** Air University, Aerospace and Aviation Campus, Kamra**

**Department of Computer Science**

**BS. Cyber Security**

CS325L: Operating Systems Lab Semester Lab Project Report BSCYS – IV

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Project Title** | Advanced Process & Thread Management System | | | |
| **Lab No.** | **13** | | | |
| **Semester** | BS(CYS) – IV (Spring 2025) | | | |
| **Project ID** | 10 | | | |
|  | 1. Zunaira Shafqat | (235051) | (Leader) | Sign |
| **Name of Group** | 2. Hajra | (235063) |  | Sign |
| **Members** |  |  |  |  |
|  | 3. Fakhra | (235057) |  | Sign |
|  | 4. Maimoona | (235056) |  | Sign |
| **Date of Submission** | Monday, May 26, 2025 | | | |
| **Date of**  **Presentation** | Tuesday, May 27, 2025 | | | |
| **Submitted To** | **Waqar Azeem**  Lab Engineer (Cyber Security) Department of Computer Science, AACK | | | |
| **Marks Obtained** | **/ 20** | | | |
| **Remarks & Signature by Instructor:** |  | | | |

Table of Contents:

[1. Introduction 3](#_Toc199163369)

[2. Scope of Work 3](#_Toc199163370)

[3. Experimental Setup 4](#_Toc199163371)

[4. Implementation Details 5](#_Toc199163372)

[5. Experimentation & Results 5](#_Toc199163373)

[6. Results: 7](#_Toc199163374)

[6. Conclusion / Lessons Learnt 9](#_Toc199163375)

[7. References: 10](#_Toc199163376)

# Introduction

**Overview:**

This project focuses on implementing a **Process and Thread Management System** using the C language under a Linux environment. It covers critical operating system (OS) concepts such as process management, thread management, inter-process communication (IPC), synchronization primitives, and signal handling. The project simulates core OS functionalities, making it a practical exercise in systems programming.

**Purpose:**

The purpose is to deepen understanding of:

* OS-level process lifecycle and scheduling.
* POSIX-compliant multithreading models.
* Classic and modern IPC techniques.
* Race condition prevention through proper synchronization.
* Robust system design involving signal trapping and logging.

**Objectives:**

* Demonstrate fork(), exec(), wait(), kill() for process management.
* Implement POSIX threads with various synchronization strategies.
* Simulate real-world IPC mechanisms like pipes, message queues, and shared memory.
* Apply synchronization constructs like mutexes, semaphores, and condition variables.
* Practice robust programming by implementing signal handlers and loggers.

# Scope of Work

**Functional Modules Implemented**

1. **Process Management**
   * Creation and termination of child processes.
   * Display of process hierarchy tree using recursive logic.
2. **Thread Management**
   * Standard threads with shared memory counters.
   * Threads coordinated with condition variables and mutexes.
3. **Inter-Process Communication**
   * Anonymous pipes.
   * Named pipes (FIFOs).
   * Shared memory segments **(**shmget(), shmat(), shmdt()**)**.
   * UNIX domain sockets.
   * POSIX message queues.
4. **Synchronization Mechanisms**
   * POSIX semaphores **(**sem\_init(), sem\_wait(), sem\_post()**)**.
   * Mutex locks **(**pthread\_mutex\_lock() / unlock()**)**.
   * Condition variables for producer-consumer problems.
5. **Utilities and Logging**
   * Graceful exit with signal handling (SIGINT, SIGTERM).
   * System activity logs are written to process\_manager.log.

# Experimental Setup

**Hardware and Software Requirements**

|  |  |
| --- | --- |
| **Component** | **Specification** |
| OS | Ubuntu 20.04 / Kali Linux |
| Language | C with POSIX support |
| Compiler | GCC (GNU Compiler Collection) |
| Debugging Tool | GDB |
| Additional | Make utility for build automation |

**Key Libraries**

* <pthread.h> for thread APIs
* <semaphore.h>, <fcntl.h> for synchronization and file descriptors
* <sys/ipc.h>, <sys/shm.h>, <sys/msg.h> for IPC
* <signal.h> for signal handling

**Configuration** **Steps**:

* Install build tools: sudo apt install build-essential
* Compile code: gcc -o process\_manager project.c -lpthread
* Run with: ./process\_manager
* Log file process\_manager.log is created automatically

# Implementation Details

**Key** **Snippets**:

* fork() for processes
* pthread\_create() for threads
* sem\_init, sem\_wait, sem\_post for semaphores
* shmget, shmat, shmdt for shared memory
* pthread\_cond\_wait, pthread\_cond\_signal for CVs
* Logging via log\_msg() wrapper

**Challenges**

* Handling race conditions in shared memory.
* Avoiding deadlocks with improper mutex usage.
* Cleaning up semaphores and shared memory using shmctl() and sem\_destroy().
* Graceful termination of multiple threads using pthread\_cancel().

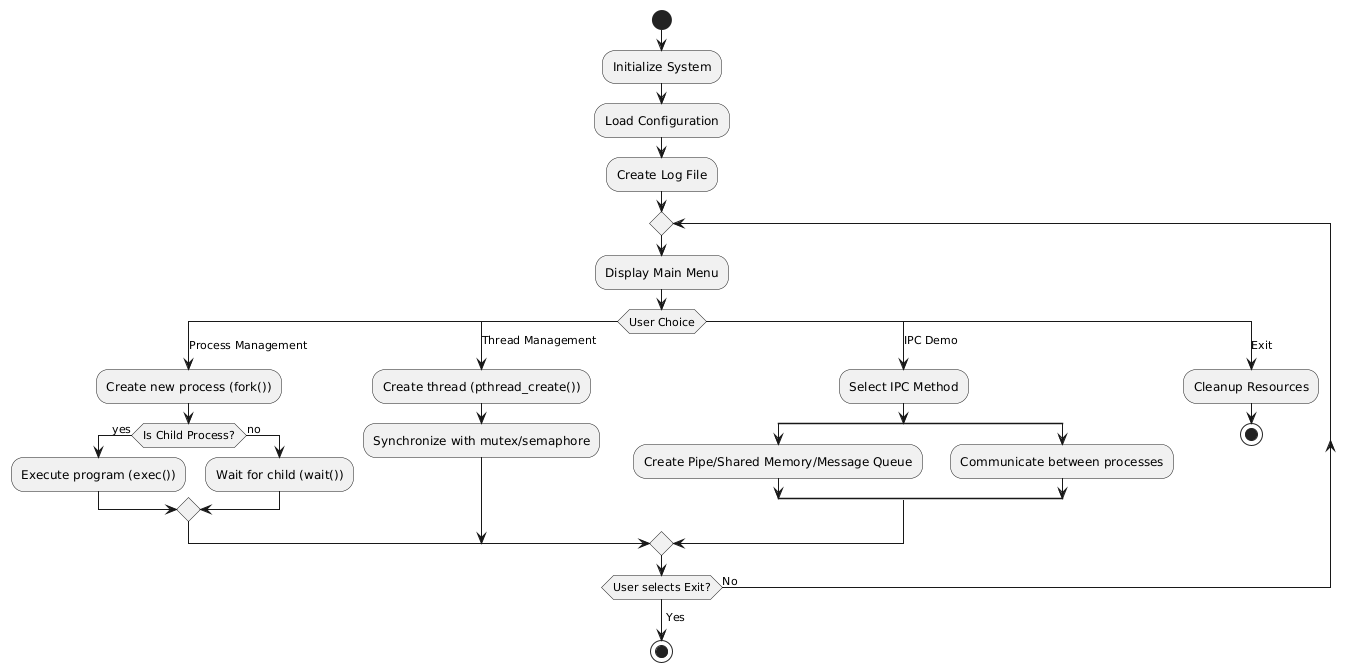
### **Experimentation & Results**

#### **Experimental Verification:**

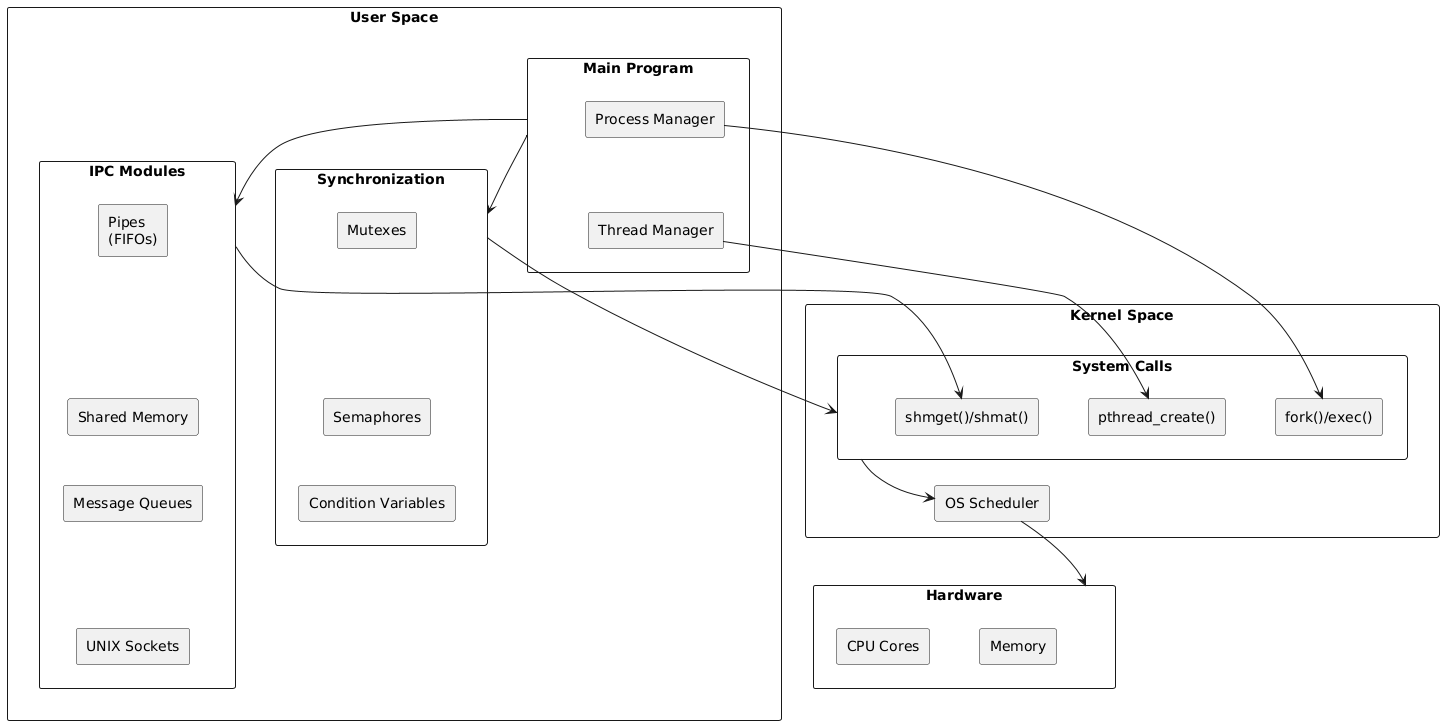
|  |  |
| --- | --- |
| **Feature** | **Observation** |
| Process Tree | Displays parent-child structure accurately. |
| Threading | Handles multiple threads with shared counters. |
| Shared Memory | Memory blocks are shared among processes. |
| Message Queues | Messages passed with prioritization. |
| Condition Variables | Correctly synchronizes thread execution. |
| Signal Handling | The program exits gracefully on Ctrl+C. |

**Working Methodology:**

**Flowchart:**



**Architecture diagram:**

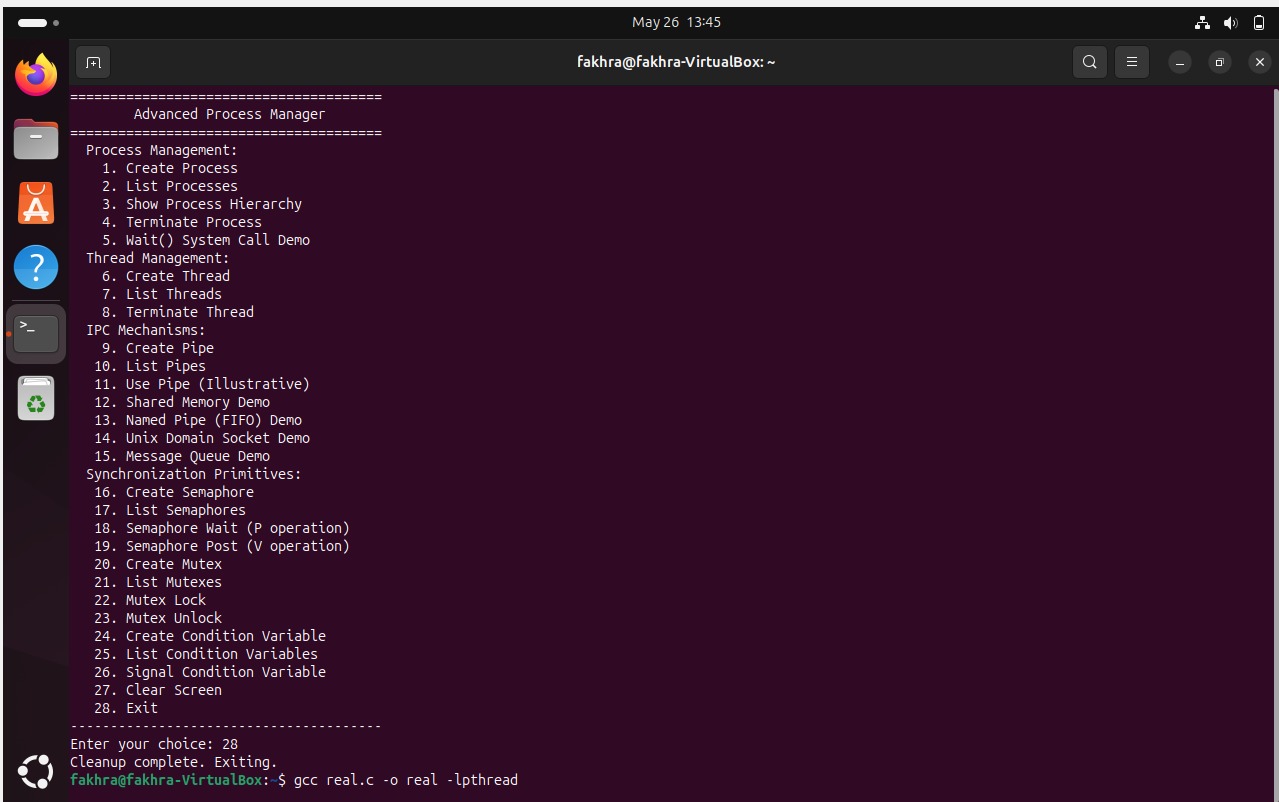


### **Results:**

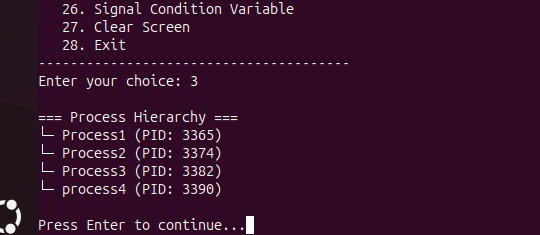
Key Observations:

* Process creation with fork() and hierarchy displayed correctly
* Threads respond to cancellation signals properly
* Shared counter works across threads within a process
* The shared memory demo reads/writes correctly
* Message queue and named pipe demos successfully pass messages
* Synchronization works via semaphores and mutexes

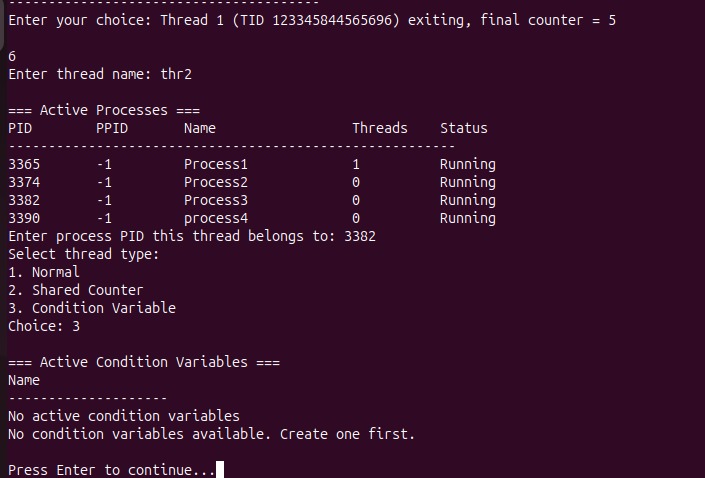
**Menu Interface:**



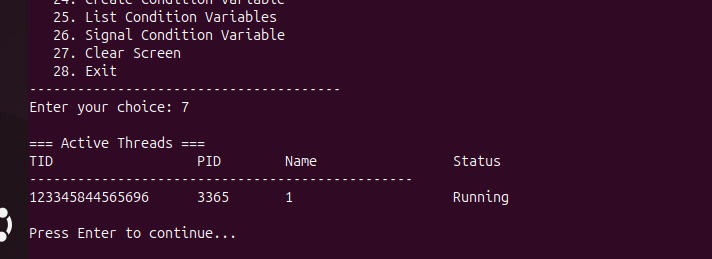
**Process Hierarchy Tree:**



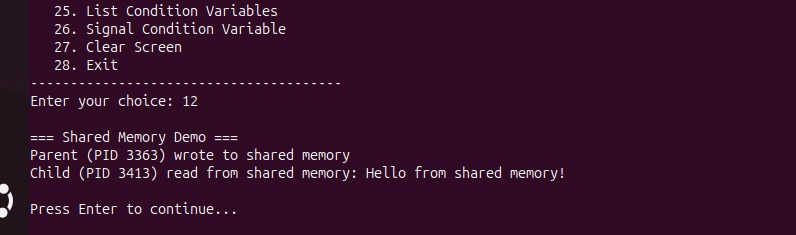
**Thread Logs and Execution:**



**Shared Memory Output:**



**IPC via Message Queue / Pipe:**



# Conclusion / Lessons Learnt

* Gained comprehensive insight into process/thread management and their life cycles.
* Understood and implemented key IPC mechanisms.
* Identified and resolved synchronization issues through mutexes and condition variables.
* Realized the importance of resource cleanup and robust signal handling in production-level code.
* Enhanced proficiency in debugging concurrent and multi-process C programs.

# References:

* Silberschatz, A., Galvin, P. B., & Gagne, G. *Operating System Concepts*, 10th Edition.
* Stevens, W. R. *Advanced Programming in the UNIX Environment*.
* Tanenbaum, A. S. *Modern Operating Systems*.
* Linux Man Pages – [man7.org](https://man7.org/)
* GeeksforGeeks – POSIX Thread Programming
* cppreference.com – C Standard Library & POSIX Reference