SIMPLE PROCESS MANAGER

MINI PROJECT REPORT

Submitted by

SANJAY K N (9517202309101)

JAYAVELAN S (9517202309041)

MAHESH BHOOPATHI C S (9517202309064)

in

23AD481 – OPERATING SYSTEM PRINCIPLES

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

MEPCO SCHLENK ENGINEERING COLLEGE

SIVAKASI

MAY 2025

MEPCO SCHLENK ENGINEERING COLLEGE, SIVAKASI

AUTONOMOUS

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

BONAFIDE CERTIFICATE

This is to certify that it is the bonafide work of SANJAY K N (95172023101), JAYAVELAN S (9517202309041), MAHESH BHOOPATHI C S (9517202309064) for the mini project titled **Simple Process Manager** in 23AD481 – Operating System Principles during the fourth semester January 2025 – May 2025 under my supervision.

SIGNATURE

Dr. P. Swathika,
Asst. Professor (Senior Grade),
AI&DS Department,
Mepco Schlenk Engg. College, Sivakasi

SIGNATURE

Dr. J. Angela Jennifa Sujana,
Professor & Head,
AI&DS Department
Mepco Schlenk Engg. College, Sivakasi

ABSTRACT

This project presents the development of a Mini Task Manager application using Python's Tkinter for the graphical user interface (GUI), providing users with an efficient and interactive way to monitor and manage system processes. The tool connects to a Unix-based process manager via a socket connection to retrieve and display real-time data on running processes, including details such as PID, State, Priority, and CPU usage. The application supports essential task management functions such as Pause, Resume, and Kill for selected processes, alongside priority adjustment. Additionally, the app features a responsive GUI, allowing users to seamlessly interact with process data, with color-coded tags to denote different process states (e.g., running, sleeping, zombie).

The main goal of this tool is to provide a lightweight, easy-to-use alternative for monitoring and managing system processes directly from the desktop. This report outlines the technical details of the application's development, the integration of socket communication, and the design choices made to ensure both functionality and user-friendly interaction. The project demonstrates the utility of Python, Tkinter, and socket programming in building robust system administration tools with graphical interfaces.

TABLE OF CONTENTS

Chapter 1
Introduction

1.1 Introduction

In today's digital age, efficient management and monitoring of system processes have become crucial for maintaining system health and performance. System administrators and users often need tools that can help them view, manage, and troubleshoot system processes in real-time. This project focuses on the development of a **Mini Task Manager**, a Python-based application designed to provide users with a graphical interface to monitor and control system processes. The tool is lightweight, user-friendly, and integrates socket programming for seamless communication with the underlying system. By leveraging the Tkinter library for the GUI, the project aims to present a comprehensive view of active processes, allowing users to perform essential functions such as pausing, resuming, killing, and setting priorities for processes directly from the interface.

1.2 Objectives

The main objectives of this project are as follows:

- **Development of a Task Manager**: To create a simple, interactive tool for managing system processes on Unix-based systems.
- **Real-Time Process Monitoring**: To implement functionality for retrieving live data about processes, including **PID**, **state**, and **priority**.
- **User-Friendly Interface**: To design an intuitive and responsive graphical user interface (GUI) that allows users to easily interact with system processes.
- **Process Control**: To enable users to control system processes by pausing, resuming, or killing them, and adjusting their priorities.
- **Socket Communication**: To utilize socket programming for communication between the application and the process manager, enabling efficient data exchange.

1.3 Scope of the Project

The **Mini Task Manager** is designed to provide basic functionality for process management in Unix-based systems. It is intended for users who need to monitor and manage processes without the complexity of more advanced tools. The scope of the project includes:

- **Viewing Processes**: Displaying system processes with relevant details such as **PID**, **state**, and **priority**.
- **Managing Processes**: Allowing the user to pause, resume, kill, and adjust the priority of processes.
- **User Interaction**: Providing a simple GUI to interact with the system processes.
- **Socket-Based Communication**: Using socket programming to interact with the Unix system's process manager for real-time data retrieval and manipulation.

This project does not cover complex functionalities like resource consumption monitoring, multi-platform support, or advanced process management, as it is focused on providing a lightweight alternative to traditional task management tools.

1.4 Proposed System

The proposed system consists of the following key components:

1. **Graphical User Interface (GUI)**: A Tkinter-based GUI that displays system processes in a treeview format with relevant process information, including **PID**, **State**, and **Priority**. The interface is color-coded to represent different process states, making it visually intuitive for users.
2. **Socket Communication**: The system uses socket programming to establish a connection with a Unix-based process manager, retrieving real-time data about running processes.
3. **Process Management Functions**: The user can interact with processes, pausing, resuming, killing, or modifying their priority through the interface.
4. **Refresh Interval**: The process list is automatically updated at regular intervals to provide real-time information.

The system is designed with simplicity in mind and aims to make process management more accessible to users, particularly those without advanced knowledge of system administration.

1.5 System Design

The system design is divided into the following components:

1. **User Interface (UI)**: The GUI is built using Tkinter, Python's standard library for creating desktop applications. It features a treeview that displays process details like **PID**, **State**, and **Priority**. Each process is color-coded based on its current state, making it easier to distinguish between different types of processes (e.g., running, sleeping, stopped, zombie).
2. **Backend**: The backend of the system uses **socket programming** to interact with the process manager. The application sends commands to the process manager via a Unix domain socket and receives the response, which is then displayed on the GUI.
3. **Process Management**: The application implements basic process management operations such as:
   - **Pause**: Suspends the execution of a selected process.
   - **Resume**: Resumes the execution of a paused process.
   - **Kill**: Terminates a selected process.
   - **Set Priority**: Allows the user to adjust the priority of a process.
4. **Threading**: To ensure that the user interface remains responsive while the application communicates with the process manager, **threading** is used to handle the process list update and user interactions in parallel.
5. **System Calls Used**:

   \>>fdopen(int fildes, const char *mode);

   \>>char *strtok(char *str, const char *delims);

   \>>int fflush(FILE* stream);

   \>>int unlink(const char *pathname);

   \>>int FD_ISSET(int fd, fd_set *set);

   \>>void FD_SET(int fd, fd_set *set);

   \>>void FD_ZERO(fd_set *set);

   \>>fork();

   \>>SIGSTOP, SIGCONT, SIGKILL, SIGINIT, SIGTERM

CHAPTER 2

IMPLEMENTATION

2.1 – Program Coding

Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <unistd.h>
#include <signal.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <sys/resource.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/types.h>

#define SOCKET_PATH    "/tmp/manager_socket"
#define PID_FILE       "/tmp/producer_pids"
#define PIPE_PATH      "/tmp/producer_pipe"
#define MAX_PRODUCERS  1024

static int read_pids(pid_t *pids) {
    FILE *fp = fopen(PID_FILE, "r");
    if (!fp) return 0;
    int count = 0;
    while (count < MAX_PRODUCERS && fscanf(fp, "%d\n", &pids[count]) == 1)
count++;
    fclose(fp);
    return count;
}

static char get_state(pid_t pid) {
    char path[64], buf[256];
    snprintf(path, sizeof(path), "/proc/%d/stat", pid);
    FILE *fp = fopen(path, "r");
    if (!fp) return '?';
    fscanf(fp, "%*d (%*[^)]) %c", &buf[0]);
    fclose(fp);
    return buf[0];
}

static const char* state_str(char s) {
    switch (s) {
        case 'R': return "Running";
        case 'S': return "Sleeping";
        case 'D': return "Uninterruptible";
        case 'T': case 't': return "Stopped";
        case 'Z': return "Zombie";
        default:  return "Unknown";
    }
}

static void handle_client(int client) {
    FILE *in = fdopen(client, "r");
    FILE *out = fdopen(dup(client), "w");
    if (!in || !out) { close(client); return; }
    char line[128];
    while (fgets(line, sizeof(line), in)) {
        char *cmd = strtok(line, " \t\n");
        if (!cmd) continue;

        if (strcmp(cmd, "LIST") == 0) {
            pid_t pids[MAX_PRODUCERS];
            int n = read_pids(pids);
            for (int i = 0; i < n; i++) {
                if (kill(pids[i], 0) == -1 && errno == ESRCH) continue;
                char s = get_state(pids[i]);
                int prio = getpriority(PRIO_PROCESS, pids[i]);
                fprintf(out, "%d %s %d\n", pids[i], state_str(s), prio);
            }
            fprintf(out, "END\n");
            fflush(out);
        }
        else if (strcmp(cmd, "PAUSE") == 0 || strcmp(cmd, "RESUME") == 0 || strcmp(cmd,
"KILL") == 0) {
            pid_t pid = (pid_t)atoi(strtok(NULL, " \t\n"));
            int sig = strcmp(cmd, "PAUSE") == 0 ? SIGSTOP :
                      strcmp(cmd, "RESUME") == 0 ? SIGCONT : SIGKILL;
            if (kill(pid, sig) == 0) fprintf(out, "OK\n");
            else fprintf(out, "ERROR %s\n", strerror(errno));
            fflush(out);
        }
        else if (strcmp(cmd, "PRIORITY") == 0) {
            pid_t pid = (pid_t)atoi(strtok(NULL, " \t\n"));
            int pr = atoi(strtok(NULL, " \t\n"));
            if (setpriority(PRIO_PROCESS, pid, pr) == 0) fprintf(out, "OK\n");
            else fprintf(out, "ERROR %s\n", strerror(errno));
            fflush(out);
        }
    }
    fclose(in);
    fclose(out);
}

int main(void) {
    signal(SIGPIPE, SIG_IGN);

    unlink(SOCKET_PATH);
    unlink(PIPE_PATH);

    int sock = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sock < 0) { perror("socket"); exit(1); }

    struct sockaddr_un addr = { .sun_family = AF_UNIX };
    strncpy(addr.sun_path, SOCKET_PATH, sizeof(addr.sun_path) - 1);

    if (bind(sock, (struct sockaddr*)&addr, sizeof(addr)) < 0) {
        perror("bind");
        exit(1);
    }

    chmod(SOCKET_PATH, 0666);

    if (listen(sock, 5) < 0) {
        perror("listen");
        exit(1);
    }
```

```
// Create the named pipe
if (mkfifo(PIPE_PATH, 0666) == -1 && errno != EEXIST) {
    perror("mkfifo");
    exit(1);
}

printf("Manager listening on %s\n", SOCKET_PATH);

int pipe_fd = open(PIPE_PATH, O_RDONLY | O_NONBLOCK);
if (pipe_fd == -1) {
    perror("open pipe");
    exit(1);
}

fd_set fds;
int max_fd = (sock > pipe_fd ? sock : pipe_fd) + 1;

while (1) {
    FD_ZERO(&fds);
    FD_SET(sock, &fds);
    FD_SET(pipe_fd, &fds);

    if (select(max_fd, &fds, NULL, NULL, NULL) > 0) {
        if (FD_ISSET(sock, &fds)) {
            int client = accept(sock, NULL, NULL);
            if (client >= 0) {
                pid_t pid = fork();
                if (pid == 0) {
                    close(sock);
                    handle_client(client);
                    close(client);
                    exit(0);
                }
                close(client);
            }
        }
        if (FD_ISSET(pipe_fd, &fds)) {
            char buf[128];
            int bytes = read(pipe_fd, buf, sizeof(buf) - 1);
            if (bytes > 0) {
                buf[bytes] = '\0';
                printf("Received from producer: %s", buf);
            }
        }
    }
  }
}
```

Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
#include <signal.h>

// PID file path and pipe path
#define PID_FILE "/tmp/producer_pids"
#define PIPE_PATH "/tmp/producer_pipe"
```

```
void remove_pid(pid_t pid) {
    FILE *fp = fopen(PID_FILE, "r+");
    if (!fp) {
        fprintf(stderr, "Failed to open PID file for cleanup: %s\n", strerror(errno));
        return;
    }

    char buffer[4096];
    size_t size = 0;
    pid_t temp_pid;

    while (fscanf(fp, "%d\n", &temp_pid) == 1) {
        if (temp_pid != pid) {
            size += snprintf(buffer + size, sizeof(buffer) - size, "%d\n", temp_pid);
        }
    }

    freopen(PID_FILE, "w", fp);
    fwrite(buffer, 1, size, fp);
    fclose(fp);
}

void cleanup(int signum) {
    pid_t pid = getpid();
    printf("Cleaning up PID %d...\n", pid);
    remove_pid(pid);
    exit(0);
}

int main(int argc, char *argv[]) {
```

```
const char *name = argc > 1 ? argv[1] : "Producer";

signal(SIGTERM, cleanup);
signal(SIGINT, cleanup);

FILE *fp = fopen(PID_FILE, "a");
if (!fp) {
   fprintf(stderr, "Failed to open PID file %s: %s\n", PID_FILE, strerror(errno));
   exit(EXIT_FAILURE);
}
pid_t pid = getpid();
fprintf(fp, "%d\n", pid);
fclose(fp);

// Open the pipe
int pipe_fd;
while ((pipe_fd = open(PIPE_PATH, O_WRONLY)) == -1) {
   perror("Waiting for manager to create pipe");
   sleep(1);
}

int counter = 0;
while (1) {
   dprintf(pipe_fd, "%d\n", counter);  // send number to pipe
   printf("%s (PID %d): running (%d)\n", name, pid, counter);
   fflush(stdout);
   counter++;
   sleep(1);
}

close(pipe_fd);
```

```
    return 0;
}
```

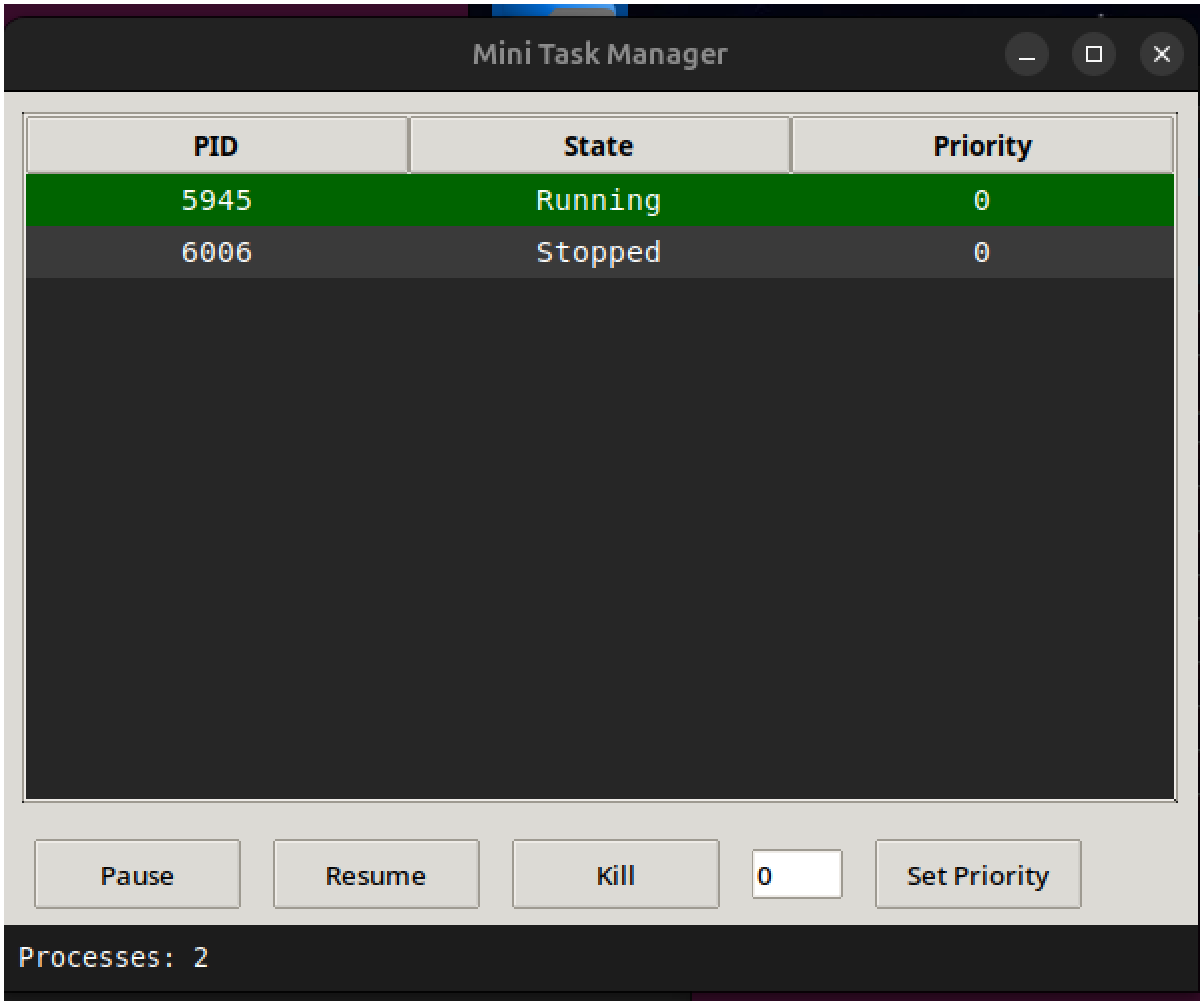
Gui

```
import tkinter as tk
from tkinter import ttk, messagebox
import socket
import threading

SOCKET_PATH = "/tmp/manager_socket"
REFRESH_INTERVAL = 500  # ms

def send_command(cmd):
    try:
        sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
        sock.settimeout(1)
        sock.connect(SOCKET_PATH)
        sock.sendall((cmd + "\n").encode())
        data = b""
        while True:
            try:
                chunk = sock.recv(1024)
                if not chunk:
                    break
                data += chunk
            except socket.timeout:
                break
        sock.close()
        return data.decode()
    except Exception:
        return None

def update_list_async():
    raw = send_command("LIST")
    if not raw or raw.strip() == "":
        return
    rows = [line.split() for line in raw.splitlines() if line and line != 'END']

    sel = tree.selection()
    selected = tree.item(sel[0])['values'][0] if sel else None

    def ui():
        tree.delete(*tree.get_children())
        for pid, state, prio in rows:
            display_state = "Running" if state.lower() == "sleeping" else state
            display_tag = "running" if state.lower() == "sleeping" else state.lower()
            tree.insert('', 'end', values=(pid, display_state, prio), tags=(display_tag,))
        if selected:
            for iid in tree.get_children():
                if tree.item(iid)['values'][0] == selected:
                    tree.selection_set(iid)
                    break
        status_var.set(f"Processes: {len(rows)}")

    root.after(0, ui)

def schedule_update():
    threading.Thread(target=update_list_async, daemon=True).start()
    root.after(REFRESH_INTERVAL, schedule_update)

def get_selected_pid():
    sel = tree.selection()
    if not sel:
        messagebox.showwarning("Select PID", "Please select a process.")
        return None
    return tree.item(sel[0])['values'][0]

def pause_proc():
    pid = get_selected_pid()
    if pid:
        threading.Thread(target=lambda: send_command(f"PAUSE {pid}"),
daemon=True).start()

def resume_proc():
    pid = get_selected_pid()
    if pid:
        threading.Thread(target=lambda: send_command(f"RESUME {pid}"),
daemon=True).start()

def kill_proc():
    pid = get_selected_pid()
    if pid:
        threading.Thread(target=lambda: send_command(f"KILL {pid}"),
daemon=True).start()

def set_priority():
    pid = get_selected_pid()
    pr = prio_entry.get()
    if pid and pr.isdigit():
        threading.Thread(target=lambda: send_command(f"PRIORITY {pid} {pr}"),
daemon=True).start()
    else:
        messagebox.showwarning("Priority", "Enter a valid integer priority.")

def show_context_menu(event):
    iid = tree.identify_row(event.y)
    if iid:
        tree.selection_set(iid)
        menu.post(event.x_root, event.y_root)

# ------------------------- UI -------------------------------

root = tk.Tk()
root.title("Mini Task Manager")
root.geometry("600x450")
root.configure(bg="#1c1c1c")

style = ttk.Style()
style.theme_use("clam")
style.configure("Treeview",
                background="#262626",
                foreground="white",
                fieldbackground="#262626",
                rowheight=26,
                font=('Consolas', 11))
style.map("Treeview",
          background=[('selected', '#3a3a3a')],
          foreground=[('selected', 'white')])

frame = ttk.Frame(root, padding=10)
frame.pack(fill='both', expand=True)

tree = ttk.Treeview(frame, columns=('PID', 'State', 'Priority'), show='headings')
```

```
for col in ('PID', 'State', 'Priority'):
    tree.heading(col, text=col, anchor='center')
    tree.column(col, anchor='center', width=150)

# Tag colors
tree.tag_configure('running', background='#006400')      # dark green for running
tree.tag_configure('sleeping', background='#333333')      # gray for original sleeping (won't
be used now)
tree.tag_configure('stopped', background='#8B8000')       # darker yellow
tree.tag_configure('zombie', background='#8B0000')        # dark red
tree.tag_configure('unknown', background='#555555')       # dark gray
tree.pack(fill='both', expand=True)

menu = tk.Menu(root, tearoff=0)
menu.add_command(label="Pause",  command=pause_proc)
menu.add_command(label="Resume", command=resume_proc)
menu.add_command(label="Kill",   command=kill_proc)
menu.add_separator()
menu.add_command(label="Set Priority", command=lambda: prio_entry.focus_set())
tree.bind("<Button-3>", show_context_menu)

btn_frame = ttk.Frame(root, padding=8)
btn_frame.pack(fill='x')
for text, cmd in [("Pause", pause_proc), ("Resume", resume_proc), ("Kill", kill_proc)]:
    ttk.Button(btn_frame, text=text, command=cmd).pack(side='left', padx=8)
prio_entry = ttk.Entry(btn_frame, width=5)
prio_entry.pack(side='left', padx=8)
prio_entry.insert(0, '0')
ttk.Button(btn_frame, text="Set Priority", command=set_priority).pack(side='left',
padx=8)
```

```
status_var = tk.StringVar(value="Connecting to manager...")
status = ttk.Label(root, textvariable=status_var, anchor='w',
                   background="#1c1c1c", foreground='white', padding=6,
                   font=('Consolas', 10))
status.pack(fill='x', side='bottom')

root.after(100, schedule_update)
root.mainloop()
```

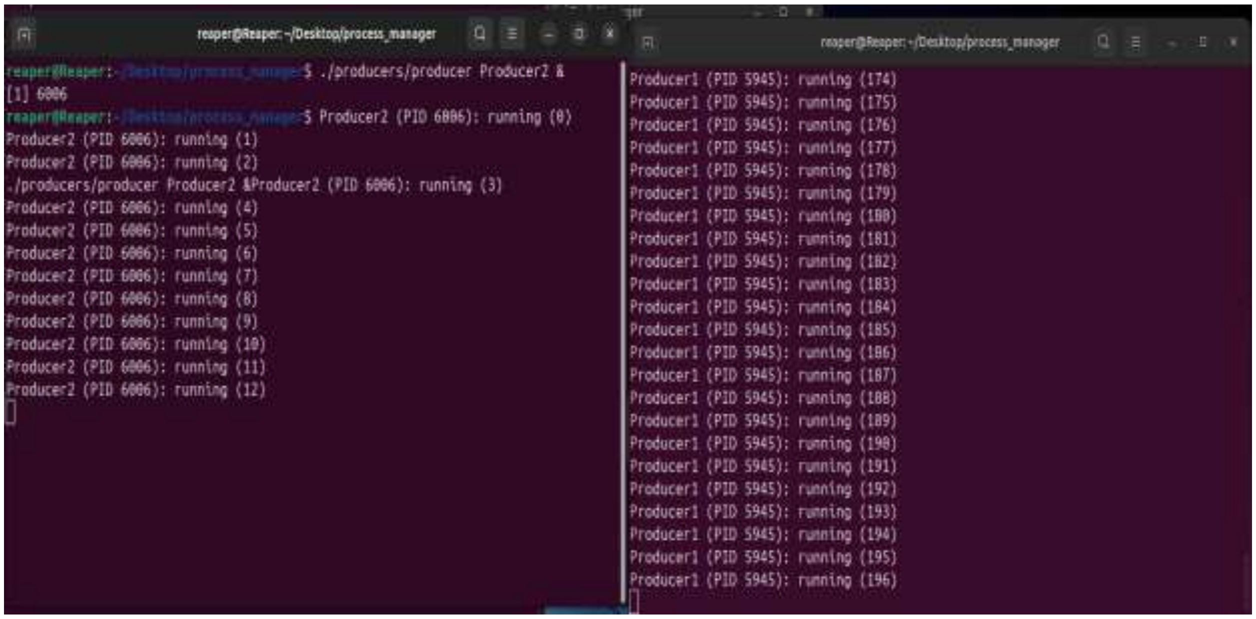
2.2 – Output



PID	State	Priority
5945	Running	0
6006	Stopped	0

```
Processes: 2
```

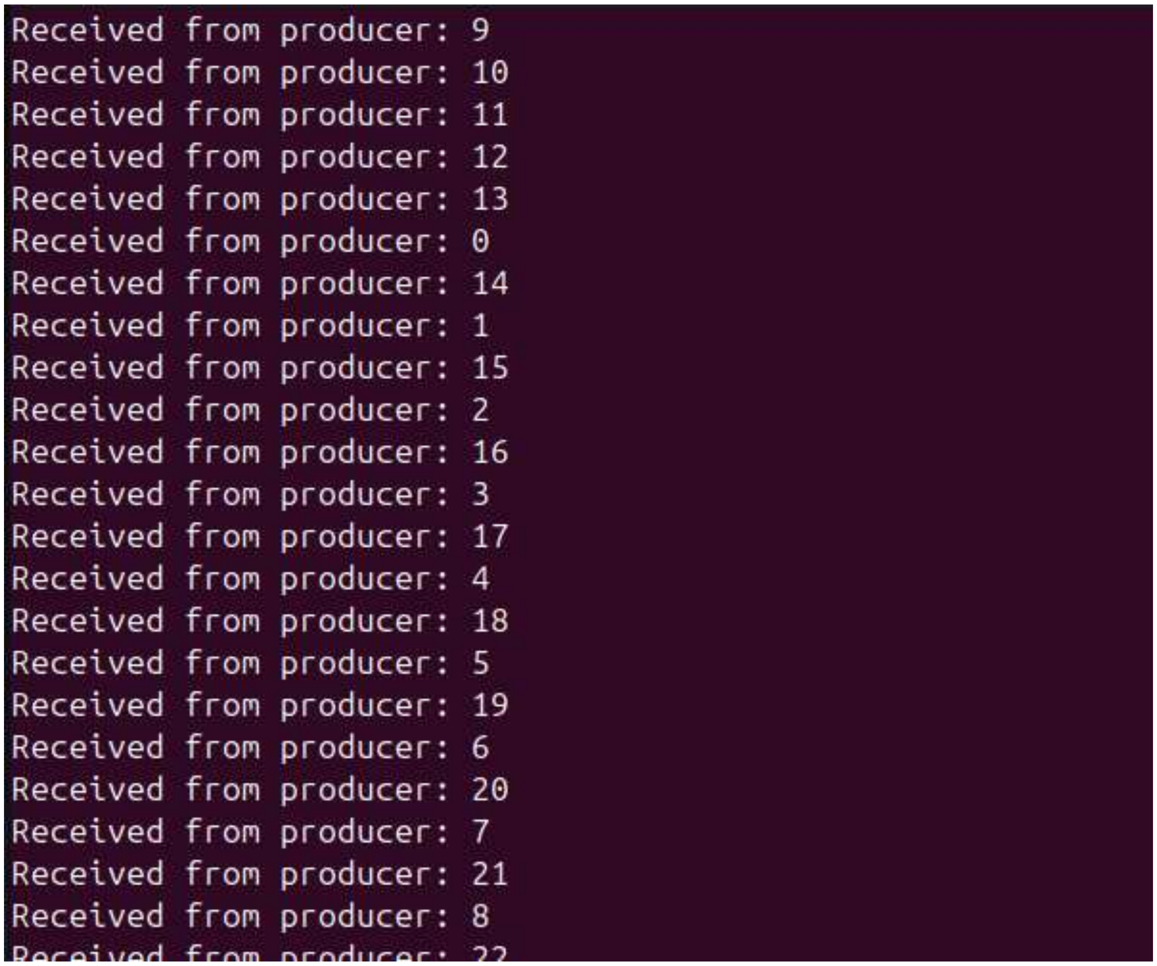
Figure 2.1 : User Interface



```
reaper@Reaper: ~/Desktop/process_manager
reaper@Reaper:~/Desktop/process_manager$ ./producers/producer Producer2 &
[1] 6006
reaper@Reaper:~/Desktop/process_manager$ Producer2 (PID 6006): running (0)
Producer2 (PID 6006): running (1)
Producer2 (PID 6006): running (2)
./producers/producer Producer2 &Producer2 (PID 6006): running (3)
Producer2 (PID 6006): running (4)
Producer2 (PID 6006): running (5)
Producer2 (PID 6006): running (6)
Producer2 (PID 6006): running (7)
Producer2 (PID 6006): running (8)
Producer2 (PID 6006): running (9)
Producer2 (PID 6006): running (10)
Producer2 (PID 6006): running (11)
Producer2 (PID 6006): running (12)
```

```
reaper@Reaper: ~/Desktop/process_manager
Producer1 (PID 5945): running (174)
Producer1 (PID 5945): running (175)
Producer1 (PID 5945): running (176)
Producer1 (PID 5945): running (177)
Producer1 (PID 5945): running (178)
Producer1 (PID 5945): running (179)
Producer1 (PID 5945): running (180)
Producer1 (PID 5945): running (181)
Producer1 (PID 5945): running (182)
Producer1 (PID 5945): running (183)
Producer1 (PID 5945): running (184)
Producer1 (PID 5945): running (185)
Producer1 (PID 5945): running (186)
Producer1 (PID 5945): running (187)
Producer1 (PID 5945): running (188)
Producer1 (PID 5945): running (189)
Producer1 (PID 5945): running (190)
Producer1 (PID 5945): running (191)
Producer1 (PID 5945): running (192)
Producer1 (PID 5945): running (193)
Producer1 (PID 5945): running (194)
Producer1 (PID 5945): running (195)
Producer1 (PID 5945): running (196)
```

Figure 2.2 : Producers creating new numbers



```
Received from producer: 9
Received from producer: 10
Received from producer: 11
Received from producer: 12
Received from producer: 13
Received from producer: 0
Received from producer: 14
Received from producer: 1
Received from producer: 15
Received from producer: 2
Received from producer: 16
Received from producer: 3
Received from producer: 17
Received from producer: 4
Received from producer: 18
Received from producer: 5
Received from producer: 19
Received from producer: 6
Received from producer: 20
Received from producer: 7
Received from producer: 21
Received from producer: 8
Received from producer: 22
```

Figure 2.3 : Manager receiving Numbers

Chapter 3

Conclusion

In conclusion, the **Mini Task Manager** application provides a simple yet powerful solution for managing and monitoring system processes on Unix-based systems. By leveraging Python's Tkinter library for the GUI and socket programming for real-time communication with the process manager, the application delivers essential functionality for process management in an intuitive and user-friendly interface. The ability to view, control, and modify processes directly from the application enhances the user experience and provides an accessible tool for users who may not be familiar with more complex system management tools.

The project successfully demonstrates the integration of graphical interfaces and system-level programming, offering a lightweight alternative to traditional task management applications. While this tool is designed with simplicity in mind, it has the potential to be extended further with additional features, such as resource consumption monitoring or support for multiple platforms. The application's modular design ensures that it can be easily adapted or expanded as needed to meet future requirements.

The project highlights the power of Python and its libraries in building efficient and effective system tools, providing a valuable learning experience in both **GUI development** and **system administration**.

References

1. **Python Documentation**: Official Python documentation, available at https://docs.python.org.
2. **Tkinter Documentation**: Tkinter documentation for Python, available at https://docs.python.org/3/library/tkinter.html.
3. **Socket Programming in Python**: A comprehensive guide to socket programming in Python, available at https://realpython.com/python-sockets/.
4. **Unix Manual Pages**: Official Unix man pages for system commands and processes, available at https://man7.org/linux/man-pages/.
5. **Python Threading**: Guide on threading in Python, available at https://docs.python.org/3/library/threading.html.