Banker’s Algorithm for Deadlock Avoidance

Author: Harsh Kumar
Course: Operating Systems Lab
Submission: Lab 8 — Banker’s Algorithm
Date:

$Enterdatehere$

Abstract

This report presents an implementation of the Banker’s Algorithm for deadlock avoidance in an operating system. The program, written in C, computes the `Need` matrix from given `Max` and `Allocation` matrices, checks whether the current state of the system is safe using the Safety Algorithm, and processes additional resource requests using the Resource-Request Algorithm. The source code is heavily commented for learning purposes and the report contains step-by-step instructions to compile and execute the program.

Keywords

Banker’s Algorithm, Deadlock avoidance, Safety Algorithm, Resource-Request Algorithm, C programming

---

1. Introduction

Deadlocks occur in multi-process systems when a set of processes become permanently blocked, each waiting for resources held by the others. The Banker’s Algorithm is a classic method for avoiding deadlocks by ensuring that resource allocation decisions never leave the system in an unsafe state. The algorithm was proposed by Dijkstra and is particularly useful in systems where processes declare their maximum resource needs in advance.

---

2. Problem Statement

Given: - `n` processes: P0, P1, ..., P(n-1) - `m` resource types: R0, R1, ..., R(m-1) - `Available` vector: number of available instances of each resource type - `Max` matrix: maximum demand of each process for each resource type - `Allocation` matrix: current allocation of resources to each process

Compute: - `Need[i][j] = Max[i][j] - Allocation[i][j]` for all processes and resource types - Determine whether the system is in a safe state using the Safety Algorithm - Process resource requests using the Resource-Request Algorithm and decide whether they can be granted safely

---

3. Algorithm Overview

3.1 Need Matrix

For each process `i` and resource `j`:

```
Need[i][j] = Max[i][j] - Allocation[i][j]
```

3.2 Safety Algorithm (high-level)

1. Let `Work` = `Available` (copy the available vector).
2. Let `Finish[i] = false` for all processes.
3. Find a process `i` such that `Finish[i] == false` and `Need[i] <= Work` (component-wise). If found:
4. `Work = Work + Allocation[i]` (release the resources when process completes)
5. `Finish[i] = true` and add `i` to safe sequence.
6. Repeat step 3 until all processes finish or no such process is found.
7. If all `Finish[i] == true`, the system is in a safe state and the safe sequence is valid. Otherwise the system is unsafe.

3.3 Resource-Request Algorithm (high-level)

When process `p` requests resources `Request[]`: 1. If `Request[j] > Need[p][j]` for any `j`, error — process has exceeded its declared maximum. 2. If `Request[j] > Available[j]` for any `j`, the request cannot be granted immediately (not enough resources). 3. Otherwise, tentatively allocate: - `Available = Available - Request` - `Allocation[p] = Allocation[p] + Request` - `Need[p] = Need[p] - Request` and run the Safety Algorithm. If the system remains safe, grant the request. Otherwise rollback and deny.

---

4. Implementation (C Language)

> The program below is written in standard C. All parts are explained with detailed comments intended for beginners. Read the comments carefully to understand the control flow and data structures.

```c
/*
 * Banker's Algorithm Implementation in C
 * Author: Harsh Kumar
 * Purpose: Educational implementation with detailed comments for beginners
 * Note: The program uses fixed maximum sizes for simplicity. These can be
 *       adjusted as needed.
 */

#include <stdio.h>
#include <stdlib.h>

/* Maximum limits; change these if you need larger systems. */
```

```c
#define MAX_PROCESSES 10
#define MAX_RESOURCES 10

/* Global variables to store system state */
int n; /* number of processes */
int m; /* number of resource types */
int Available[MAX_RESOURCES];               /* Available[j] = # of available
instances of resource j */
int Max[MAX_PROCESSES][MAX_RESOURCES];      /* Max[i][j] = maximum demand of
process i for resource j */
int Allocation[MAX_PROCESSES]
[MAX_RESOURCES]; /* Allocation[i][j] = currently allocated resources to
process i */
int Need[MAX_PROCESSES][MAX_RESOURCES];     /* Need[i][j] = Max[i][j] -
Allocation[i][j] */

/* Function: calculateNeed
 * Purpose: Compute the Need matrix from Max and Allocation
 */
void calculateNeed() {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            Need[i][j] = Max[i][j] - Allocation[i][j];
            /* For safety, ensure Need is not negative. If it is, input is
invalid. */
            if (Need[i][j] < 0) {
                fprintf(stderr, "Invalid input: Allocation exceeds Max for
process %d resource %d\n", i, j);
                exit(EXIT_FAILURE);
            }
        }
    }
}

/* Function: isSafe
 * Purpose: Check whether the current system state is safe.
 * If safe, it fills safeSeq[] with an ordering of processes that forms the
safe sequence.
 * Returns: 1 if safe, 0 otherwise
 */
int isSafe(int safeSeq[]) {
    int Work[MAX_RESOURCES];
    int Finish[MAX_PROCESSES];

    /* Initialize Work = Available and Finish[] = false */
    for (int j = 0; j < m; j++)
        Work[j] = Available[j];
    for (int i = 0; i < n; i++)
        Finish[i] = 0; /* 0 means false */

    int count = 0; /* number of processes added to safe sequence */
```

```
    /* Repeat until we either find a safe sequence for all processes or
cannot proceed */
    while (count < n) {
        int found = 0; /* indicates if a suitable process was found in this
iteration */

        for (int p = 0; p < n; p++) {
            if (!Finish[p]) {
                /* Check if Need[p] <= Work component-wise */
                int canProceed = 1;
                for (int j = 0; j < m; j++) {
                    if (Need[p][j] > Work[j]) {
                        canProceed =
0; /* cannot satisfy process p right now */
                        break;
                    }
                }

                /* If process p can proceed, simulate its completion */
                if (canProceed) {
                    for (int j = 0; j < m; j++)
                        Work[j] += Allocation[p][j]; /* release resources */

                    safeSeq[count++] = p; /* add this process to safe
sequence */
                    Finish[p] = 1; /* mark as finished */
                    found = 1;
                }
            }
        }

        /* If no unfinished process could be satisfied in this pass, system
is unsafe */
        if (!found)
            return 0; /* unsafe */
    }

    return 1; /* safe */
}

/* Function: requestResources
 * Purpose: Attempt to grant resources requested by process p according to
Banker's rules
 * Returns: 1 if request granted, 0 if denied
 */
int requestResources(int p, int Request[]) {
    /* Step 1: Check Request <= Need */
    for (int j = 0; j < m; j++) {
        if (Request[j] > Need[p][j]) {
            printf("Error: Process P%d has exceeded its maximum claim for
```

```
resource %d.\n", p, j);
            return 0; /* invalid request */
        }
    }

    /* Step 2: Check Request <= Available */
    for (int j = 0; j < m; j++) {
        if (Request[j] > Available[j]) {
            printf("Request by P%d cannot be granted immediately; not enough
available resources.\n", p);
            return 0; /* not enough resources right now */
        }
    }

    /* Step 3: Pretend to allocate and check safety */
    for (int j = 0; j < m; j++) {
        Available[j] -= Request[j];
        Allocation[p][j] += Request[j];
        Need[p][j] -= Request[j];
    }

    int safeSeq[MAX_PROCESSES];
    int safe = isSafe(safeSeq);

    if (safe) {
        printf("Request can be safely granted.\n");
        printf("New Safe Sequence: ");
        for (int i = 0; i < n; i++)
            printf("P%d ", safeSeq[i]);
        printf("\n");
        return 1; /* request granted */
    }

    /* Step 4: Rollback since not safe */
    for (int j = 0; j < m; j++) {
        Available[j] += Request[j];
        Allocation[p][j] -= Request[j];
        Need[p][j] += Request[j];
    }

    printf("Request denied. Granting it would leave the system in an unsafe
state.\n");
    return 0; /* request denied */
}

/* Function: printMatrix
 * Purpose: Helper to print matrices (Allocation, Max, Need) in a readable
format
 */
void printMatrix(const char *title, int mat[MAX_PROCESSES][MAX_RESOURCES]) {
    printf("%s\n", title);
```

```
    for (int i = 0; i < n; i++) {
        printf("P%d: ", i);
        for (int j = 0; j < m; j++)
            printf("%d ", mat[i][j]);
        printf("\n");
    }
}

int main() {
    /* Input: number of processes and resource types */
    printf("Enter number of processes (n): ");
    if (scanf("%d", &n) != 1) {
        fprintf(stderr, "Invalid input for number of processes.\n");
        return EXIT_FAILURE;
    }
    if (n <= 0 || n > MAX_PROCESSES) {
        fprintf(stderr, "Number of processes must be between 1 and %d.\n",
MAX_PROCESSES);
        return EXIT_FAILURE;
    }

    printf("Enter number of resource types (m): ");
    if (scanf("%d", &m) != 1) {
        fprintf(stderr, "Invalid input for number of resources.\n");
        return EXIT_FAILURE;
    }
    if (m <= 0 || m > MAX_RESOURCES) {
        fprintf(stderr, "Number of resource types must be between 1 and %d.
\n", MAX_RESOURCES);
        return EXIT_FAILURE;
    }

    /* Input: Available vector */
    printf("Enter Available vector (space-separated %d values):\n", m);
    for (int j = 0; j < m; j++) {
        if (scanf("%d", &Available[j]) != 1) {
            fprintf(stderr, "Invalid input for Available[%d].\n", j);
            return EXIT_FAILURE;
        }
        if (Available[j] < 0) {
            fprintf(stderr, "Available resources cannot be negative.\n");
            return EXIT_FAILURE;
        }
    }

    /* Input: Max matrix */
    printf("Enter Max matrix (n rows each with m values):\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (scanf("%d", &Max[i][j]) != 1) {
                fprintf(stderr, "Invalid input for Max[%d][%d].\n", i, j);
```

```
                return EXIT_FAILURE;
            }
            if (Max[i][j] < 0) {
                fprintf(stderr, "Max values cannot be negative.\n");
                return EXIT_FAILURE;
            }
        }
    }

    /* Input: Allocation matrix */
    printf("Enter Allocation matrix (n rows each with m values):\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (scanf("%d", &Allocation[i][j]) != 1) {
                fprintf(stderr, "Invalid input for Allocation[%d][%d].\n",
i, j);
                return EXIT_FAILURE;
            }
            if (Allocation[i][j] < 0) {
                fprintf(stderr, "Allocation values cannot be negative.\n");
                return EXIT_FAILURE;
            }
            if (Allocation[i][j] > Max[i][j]) {
                fprintf(stderr, "Invalid input: Allocation[%d][%d] > Max[%d]
[%d].\n", i, j, i, j);
                return EXIT_FAILURE;
            }
        }
    }

    /* Calculate Need matrix */
    calculateNeed();

    /* Print Need matrix */
    printf("\nNeed Matrix:\n");
    for (int i = 0; i < n; i++) {
        printf("P%d: ", i);
        for (int j = 0; j < m; j++)
            printf("%d ", Need[i][j]);
        printf("\n");
    }

    /* Check whether system is currently safe */
    int safeSeq[MAX_PROCESSES];
    if (isSafe(safeSeq)) {
        printf("\nSystem is in a SAFE STATE.\nSafe Sequence: ");
        for (int i = 0; i < n; i++)
            printf("P%d ", safeSeq[i]);
        printf("\n");
    } else {
        printf("\nSystem is NOT in a safe state.\n");
```

```
        /* Even if unsafe, the program allows trying a request, but typically
no safe allocation exists now. */
    }

    /* Ask user whether they want to make a resource request */
    char choice;
    printf("\nDo you want to make a resource request? (y/n): ");
    scanf(" %c", &choice); /* space before %c skips whitespace/newline */

    if (choice == 'y' || choice == 'Y') {
        int p;
        printf("Enter process number (0 to %d) making the request: ", n - 1);
        if (scanf("%d", &p) != 1 || p < 0 || p >= n) {
            fprintf(stderr, "Invalid process number.\n");
            return EXIT_FAILURE;
        }

        int Request[MAX_RESOURCES];
        printf("Enter request vector for P%d (m values):\n", p);
        for (int j = 0; j < m; j++) {
            if (scanf("%d", &Request[j]) != 1) {
                fprintf(stderr, "Invalid input for Request[%d].\n", j);
                return EXIT_FAILURE;
            }
            if (Request[j] < 0) {
                fprintf(stderr, "Request values cannot be negative.\n");
                return EXIT_FAILURE;
            }
        }

        /* Try to grant the request */
        int granted = requestResources(p, Request);

        /* If granted, print updated Allocation, Need and Available */
        if (granted) {
            printf("\nUpdated Allocation Matrix:\n");
            printMatrix("Allocation:", Allocation);
            printf("\nUpdated Need Matrix:\n");
            printMatrix("Need:", Need);
            printf("\nUpdated Available Vector:\n");
            for (int j = 0; j < m; j++)
                printf("%d ", Available[j]);
            printf("\n");
        }
    }

    printf("\nProgram finished.\n");
    return 0;
}
```

5. How to Compile and Run (Step-by-step)

The following steps assume you are using a Unix-like environment (Linux or macOS) or Windows with a C compiler such as `gcc` (MinGW, TDM-GCC, or WSL). The commands shown use `gcc`.

1. **Save the source code**
2. Create a new text file named `banker.c` and paste the entire C source code above into it.
3. **Open a terminal / command prompt**
4. Linux / macOS: Open Terminal.
5. Windows: Open Command Prompt (cmd) or PowerShell, or use Git Bash or WSL.
6. **Compile the program**
7. Run:

```
gcc -o banker banker.c
```

8. This tells `gcc` to compile `banker.c` and create an executable named `banker` (or `banker.exe` on Windows).
9. If you get warnings, read them and fix inputs if necessary; warnings do not always prevent an executable.
10. **Run the program**
11. On Linux / macOS:

```
./banker
```

12. On Windows (Command Prompt / PowerShell):

```
banker.exe
```

13. **Provide input when prompted**
14. The program will ask for:
    - Number of processes `n` and number of resources `m`.
    - `Available` vector (m integers).
    - `Max` matrix (n lines with m integers each).
    - `Allocation` matrix (n lines with m integers each).
15. Optionally, you may choose to make a resource request for a specific process by entering `y` when prompted, then providing the `Request` vector.

16. **Example run (sample inputs)**

Sample input (one-line comments show what the user should type):

```
Enter number of processes (n): 5
Enter number of resource types (m): 3
Enter Available vector (space-separated 3 values):
3 3 2
Enter Max matrix (n rows each with m values):
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter Allocation matrix (n rows each with m values):
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
```

After these inputs the program will compute and display the `Need` matrix, check safety, and print a safe sequence if one exists.

1. **Example resource request**
2. If you choose to make a request, example:

   ```
   Do you want to make a resource request? (y/n): y
   Enter process number (0 to 4) making the request: 1
   Enter request vector for P1 (m values):
   1 0 2
   ```

3. The program will check the request and either grant it and display the updated safe sequence or deny it.

---

6. Sample Output (from the example inputs above)

> The exact output formatting may vary slightly depending on terminal and compiler, but it will follow the structure below.

```
Need Matrix:
P0: 7 4 3
P1: 1 2 2
P2: 6 0 0
P3: 0 1 1
P4: 4 3 1
```

```
System is in a SAFE STATE.
Safe Sequence: P1 P3 P4 P0 P2

Do you want to make a resource request? (y/n): y
Enter process number (0 to 4) making the request: 1
Enter request vector for P1 (m values):
1 0 2
Request can be safely granted.
New Safe Sequence: P1 P3 P4 P0 P2

Updated Allocation Matrix:
Allocation:
P0: 0 1 0
P1: 3 0 2
P2: 3 0 2
P3: 2 1 1
P4: 0 0 2

Updated Need Matrix:
Need:
P0: 7 4 3
P1: 0 2 0
P2: 6 0 0
P3: 0 1 1
P4: 4 3 1

Updated Available Vector:
2 3 0

Program finished.
```

---

7. Notes and Recommendations

- Carefully validate input data: `Allocation[i][j]` must not exceed `Max[i][j]` and all inputs should be non-negative.
- The program uses fixed maximum sizes (`MAX_PROCESSES` and `MAX_RESOURCES`) for simplicity; increase these values if you need to test larger systems.
- For large inputs consider redirecting input from a file to avoid typing large matrices manually.

---

8. Conclusion

This report provides a beginner-friendly, well-commented C implementation of the Banker's Algorithm, explains the theory, and demonstrates how to compile and run the program. The program can be used as a learning tool in Operating Systems labs and extended for larger or more complex test cases.

References

1. A. Tanenbaum, *Modern Operating Systems*, 4th ed. (Chapter on Deadlocks).
2. A. Silberschatz, P. Galvin, G. Gagne, *Operating System Concepts* (Chapter on Deadlocks).

End of report.