations and remaining needs
- explain: Explain why a request was denied
- test <filepath>: Execute a batch of commands from a test file
- savepoint<name>: Save the current system state under a custom label
- undo: Revert the system to the last saved snapshot
- compare<name>: Compare current system state to a named savepoint
- diff<name>: Show matrix-level differences from a named savepoint
- preview i R0 R1R2 R3: Simulate a resource request without applying changes
- summary: Print a summary of all command usage statistics
- history: View command history from the current sessioion
- !N: Recall and re-execute the N-th command from history
- help [command]: Display help documentation (general or command-specific)
- color on/off: Toggle color terminal output
- verbose on/off: Toggle detailed session logging
- exit: End the session, print a final summary, and write all logs

All commands are parsed and validated by the CommandHandler module. Feedback is printed to the terminal and logged to *full_session.txt* and *events.txt*. Safety checks are performed by the Banker class to ensure that any request leaves the system in a safe state. Bonus commands such as savepoint, compare, and preview extend the core functionality with state tracking and rollback support.

2.3. Banker's Algorithm and Safety Checks

At the core of the simulator is the **Banker's Algorithm**, which determines whether a resource request can be granted without risking a deadlock. Each incoming RQ command initiates a safety evaluation before any actual allocation is committed.

2.3.1. Resource Request Flow – banker.cpp

For each resource request, the system performs the following safety check:

- **Simulates** the requested allocation
- **Validates** whether a safe sequence of process completions still exits
- **Rolls back** the temporary allocation if the system would enter an unsafe state

This logic is executed internally via the method:

- **Key Method**: isSafeState() – Returns true if a safe sequence exists after a hypothetical allocation, otherwise false

Additional contest is provided by the explain command, which outputs the reason for denial:

- Unsafe system state
- Exceeds available resources
- Exceeds declared need

The preview command also uses this logic, allowing users to test a request without modifying the system.

2.3.2. Release and Reset

- **Release**: Resources can be released via RL commands, which update the allocation, need, and available matrices accordingly
- **Reset**: A reset command (from test files) may also be implemented to reinitialize the system or reload *maximum.txt* and the initial state if supported.

2.4. Logging and Output

- *events.log*: Human-readable, sequential summary of all command results (granted, denied, released)
- *full_session.txt*: Complete terminal output including commands, matrix snapshots, and verbose diagnostics
- *report.csv*: System-wide summary of maximum, allocation, need, and available resources
- *per_customer_log.csv*: Metrics per customer: wait time retry count, turnaround time, etc.
- *log_summary.csv*: Total counts for RQ, RL, safe/unsafe/denied requests, exit status, etc.
- *deadlock_log.csv*: Timestamps and customer/resource info for each detected deadlock
- *request_heatmap.csv*: Matrix request and release frequencies for heatmap analysis
- *customer_P*.txt*: Per-customer logs written at simulation end with detailed action history
- *session_summary*.txt: Text summary printed after exit, also saved to logs

Color Output: Green messages for granted requests, yellow for releases, red for denials, cyan for summaries (optional enhancements via *logger.cpp,* toggleable with the color command)

2.5. Test Automation

The simulator supports processing of scripted commands via *.txt* files located in the tests/ directory. Tests cover a wide range of conditions including:

- Safe and unsafe resource request sequences
- Invalid or malformed inputs
- Requests exceeding available or declared need
- Command chaining (e.g., request -> savepoint -> undo -> compare)
- Denied request tracking and logging
- Resettable simulations for clean reruns
- Verification of savepoint behavior (snapshot, undo, compare, diff)
- Preview-only requests to simulate what-if scenarios

Each test logs command behavior, matrix state transitions, and relevant metrics to ensure correctness and repeatability.

3. Results

This section outlines the design and flow of ZotBank, emphasizing modular structure, control flow, and critical sections that govern safe resource management. Each part of the system plays a specific role in upholding Banker's Algorithm and ensuring correct interactions between customers and shared resources.

3.1. Safe Request Scenario

This test demonstrates a **safe resource request** from customer **P0**. The system verifies that granting this request will leave the system in a state and then updates the allocation and need matrices accordingly.

Before executing the request, the safety command enables to display the computed safe sequence. The request RQ 0 1 1 1 0 is with P0's declared maximum and does not exceed available resources. The system verifies that a complete sequence exists (P0 -> P1 -> P2 -> P3 -> P4) and grants the request

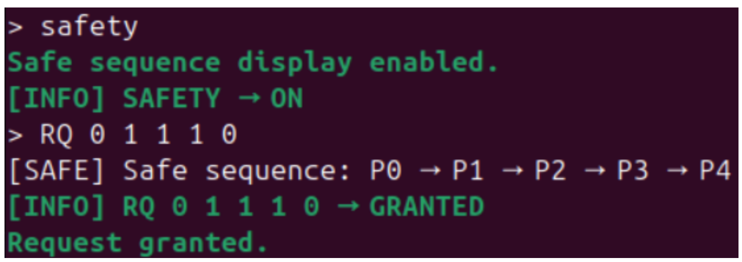


Fig. 3.1.1. Safe request from **P0** granted with verified sequence
(*captured from a manual run that mirrors the test_safe_sequence.txt case.)*

This confirms that the system enforces safety constraints before granting any request and logs the outcome clearly in both the terminal and log files.

3.2. Unsafe Request Scenario

This scenario demonstrates a resource request that is denied because it would place the system in an unsafe state and potentially cause a deadlock. Customer **P0** requests its full maximum need: RQ 0 3 2 2 2.

Although the system has enough available resources to satisfy this request, it would leave no safe sequence. The safety check determines that customer **P2** would be blocked and unable to complete, due to unavailable resources needed to fulfill its demand.

The system correctly

- Detects this as a **potential deadlock**
- Identifies the blocked customer and missing resources
- Logs the current holders of each source
- Automatically creates a savepoint (auto_P3_deadlock)
- Denies the request to preserve system safety

```
================================
ZotBank v1.0 - EECS 111 Project #3
================================
Type 'help' for command syntax.

> RQ 0 3 2 2 2
[DEADLOCK] No process can proceed – potential deadlock state.
Blocked customers: P2
  - P2 is blocked because it needs: R0(2) R1(3) R2(2) R3(2)

Resource holders:
  - R0 held by: P0(3)
  - R1 held by: P0(2)
  - R2 held by: P0(2)
  - R3 held by: P0(2)
[INFO] SAVEPOINT → Named savepoint "auto_P3_deadlock" created
[INFO] SAVEPOINT → Automatically saved as "auto_P3_deadlock" before deadlock exit
[WARN] Request denied: would lead to unsafe state.
[ERROR] RQ 0 3 2 2 2 → DENIED
Request denied.
```

Fig. 3.2.1. Unsafe request **P0** denied due to unsafe state and potential deadlock
(*captured from a manual run using reduced resource availability to trigger denial.)*

This interaction validates the critical logic inside Banker::isSafe() and CommandHandler::process(), and demonstrates how the simulator's denial reasoning and logging mechanisms operate in real time.

3.3. Invalid Input Handling

The simulator includes robust input validation through the Validator module to reject any resource request that are malformed or logically invalid before reaching the Banker's Algorithm.

In this test, two invalid commands are submitted:

1. RQ 0 4 2 2 2: This request exceeds the declared maximum for P0 (10 5 7 8) as is denied with an error indicating that the request would exceed need.
2. RQ 0 -1 0 0 0: This command includes a negative value and is immediately marked as invalid without altering the system state.

These checks are designed to:

- Prevent logical inconsistencies
- Preserve system integrity
- Ensure the user receives clear feedback about the issue

```
callachen@EECS111:~/shared_folder/project3$ ./zotbank maximum.txt 10 5 7 8
================================
ZotBank v1.0 - EECS 111 Project #3
================================
Type 'help' for command syntax.

> RQ 0 4 2 2 2
[WARN] Request denied: exceeds declared need.
[ERROR] RQ 0 4 2 2 2 → DENIED
Request denied.
> RQ 0 -1 0 0 0
Invalid request: bad customer ID or negative values.
[WARN] RQ 0 -1 0 0 0 → INVALID
```

Fig. 3.3.1. Invalid requests denied by the system validator
(*captured from a manual test session; included in test_invalid.txt.)*

This demonstrates that ZotBank correctly enforces input rules and gracefully handles improper commands.

3.4. Resource Releases and State Recovery

This scenario demonstrates the system's ability to recover resources after a customer releases them. Customer **P1** first requests resources using RQ 1 1 1 0 0, which the system grants after verifying safety. The customer then releases the same resources with RL 1 1 1 0 0.

Fig. 3.4.1 shows the successful grant and release sequence.

After the release the system updates:

- The **allocation matrix**, subtracting the released values from P1
- The **need matrix**, restoring P1's full demand
- The **available vector**, returning the resources back to the pool

As shown in *Fig. 3.4.2*, the system state is reset to its original configuration, confirming correct recovery and reusability of resources.

This behavior validates the logic inside Banker::release() and demonstrates how the simulator enables dynamic sharing of resources without violating safety constraints.

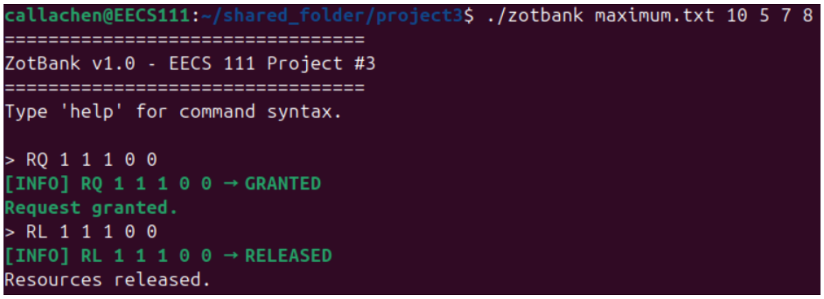


Fig. 3.4.1. Request and release for customer **P1**
(*captured from a manual test session.*)

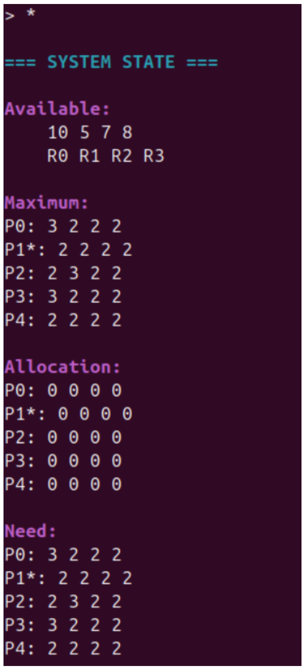


Fig. 3.4.2. Final system state showing successful resource recovery
(*Result of * command after release.*)

3.5. Bonus Feature Demonstrations

In addition to the required functionality, this project implements multiple bonus features that extend the capabilities of Banker's Algorithm simulator. These features improve usability, allow advanced state management, and enhance testing and analysis. All bonus features are fully integrated with logging and validation and have been verified via both manual input and automated test scripts.

The bonus features demonstrated below include:

- Named savepoints and comparison tools (savepoint, compare, diff)
- Simulation previews (preview) to test requests without committing
- Recovery tools (snapshot, undo) for manual rollback
- Command usage tracking and request heatmaps (summary, *request_heatmap.csv*)

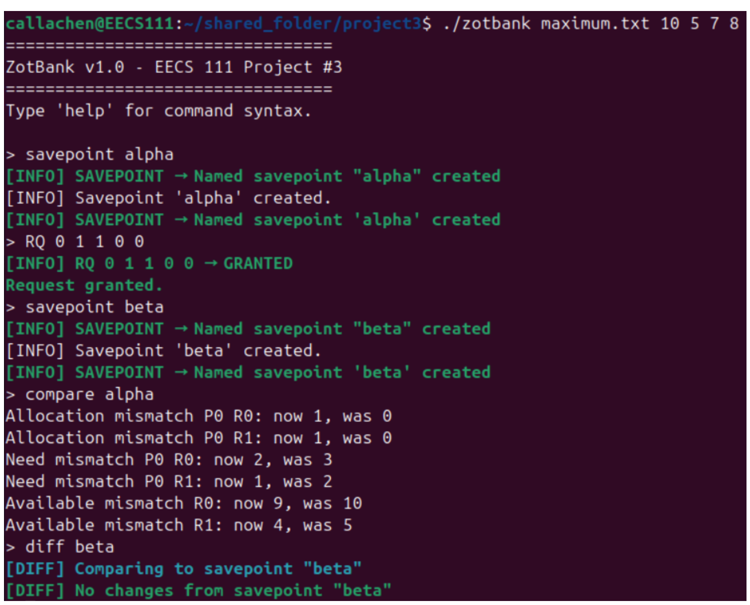
Each feature is implemented with CommandHandler, Banker, and Logger modules and is accessible through the interactive command-line interface.

3.5.1. compare/diff

The compare and diff feature allows users to analyze changes between different system states. Users may label and store snapshots of the system using the savepoint command, then later inspect how the system has evolved.

In this demonstration:

1. A savepoint named alpha is created before any request
2. P0 is granted a request RQ 0 1 1 0 0
3. A second savepoint beta is created immediately after
4. The command compare alpha outputs all matrix-level differences between the current state and alpha, identifying mismatches in **allocation**, **need**, and **available**.
5. The diff beta command confirms that the system has not changed since beta was saved.



```
callachen@EECS111:~/shared_folder/project3$ ./zotbank maximum.txt 10 5 7 8
================================
ZotBank v1.0 - EECS 111 Project #3
================================
Type 'help' for command syntax.

> savepoint alpha
[INFO] SAVEPOINT → Named savepoint "alpha" created
[INFO] Savepoint 'alpha' created.
[INFO] SAVEPOINT → Named savepoint 'alpha' created
> RQ 0 1 1 0 0
[INFO] RQ 0 1 1 0 0 → GRANTED
Request granted.
> savepoint beta
[INFO] SAVEPOINT → Named savepoint "beta" created
[INFO] Savepoint 'beta' created.
[INFO] SAVEPOINT → Named savepoint 'beta' created
> compare alpha
Allocation mismatch P0 R0: now 1, was 0
Allocation mismatch P0 R1: now 1, was 0
Need mismatch P0 R0: now 2, was 3
Need mismatch P0 R1: now 1, was 2
Available mismatch R0: now 9, was 10
Available mismatch R1: now 4, was 5
> diff beta
[DIFF] Comparing to savepoint "beta"
[DIFF] No changes from savepoint "beta"
```

Fig. 3.5.1.1. Savepoint comparison with compare alpha and diff beta

3.5.2. preview

The preview command allows users to simulate resources requests and observe whether the system would grant or deny them – without committing any changes to the state. This is especially useful for explanatory testing or debugging multi-step simulations.

In this demonstration:

- The command preview 1 1 1 0 0 is rejected due to an unsafe state, meaning that even though the request may be valid in format, it would result in no safe completion sequence.
- A second request, preview 0 3 3 3 3, is rejected outright as it exceeds the declared minimum need for customer P0.

Imporantly, no changes are made to the available, allocation, or need matrices in either case.

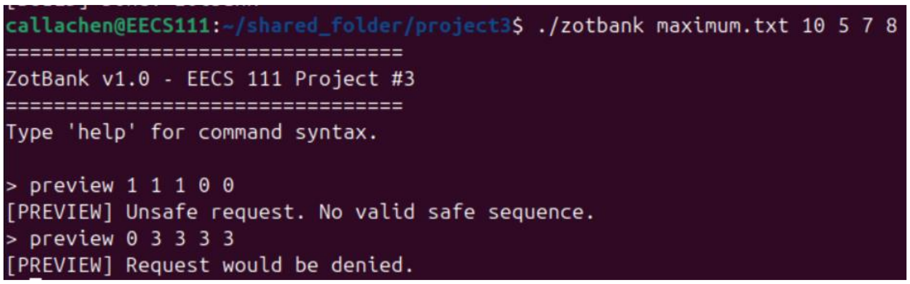


Fig. 3.5.2.1. Previewing unsafe and invalid requests without committing

This feature uses the same internal logic as Banker::isSafe() and Validator::isValidRequest() but performs these tests in non-mutating mode. It provides safer, more transparent way for users to understand how the system responds to their input.

3.5.3. savepoint and undo

The snapshot and undo commands provide users with manual rollback capability. Unlike named savepoints which are used for comparison (compare, diff), snapshot captures the entire system state and enables full restoration through undo.

This test demonstrates:

1. Creating a snapshot at the initial clean state
2. Issuing a valid resource request to customer P1
3. Using undo to restore the system to its previous snapshot
4. Verifying that all matrices (available, allocation, and need) return to their original values

```
> *

=== SYSTEM STATE ===

Available:
    10 5 7 8
    R0 R1 R2 R3

Maximum:
P0: 3 2 2 2
P1: 2 2 2 2
P2: 2 3 2 2
P3: 3 2 2 2
P4: 2 2 2 2

Allocation:
P0: 0 0 0 0
P1: 0 0 0 0
P2: 0 0 0 0
P3: 0 0 0 0
P4: 0 0 0 0

Need:
P0: 3 2 2 2
P1: 2 2 2 2
P2: 2 3 2 2
P3: 3 2 2 2
P4: 2 2 2 2
```

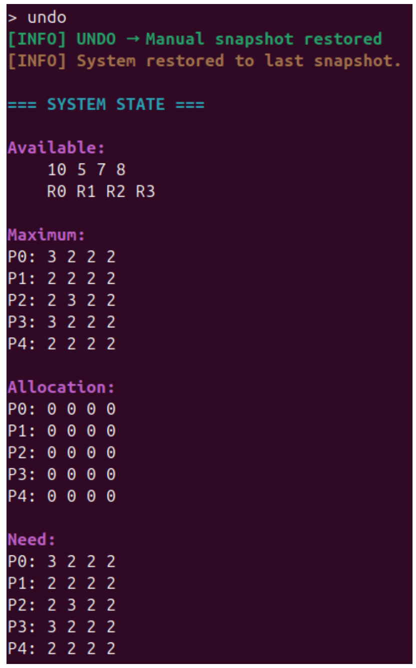
Fig. 3.5.3.1. Initial system state prior to snapshot creation
(Shows that all customers have no allocations and all resources are fully available)

```
> snapshot
[INFO] SNAPSHOT → Manual snapshot saved
[INFO] Manual snapshot saved.
> RQ 1 1 0 0 0
[INFO] RQ 1 1 0 0 0 → GRANTED
Request granted.
```

Fig. 3.5.3.2. Snapshot creation and successful resource request by P1
(P1 is granted R0 = 1; state is now modified)

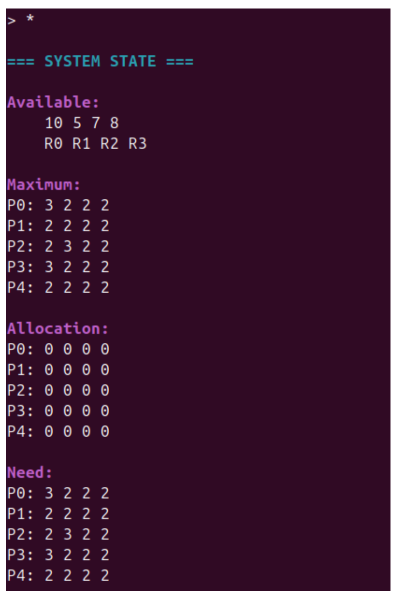
```
> *

=== SYSTEM STATE ===

Available:
    9 5 7 8
    R0 R1 R2 R3

Maximum:
P0: 3 2 2 2
P1*: 2 2 2 2
P2: 2 3 2 2
P3: 3 2 2 2
P4: 2 2 2 2

Allocation:
P0: 0 0 0 0
P1*: 1 0 0 0
P2: 0 0 0 0
P3: 0 0 0 0
P4: 0 0 0 0

Need:
P0: 3 2 2 2
P1*: 1 2 2 2
P2: 2 3 2 2
P3: 3 2 2 2
P4: 2 2 2 2
```

Fig. 3.5.3.3. System state after request is applied
(Allocation and need matrices are updated for P1; available reflects the deduction)



```
> undo
[INFO] UNDO → Manual snapshot restored
[INFO] System restored to last snapshot.

=== SYSTEM STATE ===

Available:
    10 5 7 8
    R0 R1 R2 R3

Maximum:
P0: 3 2 2 2
P1: 2 2 2 2
P2: 2 3 2 2
P3: 3 2 2 2
P4: 2 2 2 2

Allocation:
P0: 0 0 0 0
P1: 0 0 0 0
P2: 0 0 0 0
P3: 0 0 0 0
P4: 0 0 0 0

Need:
P0: 3 2 2 2
P1: 2 2 2 2
P2: 2 3 2 2
P3: 3 2 2 2
P4: 2 2 2 2
```

Fig. 3.5.3.4. undo restores the system to last snapshot
(Log confirms restoration and system state returns to snapshot baseline)



```
> *

=== SYSTEM STATE ===

Available:
    10 5 7 8
    R0 R1 R2 R3

Maximum:
P0: 3 2 2 2
P1: 2 2 2 2
P2: 2 3 2 2
P3: 3 2 2 2
P4: 2 2 2 2

Allocation:
P0: 0 0 0 0
P1: 0 0 0 0
P2: 0 0 0 0
P3: 0 0 0 0
P4: 0 0 0 0

Need:
P0: 3 2 2 2
P1: 2 2 2 2
P2: 2 3 2 2
P3: 3 2 2 2
P4: 2 2 2 2
```

Fig. 3.5.3.5. Final system state matches the initial snapshot
(All matrices confirm rollback: allocations cleared, availability reset, needs restored)

Together, the above figures confirm that the snapshot and undo mechanism functions as intended: allowing temporary changes to be reversed, with restoration of all resource tracking matrices and system state.

3.5.4. summary and heatmap logs

ZotBank logs internal statistics during the simulation and produces detailed breakdowns upon session completion. These logs assist in evaluating system behavior, usage frequency, and fairness across customers.

During execution, the summary command reports the frequency of each supported command type. Upon calling exit, the simulator writes structured logs to the logs/ directory, including:

- *request_heatmap.csv*: Per-customer RQ/RL counts
- *log_summary.csv*: System-wide command breakdown
- *session_summary.txt*: End-of-session metrics, including deadlocks and denial reasons

These files support visual analysis and allow external tools like the provided analysis.py to generate plots.

```
callachen@EECS111:~/shared_folder/project3$ ./zotbank maximum.txt 10 5 7 8
=================================
ZotBank v1.0 - EECS 111 Project #3
=================================
Type 'help' for command syntax.

> RQ 0 1 0 0 0
[INFO] RQ 0 1 0 0 0 → GRANTED
Request granted.
> RQ 1 0 1 0 0
[INFO] RQ 1 0 1 0 0 → GRANTED
Request granted.
> RL 0 1 0 0 0
[INFO] RL 0 1 0 0 0 → RELEASED
Resources released.
```

Fig. 3.5.4.1. Interactive session with granted request and release commands
(Customer P0 and P1 each issue a successful resource request. P0 then releases previously acquired resources)

```
> summary
Command Usage Breakdown:
  RQ:         2
  RL:         1
  safety:     0
  reset:      0
  report:     0
  explain:    0
  undo:       0
  snapshot:   0
  history:    0
  recap:      0
  help:       0
  verbose:    0
  color:      0
  savepoint:  0
  rollback:   0
  preview:    0
  compare:    0
  diff:       0
  unknown:    0
```

Fig. 3.5.4.2. Output of the summary command showing command usage counts
(The internal counter tracks each command type. This shows 2 requests and 1 release issued)

```
> exit
[INFO] EXIT → Session ended
[INFO] HEATMAP → logs/request_heatmap.csv written
[HEATMAP] Request/Release counts:
  P0 → RQ: 1, RL: 1
  P1 → RQ: 1, RL: 0
  P2 → RQ: 0, RL: 0
  P3 → RQ: 0, RL: 0
  P4 → RQ: 0, RL: 0

===== Session Summary =====
Total Requests:  2
Total Releases:  1
Denied Requests: 0  (RQ only)
Safe Requests:   2  (RQ only)
Unsafe Requests: 0  (RQ only)
 > Exceeds Need:    0
 > Exceeds Avail:   0
 > Unsafe State:    0
Deadlocks detected: 0
===========================
```

Fig. 3.5.4.1. Interactive session with granted request and release commands
(The report confirms total requests and releases, all granted safely. Heatmap logs are also written to logs/request_heatmap.csv for visualization)

3.6. Critical Implementation Sections

This section highlights the most important implementation areas within the ZotBank simulation that enforce safe execution and correctness of resource allocation and system state transitions. The following figures showcase the logic behind request validation, release handling, and safety checks – key components of Banker's Algorithm.

```cpp
    else if (cmd == "RQ") {
        // Handle resource request from a customer
        globalStats.countRQ++;              // Track RQ usage count
        globalStats.commandUsage["RQ"]++;   // Record usage in detailed command app
        res.isRequest = true;

        int cust, req[NUMBER_OF_RESOURCES]; // Array to store requested resources
        stringstream ss(trimmed.substr(3)); // Parse after "RQ"
        ss >> cust;
        for (int j = 0; j < NUMBER_OF_RESOURCES; ++j) ss >> req[j];

        // Track per-customer request count
        if (cust >= 0 && cust < NUMBER_OF_CUSTOMERS) {
            globalStats.requestCount[cust]++;
        }

        // Track first arrival time if this is the customer's first appearance
        if (customerArrivalTimes[cust] == -1)
            customerArrivalTimes[cust] = time(NULL);

        // Validate the request: valid customer and no negative values
        if (!Validator::isValidCustomer(cust) || !Validator::isValidRequest(req)) {
            string msg = "Invalid request: bad customer ID or negative values.\n";
            cout << msg;
            fullLog << msg;
            Logger::log("RQ " + trimmed.substr(3) + " → INVALID", Logger::WARN);

            if (verboseMode)
                fullLog << "[VERBOSE] RQ " << trimmed.substr(3) << "  → INVALID input\n";

            res.wasDenied = true;
            return res;
        }

        // Attempt to grant the request using Banker's Algorithm
        bool granted = (banker.request(cust, req) == Banker::GRANTED);

```

Fig. 3.6.1 CommandHandler::process() – RQ Command handling and Logging (command_handler.cpp)

```cpp
        // Verbose logging output
        if (verboseMode) {
            fullLog << "[VERBOSE] RQ " << cust << " ";
            for (int i = 0; i < NUMBER_OF_RESOURCES; ++i)
                fullLog << req[i] << " ";
            fullLog << "→ " << (granted ? "GRANTED" : "DENIED") << "\n";
        }

        // Record whether the request was granted or denied
        int result = granted ? Banker::GRANTED : -1;
        res.wasDenied = (result != Banker::GRANTED);
        res.status = (result == Banker::GRANTED) ? CONTINUE : DENIED;

        // If denied, then count retry and classify the reason
        if (res.wasDenied)
            customerRetryCounts[cust]++;

        if (res.wasDenied) {
            string reason = banker.getLastDenialReason();
            if (reason.find("need") != string::npos) res.exceedsNeed = true;
            else if (reason.find("available") != string::npos) res.exceedsAvail = true;
            else if (reason.find("unsafe") != string::npos) res.wasUnsafe = true;
        }

        // Print the final outcome and log to console/log files
        string statusStr = (result == Banker::GRANTED) ? "GRANTED" : "DENIED";
        string outputMsg = "Request " + string(result == Banker::GRANTED ? "granted.\n" : "(

        Logger::log("RQ " + trimmed.substr(3) + " → " + statusStr,
                result == Banker::GRANTED ? Logger::INFO : Logger::ERROR);

        cout << (result == Banker::GRANTED ? COLOR_GREEN : COLOR_RED)
             << outputMsg << COLOR_RESET;
        fullLog << outputMsg;
```

Fig. 3.6.2 CommandHandler::process() – RQ (cont.) (command_handler.cpp)

```cpp
        // Log to per-customer CSV if open
        if (cust >= 0 && cust < 10 && customerLogs[cust].is_open()) {
            customerLogs[cust] << "[" << currentTimestamp() << "] ";
            customerLogs[cust] << trimmed << " → " << statusStr << "\n";
            customerLogs[cust].flush();
        }
```

Fig. 3.6.3 CommandHandler::process() – RQ (cont.) (command_handler.cpp)

This section processes resource requests, validates input, invokes the Banker for approval and logs the results accordingly.

```cpp
int Banker::request(int customerNum, int request[]) {
    // Step 1: Check if request exceeds customer's declared need
    for (int j = 0; j < NUMBER_OF_RESOURCES; ++j) {
        if (request[j] > need[customerNum][j]) {
            lastDenialReason = "Request denied: exceeds declared need.";
            Logger::log(lastDenialReason, Logger::WARN);
            return DENIED_NEED; // Equivalent to error conditions in ZyBook Section 8.6 Step 1
        }
    }

    // Step 2: Check if request exceeds currently available resources
    for (int j = 0; j < NUMBER_OF_RESOURCES; ++j) {
        if (request[j] > available[j]) {
            lastDenialReason = "Request denied: exceeds available resources.";
            Logger::log(lastDenialReason, Logger::WARN);
            return DENIED_AVAIL; // Equivalent to must wait in Zybook Section 8.6 Step 2
        }
    }

    // Step 3: Tentatively allocate resources
    snapshot(); // Save current state in case we need to roll back

    // [CRITICAL SECTION START] Tentative allocation for safety check
    for (int j = 0; j < NUMBER_OF_RESOURCES; ++j) {
        available[j] -= request[j];                      // Available -= Request
        allocation[customerNum][j] += request[j];        // Allocation += Request
        need[customerNum][j] -= request[j];             // Need -= Request
    }
    // [CRITICAL SECTION END] Tentative allocation
```

Fig. 3.6.4 Banker::request() – Safe validation logic (banker.cpp)

This segment of code verifies resource safety by:

1. Ensures that the request does not exceed the customer's declared maximum.
2. Checking that the system has enough available resources.
3. Marking the tentative allocation region (highlighted by comments)

The actual allocation is done directly to the system's state (available, allocation, need) but it is protected by a prior snapshot() in case the system must roll back. The [CRITICAL SECTION] markers are internal comments denoting where this tentative allocation occurs, not isolation primitives themselves. This routine is central part of the Banker's Algorithm, ensuring is preserved before committing to a request.

```
else if (cmd == "RL") {
    // Handles resource request from customer
    globalStats.countRL++;              // Track global RL usage count
    globalStats.commandUsage["RL"]++;   // Track usage in detailed command map
    res.isRelease = true;

    int cust, rel[NUMBER_OF_RESOURCES]; // rel[] holds release amounts for each resource
    stringstream ss(trimmed.substr(3)); // Prase after "RL "
    ss >> cust;
    for (int j = 0; j < NUMBER_OF_RESOURCES; ++j) ss >> rel[j];

    // Track now how many times this customer has released resources
    if (cust >= 0 && cust < NUMBER_OF_CUSTOMERS) {
        globalStats.releaseCount[cust]++;
    }

    // Validate customer and release vector using current allocation
    const int (*alloc)[NUMBER_OF_RESOURCES] = banker.getAllocation();
    bool valid = Validator::isValidCustomer(cust) &&
         Validator::isValidRelease(rel, alloc, cust);

    if (verboseMode) {
        // verbose log for release command
        fullLog << "[VERBOSE] RL " << cust << " ";
        for (int i = 0; i < NUMBER_OF_RESOURCES; ++i)
            fullLog << rel[i] << " ";
        fullLog << "→ " << (valid ? "RELEASED" : "INVALID") << "\n";
    }

    // Showing error and log if release is invalid
    if (!Validator::isValidCustomer(cust) || !Validator::isValidRelease(rel, alloc, cust)) {
        string msg = "Invalid release: too much released or bad customer ID.\n";
        cout << msg;
        fullLog << msg;
        Logger::log("RL " + trimmed.substr(3) + " → INVALID", Logger::WARN);
        return res;
    }

    // Perform the release
    banker.release(cust, rel);
    Logger::log("RL " + trimmed.substr(3) + " → RELEASED", Logger::INFO);
    globalStats.totalReleases++;
```

Fig. 3.6.5 CommandHandler::process() – RL command: Validation and Release (command_handler.cpp)

This code segment parses the customer ID and release vector, using the current allocation state, and then processed to release the resources if valid. It logs success or failure, increments totalReleases, and updates per-customer release counters. This section ensures no customer can release more than allocated.

```cpp
bool Banker::isSafe() {
    int work[NUMBER_OF_RESOURCES];
    bool finish[NUMBER_OF_CUSTOMERS] = { false };
    int safeSequence[NUMBER_OF_CUSTOMERS];
    int idx = 0;

    // Step 1: Initialize Work = Available
    for (int j = 0; j < NUMBER_OF_RESOURCES; ++j)
        work[j] = available[j];

    bool progress = true;
    while (progress) {
        progress = false;

        // Step 2. Find an unfinished customer i such that Need[i] <= Work
        for (int i = 0; i < NUMBER_OF_CUSTOMERS; ++i) {
            if (!finish[i]) {
                bool canFinish = true;
                for (int j = 0; j < NUMBER_OF_RESOURCES; ++j) {
                    if (need[i][j] > work[j]) {
                        canFinish = false;
                        break;
                    }
                }

                // Step 3. If found, simulate completion: Work += Allocation[i]
                if (canFinish) {
                    for (int j = 0; j < NUMBER_OF_RESOURCES; ++j)
                        work[j] += allocation[i][j];
                    finish[i] = true;
                    safeSequence[idx++] = i;
                    progress = true;
                }
            }
        }
    }

    // Step 4: Check if all customers are finished
    bool deadlockDetected = false;
    for (int i = 0; i < NUMBER_OF_CUSTOMERS; ++i) {
        if (!finish[i]) {
            deadlockDetected = true;
```

Fig. 3.6.6 Banker::isSafe() – Initializing and Safe Sequence Evaluation & Deadlock Detection (banker.cpp)

```cpp

    // Step 4: Check if all customers are finished
    bool deadlockDetected = false;
    for (int i = 0; i < NUMBER_OF_CUSTOMERS; ++i) {
        if (!finish[i]) {
            deadlockDetected = true;
            break;
        }
    }

    if (deadlockDetected) {
        globalStats.countDeadlock++;

        cout << COLOR_RED << "[DEADLOCK] No process can proceed — potential deadlock state.\n";
        cout << "Blocked customers: ";
        fullLog << "[DEADLOCK] No process can proceed — potential deadlock state.\n";
        fullLog << "Blocked customers: ";
        for (int i = 0; i < NUMBER_OF_CUSTOMERS; ++i) {
            if (!finish[i]) {
                cout << "P" << i << " ";
                fullLog << "P" << i << " ";
            }
        }
        cout << "\n";
        fullLog << "\n";

        // Detailed Explanation: Why each process is blocked
        for (int i = 0; i < NUMBER_OF_CUSTOMERS; ++i) {
            if (!finish[i]) {
                cout << COLOR_RED << "  - P" << i << " is blocked because it needs: ";
                fullLog << "  - P" << i << " is blocked because it needs: ";
                for (int j = 0; j < NUMBER_OF_RESOURCES; ++j) {
                    if (need[i][j] > available[j]) {
                        cout << COLOR_YELLOW << "R" << j << "(" << need[i][j] << ") " << COLOR_RED;
                        fullLog << "R" << j << "(" << need[i][j] << ") ";
                    }
                }
                cout << "\n";
                fullLog << "\n";
            }
        }
```

Fig. 3.6.7 Banker::isSafe() cont (banker.cpp)

```cpp
        // Shows who is holding each critical resource
        cout << "\nResource holders:\n";
        fullLog << "\nResource holders:\n";
        for (int j = 0; j < NUMBER_OF_RESOURCES; ++j) {
            if (available[j] == 0) {
                cout << "  - R" << j << " held by: ";
                fullLog << "  - R" << j << " held by: ";
                for (int i = 0; i < NUMBER_OF_CUSTOMERS; ++i) {
                    if (allocation[i][j] > 0) {
                        cout << "P" << i << "(" << allocation[i][j] << ") ";
                        fullLog << "P" << i << "(" << allocation[i][j] << ") ";
                    }
                }
                cout << "\n";
                fullLog << "\n";
            }
        }
```

Fig. 3.6.8 Banker::isSafe() cont (banker.cpp)

The isSafe() function verifies system safety by first simulating resource availability and checking if all customers can complete, constructing a safe sequence if possible. If no such sequence exists, it identifies blocked customers, explains which resources are insufficient, and prints out which customers are holding them. This critical section concludes by logging the deadlock event to *deadlock_log.csv* and automatically creating a recovery savepoint called auto_P3_deadlock.

```cpp
void Banker::release(int customerNum, int release[]) {
    // [CRITICAL SECTION START] Releasing resources back to system
    for (int j = 0; j < NUMBER_OF_RESOURCES; ++j) {
        allocation[customerNum][j] -= release[j];
        available[j] += release[j];
        need[customerNum][j] += release[j];
    }
    // [CRITICAL SECTION END] Release complete
}
```

Fig. 3.6.9 Banker::release()

This critical section updates internal state matrices to return released resources to the system and adjust the customer's allocations and remaining need accordingly.

4. Submission Details.

The project submission .zip folder contains the following components:

- **Source code files**
  All *.cpp* and *.h* files implementing the ZotBank simulator, including Banker's Algorithm logic, resource request/release handling, safety checking, command pasring, logging utilities, and helper modules. All code is compliant with C++98 and designed to run on UCI's EECS Linux servers
- **Makefile**
  A working *Makefile* is included for compiling the simulator. The main executable ./zotbank can be built by running make, and invoked using

  ```
  ./zotbank maximum.txt 10 5 7 8
  ```

- **Automated tests**
  A set of 20 *.txt* files under the tests/ directory are included for auto-testing core behavior (e.g., safe/unsafe requests, invalid inputs, summary commands, etc)
- **PDF Report**
  This document, serving as the full design and implementation write-up, complete with test cases, and critical section screenshots
- **README.md**
  A markdown file with setup, usage instructions, and descriptions of the bonus features