FATİH SULTAN MEHMET VAKIF UNIVERSITY

ProcX – Advanced Process Management System

TERM PROJECT REPORT

Students Name : Zeynep Dağtekin

Student ID : 2121251007

Project Topic : ProcX – Advanced Process Management System

Course Instructor : RA ERTUGRUL ISLAMOGLU

1. Project Description

ProcX is an advanced process management system designed to operate in a Linux environment with multi-terminal (multi-instance) support. The project implements an interactive shell-like interface that allows users to create, list, and terminate processes while ensuring synchronization across different terminal instances.

The system leverages core operating system concepts such as **Shared Memory**, **Semaphores**, and **Message Queues** to maintain data consistency and enable Inter-Process Communication (IPC).

---

2. System Architecture

The project is built upon three main architectural components:

2.1. Data Structures

Two primary structures were defined to manage process states:

- **`ProcessInfo`:** Stores detailed information for each process, including PID, Owner PID, Command, Mode (Attached/Detached), Status, and Start Time.
- **`SharedData`:** A structure mapped to shared memory that holds the process table (max 50 processes) and the count of active ProcX instances to manage the multi-terminal logic.

2.2. IPC Mechanisms

A hybrid approach of POSIX and System V standards was implemented to maximize efficiency:

- **Shared Memory (POSIX):** Used to store the global process table (`/procx_shm`). Accessed via `shm_open` and `mmap`.
- **Semaphores (POSIX):** Used to synchronize access to the shared memory region (`/procx_sem`) to prevent Race Conditions during read/write operations.
- **Message Queues (System V):** Used for real-time broadcasting of events between terminals (See Section 4.1 for design justification).

2.3. Threading Model

To ensure a non-blocking user experience, the system utilizes a multi-threaded design:

1. **Main Thread:** Handles user input and executes commands.

2. **Monitor Thread:** Runs in the background to clean up terminated processes using `waitpid(WNOHANG)` and performs periodic checks.
3. **IPC Listener Thread:** Listens for incoming messages from other instances and displays notifications.

---

3. Implementation of Functional Requirements

3.1. Process Creation (fork & exec)

Processes are created using `fork()` and `execvp()`.

- **Detached Mode:** For background processes, `setsid()` is called in the child process. This creates a new session, ensuring the process continues running even if the ProcX terminal is closed.

3.2. Process Termination

When a user terminates a process via PID:

- A `SIGTERM` signal is sent using `kill()`.
- **Immediate Update:** Upon successful signal delivery, the process is **immediately** removed from the shared memory table. This ensures that subsequent `List` commands reflect the changes instantly, without waiting for the Monitor Thread's cycle.

3.3. Exit and Cleanup Logic

Upon exiting the program (Option 0):

- Only **Attached** processes belonging to the current instance are terminated.
- **Detached** processes are left running as per requirements.
- Threads are joined using `pthread_join` to ensure a clean exit.

---

4. Design Decisions & Technical Details

4.1. Why System V Message Queue?

Although POSIX IPC was used for Shared Memory and Semaphores, **System V Message Queues** (`msgget`, `msgsnd`) were specifically chosen for messaging.

- **Justification:** System V queues support a `mtype` (message type) field. In this multi-instance architecture, each terminal is assigned a unique ID. Using `mtype`

allowed for a robust **"Targeted Broadcast"** mechanism where messages could be filtered or directed to specific instances. Implementing this logic with POSIX MQ would have been significantly more complex and less efficient.

4.2. IPC Cleanup Strategy (Reference Counting)

The project requirements state that IPC resources should be removed upon exit. However, in a multi-terminal environment, blindly removing resources (e.g., `shm_unlink`) when one terminal exits would cause a **Segmentation Fault** in other active terminals.

- **Solution:** A **Reference Counting** mechanism was implemented using `instance_count` in Shared Memory.
- Resources are unlinked and removed from the system **ONLY when the last active ProcX instance terminates**.
- This design ensures strict compliance with memory safety rules while supporting concurrent usage.

4.3. Monitor Thread Timing

To strictly adhere to the project document requirements, the Monitor Thread is set to sleep for exactly **2 seconds** (`sleep(2)`). While a lower interval could provide faster feedback, this value was chosen to ensure full compliance with the functional requirements specification.

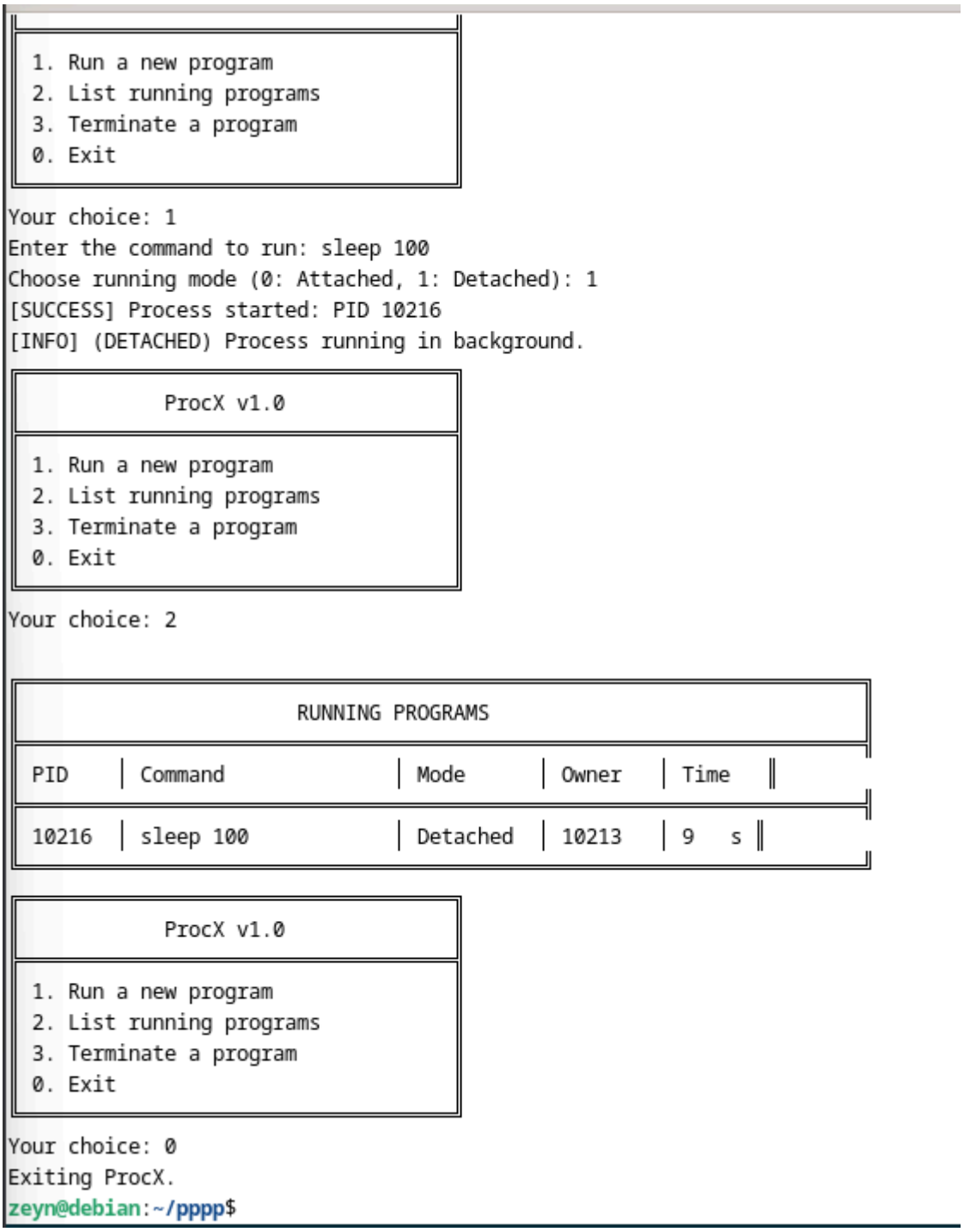
Compilation and Execution

```
zeyn@debian:~$ cd pppp/
zeyn@debian:~/pppp$ make clean
rm -f procx
zeyn@debian:~/pppp$ make
gcc -Wall -Wextra -pthread -g -o procx procx.c -lrt
zeyn@debian:~/pppp$ ./procx
╔═══════════════════════════════╗
║           ProcX v1.0          ║
╠═══════════════════════════════╣
║ 1. Run a new program          ║
║ 2. List running programs      ║
║ 3. Terminate a program        ║
║ 0. Exit                       ║
╚═══════════════════════════════╝
Your choice:
```

5. Test Scenarios

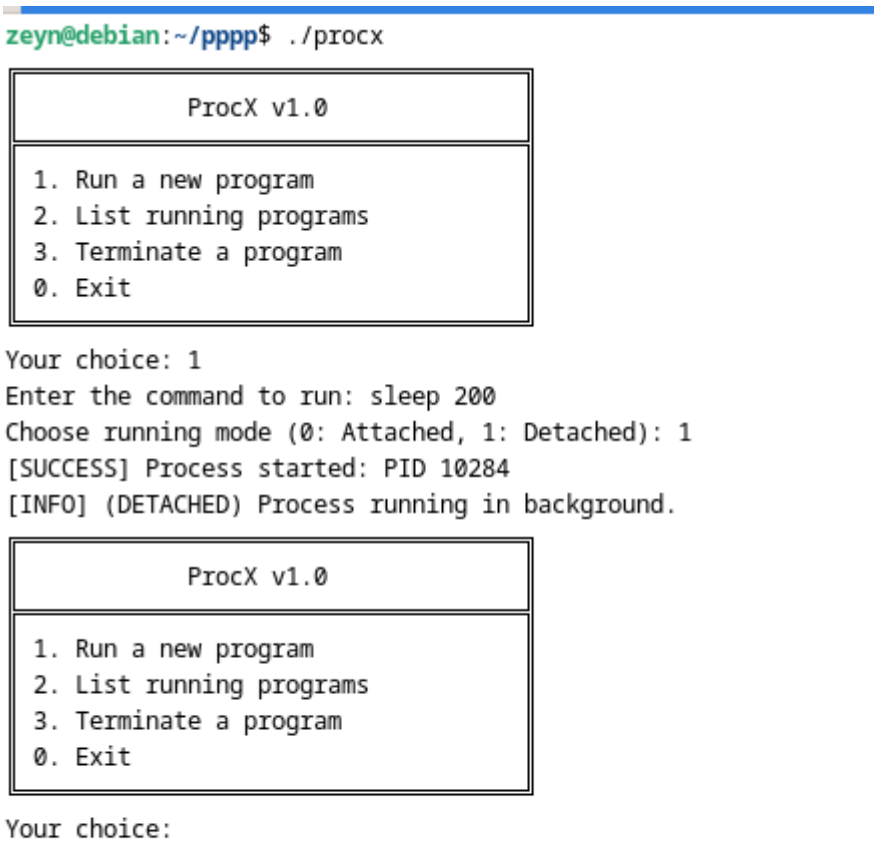
Test 1: Single Instance - Process Start and List

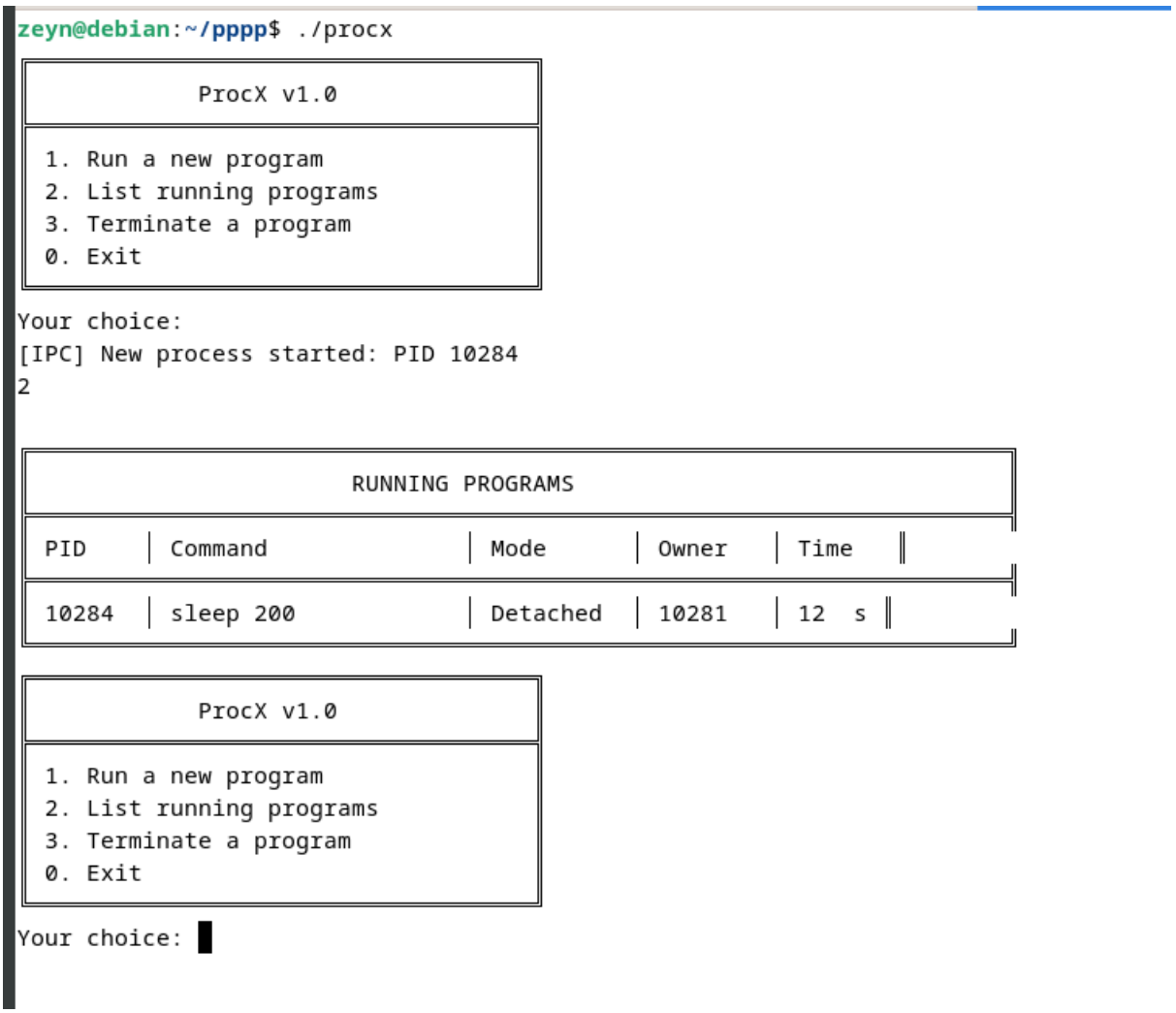
Verification of process creation and listing capabilities. A new process (sleep command) was created using `fork()` and `execvp()`. The `List` command confirms that the process is successfully registered in Shared Memory with "Running" status.



Test 2: Multi-Instance Communication

Multi-instance synchronization via IPC. A process started in Terminal 1 triggered a real-time broadcast notification in Terminal 2 via System V Message Queues, demonstrating successful data consistency across instances.





Test 3: Attached vs. Detached Termination

Exit logic and orphan process management. Upon terminating the ProcX shell, the Attached process was cleaned up, while the Detached process (PID 10298) continued running in the background due to `setsid()` usage. The `ps aux` output confirms the detached process is still active.

```
zeyn@debian: ~/pppp
zeyn@debian:~/pppp$ ./procx
```

```
╔═══════════════════╗
║      ProcX v1.0   ║
╠═══════════════════╣
║ 1. Run a new program
║ 2. List running programs
║ 3. Terminate a program
║ 0. Exit
╚═══════════════════╝
Your choice: 1
Enter the command to run: sleep 300
Choose running mode (0: Attached, 1: Detached): 0
[SUCCESS] Process started: PID 10298
[INFO] (ATTACHED) Process running linked to this terminal.
╔═══════════════════╗
║      ProcX v1.0   ║
╠═══════════════════╣
║ 1. Run a new program
║ 2. List running programs
║ 3. Terminate a program
║ 0. Exit
╚═══════════════════╝
Your choice: 1
Enter the command to run: sleep 400
Choose running mode (0: Attached, 1: Detached): 1
[SUCCESS] Process started: PID 10299
[INFO] (DETACHED) Process running in background.
╔═══════════════════╗
║      ProcX v1.0   ║
╠═══════════════════╣
║ 1. Run a new program
║ 2. List running programs
║ 3. Terminate a program
║ 0. Exit
╚═══════════════════╝
Your choice: 0
Exiting ProcX.
```

```
zeyn@debian:~/pppp$ ps aux | grep sleep
zeyn       10299  0.0  0.0   5472   888 ?        Ss   21:23   0:00 sleep 400
zeyn       10301  0.0  0.0   6340  2188 pts/1    S+   21:23   0:00 grep sleep
zeyn@debian:~/pppp$
```

Test 4: Process Termination

Manual termination handling. The user sent a `SIGTERM` signal to a specific PID. The Monitor Thread subsequently detected the terminated process, removed it from the shared list, and displayed the required cleanup message.

```
zeyn@debian:~/pppp$ ./procx
```

```
┌──────────────────────┐
│       ProcX v1.0     │
├──────────────────────┤
│ 1. Run a new program │
│ 2. List running programs │
│ 3. Terminate a program │
│ 0. Exit              │
└──────────────────────┘
```

```
Your choice: 1
Enter the command to run: sleep 300
Choose running mode (0: Attached, 1: Detached): 1
[SUCCESS] Process started: PID 3777
[INFO] (DETACHED) Process running in background.
```

```
┌──────────────────────┐
│       ProcX v1.0     │
├──────────────────────┤
│ 1. Run a new program │
│ 2. List running programs │
│ 3. Terminate a program │
│ 0. Exit              │
└──────────────────────┘
```

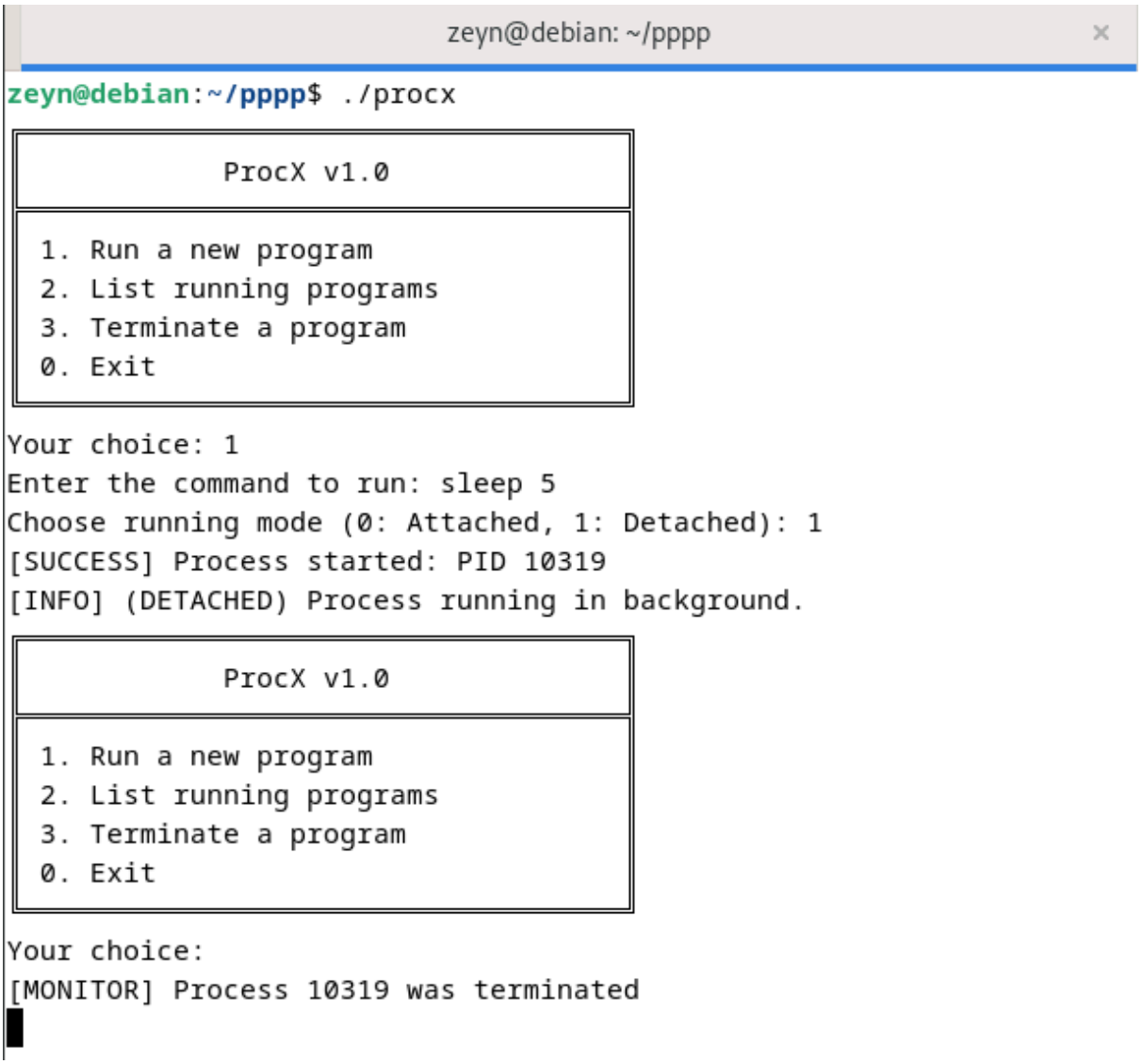
```
Your choice: 3
Process PID to terminate: 3777
[INFO] SIGTERM sent to 3777. Removing from list.
```

```
┌──────────────────────┐
│       ProcX v1.0     │
├──────────────────────┤
│ 1. Run a new program │
│ 2. List running programs │
│ 3. Terminate a program │
│ 0. Exit              │
└──────────────────────┘
```

```
Your choice:
[MONITOR] Process 3777 was terminated
```

Test 5: Monitor Thread

Automatic zombie process cleanup. A short-lived process finished execution naturally. The background Monitor Thread detected the state change using `waitpid(WNOHANG)` and released the resources without user intervention.



---

6. Conclusion

The development of the **ProcX** system provided a comprehensive practical application of core operating system concepts. Throughout the project, the complexities of **Process Management**, **Inter-Process Communication (IPC)**, and **Concurrency** were successfully addressed and implemented in a Linux environment.

Key achievements of this project include:

1. **Robust Process Control:** The system effectively manages the lifecycle of processes using `fork` and `exec` families, while strictly distinguishing between **Attached** and **Detached** execution modes.
2. **Advanced Synchronization:** Race conditions, a common pitfall in multi-process environments, were prevented using **POSIX Semaphores**. Data consistency within the **Shared Memory** segment was maintained even under high-concurrency scenarios where multiple ProcX instances operated simultaneously.
3. **Architectural Stability:** A significant design challenge safely cleaning up IPC resources in a multi-terminal setup was resolved by implementing a **Reference Counting** strategy. This approach prevents "Segmentation Faults" by ensuring resources are released only when the final active instance terminates. Furthermore, the integration of **System V Message Queues** enabled a highly efficient, targeted broadcasting mechanism for real-time notifications.
4. **Resource Efficiency:** The implementation of a background **Monitor Thread** utilizing `waitpid` with the `WNOHANG` flag ensures that "Zombie Processes" are instantly reaped, preventing resource leakage.

In conclusion, ProcX is not merely a process lister but a resilient process manager that mimics the behavior of production-grade shells. The project meets all functional requirements while demonstrating an advanced understanding of system stability and memory safety.