CSE316

(OPERATING SYSTEMS)

COMPUTER SCIENCE AND ENGINEERING

Submitted by:

NAME: SARTHAK KAUNDAL

REGISTRATION NUMBER: 12300561

ROLL NUMBER: 58

NAME: ARPIT KUMAR

REGISTRATION NUMBER: 12301610

ROLL NUMBER : 46

Submitted to:

DR. PARVINDER SINGH

Abstract

Deadlocks is a significant problem in operating systems where multiple processes take and request resources at the same time, leading to a deadlock situation in which none of the processes are able to go any further. In this project, we introduce the Deadlock Detection Tool that is a computer program to recognize deadlock states based on system resource allocation and process dependency analysis using graph-based methods. The tool constructs a Wait-For Graph (WFG) of the dynamic process-resource dependencies. The tool uses cycle detection routines that are effective in performing Depth-First Search (DFS) to detect circular wait conditions that form deadlocks. Upon detecting a deadlock, the system gathers detailed information and recommends recovery options like resource preemption or abortion of processes.

This report offers a thorough description of the design, implementation, and assessment of the tool. We cover theoretical background, present software design, provide detailed code reviews, and present experimental results based on simulated test inputs. We finally recommend possible directions for improvement and future work that would make real-time monitoring and resolution more effective. This project is aimed at enhancing the reliability of operating systems through providing system administrators and developers with an effective means of early deadlock detection and prevention.

Table of Contents

Revision Tracking on GitHub: -

https://github.com/arpitkumar08/Deadlock-Detector

1. Introduction

A modern operating system, in most cases, is never a one-process system. These competing processes are fighting for the resources of another process, which comprises CPU time, memory, and peripheral devices. Deadlock occurs when a set of processes is blocked because every process in the set is waiting for a resource held by another process in the set. Deadlock can cause freezing of the system in any one process under consideration cannot go further until deadlock is removed.

Deadlock detection is concerned as:

• There is guarantee reliability of the system and no indefinite blocking.

• Diagnosis and correction of resource conflict problems become a bit less tedious.

• It also serves to improve the efficiency of the system overall since the process waiting time is lessened.

Deadlock detection tool is the automatic form of solving the detection and recovery of deadlocks through systematic observation of process-resource allocation status in the system. Indeed, it identifies deadlock situations through advanced graph-based algorithmic methods, and gives recommendations for recovery.

---

2. Deadlock Conditions

There are four conditions which must hold simultaneously in order to have a deadlock:

2.1 Mutual Exclusion

• Definition: A process at a given instant might utilize a resource.

• Description: Certain resources, such as disk drives or printers, are nonshared resources. A process that accesses one of these types of resources will cause all the other processes to wait until this resource becomes available again.

• Effect: This renders the resource less accessible and aids in the formation of potential bottlenecks.

2.2 Hold and Wait

• Definition: Holding processes can request additional resources.

• Explanation: A process can hold one resource and hold off for additional resources so multiple processes hold a few resources and hold off for others.

• Impact: Makes deadlock more likely because the processes are not releasing resources upon holding off.

2.3 No Pre-emption

• Definition: Resources are not pre-eminently taken from a process.

• Reason: After a process gains a resource, it has to relinquish it voluntarily. It is not easy for the system to break deadlock cycles by reallocating resources.

• Effect: Prevents the system from doing anything to terminate deadlock cycles without relying on external actions.

2.4 Circular Wait

• Definition: There is a circular chain of processes where every process possesses a resource which the next process requires.

• Explanation: A circular wait is an occurrence when a series of processes create a circular chain where every process waits for a resource held by the next process.

• Impact: Most vital requirement to detect deadlocks and the main issue of the graph-based approach used by our tool.

---

3. Deadlock Prevention

Elimination of deadlocks dealt with by operating systems using one out of three strategies:

3.1 Deadlock Prevention:

• Approach: System policy to be modified such that at least one of the four deadlock conditions cannot hold true at a given time.

• Examples:

- **Resource ordering**
- **Request limiting on resources**

• Trade-offs: This can render resources substantially underutilized.

3.2 Deadlock Avoidance:

• Approach: The resource allocation state is examined on a dynamic basis, to avoid the system reaching an unsafe state with respect to a request for a resource.

• Example:

Banker's Algorithm

• Trade-offs: Requires a priori knowledge of resource needs and may prove unrealistic in highly dynamic environments.

3.3 Deadlock Detection and Recovery:

• Strategy: Permit deadlocks to occur, and then detect and correct them with recovery methods.

• Method Applied in This Project:

o The tool continuously monitors the system, constructs a Wait-For Graph, and detects cycles using DFS.

• Recovery Techniques:

- **Killing one or more processes involved in the deadlock.**
- **Pre-emption of resources from processes.**

• Trade-offs: Incur the cost of overheads during detection and recovery but flexible and adaptable for different situations.

4. Project Objective The following constraints were aimed by cycle detection tool:

•Accuracy in detection: a sound system must be able to detect deadlock in a satisfactory manner with respect to complex process-resource bindings.

•Graph-theoretic analysis: dynamic interdependencies of processes are subjected to analysis by means of graph theory and DFS.

•Full logging: most importantly logging that references deadlock detection for debugging and system analysis.

•Output User-centric: Clearly stated, actionable outputs that enable system administrators to understand and resolve deadlocks.

•Scalability: so that the tool can capably travail in the presence of many processes and an enormous pool of resources.

•Extensibility: making the solution such that it can easily extend to the addition of further detection methods or resolution methods.

5. Methodology

This project will be executed in a planned and systematic manner with the following phases:

5.1 Data gathering

•Input Datagen:

The tool is started by the collection of views about the observed system state: running processes, consumed resources, and pending resource requests.

•Data sources:

No logs, process monitors, or user input.

5.2 Graph Construction

•Wait-For Graph (WFG):

In this directed graph, each process is a node. o An edge from Process A to Process B indicates that Process A is waiting for a resource held by Process B.•\tSignificance:

The WFG is shown a computational and graphical model to identify dependencies and cycles.

5.3 Cycle detection

•Algorithm:

Depth-First Search (DFS) is employed by the tool to traverse the WFG.

If DFS identifies a back edge creating a cycle, a deadlock is identified.

•Cycle Analysis:

A detailed log of the cycle is captured to identify all processes in the deadlock.

5.4 Logging and Reporting

•Logging:

Deadlock information like process IDs and resource states are kept in a SQLite database.

•Reporting:

The system offers an explicit description of the identified deadlock and recovery actions which may be undertaken.

5.5 Resolution Strategy

User Guidance:

The utility suggests solutions such as killing a process in the loop or resource preemption.

Implementation:

Resolution automatically is engaged, but adequate information is given for a system administrator to select the best option.

---

6. System Design

6.1-Architectural Overview

The system itself consists of various interactive modules.

• Input Handler:

Reads current process-resource allocation from simulated input or system files.

• Graphics Constructor:

Converts input data to a Wait-For Graph representation.

• Detection Module:

Strict application of DFS for the detection of cycles.

• Logger Module:

Log occurrences of deadlocks detected into the database.

• Resolution Module:

Suggest remedies for possible identified deadlocks.

• User Interface:

A graphical user interface (GUI) to show the status of processes and resources, present deadlocks that were detected, and logging capabilities.

6.2 Module Descriptions

6.2.1 Input Handler

• Responsibilities:

To validate the input data.

To preprocess data for future use in graph construction.

• Design Considerations:

Extensive error handling dealing with invalid or missing input.

6.2.2 Graph Constructor

• Responsibilities:

It builds a WFG based on valid inputs.

• Design Considerations:

Be optimized for large inputs to prevent performance bottlenecks.

6.2.3 Detection Module

- Traverse the WFG using DFS

- Identify loops referring to deadlock

Design Considerations: Key-Recursion or iterative approach with a stack for enabling DFS.

6.2.4 Logger Module

Responsibilities- Maintain database using SQLite over deadlock incidents-

Audit trail examination. Design Considerations-

Fast database transactions with minimum user disturbances for detection.

6.2.5 Module of Resolution

Responsibilities-

Cycle detection.

Offer potential resolution methods.

Design Considerations- Consider system impacts before suggesting termination of processes.

---

7. Implementation Details

The code for the project is in Python and structured as a sequence of modules. Following is a detailed explanation of the key modules and their function.

7.1 Main Program (main.py)

Purpose:

• The entry point of the program.

• Initiates data collection, deadlock detection, logging, and reporting.

Main Program Key Steps:

python

CopyEdit

```
from core.deadlock_detector import detect_deadlock
from core.logger import log_deadlock
def main():
    # Mocked process and resource allocation
    processes = ["P1", "P2", "P3"]
resources = ["R1", "R2", "R3"]
    # Sample allocation and request mappings

    allocations = {"P1": "R1", "P2": "R2", "P3": "R3"}
requests = {"P1": "R2", "P2": "R3", "P3": "R1"}
# Deadlock detection

    deadlock_found = detect_deadlock(processes, allocations, requests)
    if deadlock_found:
        print("Deadlock detected!")
```

log_deadlock(deadlock_found)

else:

print("No deadlock detected.")

if __name__ == "__main__":

main()

Discussion:

• A module illustrating that the tool performs a simulation of an average deadlock scenario.

• The code may be updated to be capable of parsing real system data from the operating system.

7.2 Deadlock Detection Module (deadlock_detector.py)

Purpose:

• Builds the Wait-For Graph (WFG) and subsequently performs DFS for cycle detection.

Algorithm Explanation:

python

CopyEdit

def detect_deadlock(processes, allocations, requests):

 # Build a Wait-For Graph as a dictionary

graph = {p: [] for p in processes}

for process, resource in requests.items():

if resource in allocations.values():

owner = [p for p, r in allocations.items() if r == resource][0]

graph[process].append(owner)

 # Recursive DFS function to detect cycles

```
def dfs(visited, stack, node):
    visited.add(node)
stack.add(node)
for neighbor in graph[node]:
if neighbor not in visited:
        if dfs(visited, stack, neighbor):
return True
elif node in stack:
            return True  # Cycle found
    stack.remove(node)
return False
visited, stack = set(), set()


for process in processes:
  if process not in visited:
    if dfs(visited, stack, process):
      return f"Deadlock Detected: Cycle involving process {process}"
return None
```

Discussion:

• The DFS algorithm visits each node recursively.

• A deadlock cycle is created when a node is met that is present in the recursion stack when the node has already been visited.

• The architecture ensures even deeply complicated dependency graphs are processed well.

7.3 Logger Module (logger.py)

Purpose:

• Triggers identified deadlock events into an SQLite database to be investigated post-mortem.

Implementation:

python

CopyEdit

```
import sqlite3

def log_deadlock(deadlock_message):

   connection = sqlite3.connect("database/deadlock_logs.db")

   cursor = connection.cursor()

cursor.execute("CREATE TABLE IF NOT EXISTS logs (id INTEGER PRIMARY KEY, message TEXT)")

cursor.execute("INSERT INTO logs (message) VALUES?", (deadlock_message,))

connection.commit()

connection.close()
```

Discussion:

•This module creates a database table if it does not exist.

•Each deadlock occurrence is traced with a traceable unique identifier.

7.4 GUI Module (gui/app.py)

Purpose:

•It provides a graphical representation for displaying the Wait-For Graph and deadlock condition.

•It gives interactive choices for solving deadlock (e.g., marking processes engaged in deadlock).

Discussion

- **The GUI is interactive with immediate feedback.**
- **Screenshot placeholders tell the user what they will be seeing when the tools are executed.**

_____

8. Deadlock Detection Algorithm – Step-by-Step Walkthrough

8.1 Wait-For Graph (WFG) Construction

•Concept:

A graph node is employed to represent each process.

A directed edge from process A to process B indicates that A is waiting for a resource held by B.

•Example:

If Process P1 is waiting for R1 and holding R2, P2 is waiting for R2 and holding R3, and P3 is waiting for R3 and holding R1, the resulting WFG illustrates a cycle.

8.2 Cycle Detection Using DFS

•Step-by-Step Process:

1.

Initialization:

Mark all nodes as unvisited.

2.

DFS Traversal:

For every unvisited node, initiate DFS.

Mark nodes as visited and add them to the current recursion stack.

3.

Cycle Identification:

If it travels to a node already in the recursion stack, then there is a cycle.

4. Deadlock Logging:

The detected cycle is logged for diagnosis.

8.3 Pseudocode Summary

CopyEdit

function detect_deadlock(processes, allocations, requests):

build graph from processes and requests

for each process in processes:

if process not visited:

if DFS(process) is true:

log cycle and deadlock information

return no deadlock

function DFS(node):

mark node as visited

add node to recursion stack

for each neighbor in graph[node]:

if neighbor not visited:

if DFS(neighbor) returns true:

return true

elif neighbor in recursion stack:

return true

remove node from recursion stack

return false

---

9. Experimental Results & Case Study

9.1 Test Case Description

We tested our system using a simulated system with three processes (P1, P2, P3) and three resources (R1, R2, R3) with the following allocation:

Process Allocated Resource Requested Resource

P1 R1 R2

P2 R2 R3

P3 R3 R1

9.2 Observations

•Wait-For Graph Analysis:

The graph shows a cycle: P1 → P2 → P3 → P1.

•Detection Output:

The tool detects the cycle accurately and prints:

"Deadlock Detected: Cycle involving process P1"

•Logging:

The occurrence of deadlock is recorded in the database to be analyzed later.

9.3 Detailed Discussion

Experimental results confirm the effectiveness of our tool:

•Scalability:

The tool is applicable for bigger systems containing numerous processes.

•Efficiency:

DFS-based cycle detection exhibits acceptable performance in simulated systems.

•Practical Impact:

Early identification can significantly avoid the downtime caused by deadlocks in critical systems.

---

10. GUI & Database Analysis

10.1 GUI Overview

The GUI module, implemented in Python, offers:

•Real-time Monitoring:

Live visualization of process-resource relationships.

•Interactive Controls:

Facilities to simulate resource requests and analyze detailed logs.

User Feedback:

Visual cues (e.g., color-coded nodes) indicate deadlock cycles.

10.2 Database Logging

The tool uses an SQLite database (deadlock_logs.db) to record data on detected deadlocks:

•Structure:

Table with columns for auto-incremented ID and the deadlock message.

•Usage:

System administrators may ask the database how past deadlock occurrences have happened and review system activity.

•Benefits:

Provides an audit trail that can be used for further performance and troubleshooting analysis.

---

11. Future Improvements

While the current tool is a solid foundation to build from when it comes to deadlock detection, there are many possible areas of improvement:

11.1 Real-Time Monitoring

•Goal:

On add the tool with real-time OS information in order to track resource allocation in real-time.

•Potential Methods:

To use OS APIs in order to get real-time process and resource information.

To provide event-based notifications via the GUI.

11.2 Advanced Resolution Strategies

•Goal:

To Automate the recovery with decision algorithms.

•Possible Methods:

To Create checkpointing and rollback of processes.

To Create ways to preempt resources dynamically without drastic system disruption.

11.3 Improved Visualization

•Goal:

To enhance user interaction and comprehension with thoughtful visual presentations.

•Possible Methods:

To Utilize sophisticated graph libraries to support interactive WFG visualizations.

To Employ animation to illustrate the deadlock formation and resolution process.

11.4 Scalability Improvements

•Objective:

to Improve the tool for large systems.

•Potential Solutions:

To parallelize DFS for effective cycle detection.

To optimize database operations to allow a high rate of deadlock detection.

_____

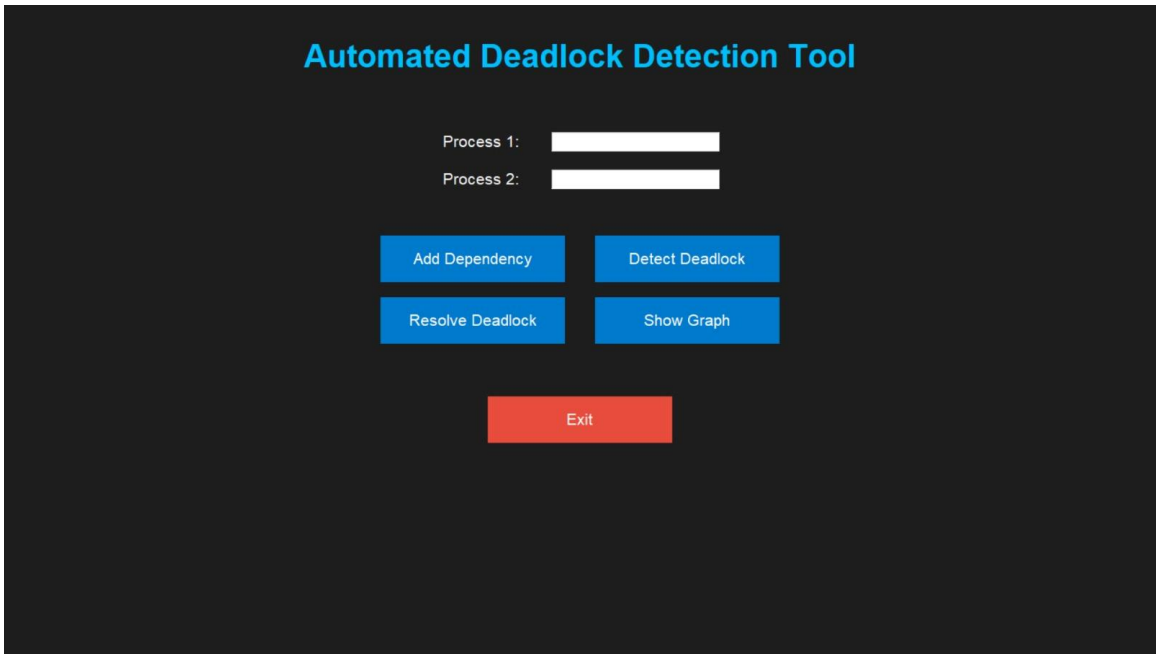
12. Conclusion

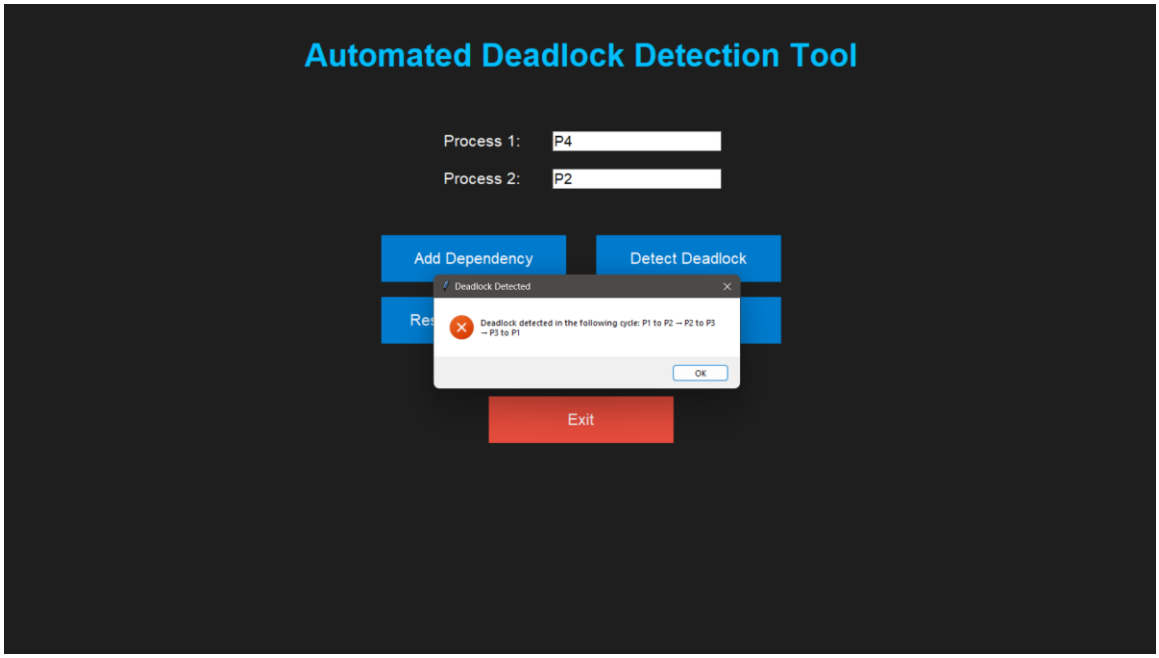
Deadlock Detection Tool is an end-to-end solution that solves one of the biggest operating system problems. Using graph-based analysis and DFS, the tool successfully detects deadlocks by identifying cycles in the process-resource allocation graph. Fine-grained logging and usability also facilitate on-time deadlock resolution.

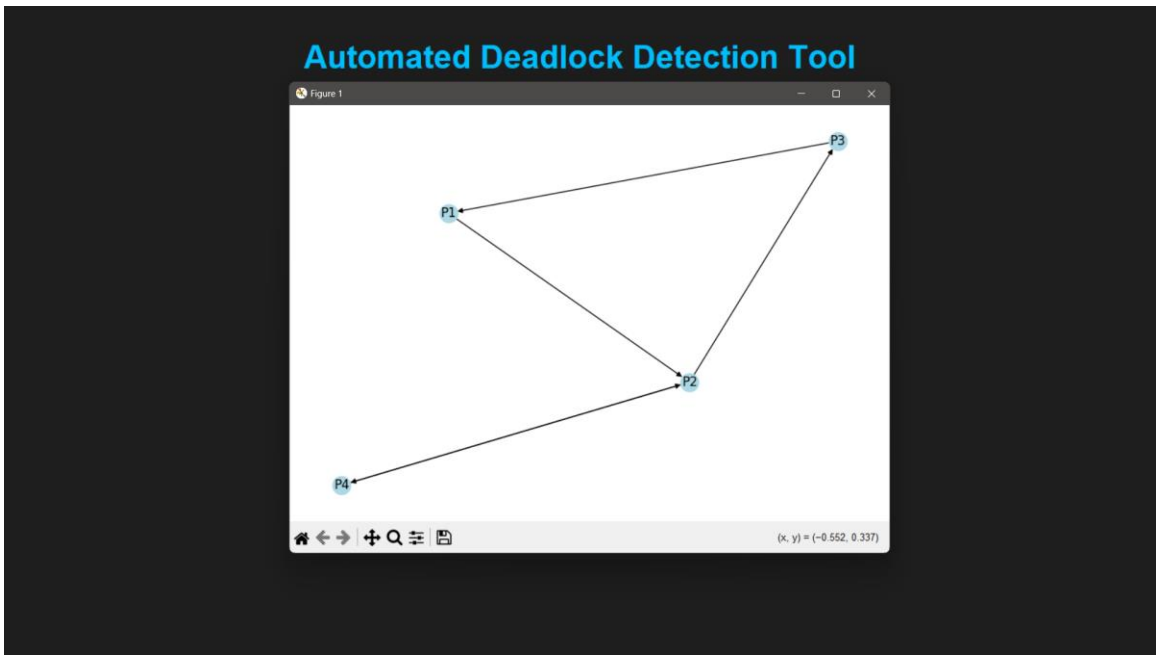
Our experiment in the case study demonstrates the effectiveness of the tool in a test environment, and taking future enhancement into account gives the clear direction necessary for further system improvement and scalability. The project is not only an educational tool but also an effective guide to system reliability and resource allocation.

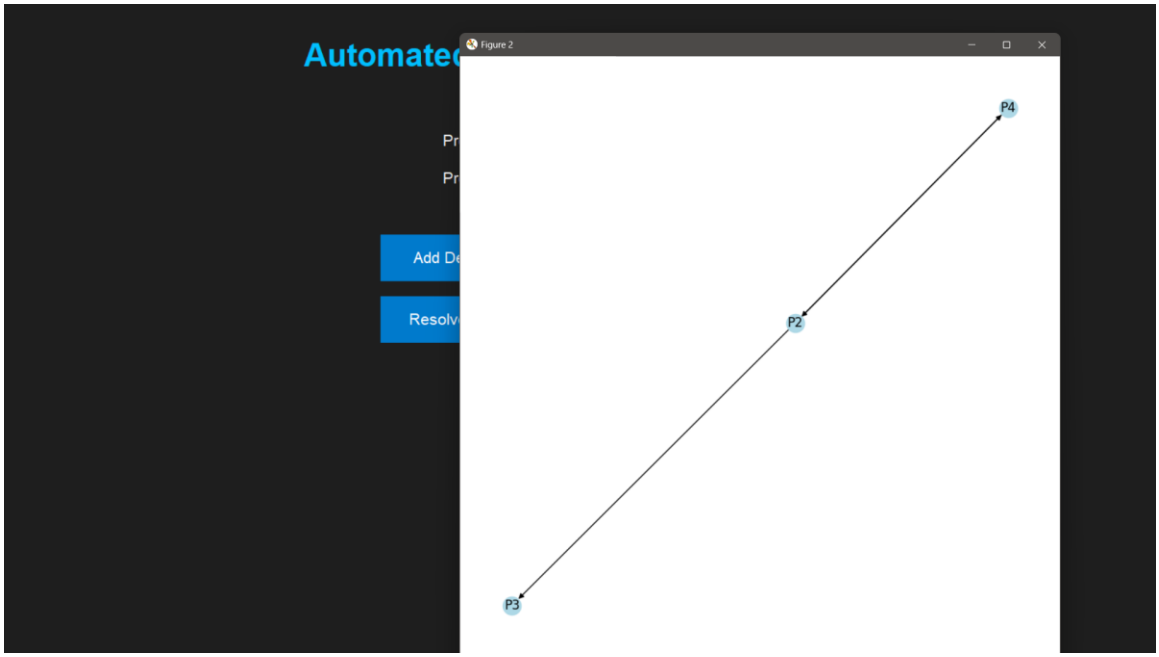
13. Appendix

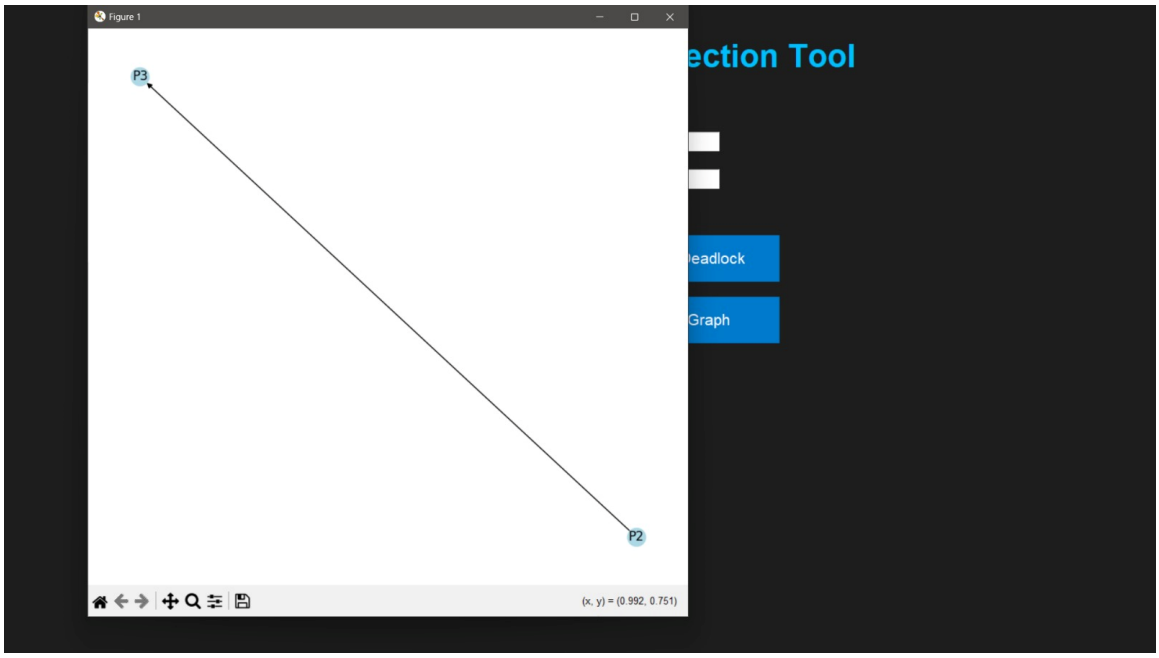
13.1 Screenshots





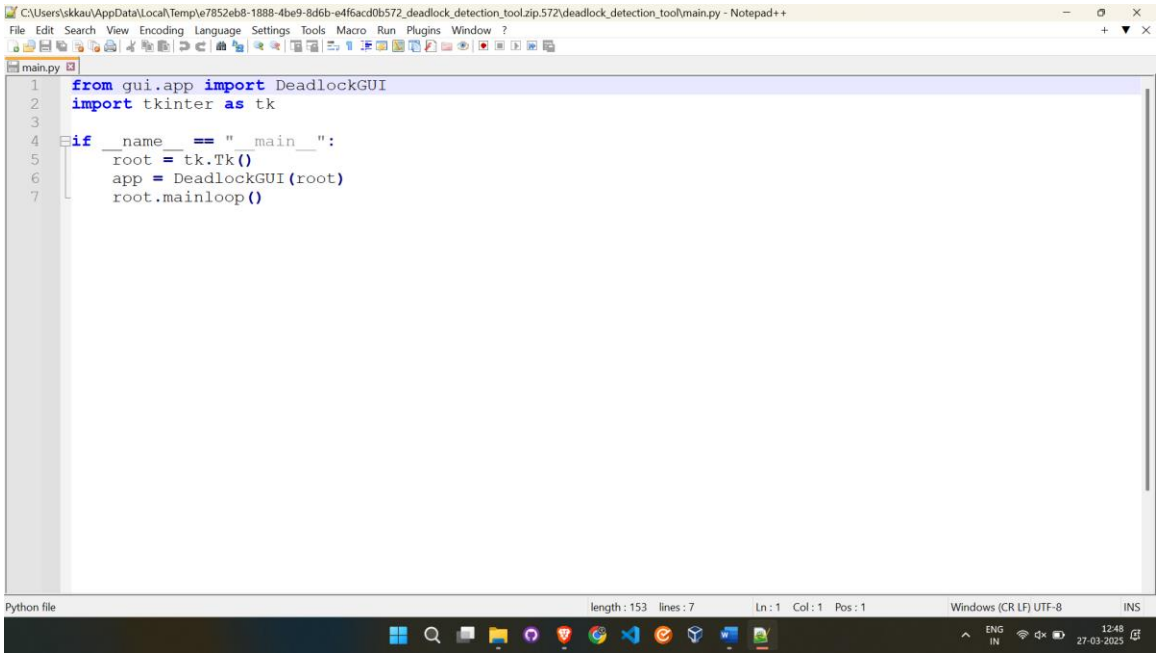

Automated Deadlock Detection Tool
Figure 1
P1
P2
P3
P4
(x, y) = (−0.552, 0.337)


Automate
Add De
Resolv
Figure 2
P2
P3
P4

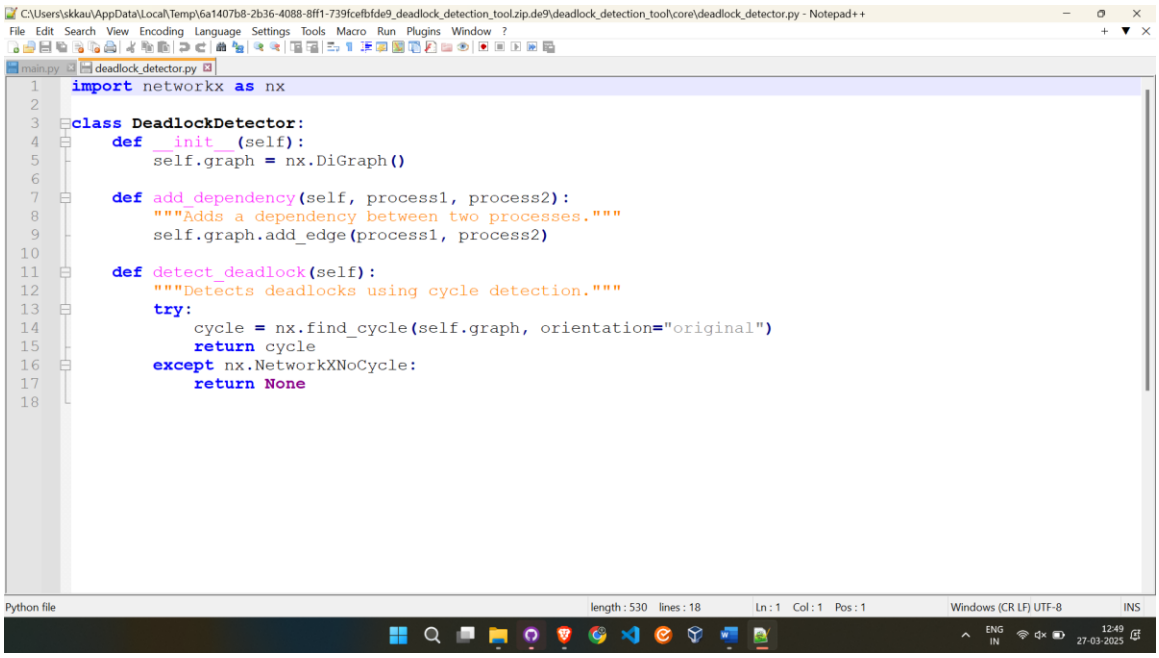


14.2 Code Listings

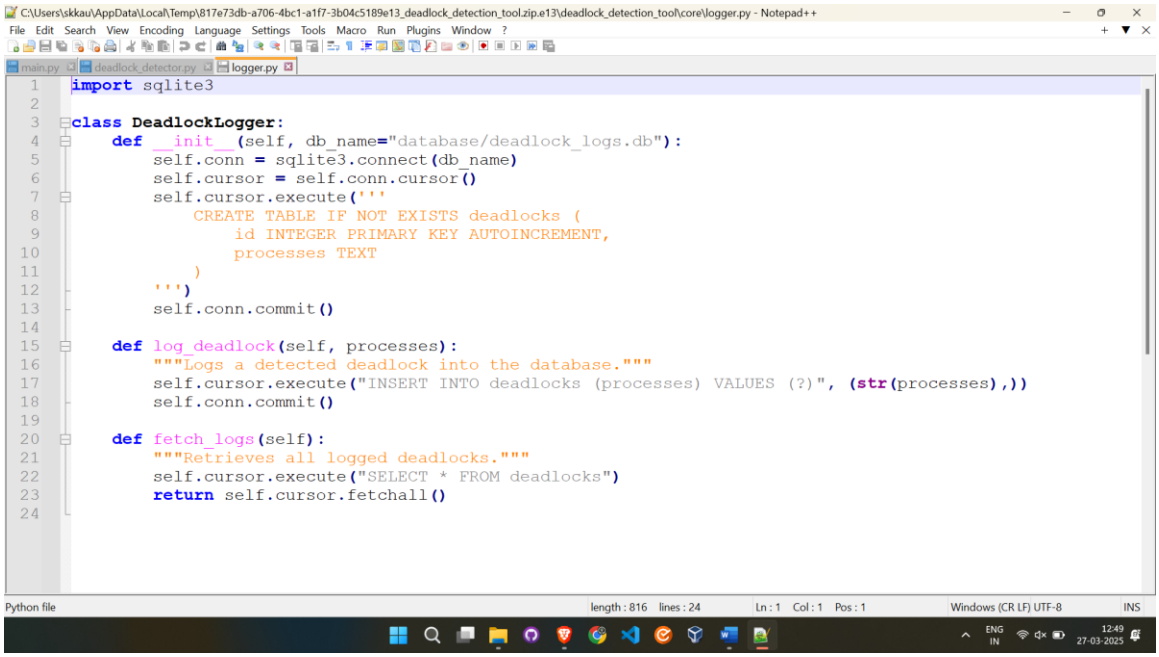
• Listing 1: Principal program (main.py)



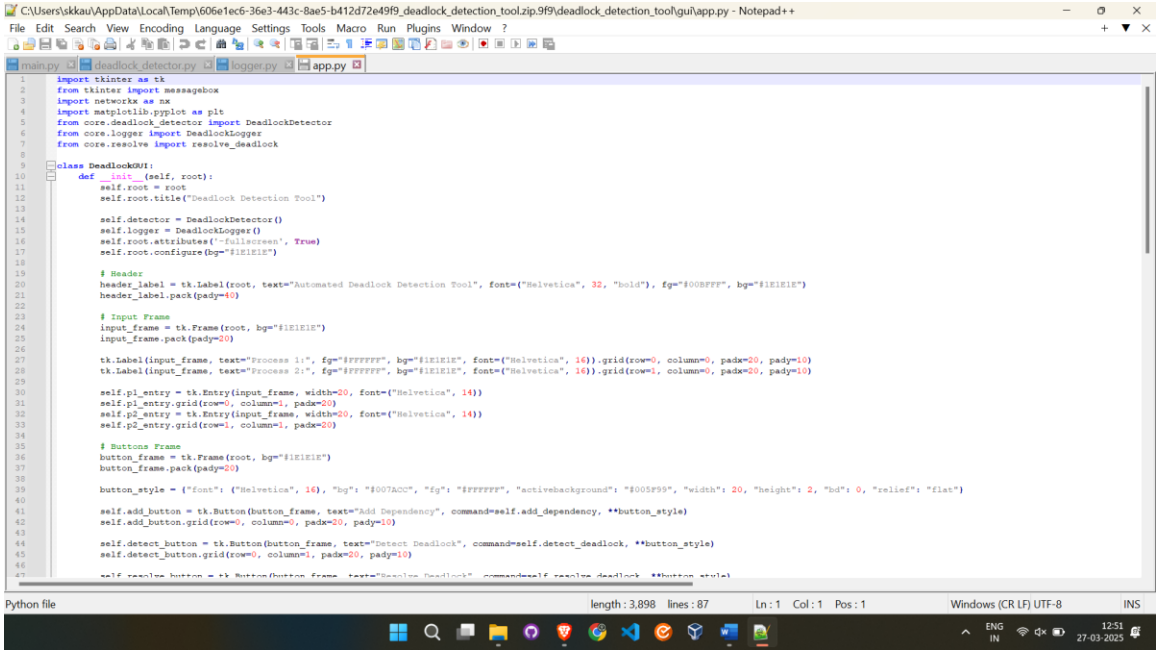
• Listing 2: Deadlock detection module (deadlock_detector.py)

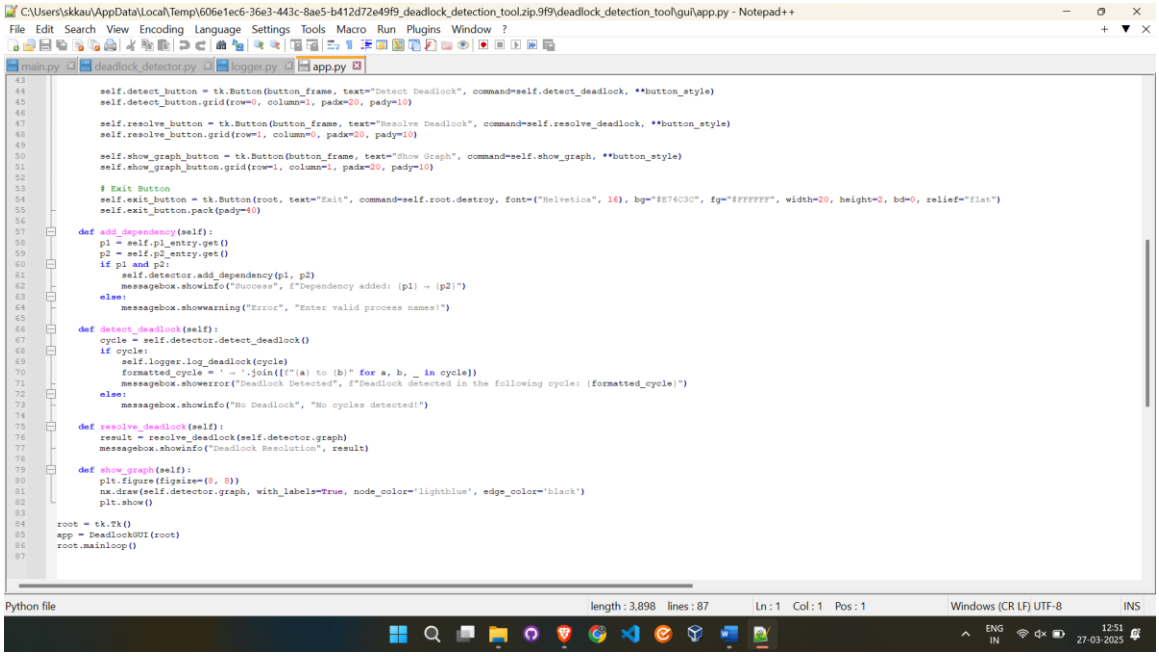


• **Listing 3: Logger module (logger.py)**



• **Listing 4: GUI module (gui/app.py)**


C:\Users\skkau\AppData\Local\Temp\606e1ec6-36e3-443c-8ae5-b412d72e49f9_deadlock_detection_tool.zip.9f9\deadlock_detection_tool\gui\app.py - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
main.py deadlock_detector.py logger.py app.py
import tkinter as tk
from tkinter import messagebox
import networkx as nx
import matplotlib.pyplot as plt
from core.deadlock_detector import DeadlockDetector
from core.logger import DeadlockLogger
from core.resolve_deadlock
class DeadlockGUI:
def __init__(self, root):
self.root = root
self.root.title("Deadlock Detection Tool")
self.detector = DeadlockDetector()
self.logger = DeadlockLogger()
self.root.attributes('-fullscreen', True)
self.root.configure(bg="#1E1E1E")
# Header
header_label = tk.Label(root, text="Automated Deadlock Detection Tool", font=("Helvetica", 32, "bold"), fg="#00BFFF", bg="#1E1E1E")
header_label.pack(pady=40)
# Input Frame
input_frame = tk.Frame(root, bg="#1E1E1E")
input_frame.pack(pady=20)
tk.Label(input_frame, text="Process 1:", fg="#FFFFFF", bg="#1E1E1E", font=("Helvetica", 16)).grid(row=0, column=0, padx=20, pady=10)
tk.Label(input_frame, text="Process 2:", fg="#FFFFFF", bg="#1E1E1E", font=("Helvetica", 16)).grid(row=1, column=0, padx=20, pady=10)
self.p1_entry = tk.Entry(input_frame, width=20, font=("Helvetica", 14))
self.p1_entry.grid(row=0, column=1, padx=20)
self.p2_entry = tk.Entry(input_frame, width=20, font=("Helvetica", 14))
self.p2_entry.grid(row=1, column=1, padx=20)
# Buttons Frame
button_frame = tk.Frame(root, bg="#1E1E1E")
button_frame.pack(pady=20)
button_style = {"font": ("Helvetica", 16), "bg": "#007ACC", "fg": "#FFFFFF", "activebackground": "#005F99", "width": 20, "height": 2, "bd": 0, "relief": "flat"}
self.add_button = tk.Button(button_frame, text="Add Dependency", command=self.add_dependency, **button_style)
self.add_button.grid(row=0, column=0, padx=20, pady=10)
self.detect_button = tk.Button(button_frame, text="Detect Deadlock", command=self.detect_deadlock, **button_style)
self.detect_button.grid(row=0, column=1, padx=20, pady=10)
Python file
length : 3,898 lines : 87
Ln : 1 Col : 1 Pos : 1
Windows (CR LF) UTF-8
INS
ENG IN
12:51 27-03-2025


C:\Users\skkau\AppData\Local\Temp\606e1ec6-36e3-443c-8ae5-b412d72e49f9_deadlock_detection_tool.zip.9f9\deadlock_detection_tool\gui\app.py - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
main.py deadlock_detector.py logger.py app.py
self.detect_button = tk.Button(button_frame, text="Detect Deadlock", command=self.detect_deadlock, **button_style)
self.detect_button.grid(row=0, column=1, padx=20, pady=10)
self.resolve_button = tk.Button(button_frame, text="Resolve Deadlock", command=self.resolve_deadlock, **button_style)
self.resolve_button.grid(row=1, column=0, padx=20, pady=10)
self.show_graph_button = tk.Button(button_frame, text="Show Graph", command=self.show_graph, **button_style)
self.show_graph_button.grid(row=1, column=1, padx=20, pady=10)
# Exit Button
self.exit_button = tk.Button(root, text="Exit", command=self.root.destroy, font=("Helvetica", 16), bg="#E74C3C", fg="#FFFFFF", width=20, height=2, bd=0, relief="flat")
self.exit_button.pack(pady=40)
def add_dependency(self):
p1 = self.p1_entry.get()
p2 = self.p2_entry.get()
if p1 and p2:
self.detector.add_dependency(p1, p2)
messagebox.showinfo("Success", f"Dependency added: {p1} → {p2}")
else:
messagebox.showwarning("Error", "Enter valid process names!")
def detect_deadlock(self):
cycle = self.detector.detect_deadlock()
if cycle:
self.logger.log_deadlock(cycle)
formatted_cycle = ' → '.join([f"{a} to {b}" for a, b, _ in cycle])
messagebox.showerror("Deadlock Detected", f"Deadlock detected in the following cycle: {formatted_cycle}")
else:
messagebox.showinfo("No Deadlock", "No cycles detected!")
def resolve_deadlock(self):
result = resolve_deadlock(self.detector.graph)
messagebox.showinfo("Deadlock Resolution", result)
def show_graph(self):
plt.figure(figsize=(8, 6))
nx.draw(self.detector.graph, with_labels=True, node_color='lightblue', edge_color='black')
plt.show()
root = tk.Tk()
app = DeadlockGUI(root)
root.mainloop()
Python file
length : 3,898 lines : 87
Ln : 1 Col : 1 Pos : 1
Windows (CR LF) UTF-8
INS
ENG IN
12:51 27-03-2025