1. Introduction

The LinuxOps Management System is an integrated software suite designed to provide a comprehensive understanding of Linux process management, scheduling, and monitoring. The project explores the interaction between system calls in C and high-level visualization using Python, focusing on the lifecycle of processes, multithreading, and signal handling in a Linux environment.

2. System Components:

The project is divided into three components that work together to manage and visualize Linux processes:

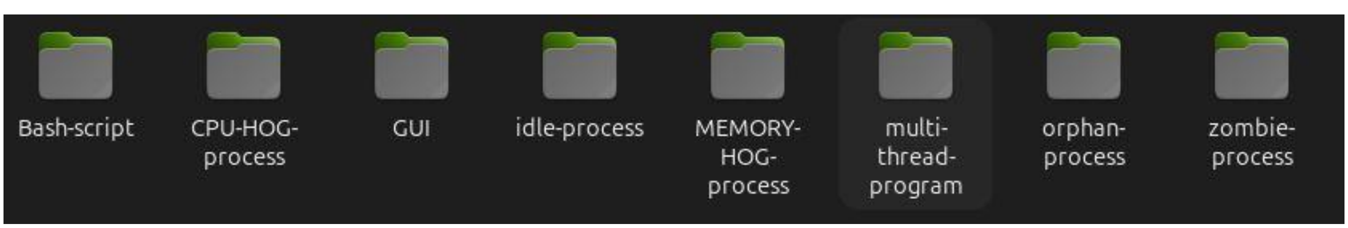
2.1. The Processes (C Language)

This layer consists of several C programs designed to simulate different types of system behaviors:

- **CPU & Memory Intensive:**
  - **CPU-HOG**: A program that loops infinitely to use 100% of the CPU.
  - **MEMORY-HOG**: A program that allocates large amounts of RAM to test memory management.
- **Process Lifecycle:**
  - **Zombie Process**: A child process that finishes its job, but the parent doesn't "clean it up" yet.
  - **Orphan Process**: A process whose parent has finished, so it gets a new "parent" .
- **Idle Process**: A process that does nothing but "sleep," used to test how the system wakes it up or shuts it down politely.
- **Multi-threading**: A program that creates multiple "tasks" (threads) inside a single process using **pthreads**.

2.2. (Bash Script)

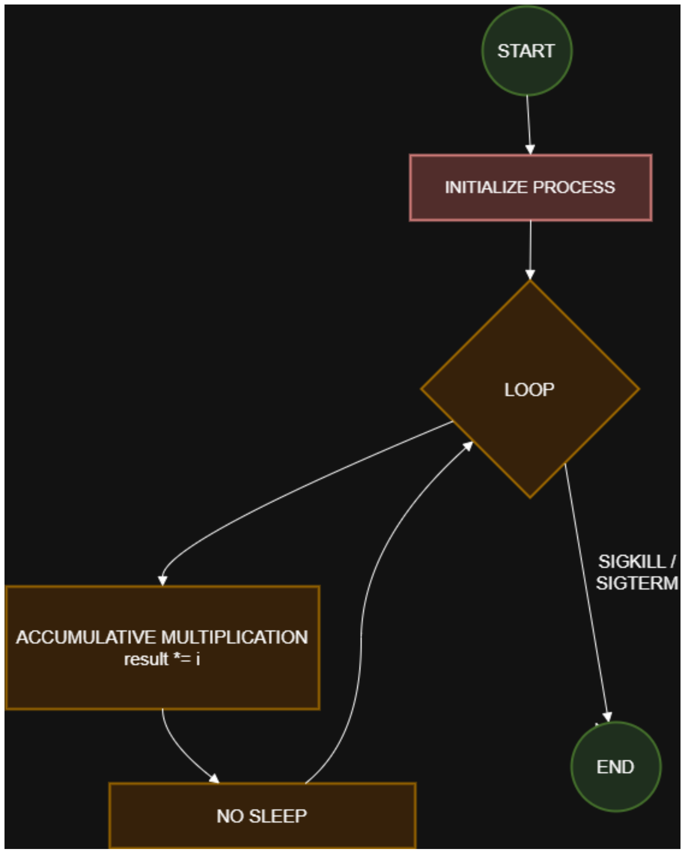
2.3. (GUI)



3. Implementation Details:

3.1. CPU-Intensive Process (CPU Hog)

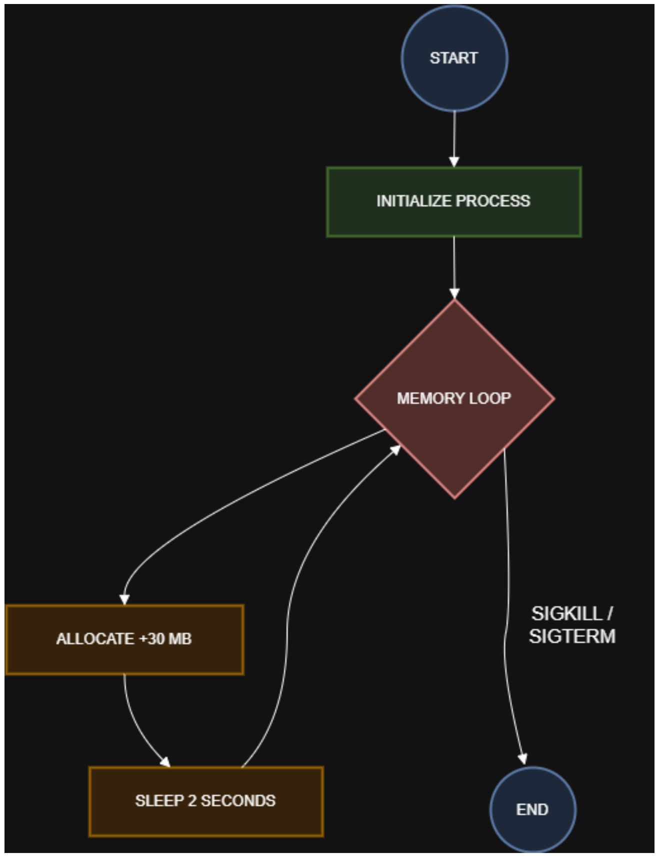
The first core component is the **CPU Hog**, a C program designed to stress the processor.



To achieve maximum CPU usage, we implemented an infinite while(1) loop. Inside the loop, the program performs continuous arithmetic operations (i = i * j), ensuring the process stays at the top of the CPU execution queue unless its priority is adjusted.

3.2. Memory-Intensive Process (Memory Hog)

The **Memory Hog** is a C program implemented to simulate high memory consumption and observe how the Linux system manages dynamic memory allocation.



- Dynamic Allocation: The program defines a "chunk" size of 30 MB (30 * 1024 * 1024 bytes).

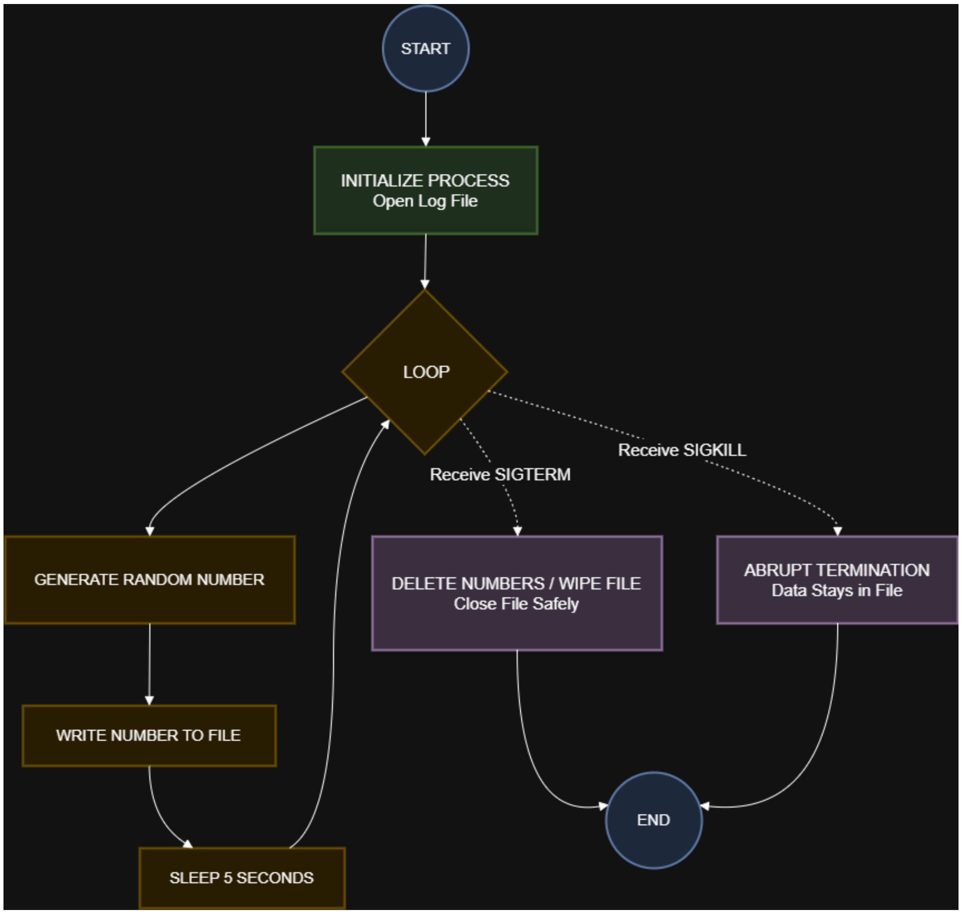
- Continuous Consumption: Inside an infinite while(1) loop, the program uses malloc(chunk) to allocate this 30 MB block in the heap memory during every iteration.

- Real Memory Usage: To ensure the kernel allocates "Physical RAM" instead of just "Virtual Memory," the program uses memset(ptr, 0, chunk) to write zeros into the allocated space.

- Controlled Growth: A sleep() function is included to slow down the allocation rate, allowing us to monitor the memory usage increasing cumulatively every second in the monitoring tools.

3.3. Idle Process

The **Idle Process** is a C program designed to perform minimal background tasks while demonstrating advanced **Signal Handling** capabilities in a Linux environment.



- Signal Handling Functions:

- **SIGTERM (Signal 15)**: A custom handler handle_sigterm is implemented to perform a "Clean Shutdown". It calls clear_content() to wipe the data from temp_data.txt before exiting.

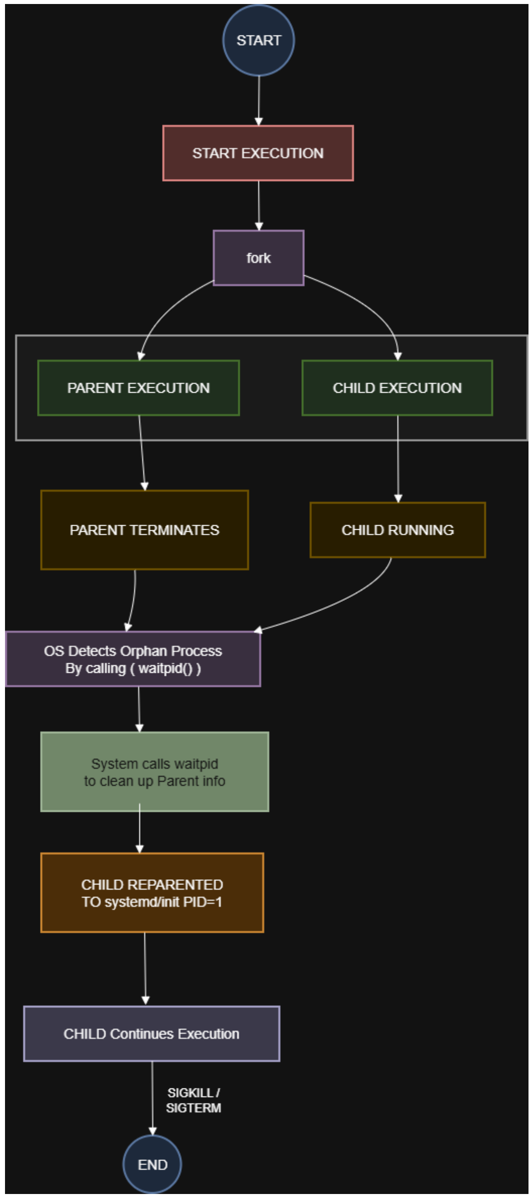
- **SIGKILL (Signal 9)**: A handler handle_sigkill is defined to demonstrate that this signal is generally uninterruptible by the process itself, forcing an immediate exit.

- Background Task: The program runs a while(1) loop that generates a random number every 5 seconds and appends it to temp_data.txt. This simulates a logging process or a background daemon.

- File Management: A helper function clear_content() is used to open the temporary data file in "write" mode without adding content, effectively clearing it.

3.4. Orphan Process Creation

The **Orphan Process** module demonstrates a specific scenario in Linux process management where a child process continues to run after its parent has terminated.



- Process Spawning: The program uses the fork() system call to create a parent-child relationship.

- The "Card" Analogy: the pid_t pid acts like an identification card; the parent receives a card containing the child's PID, while the child receives a card with a value of 0.

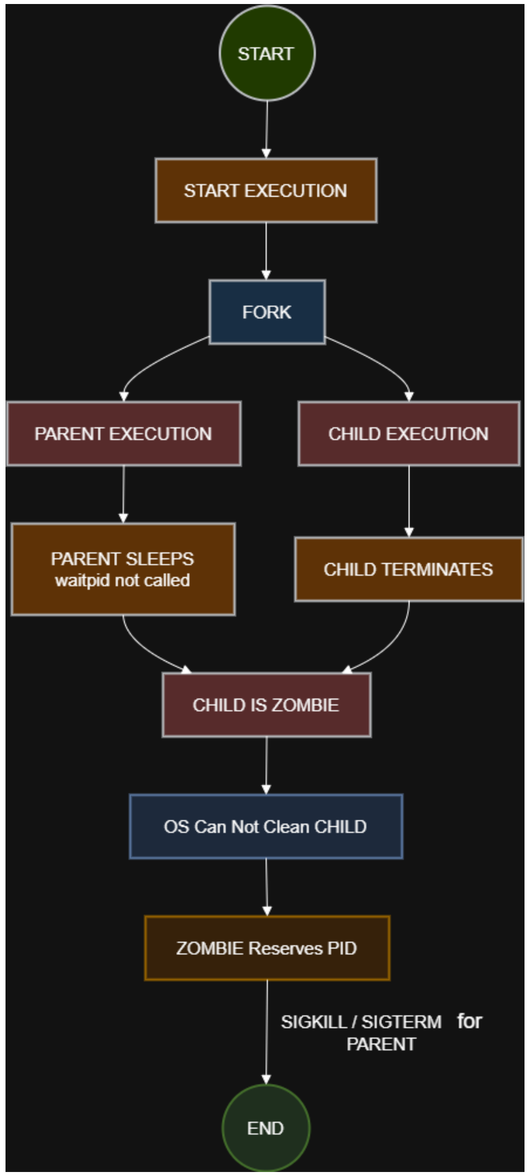
- Parent Termination: To create the orphan state, the parent process (where pid > 0) is forced to exit(0) immediately.

- Child Persistence: The child process (where pid == 0) is programmed to sleep(1) initially, it is will be reparented by system/init and its ppid = pid of systemd.

- Infinite Background Execution: After the parent dies, the child enters an infinite loop with sleep(3), remaining active in the system background.

3.5. Zombie Process Creation

The **Zombie Process** module demonstrates a scenario where a process has completed execution but still has an entry in the process table.



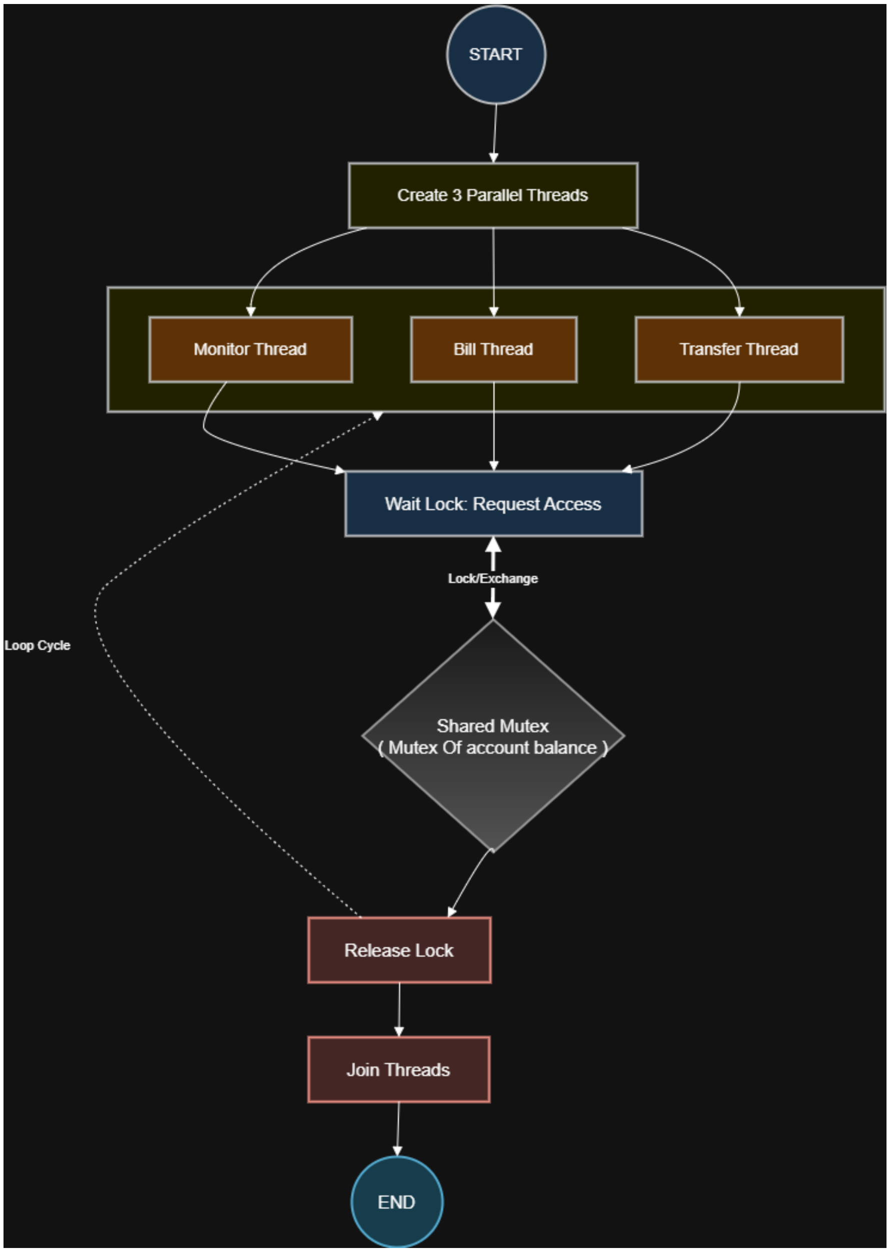
- **Process Spawning**: The program uses the fork() system call to create a child process.

- **Parent Suspension**: To ensure the child becomes a zombie, the parent process (where pid > 0) enters an infinite loop with sleep(5). By doing this, the parent is prevented from calling the wait() system call, which is necessary to read the child's exit status and remove it from memory.

- **Child Termination**: The child process executes exit(0) to terminate itself immediately. Because the parent is "sleeping" and hasn't acknowledged the death of its child, the child remains in the system in a **Zombie State**.

3.6. Multi-threaded Bank System Simulation

The **Multi-threaded Program** is a sophisticated C application that simulates a real-world banking system. It demonstrates how multiple threads can safely interact with shared data using the **POSIX threads (pthreads)** library.



- **Shared Resources & Mutex**: The system defines a global variable account_balance shared across all threads. To prevent **Race Conditions** (where multiple threads try to modify the balance simultaneously), a pthread_mutex_t named bank_vault is used as a lock.

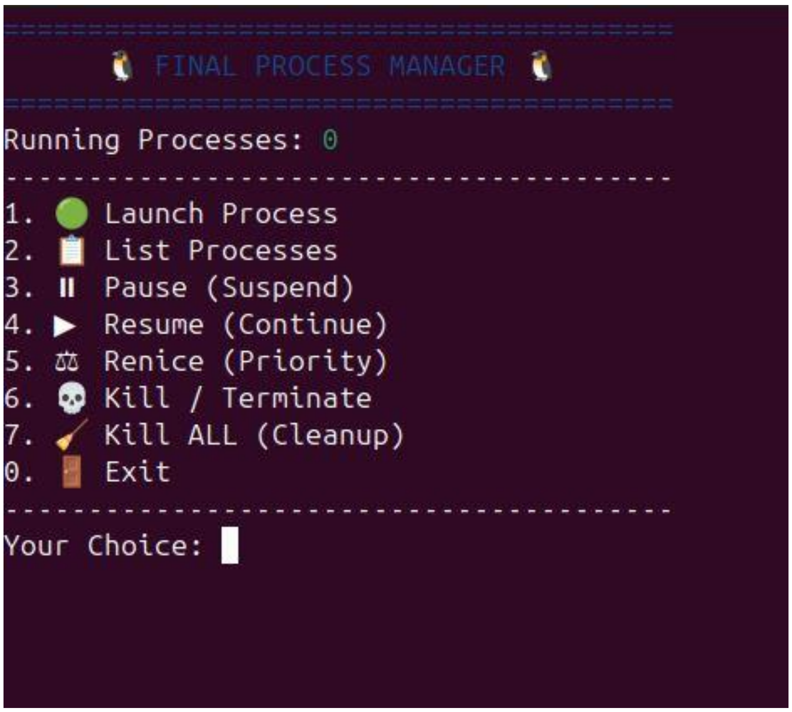
1- Monitor Thread (monitor): Constantly checks the account_balance. It uses the mutex to safely read the current balance and prints a notification only if it detects a change from the last known state.

2- International Transfer Thread (long_transfer): Simulates a secure, time-consuming transaction. It acquires the lock, performs a 30-step verification process (simulated with a loop and sleep), deduces 2000 EGP, and finally releases the lock.

3- Bill Payment Thread (bill): Acts as a standard utility payment. It acquires the lock, deduces 500 EGP after a short delay, and releases the lock.

3.7.(Bash Script)

The **Bash Script** serves as the interactive control center for the entire suite, automating complex Linux commands through a user-friendly Command Line Interface (CLI).



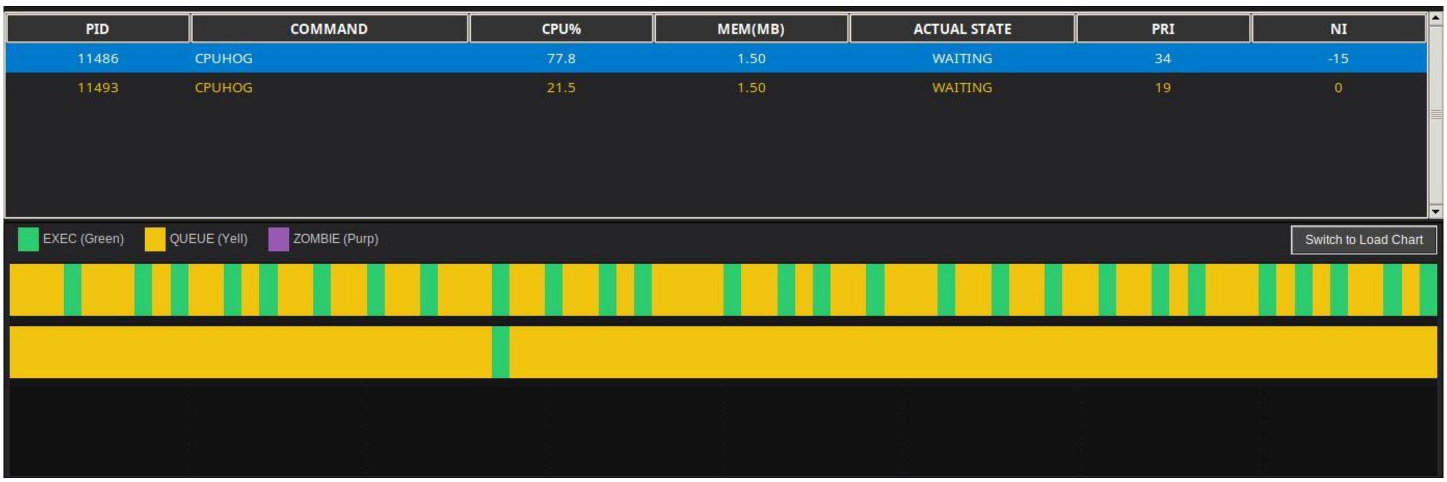
- Menu-Driven Interface:

- The script utilizes a main menu with numbered options, allowing the user to **Launch**, **List**, **Suspend**, **Resume**, **Renice**, and **Terminate** processes.

4.(The Python GUI)

The **LinuxOps Management System** features a high-level dashboard developed in Python. This GUI serves as a sophisticated monitoring and management tool that translates raw system data into actionable visual insights.

- CPU Affinity (Taskset): Processes (CPU HOG – MEMORY HOG – THREADS – IDLE – ORPHAN - ZOMBIE) are launched using the taskset -c 0 command, forcing them to run on a single CPU core. This is a critical technical choice that enables the observation of "Context Switching" as different processes compete for the same core.



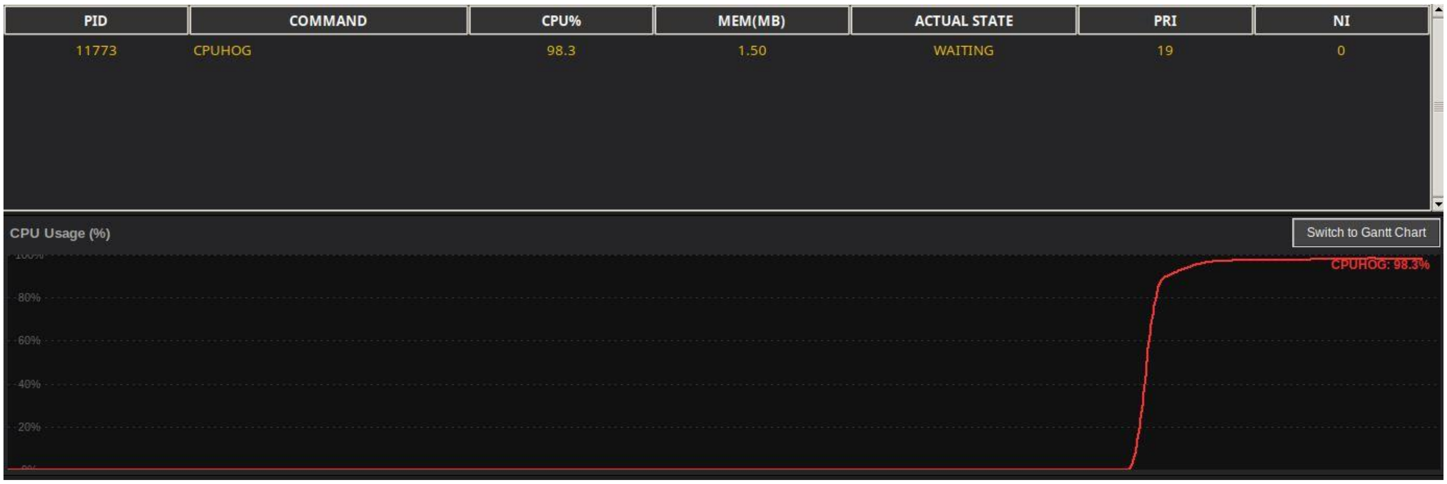
- The Gantt Chart: This chart visualizes the execution timeline of processes (CFS role). It uses a color-coded legend.

Process A (PID 11486): We gave it a **Nice value of -15** (High Priority).

Process B (PID 11493): We gave it a **Nice value of 0** (Normal Priority).

The Table Results: The High Priority process took **77.8%** of the CPU, while the Normal one only got **21.5%**.

The Gantt Chart: The High Priority process has long **Green (EXEC)** bars, meaning it spent most of its time actually running.



- The Load Chart: Users can switch to a "Load Chart" view to see a real-time line graph of CPU usage. For instance, a **CPUHOG** process can be seen spiking to **98.3%** CPU consumption, providing immediate visual proof of high-intensity execution.