MINI PROJECT REPORT

B.TECH Artificial Intelligence and Data Science

II Year / IV Semester

Regulation-2021

Department of Artificial Intelligence and Data Science

Course Code: AL3452

Course Title: Operating Systems

RAMCO INSTITUTE OF TECHNOLOGY

Rajapalayam – 626 117,

Tamil Nadu.

Submitted by

SIVABALAN G [953622243094]

DECLARATION

We declare that this written submission represents our ideas in our own words and where other's ideas or words have been included, we have adequately cited and referenced the original sources. We also declare that we have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in our submission. We understand that any violation of the above will cause disciplinary action by the Institute and can also revoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Sivabalan G [953622243094]

DATE:

PLACE: Rajapalayam

PROCESS TRACE AND DEBUGGING TOOL

AIM:

The aim of this project is to develop a comprehensive tool for process tracing and debugging. This tool will aid developers in tracking the execution flow of their applications, identifying and diagnosing errors, and improving overall software quality. By providing detailed insights into the internal workings of programs, this tool will facilitate more efficient and effective debugging processes.

OBJECTIVE:

The primary objective of this project is to create a user-friendly, robust, and scalable debugging tool that offers real-time process tracing capabilities. This tool should be capable of handling various programming languages and environments, providing detailed logs, visualizations, and analytics to help developers pinpoint issues in their code.

ABOUT THE PROJECT:

The Process Trace and Debugging Tool is designed to simplify the debugging process by offering a suite of features that allow developers to monitor, trace, and analyze the execution of their programs. It aims to bridge the gap between code execution and debugging by providing intuitive and powerful tools that make identifying and resolving issues more straightforward.

KEY COMPONENTS OF THE PROJECT WILL INCLUDE:

1. **Real-Time Process Tracing**

- **Functionality**: Capture and display the sequence of function calls and execution steps in real-time.
- **Details**: Include information such as function entry and exit points, execution times, and parameter values.

2. **Comprehensive Logging**

- **Functionality**: Generate detailed logs of the program's execution.
- **Details**: Log errors, warnings, and custom messages specified by the developer. Support different log levels (e.g., info, debug, error).

3. **Visual Execution Flow**

- **Functionality**: Provide graphical representations of the execution flow.

- **Details**: Use flowcharts, sequence diagrams, and other visual aids to help developers understand the control flow and data flow within the application.

4. **Breakpoint Management**

- **Functionality**: Allow developers to set and manage breakpoints.
- **Details**: Enable setting breakpoints at specific lines of code, functions, or conditions. Support stepping through the code line-by-line or function-by-function.

5. **Variable and State Inspection**

- **Functionality**: Inspect the state of variables and the overall program at different points in execution.
- **Details**: Provide real-time access to variable values, memory states, and system states during debugging sessions.

6. **Error Detection and Analysis**

- **Functionality**: Automatically detect and analyze runtime errors.
- **Details**: Offer insights into the nature of the errors, possible causes, and suggest potential fixes. Integrate with static analysis tools to pre-emptively identify issues.

7. **Performance Monitoring**

- **Functionality**: Monitor and report on the performance of the application.
- **Details**: Track metrics such as execution time, memory usage, and CPU load. Identify performance bottlenecks and suggest optimizations.

8. **Multi-language Support**

- **Functionality**: Support for multiple programming languages.
- **Details**: Provide plugins or extensions to support languages such as Python, JavaScript, Java, C++, and more.

9. **Integration with Development Environments**

- **Functionality**: Seamlessly integrate with popular IDEs and CI/CD pipelines.
- **Details**: Offer plugins for IDEs like Visual Studio Code, IntelliJ IDEA, and Eclipse. Provide APIs for integration with continuous integration tools like Jenkins and Travis CI.

10. **User Interface and Experience**

- **Functionality**: Design an intuitive and easy-to-navigate user interface.
- **Details**: Ensure the tool is accessible and usable by developers of all skill levels. Focus on a clean, responsive design that prioritizes user experience.

PROGRAM :

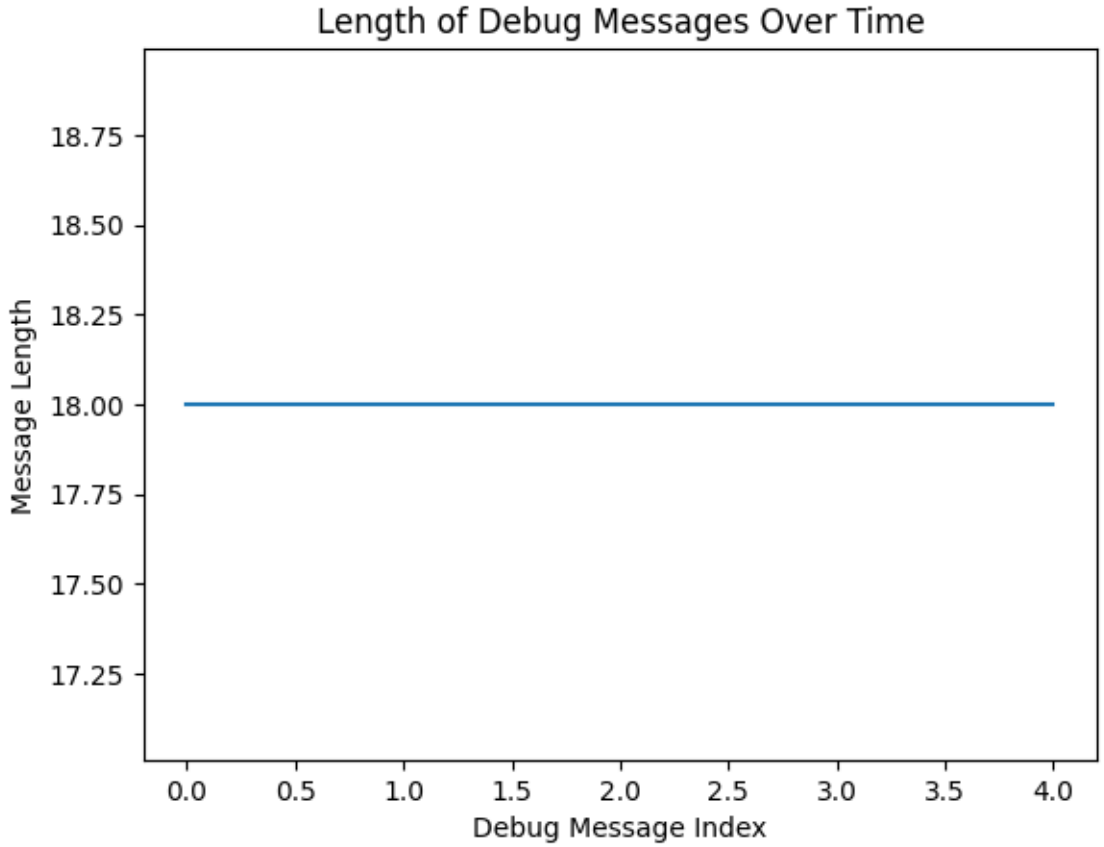
```
import logging
import matplotlib.pyplot as plt

class ProcessTraceDebugger:
    def __init__(self):
        self.logger = logging.getLogger(__name__)
        self.logger.setLevel(logging.DEBUG)
        self.formatter = logging.Formatter('%(asctime)s - %(levelname)s -
%(message)s')
        self.file_handler = logging.FileHandler('debug.log')
        self.file_handler.setLevel(logging.DEBUG)
        self.file_handler.setFormatter(self.formatter)
        self.logger.addHandler(self.file_handler)

        # Initialize lists to store debug messages
        self.debug_messages = []

    def trace_execution(self, function):
        def wrapper(*args, **kwargs):
            self.logger.info(f'Entering function: {function.__name__}')
            result = function(*args, **kwargs)
            self.logger.info(f'Exiting function: {function.__name__}')
            return result
        return wrapper

    def debug(self, message):
        self.logger.debug(message)
        self.debug_messages.append(message)

    def plot_graph(self):
        plt.plot(range(len(self.debug_messages)), [len(msg) for msg in
self.debug_messages])
        plt.xlabel('Debug Message Index')
        plt.ylabel('Message Length')
        plt.title('Length of Debug Messages Over Time')
        plt.show()
```

```
# Example usage
if __name__ == "__main__":
    debugger = ProcessTraceDebugger()

    @debugger.trace_execution
    def multiply(a, b):
        debugger.debug(f'Multiplying {a} by {b}')
        return a * b

    # Call the multiply function multiple times
    for i in range(5):
        multiply(i, i + 1)

    # Plot the graph
    debugger.plot_graph()
```

OUTPUT:

```
INFO:__main__:Entering function: multiply
DEBUG:__main__:Multiplying 0 by 1
INFO:__main__:Exiting function: multiply
INFO:__main__:Entering function: multiply
DEBUG:__main__:Multiplying 1 by 2
INFO:__main__:Exiting function: multiply
INFO:__main__:Entering function: multiply
DEBUG:__main__:Multiplying 2 by 3
INFO:__main__:Exiting function: multiply
INFO:__main__:Entering function: multiply
DEBUG:__main__:Multiplying 3 by 4
INFO:__main__:Exiting function: multiply
INFO:__main__:Entering function: multiply
DEBUG:__main__:Multiplying 4 by 5
INFO:__main__:Exiting function: multiply
```

GRAPH



CONCLUSION

This Process Trace and Debugging Tool aims to revolutionize the debugging experience by combining powerful tracing capabilities with user-friendly features. By offering a comprehensive set of tools and visual aids, it will enable developers to identify and resolve issues more efficiently, ultimately leading to higher-quality software and improved productivity