THOMPSON RIVERS UNIVERSITY

COMP 3410
Operating Systems

Submitted by

Name: Surkamal Singh Jhand

Student Number: T00602466

Section: 01

Assignment 2: Banker's Algorithm

Introduction: Operating systems employ The Banker's Algorithm, a robust deadlock avoidance approach, to manage resource distribution among several processes. Its name refers to how it resembles how a banker would distribute funds to clients. Before assigning resources, the algorithm looks for the possibility of a safe sequence of processes, preventing deadlocks and guaranteeing the system is always in a secure state [1].

Background:

Edsger Dijkstra developed the Banker's Algorithm in 1965 to address the deadlock issue in multi-process systems. First, the method compares the current resource allocation to the maximum resources each process may need. It then determines if a safe sequence occurs following the budget to see if providing the requesting process extra resources is secure. Finally, resource allocation ensures the system stays safe if a safe sequence is present [2].

Code Description and Code References:

```
# Name: Surkamal Singh Jhand
# Course: COMP 3410 Operating Systems
```

```
# Date: 2023-03-31
# Program Description: The Banker's Algorithm is a resource allocation and
#                      deadlock avoidance algorithm that tests for
#                      the safety of resource allocation to processes in a
#                      system. This program is a graphical user
#                      interface (GUI) application that allows users to
#                      input the number of processes, the number
#                      of resources, the available resources, the maximum
#                      resource allocation matrix, and the current
#                      resource allocation matrix. By clicking the "Check
#                      Safety" button, the program will determine if
#                      the system is in a safe or unsafe state.
#
#                      Upon execution, the user inputs the number of
#                      processes and resources, and enters the available
#                      resources, maximum resource allocation matrix, and
#                      current resource allocation matrix.
#                      After entering the required information, the user
#                      clicks the "Check Safety" button,
#                      and the program runs the Banker's Algorithm to check
#                      if the system is in a safe state.
#
#                      If the system is in a safe state, the program will
#                      display a message indicating the safe status
#                      and provide a safe sequence of processes that can be
#                      executed without causing a deadlock.
#                      If the system is in an unsafe state, the program
#                      will display a message indicating the
#                      unsafe status.
#
# References Cited: Below are the resources that were utilized to clarify
#                   and solve the given problem.
#
#                   1. Operating System Concepts (10th Edition) by Abraham
#                      Silberschatz, Peter B. Galvin,
#                      and Greg Gagne - The textbook provides a
#                      comprehensive explanation of the Banker's Algorithm
#                      and its implementation.
#                      Link: https://www.os-book.com/OS10/index.html
#                   2. GeeksforGeeks - Banker's Algorithm in Operating
#                      System. This article offers a detailed explanation
#                      of the Banker's Algorithm, its safety algorithm,
#                      and resource request algorithm.
#                      Link: https://www.geeksforgeeks.org/bankers-
#                      algorithm-in-operating-system/
#                   3. Programiz - Banker's Algorithm in Python. This
#                      article provides an example of Banker's Algorithm
#                      implemented in Python.
#                      Link: https://www.programiz.com/python-
#                      programming/examples/bankers-algorithm
#                   4. Tutorialspoint - Operating System - Banker's
#                      Algorithm. This article gives an overview of the
#                      Banker's Algorithm, its purpose, and how it works.
#                      Link: https://www.tutorialspoint.com
#                   5. Stack Overflow - How to implement Banker's Algorithm
#                      in Python?
#                      This discussion on Stack Overflow provides insights
```

```
#                       and code snippets related to implementing
#                       the Banker's Algorithm in Python.
#                       Link: https://stackoverflow.com/questions/47706058

import re
from tkinter import *


# Function to find the need of each process
def calculateNeed(need, maxm, allot):
    for i in range(len(need)):
        for j in range(len(need[i])):
            need[i][j] = maxm[i][j] - allot[i][j]


# Function to find whether a process can be allocated resources
def isSafe(processes, available, maxm, allot):
    need = [[0] * len(available) for i in range(len(processes))]
    calculateNeed(need, maxm, allot)

    # Mark all processes as unfinished
    finish = [False] * len(processes)

    # Initialize the work and the finish arrays
    work = available.copy()

    # Initialize the safe sequence array
    safe_sequence = []

    # Find a process which can be allocated resources
    found = True
    while found:
        found = False
        for i in range(len(processes)):
            # Check if the process has not finished and its need can be
satisfied
            if not finish[i] and all(need[i][j] <= work[j] for j in
range(len(work))):
                # Allocate resources to the process
                for j in range(len(work)):
                    work[j] += allot[i][j]

                # Mark the process as finished
                finish[i] = True

                # Add the process to the safe sequence
                safe_sequence.append(i)

                found = True
                break # Exit the loop after finding a process that can be
allocated resources

    # If all processes are finished, the system is in a safe state
    if all(finish):
        return True, safe_sequence
    else:
        return False, []
```

```
# Function to display the input data in a table format
def displayInputData(available, maxm, allot):
    output_text = "\nInput Data:\n\n"
    output_text += "Available Resources: " + ", ".join([f"{x}" for x in
available]) + "\n\n"
    output_text += "Process\tMaximum\tAllocation\n"
    for i in range(len(maxm)):
        output_text += f"P{i}\t"
        output_text += "[" + ", ".join([f"{x}" for x in maxm[i]]) + "]" +
"\t"
        output_text += "[" + ", ".join([f"{x}" for x in allot[i]]) + "]" +
"\n"
    output_text += "\n"

    return output_text


# Function to check whether the input values are valid
def validateInput(input_string):
    if not input_string:
        return False

    # Check if input is valid matrix of integers
    try:
        rows = input_string.strip().split('\n')
        for row in rows:
            values = [int(x) for x in row.strip().split(',')]
    except ValueError:
        return False

    return True

def checkSafety():
    # Get the input values
    num_processes = num_processes_spinner.get()
    num_resources = num_resources_spinner.get()

    if not num_processes or not num_resources:
        output_label.configure(text="Please enter the number of processes
and resources.")
        return

    num_processes = int(num_processes)
    num_resources = int(num_resources)

    if num_resources > len(available_spinners):
        num_resources = len(available_spinners)

    available = [int(available_spinners[i].get().strip()) for i in
range(num_resources)]  # Update the available array here
    maxm = [[int(x) for x in row.split(",")] for row in
maxm_text.get("1.0", "end-1c").split("\n")]
    allot = [[int(x) for x in row.split(",")] for row in
allot_text.get("1.0", "end-1c").split("\n")]
```

```
    # Check if the inputs are valid
    if not all([num_processes > 0, num_resources > 0] +
[validateInput(x.get()) for x in available_spinners] +
[validateInput(maxm_text.get("1.0", "end-1c")),
validateInput(allot_text.get("1.0", "end-1c"))]):
        output_label.configure(text="Invalid input.")
        return

    # Check if there are any negative values in the input matrices
    if any(any(x < 0 for x in row) for row in maxm + allot + [available]):
        output_label.configure(text="Invalid input. Matrix values should
not be negative.")
        return

    # Check if there are any values in the input matrices that exceed the
maximum resource value
    if any(any(x > 10 for x in row) for row in maxm + allot + [available]):
        output_label.configure(text="Invalid input. Matrix values should
not exceed 10.")
        return

    # Check if the number of resources requested by a process exceeds the
maximum resources available
    if any(any(x > available[i] for i, x in enumerate(row)) for row in
allot):
        output_label.configure(text="Invalid input. The number of resources
requested by a process exceeds the maximum resources available.")
        return

    # Run the banker's algorithm
    safe, sequence = isSafe(list(range(num_processes)), available, maxm,
allot)

    # Display the results
    if safe:
        output_label.configure(text="The system is in a safe state.\nSafe
sequence: " + ", ".join([f"P{x}" for x in sequence]))
    else:
        output_label.configure(text="The system is in an unsafe state.")

# Create the main window
root = Tk()
root.title("Banker's Algorithm")

# Create the widgets
num_processes_label = Label(root, text="Number of Processes:")
num_processes_spinner = Spinbox(root, from_=1, to=10)
num_resources_label = Label(root, text="Number of Resources:")
num_resources_spinner = Spinbox(root, from_=1, to=10)

available_label = Label(root, text="Available Resources:")
available_spinners = []
for i in range(10):
    spinner = Spinbox(root, from_=0, to=10)
    available_spinners.append(spinner)
```

```
maxm_label = Label(root, text="Maximum Resource Allocation (one row per
process):")
maxm_text = Text(root, width=50, height=10)
maxm_scroll = Scrollbar(root, command=maxm_text.yview)
maxm_text.config(yscrollcommand=maxm_scroll.set)

allot_label = Label(root, text="Current Resource Allocation (one row per
process):")
allot_text = Text(root, width=50, height=10)
allot_scroll = Scrollbar(root, command=allot_text.yview)
allot_text.config(yscrollcommand=allot_scroll.set)

check_button = Button(root, text="Check Safety", command=checkSafety)
output_label = Label(root, text="")

# Add the widgets to the window
num_processes_label.grid(row=0, column=0, padx=5, pady=5, sticky=W)
num_processes_spinner.grid(row=0, column=1, padx=5, pady=5, sticky=W)
num_resources_label.grid(row=1, column=0, padx=5, pady=5, sticky=W)
num_resources_spinner.grid(row=1, column=1, padx=5, pady=5, sticky=W)

available_label.grid(row=2, column=0, padx=5, pady=5, sticky=W)
for i in range(10):
    available_spinners[i].grid(row=2, column=i+1, padx=5, pady=5, sticky=W)

maxm_label.grid(row=3, column=0, padx=5, pady=5, sticky=W)
maxm_text.grid(row=3, column=1, columnspan=10, padx=5, pady=5, sticky=W)
maxm_scroll.grid(row=3, column=11, sticky=N+S+W)

allot_label.grid(row=4, column=0, padx=5, pady=5, sticky=W)
allot_text.grid(row=4, column=1, columnspan=10, padx=5, pady=5, sticky=W)
allot_scroll.grid(row=4, column=11, sticky=N+S+W)

check_button.grid(row=5, column=0, padx=5, pady=5, sticky=W)
output_label.grid(row=5, column=1, padx=5, pady=5, sticky=W)

# Start the main event loop
root.mainloop()
```

Problem Breakdown: We used a methodical approach to build the Banker's Algorithm by segmenting the task into smaller sections. The measures we took are outlined below:

1. Recognize the issue: We began by having a good grasp of the Banker's Algorithm, its function, and its part in preventing stalemate.
2. Determine inputs and data structures: Next, we determined the algorithm's necessary inputs, including the number of processes, resources, and available resources, as well as the maximum and current resource allocation matrices. We also decided which data structures would be best for storing these inputs.

3. **3.** Compute extra matrices: We computed the availability and need matrices to assess the system's safety.
4. Establish the safe sequence: We constructed the heart of the Banker's Algorithm by iteratively going through the processes and resources, determining if the process can be safely carried out by comparing its demand with the resources available. Then, we updated the available resources and added the approach to the safe sequence if it could be conducted safely.
5. Consider edge cases: We ensured our implementation could gracefully handle instances where the algorithm failed to identify a secure sequence.
6. Implement user input and output: We allowed the user to supply data for the algorithm's input and see the algorithm's output, including the safe sequence or a warning that the system is dangerous.
7. Test the implementation: To confirm our implementation's accuracy and dependability, we lastly put it through a rigorous testing process employing a variety of test scenarios. We modified and improved the code as required based on the test findings.

We swiftly and successfully built the Banker's Algorithm, guaranteeing its correctness and robustness for deadlock avoidance, by dividing the problem into smaller phases.

Implementation: After deconstructing the issue, we carried out the Banker's Algorithm implementation utilizing the subsequent steps:

1. Setting up the environment: To build the Banker's Algorithm using Tkinter, we utilised DataSpell as our development environment.
2. Several fundamental variables and data structures, including processes, availability, maxm, allot, need, work, finish, and safe sequence, were defined.
3. The maximum resource allocation matrix (maxm) and the current resource allocation matrix were used to create the need matrix using the calculateNeed() method (allot).
4. Putting the Banker's Algorithm's fundamental operation into practise: To repeatedly loop over processes and resources, evaluate each process's requirement with available resources, and update the safe sequence and work resources as necessary, we built the isSafe() method. The safe sequence is returned after this function determines if the system is safe.

5. Designing the graphical user interface (GUI): Using the Tkinter library, we built a GUI that enables users to enter data on the number of processes, the number of resources, the resources that are currently available, the maximum resource allocation matrix, and the current resource allocation matrix.
6. Validating user input: To ensure the user input is accurate, we built the validateInput() method, which checks for incorrect matrix inputs, negative values, and values that exceed the maximum resource limit.
7. The checkSafety() method was created to evaluate user inputs, run the Banker's Algorithm using the isSafe() function and display the algorithm's findings.
8. Showing input data: To display the input data in a tabular fashion, we created the displayInputData() method.
9. displaying results, The Banker's Algorithm findings, including the safe sequence (if discovered) or a warning if the system is in an unsafe condition, were shown using a Tkinter Label widget.

Using The Program:

1. Ensure that the Tkinter library and Python are installed on your machine. On most computers, Tkinter is installed with Python, but you can check if it's there by executing import tkinter in a Python shell.
2. Run the Python script that contains the Banker's Algorithm implementation to start the programme.
3. After the graphical user interface (GUI) loads, you can input the necessary data:
   a. Use the "Number of Processes" spinner to enter the number of processes.
   b. Enter the total resources using the "Number of Resources" spinner.
   c. Using the "Available Resources" spinners, specify the resources that are accessible for each resource category.
   d. Fill out the "Maximum Resource Allocation" text box with the resource allocation matrix's maximum values. Be careful to enter one row per process, with line breaks between rows and commas between resource values.
   e. Use the same format as the maximum resource allocation matrix to enter the current resource allocation matrix in the "Current Resource Allocation" text box.

After inputting all the necessary data, click the "Check Safety" button to run the Banker's Algorithm.

4. After inputting all the necessary data, click the "Check Safety" button to run the Banker's Algorithm.
5. Below the "Check Safety" button, the computer will evaluate the provided data and present the findings. The safe series of processes will be shown if the system is safe. An alert message will appear if the system is in a dangerous condition.

For more comprehensive information and use examples, please refer to the AdditionalTestCases.txt file and the associated ReadMe.txt file.

<u>Input and Output</u>: Below is a sample run of the program.

- **<u>Program Interface</u>:** If run without error, the program will present the following window.

- **<u>Input Sample</u>:** We will use the following input to test the program.

Number of Processes: 5

Number of Resources: 3

Available Resources:

R1: 10

R2: 5

R3: 7

Maximum Resource Allocation matrix (one row per process):

P0: 7, 5, 3

P1: 3, 2, 2

P2: 9, 0, 2

P3: 2, 2, 2

P4: 4, 3, 3

Current Resource Allocation matrix (one row per process):

P0: 0, 1, 0

P1: 2, 0, 0

P2: 3, 0, 2

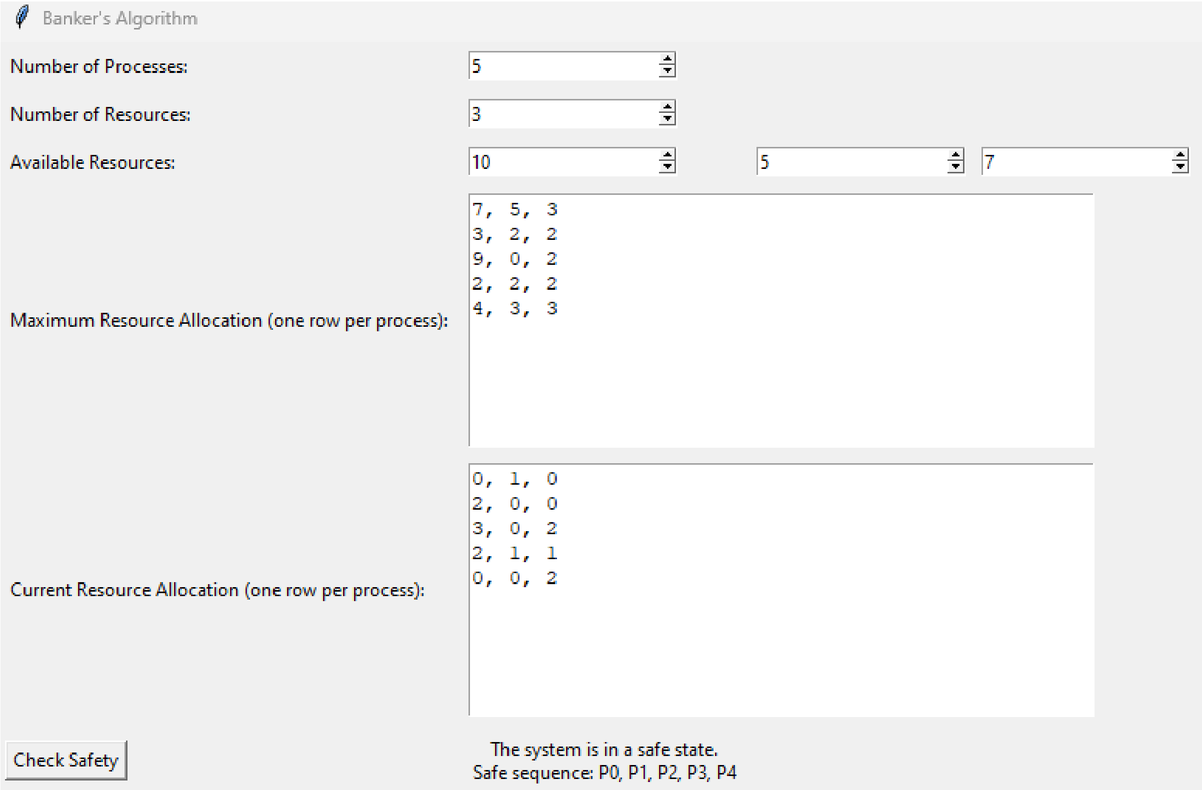
P3: 2, 1, 1

P4: 0, 0, 2

Upon entering the above test case into the program, click the "Check Safety" button. The output should display that the system is in a safe state and the safe sequence. A possible safe sequence is "P0, P1, P2, P3, P4".

Note that the actual safe sequence might vary depending on the implementation, but the result should always indicate a safe state for this test case.

- **<u>Output</u>:** Following is the expected out with the given test case.



Bibliography

[1] Dijkstra, E. W. (1965). Cooperating sequential processes. In Programming Languages, F. Genuys (Ed.), Academic Press, pp. 43-112.

[2] Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts. John Wiley & Sons.

[3] Tanenbaum, A. S., & Bos, H. (2014). Modern Operating Systems. Prentice Hall.

[4] Stallings, W. (2014). Operating Systems: Internals and Design Principles. Prentice Hall.