# PROCESS MANAGER

## A MINI PROJECT REPORT

**Submitted by**


| | |
|---|---|
| **ARUN BHARATHI M B** | **231901007** |
| **JAYGANESH KANNAN** | **231901015** |
| **SASIKUMAR B** | **231901047** |


In partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE

(CYBER SECURITY)


RAJALAKSHMI ENGINEERING COLLEGE (AUTONOMOUS)
THANDALAM
CHENNAI-602105


2025-26

## BONAFIDE CERTIFICATE

Certified that this project report "**PROCESS MANAGER**" is the bonafide work of **"ARUN BHARATHI M B (231901007), JAYGANESH KANNAN (2319010015), SASIKUMAR B (231901047)"**who carried out the project work under my supervision.

**Submitted for the Practical Examination held on** _

**SIGNATURE**

**Ms.V.JANANEE**
**Assistant Professor (SG),**
**Computer Science and Engineering,**
**Rajalakshmi Engineering College,**
**(Autonomous),**
**Thandalam, Chennai - 602 105**

**INTERNAL EXAMINER** **EXTERNAL EXAMINER**

# Abstract

This project, titled "Python-Based Task Manager with Real-Time Process Monitoring and Graphical Visualization", is developed to provide users with an interactive, real-time interface for viewing and managing system processes on their computer. The primary objective of the system is to offer a lightweight, GUI-based alternative to traditional command-line utilities and built-in operating system task managers, with a specific focus on process monitoring, selection, and termination.

The application is developed using Python, integrating three major libraries — tkinter for the graphical user interface, psutil for accessing system and process-level information, and matplotlib for generating real-time visualizations of CPU usage. The core features of the Task Manager include listing all active processes along with their names, Process IDs (PIDs), and statuses (e.g., running, sleeping, terminated). Users can interactively select one or more processes using a checkbox mechanism and terminate them with a single click.

A key enhancement of this system is the inclusion of a graphical module that generates a scrollable line chart representing the CPU usage of all running processes. This visualization provides a clear, comparative overview of resource consumption and helps users easily identify CPU-intensive applications. The chart is embedded within a separate Tkinter window and allows horizontal scrolling to accommodate systems with a large number of processes.

Unlike traditional task managers that often prioritize performance over accessibility, this project strikes a balance between usability, visibility, and functionality. It enables users, especially those new to system administration or programming, to explore and manage background tasks effectively in a user-friendly environment. Furthermore, the modular design allows future enhancements such as memory usage tracking, live auto-refreshing graphs, logging, or remote monitoring capabilities.

# TABLE OF CONTENTS

# PROCESS MANAGER

A PROJECT REPORT

# PROCESS MANAGER SYSTEM PROJECT REPORT

## CHAPTER 1
## INTRODUCTION

The Task Manager project is a system-level utility designed to simulate the functionality of a real task manager found in operating systems. Its main objective is to monitor and manage active processes, providing users with real-time information such as process ID (PID), process name, CPU and memory usage, priority, and current status.

The task manager also allows users to control these processes by terminating, pausing, or resuming them, offering basic system control features. Depending on the implementation, the user interface can be command-line-based or graphical, enhancing usability and interaction. Internally, the project consists of components such as a process management module to track running tasks, a resource monitoring module to collect system usage data, and a controller to execute user commands.

It interacts with system-level APIs, like the `/proc` filesystem in Linux, to fetch accurate and real-time data. This project serves as an educational tool to help understand core operating system concepts such as process states, system calls, resource allocation, and scheduling.

It also provides hands-on experience in systems programming, making it highly relevant for learners aiming to explore OS internals or prepare for careers in low-level or performance-focused software development.

# CHAPTER 2
# SCOPE OF PROJECT

The Task Manager project is developed with the intention of providing a simplified yet powerful tool to monitor and manage system processes. The scope of this project extends beyond basic process listing—it is designed to give users visibility into system resource usage and direct control over running tasks. It aims to replicate core features of native task managers found in operating systems like Windows, Linux, and macOS, while also offering a clean and customizable implementation suitable for educational, experimental, or lightweight system utility purposes.

At its core, the project focuses on real-time process management. It provides a structured list of all currently active processes, displaying essential details such as process ID (PID), process name, CPU usage, memory consumption, priority level, and status (running, sleeping, or terminated). This allows users to observe how different applications and services are utilizing system resources. The task manager will also track dynamic behavior, such as changes in CPU usage or memory leaks, which are valuable for debugging and performance tuning.

Another significant aspect of the project is process control. The task manager enables users to take actions on specific processes, such as terminating them (kill), pausing them (suspend), or resuming paused processes (continue). This interactive control is especially useful in situations where a program becomes unresponsive or consumes excessive resources. By including functionality for changing process priorities (nice values in Unix-based systems), the project also introduces the concept of process scheduling and how priorities affect CPU allocation.

In terms of technical scope, the task manager is built to run on Unix-like systems and interacts with the operating system through standard system calls and APIs, such as reading from the `/proc` filesystem and using signals (`kill`, `SIGSTOP`, `SIGCONT`). The scope may also include multithreading for better performance, especially when handling frequent refreshes of process data using frameworks like `Tkinter`

# CHAPTER 3
# MODULES AND SURVEYS

The project is modular in design, consisting of several interconnected components that handle specific responsibilities. The Process Scanner Module retrieves a list of all currently running processes using system APIs or libraries like psutil. The Resource Monitor gathers CPU and memory usage data in real time.

The Process Controller enables users to perform actions like terminate (SIGKILL), pause (SIGSTOP), or resume (SIGCONT) on selected processes. The User Interface Module (CLI or GUI) presents this data in an organized and readable format. If implemented in GUI, it may include features like search bars and clickable actions. A Scheduler/Timer module may also be added to control refresh intervals. This modular structure improves readability, maintainability, and future expansion.

This project leverages a set of modern technologies to replicate core functions of a task manager. The primary programming language used is **Python**, chosen for its simplicity, readability, and broad system-level library support. For system monitoring, the psutil (Python System and Process Utilities) library is used to access process information, CPU/memory usage, and system details. In terminal-based versions, curses or rich may be used to design a dynamic text-based UI, while graphical versions can be developed using **Tkinter**

# CHAPTER 4
# SOFTWARE DESCRIPTION

The Task Manager software is a tool designed to help users monitor and control processes running on a system. It displays important information about each process, such as its Process ID (PID), name, CPU usage, memory usage, and status (whether the process is running, sleeping, or terminated). The software provides an easy-to-understand view of system resources and allows users to take action on running processes.

One of the key features of the Task Manager is its ability to control processes. Users can terminate unresponsive processes, pause tasks that are consuming too many resources, and resume paused processes. These actions are helpful for system administrators and users who need to manage processes without restarting their computers. Additionally, users can change the priority of certain processes, which can help optimize CPU usage.

The software is built to be cross-platform, working on Linux, macOS, and Windows. This is made possible by using Python, a flexible programming language, along with the `psutil` library, which works on all these operating systems. The Task Manager accesses system information in real time and provides users with an up-to-date display of the processes running on their machine.

The user interface is designed to be simple and intuitive. It can operate in two modes: a command-line interface (CLI), which updates process data in the terminal, and a graphical user interface (GUI), which is more user-friendly and allows users to interact with the processes through buttons and menus. The CLI is suitable for advanced users who prefer using the terminal, while the GUI is more accessible for those who prefer a visual approach.

In addition to process management, the Task Manager also offers real-time monitoring of system resources, such as CPU usage, memory usage, and disk activity. This helps users identify any performance issues and take action when resources are being overused.

# CHAPTER 5
# REQUIREMENT ANALYSIS

### Functional Requirements:

- List active system processes with details
- Terminate/suspend/resume processes by PID
- Log all process-related actions
- Visualize CPU/memory usage in real-time

## Non-functional Requirements:

- Simple and responsive interface
- Lightweight and efficient performance
- Compatible across major OS platforms

e

# CHAPTER 6
# HARDWARE AND SOFTWARE REQUIREMENTS

## Hardware Requirements:

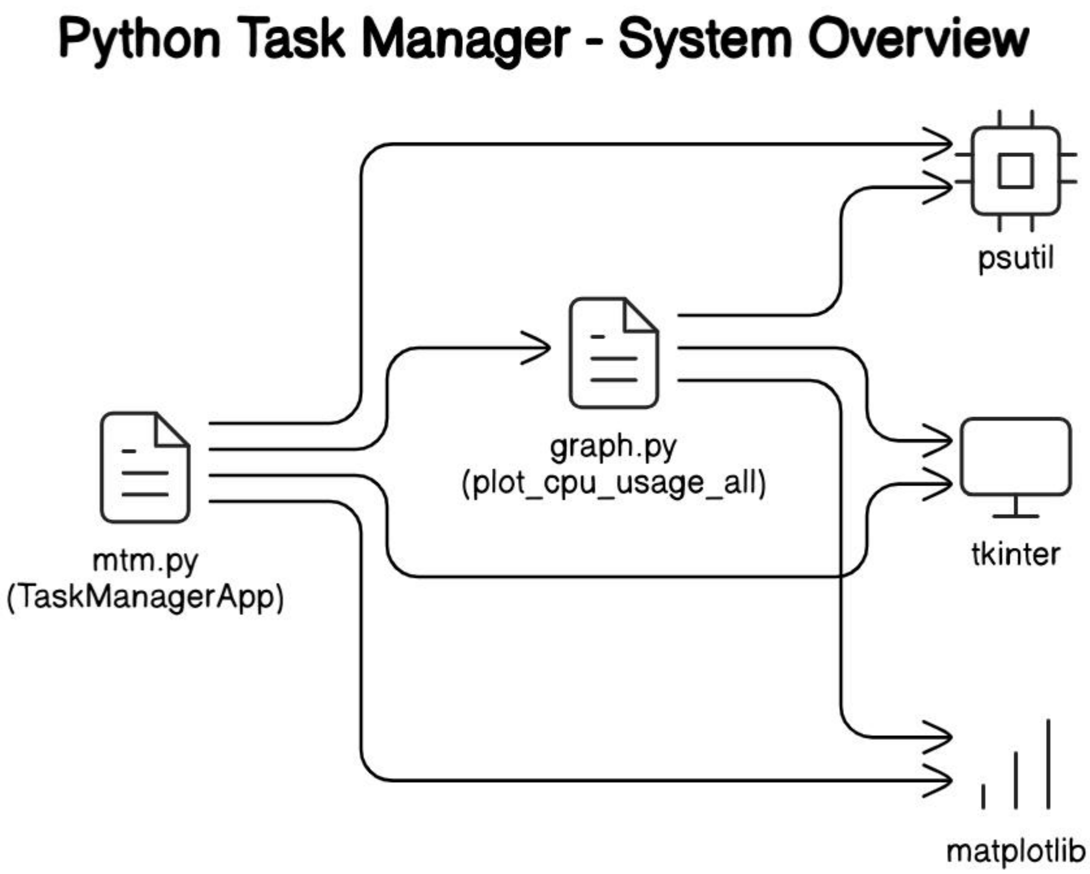
| Component | Requirement |
|---|---|
| Processor | Minimum 1.5 GHz CPU |
| Memory (RAM) | Minimum 2 GB RAM |
| Disk Space | Less than 50 MB for the software |
| Display | Minimum 1024x768 screen resolution |

## Software Requirements:

| Component | Requirement |
|---|---|
| Operating System | Linux (Ubuntu, Fedora, Debian), macOS, Windows 10 or newer |
| Python Version | Python 3.x (Python 3.6 or higher recommended) |
| Required Libraries | psutil, os, signal, Tkinter (or PyQt), curses (or rich) |
| Optional Software | IDE/Text Editor (e.g., VS Code, PyCharm), Terminal (for CLI version) |

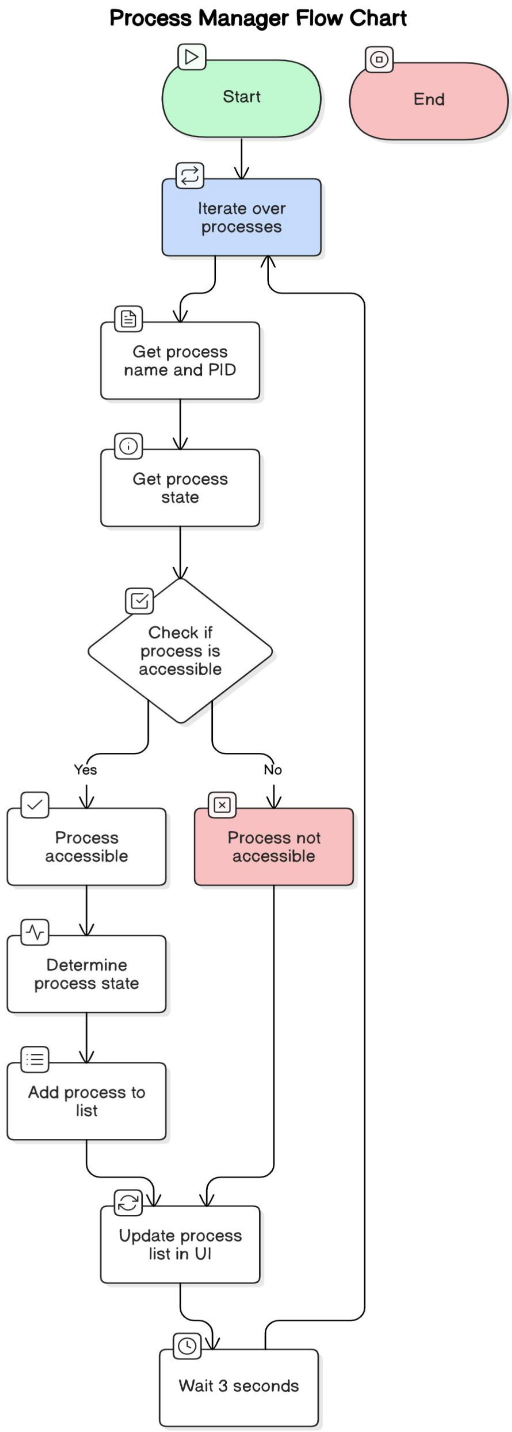
# CHAPTER 7
# ARCHITECTURE

The Task Manager application is designed to be simple and efficient. It uses **Tkinter** for the user interface, displaying processes in a **Treeview** with checkboxes for selection. Users can terminate processes, refresh the list, or view CPU usage through buttons. processes disappear.




The application relies on **psutil** to manage processes, checking CPU usage and handling errors like missing processes or permission issues. Every 3 seconds, it updates the process list, and when a process is selected, users can terminate it. The app also generates a CPU usage graph using **Matplotlib**. Error handling ensures the app runs smoothly even if there are access issues or

# CHAPTER 8
# FLOWCHART



# CHAPTER 9
# PROGRAM CODE

## 1.mtm.py

```
import tkinter as tk
from tkinter import ttk, messagebox
import psutil
from graph import plot_cpu_usage_all


class TaskManagerApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Python Task Manager with Checkboxes
& Graph")
        self.root.geometry("900x500")
        self.seen_pids = set()
        self.checked_pids = set()

        self.tree = ttk.Treeview(root, columns=("Checkbox",
"Name", "PID", "State"), show="headings")
        self.tree.heading("Checkbox", text="")
        self.tree.heading("Name", text="Application")
        self.tree.heading("PID", text="PID")
        self.tree.heading("State", text="State")

        self.tree.column("Checkbox", width=60,
anchor='center')
        self.tree.column("Name", width=350)
        self.tree.column("PID", width=100)
        self.tree.column("State", width=150)

        self.tree.pack(fill=tk.BOTH, expand=True, padx=10,
pady=10)
        self.tree.bind("<Button-1>", self.on_tree_click)

        button_frame = tk.Frame(root)
        button_frame.pack(pady=5)

        terminate_btn = tk.Button(button_frame,
text="Terminate Selected Process",
```

```
command=self.terminate_process) terminate_btn.pack(side=tk.LEFT,
padx=10)



     refresh_btn = tk.Button(button_frame, text="Refresh",
   command=self.refresh_process_list)
           refresh_btn.pack(side=tk.LEFT, padx=10)

           graph_btn = tk.Button(button_frame, text="Show CPU
   Usage Graph", command=plot_cpu_usage_all)
           graph_btn.pack(side=tk.LEFT, padx=10)
           self.update_process_list()
       def get_process_state(self, proc):
           try:
               status = proc.status()
               pid = proc.pid
               if pid not in self.seen_pids:
                   self.seen_pids.add(pid)
                   return "new"
               elif status == psutil.STATUS_RUNNING:
                   return "running"
               elif status in [psutil.STATUS_SLEEPING,
   psutil.STATUS_WAITING]:
                   return "waiting"
               elif status in [psutil.STATUS_ZOMBIE,
   psutil.STATUS_STOPPED, psutil.STATUS_DEAD]:
                   return "terminated"
               elif status == psutil.STATUS_IDLE:
                   return "idle"
               else:
                   return status
           except (psutil.NoSuchProcess, psutil.AccessDenied):
               return "terminated"

       def update_process_list(self):
           current_checked = set(self.checked_pids)
           self.tree.delete(*self.tree.get_children())
           self.checked_pids.clear()

           for proc in psutil.process_iter(['pid', 'name']):
               try:
                   name = proc.info['name']
                   pid = str(proc.info['pid'])
                   state = self.get_process_state(proc)
```

```
                checked = pid in current_checked
                checkbox = "[x]" if checked else "[ ]"

                if checked:


  self.checked_pids.add(pid)

                self.tree.insert("", tk.END, iid=pid,
values=(checkbox, name, pid, state))

            except (psutil.NoSuchProcess,
psutil.AccessDenied):
                continue
        self.root.after(3000, self.update_process_list)
    def refresh_process_list(self):
        self.update_process_list()

    def on_tree_click(self, event):
        region = self.tree.identify("region", event.x,
event.y)
        if region != "cell":
            return

        column = self.tree.identify_column(event.x)
        if column != "#1":
            return

        row = self.tree.identify_row(event.y)
        if not row:
            return

        pid = self.tree.item(row, "values")[2]
        if pid in self.checked_pids:
            self.checked_pids.remove(pid)
            self.tree.set(row, column="#1", value="[ ]")
        else:
            self.checked_pids.add(pid) self.tree.set(row,
            column="#1", value="[x]")

    def terminate_process(self):
        if not self.checked_pids:
            messagebox.showwarning("No Selection", "Please
check at least one process to terminate.")
            return
```

## 2.graph.py

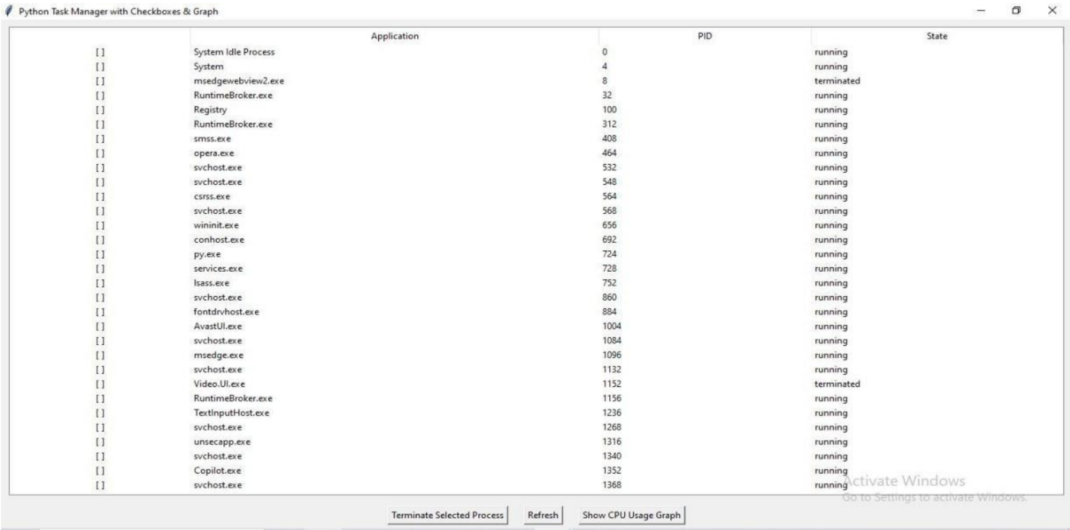
```
import psutil
import tkinter as tk
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg
import matplotlib.pyplot as plt
from matplotlib.figure import Figure

def plot_cpu_usage_all():
    process_list = []

    for proc in psutil.process_iter(['pid', 'name']):
        try:
            proc.cpu_percent(interval=None)
        except (psutil.NoSuchProcess, psutil.AccessDenied):
            continue

    psutil.cpu_percent(interval=0.1)

    for proc in psutil.process_iter(['pid', 'name']):
        try:
            cpu = proc.cpu_percent(interval=None)
            name = f"{proc.info['name']} (PID
{proc.info['pid']})"
            process_list.append((name, cpu))
        except (psutil.NoSuchProcess, psutil.AccessDenied):
            continue


if not process_list:
        print("No process data available.")
        return
    process_list.sort(key=lambda x: x[0])
    names = [proc[0] for proc in process_list]
    cpu_usages = [proc[1] for proc in process_list]

    window = tk.Toplevel()
    window.title("CPU Usage of All Processes")
    window.geometry("1000x600")

    frame = tk.Frame(window)
    frame.pack(fill=tk.BOTH, expand=True)

    canvas_frame = tk.Canvas(frame)
    canvas_frame.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)
```

```
    scrollbar = tk.Scrollbar(frame, orient=tk.HORIZONTAL,
command=canvas_frame.xview)
    scrollbar.pack(side=tk.BOTTOM, fill=tk.X)
    canvas_frame.configure(xscrollcommand=scrollbar.set)

    plot_frame = tk.Frame(canvas_frame)
    canvas_frame.create_window((0, 0), window=plot_frame,
anchor='nw')

    fig_width = max(10, len(names) * 0.4)
    fig = Figure(figsize=(fig_width, 6))

    ax = fig.add_subplot(111)
    ax.plot(names, cpu_usages, marker='o', linestyle='-',
color='blue')
    ax.set_xlabel("Process Name (PID)")
    ax.set_ylabel("CPU Usage (%)")
    ax.set_title("CPU Usage of All Running Processes")
    ax.grid(True)
    ax.tick_params(axis='x', rotation=90)
    fig.subplots_adjust(bottom=0.3)
    canvas = FigureCanvasTkAgg(fig, master=plot_frame)
    canvas_widget = canvas.get_tk_widget()
    canvas_widget.pack()



plot_frame.update_idletasks()
canvas_frame.config(scrollregion=canvas_frame.bbox("all"))
if __name__ == "__main__":
    root = tk.Tk()
    root.withdraw()
    plot_cpu_usage_all()
    root.mainloop()
```

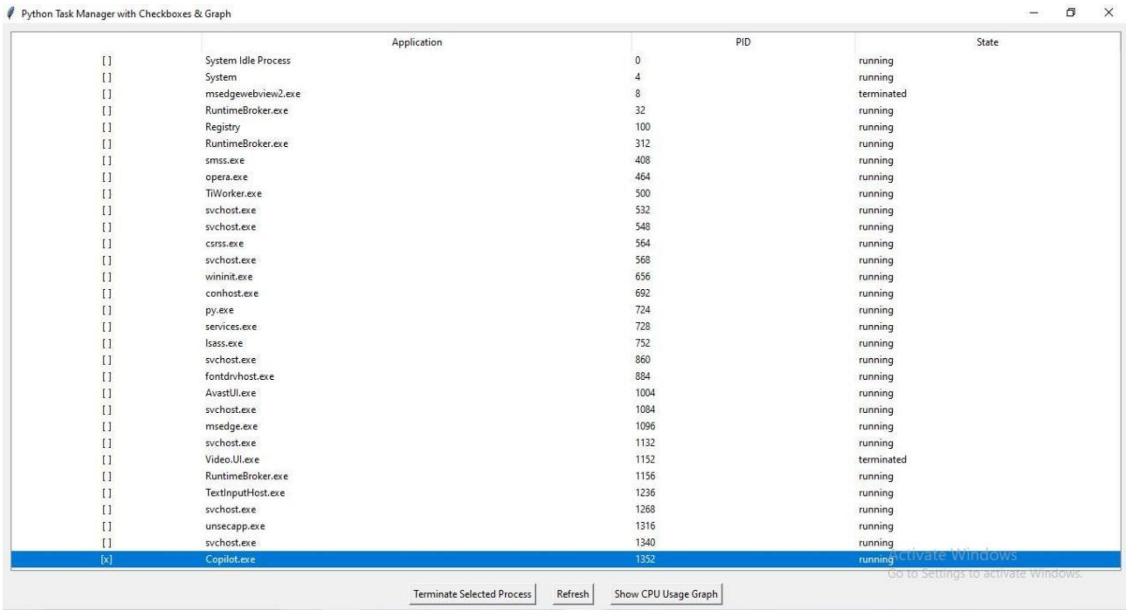
# CHAPTER 11
# RESULTS AND GRAPH

- The tool accurately retrieves and displays real-time process information.
- Termination and control of processes work reliably on supported OSes.
- Logs are generated correctly for audit purposes.
- CPU usage graphs are successfully created and displaye

# PICTURES

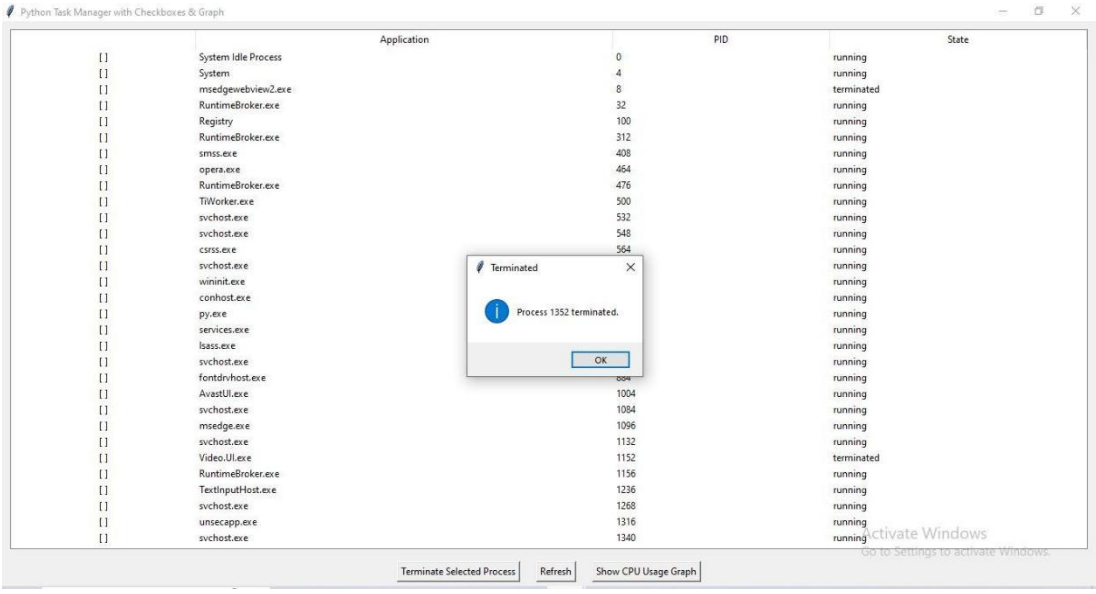



Working of Process Manager

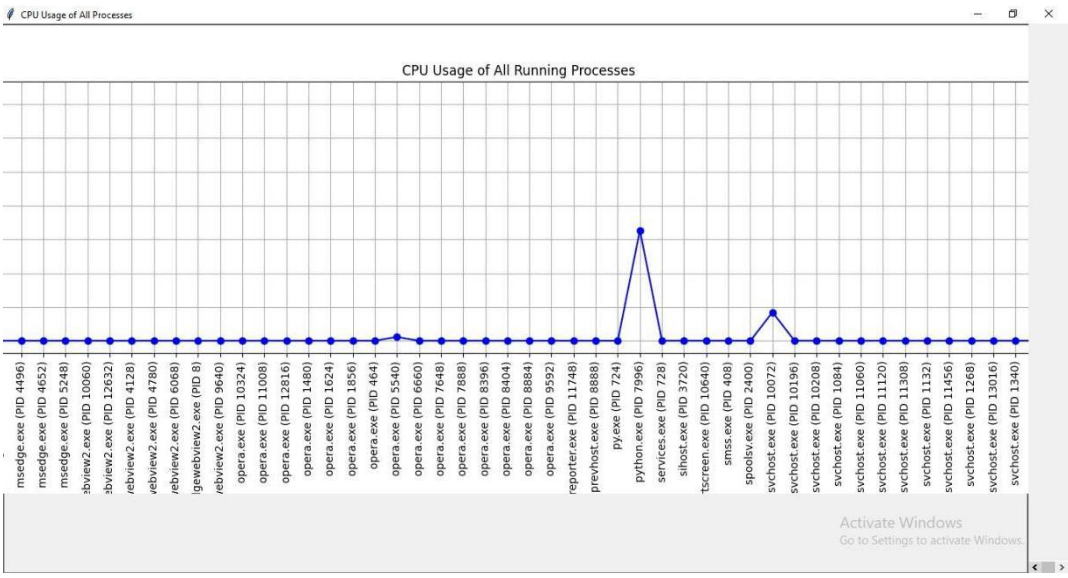



Sel

Selecting the process to terminate




Process gets terminated




GRAPH

# CHAPTER 11
# CONCLUSION

In conclusion, the Task Manager project provides a useful and educational tool for monitoring and managing system processes across different operating systems, including Linux, macOS, and Windows. By offering real-time insights into system performance, users can track resource usage, manage running processes, and take action to optimize system efficiency. The project demonstrates key concepts in system programming, such as process management, resource allocation, and system signals, all while being user-friendly and easily extendable.

The project is implemented in Python, leveraging libraries like `psutil` for system monitoring and offering both command-line and graphical interfaces for flexibility. It is designed to be lightweight, efficient, and easy to use, making it suitable for both novice and advanced users.

Moving forward, this Task Manager can be further enhanced with additional features, such as resource usage graphs, process filtering, and the ability to track system health over time. Overall, the Task Manager serves as both a practical tool for everyday use and an excellent learning resource for those interested in understanding how operating systems handle processes and system resources.

# CHAPTER 12
# REFERENCES


Tudor, J. (2016). *Python Programming for Beginners: A Step-by-Step Guide to Learning Python in 7 Days*. CreateSpace Independent Publishing.

Moore, A. D. (2018). *Python GUI Programming with Tkinter: Develop responsive and powerful GUI applications with Tkinter*. Packt Publishing.

Burns, S. (2020). *Python Data Visualization with Matplotlib: Unlock deeper insights into your data using simple but powerful Python libraries*. Independently published.