Linux Process Manager
Phase II Survey

Operating Systems

Adham Ali – 900223243
Ebram Thabet – 900214496
Omar Saqr – 900223343
Aabed Elghadban – 900223106

Submitted to Dr. Mohamed El Halaby
This paper is prepared for CSCE 3401, section 01

Department of Computer Science and Engineering
The American University in Cairo

30/11/2025

Table of contents

1 Team Contributions

1.1 Adham Ali

- Studied and demonstrated the real-time process listing functionality, showing all running processes with:
  - PID, name, CPU usage, memory, PPID, start time, nice value, user, and status.
- Mastered the process sorting mechanism, allowing sorting by:
  - CPU, memory, PID, PPID, start time, and nice value in ascending and descending order.
- Prepared documentation and demonstration materials for:
  - The real-time process list.
  - Sorting features.
- Conducted extensive testing of the process list and sorting features in:
  - TUI mode.
  - GUI mode.
- Designed and implemented confirmation prompts for all process control operations:
  - Kill, stop, continue – to prevent accidental termination.
- Developed dependency-aware termination warnings:
  - Warns the user before killing a parent process that has children.
- Implemented multi-select functionality:
  - Enables batch operations on multiple processes at once.
- Created the "start new program" feature:
  - Allows users to launch processes with custom working directories and environment variables.
- Integrated comprehensive error handling and permission checking:
  - Across all process control operations.
- Collaborated with the rest of the team:
  - To ensure process safety features are integrated cleanly with other system modules.

1.2 Ebram Thabet

- Studied and demonstrated the per-process graphs:

    - Real-time CPU and memory usage charts for individual processes.
- Mastered the global system dashboard:
    - CPU activity, memory usage, swap utilization, and overall system health.
- Prepared documentation and demonstration scenarios:
    - To showcase visualization features for system monitoring.
- Conducted comprehensive testing of graphing and dashboard functionality:
    - Ensured accurate and consistent data representation.
- Designed and implemented the advanced filtering system:
    - Multi-field queries.
    - Boolean logic (AND, OR, NOT).
    - Regular expressions.
- Created focus-mode profiles (e.g. “build”, “editing”, “presentation”):
    - Adjusts priorities.
    - Hides or highlights processes based on current workflow.
- Developed the alerts and notifications system:
    - Custom thresholds for CPU, memory, and I/O usage.
- Implemented the filter parser module:
    - Converts complex filter expressions into efficient internal filtering operations.
- Designed the profile management interface:
    - With persistent storage across application sessions.
- Integrated the alert system with process monitoring:
    - Real-time detection when thresholds are exceeded and showing active alerts to the user.

1.3 Omar Saqr

- Studied and demonstrated process control signals:
    - SIGKILL, SIGTERM, SIGSTOP, SIGCONT for full process lifecycle control.
- Mastered the priority (nice value) manager:
    - Dynamic adjustment of process priority with immediate visual feedback.
- Prepared documentation and demonstration materials:
    - For process control and priority management.
- Conducted extensive testing of:

  - Signal delivery.
  - Nice value changes with proper error handling.
- Designed and implemented the resource grouping system:
  - Groups processes by cgroups.
  - Groups by Docker/Podman containers.
  - Groups by Linux namespaces.
- Created the container view module:
  - Detects running containers.
  - Displays per-container metrics such as CPU, memory, and process counts.
- Developed namespace grouping functionality:
  - Groups processes by PID, network, and mount namespaces.
- Implemented drill-down navigation:
  - Lets users start from high-level groups and drill down to individual processes.
- Designed and implemented the job scheduling and automation system:
  - Supports cron-like scheduling for restarts and cleanup tasks.
- Created the scheduler module:
  - Handles task creation, editing, enabling, disabling.
  - Ensures tasks persist across application restarts.
- Integrated automation with process monitoring:
  - Automatic restarts on failure.
  - Idle process cleanup based on defined rules.

1.4 Aabed Elghadban

- Studied and demonstrated the process monitoring infrastructure:
  - Historical data for CPU, memory, and I/O usage over time.
- Mastered the historical data logging system:
  - Stores per-process metrics with time-based indexing for trend analysis.
- Conducted comprehensive testing of:
  - Monitoring infrastructure.
  - Historical data collection and stability.
- Designed and implemented CRIU integration (Checkpoint/Restore in Userspace):

    - Checkpointing and restoring processes.
    - Graceful fallback when CRIU is unavailable.
- Created the CRIU manager module:
    - Handles checkpoint creation and restoration workflows.
    - Includes health checks and basic failure handling.
- Implemented the coordinator module:
    - Manages connections to remote hosts.
    - Fetches remote process data.
    - Coordinates synchronized operations across hosts.
- Designed and implemented the complete GUI using the `egui` framework:
    - Tabs for Process List, Statistics, Profiles, Alerts, Checkpoints, and Hosts.
- Created interactive graphs and visualizations in the GUI:
    - Real-time updates for process and system metrics.
    - Historical trend views.
- Ensured seamless integration between TUI and GUI:
    - Shared core architecture.
    - Maintained feature parity so both interfaces expose the same main capabilities.

2 Abstract & Summary of Tool

Abstract

This Report provides a detailed manual for our Linux Task Manager, "ProcSentinel". "ProcSentinel" is a process manager that is built using Rust and has two interfaces: a Tabular User Interface (TUI) and a Graphical User Interface (GUI). Our Process Manager aims at effectively monitor and manage running system processes. Linux users are divided into two sections: some want an intuitive, accessible, easy-to-use gui interface, while others are more comfortable with a tabular form that presents more functions while giving them the option to get detailed insights without the need to leave the terminal. "ProcSentinel" bridges this with its strong TUI for terminal-based users and a user-friendly GUI for those seeking better visuals and system graphs. In both versions, real-time process monitoring, resource tracking, and the possibility of process management are enabled. "ProcSentinel" represents good evolution in Linux process management utilities by bringing together traditional approaches and several newly added features, aiming to offer users even better control and performance

optimization.

Summary of Tool

Linux Process Manager, or LPM, is a high-performance system for monitoring and controlling processes that is designed for operation on a Linux system. It was implemented using the Rust programming language to ensure memory safety and efficiency. It can handle complex queries, and system automation.

LPM provides a full range of process management functions, such as real-time examination of process metadata, including PID, CPU/memory, user, priority, status, start time, association with namespace or container, as well as basic process manipulation functions, including killing, stopping, continuing, or reprioritizing a process, along with checks for permissions or for processes to be recursively killed. It also supports exit process logging which could be benifcial for system admins during the debugging process.

One of the main advantages of LPM is that it contains a highly capable filtering engine that, besides basic filters, offers complex boolean queries, comparisons, and regular expression filters. Sorting by any of the process fields makes quick browsing of long lists of processes possible. Also, LPM offers comprehensive system statistics, including CPU, memory, disk, and process load statistics, displayed through interactive real-time graphs.

It supports multiple grouping methods, and users can aggregate processes by user, cgroup, containers, or Linux namespaces. Containerized processes, such as Docker, Kubernetes, or Podman, can be automatically identified, and system resource allocation can be analyzed using aggregate statistics for each group.

LPM also integrates a flexible automation layer, such as alerts based on thresholds, scheduled tasks that include interval, cron-like, and one-shot schedules, and a rule engine. It also integrates other fault tolerance capabilities such as checkpointing and restoration using CRIU, however these features were not in our functional requirements in the survey phase, so we didn't complete them .

It uses a modular structure, where there is a shared logic for the interface, a state that can be accessed from multiple threads using Rust concurrency primitives, and a persistent configuration stored at path `~/.lpm/`. It can sustain a thousand processes or more.

The existing shortcomings include read-only remote monitoring, lack of authentication for remote monitoring in agent mode, incomplete CRIU support, and non-persistent graphs for historical data. However, these features were not part of the functional requirments in our survey. The future enhancements include secure remote control, improved cron expression, enhanced alerts, and persistence for collected data.

3 Linux Process Manager – Requirements Summary

This section summarises the requirements for the Linux Process Manager (LPM). We distinguish between functional requirements, non-functional requirements, and known limitations of the current implementation.

3.1 Functional Requirements

Process List.

- View active processes with key information: PID, name, CPU, memory, status, user, nice value, start time, cgroup, container ID, and namespace IDs.
- Automatically refresh the list at a configurable interval in seconds (default: 1 s).
- Log process exit events with uptime measurements of the target processes.

Process Control.

- Support killing, stopping, terminating, continuing, and starting processes.
- Allow changing of priority (nice value) with validation in the range -20 to 19.
- Perform a root check before accepting negative nice values.
- Kill process trees (parent and children recursively), with a warning if a process has children.

Filtering and Sorting.

- Provide simple filtering by user, name, PID, and PPID.
- Provide advanced filtering with Boolean expressions (AND, OR, NOT), field comparisons (`==`, `!=`, `>`, `<`, `>=`, `<=`), and regular expressions.
- Support real-time filter updates.
- Support sorting by PID, name, CPU, memory, PPID, user, nice, status, and start time, in ascending or descending order.

User Interfaces.

- Provide a TUI with keyboard navigation and multiple view modes.
- Provide a GUI with tabs for: processes, statistics, graphs, profiles, alerts, checkpoints, hosts, logs, schedule, and rules.
- Use colour-coded information in both interfaces and show confirmation dialogs for important actions.

System Statistics.

- Host information: hostname, OS, kernel, boot time, uptime.
- CPU: model, frequency, cores, temperature, per-core usage, load average, context switches.
- Memory: total, used, free, cached, buffers, swap.
- Disk: total, used, free, read/write speeds, filesystem type. However, read, write speeds require permissions from the system.
- Process state counts: Running, Sleeping, Stopped, Zombie.
- Real-time graphs of CPU and memory usage over time, with per-process graph support.

Grouping.

- Group processes by cgroups, containers (Docker, Kubernetes, Podman), namespaces (pid, net, mnt, uts, ipc, user, cgroup), and username. We mainly tested using Docker.
- Display aggregate CPU and memory usage per group.
- Allow expand/collapse and drill-down into details for each group.

Profiles and Alerts.

- Profiles can:
  - highlight processes(make them appear at the beginning of the processes list),
  - filter out processes (make them disappear from the processes list), and
  - adjust nice values based on pid, or process name.
- Persist profiles to a configuration file.
- Provide alerts triggered by CPU or memory thresholds, or process exit conditions, with configurable duration and target scope (all processes, pattern match, or specific PID).
- Display active alerts in the UI.

Task Scheduling.

- Support task scheduling via:
  - fixed intervals,
  - cron expressions (basic parsing), and
  - one-shot execution.
- Supported actions: kill, stop, continue, renice processes, and apply automation rules.
- Store last-run and next-run timestamps and keep an execution log.

- Wishlist: some action types are only partially implemented; cron parsing is greatly simplified.

Automation Rules.

- Allow custom rules using the Rhai scripting language.
- Give rules access to process variables: `cpu`, `mem`, `pid`, `name`.
- Expect rules to return a boolean which is used to filter processes.
- Allow setting and clearing the active rule.

Logging.

- Log exited processes with PID, name, user, start time, exit time, and uptime.
- Allow filtering and grouping log entries by name, PPID, or user.
- Keep logs in memory only (not persisted).

3.2 Non-Functional Requirements

Performance.

- Keep the UI responsive while refreshing data (non-blocking design).
- Refresh the process list every second by default, configurable.
- Refresh graphs every 500 ms by default, configurable.
- Handle more than 1000 processes without noticeable degradation.
- Use Rust to obtain zero-cost abstractions and no Garbage collection pauses.

Reliability.

- Handle terminated processes gracefully without crashing.
- Handle kernel read failures (permission errors, missing processes).
- Handle mutex poisoning explicitly.
- Validate user inputs (nice values, PIDs).

Security.

- Check for root privileges when setting negative nice values.
- Return appropriate error messages for permission-denied operations.

Usability.

- Provide clear keyboard shortcuts and help text in the TUI.
- Provide an intuitive tabbed interface in the GUI.
- Use colour-coded information for CPU/memory usage thresholds.
- Show confirmation dialogs for destructive actions (kill).
- Display error messages clearly.
- Store user configuration in `~/.lpm/` (profiles, alerts, checkpoints).

Maintainability.

- Implement the system in Rust for memory safety.
- Use the ownership model to avoid memory safety issues.
- Use `Arc<Mutex<…> >` for shared-state coordination, with strict lock-ordering rules.
- Follow a well-structured, modular architecture with clear separation of concerns.

Compatibility and Operational Behaviour.

- Run on Linux as the primary target platform with compile-time feature detection.
- Provide fallbacks for non-Linux systems(we tried on MacOs, and WSL).
- Detect containers in Docker where possible.
- Integrate with CRIU when it is installed.
- Use standard Rust dependencies such as `sysinfo`, `tokio`, `ratatui`, `egui`, `axum`, `reqwest`, and `rhai`.
- Be buildable with `cargo build`.
- Support three modes: default (TUI), `--gui` (GUI), and `--agent [--port PORT]` (agent) (agent mode is not part of our functional requirements).
- Create all required directories on first run.

3.3 Known Limitations

Task Scheduling.

- Cron expression parsing is simplified and supports basic patterns only.

Alerts.

- The I/O monitoring alert condition is defined but not fully implemented.

Process Management.

- The system does not store process commands/arguments, so full restart capability is not currently possible.

4 Data Flow and Architecture Diagrams

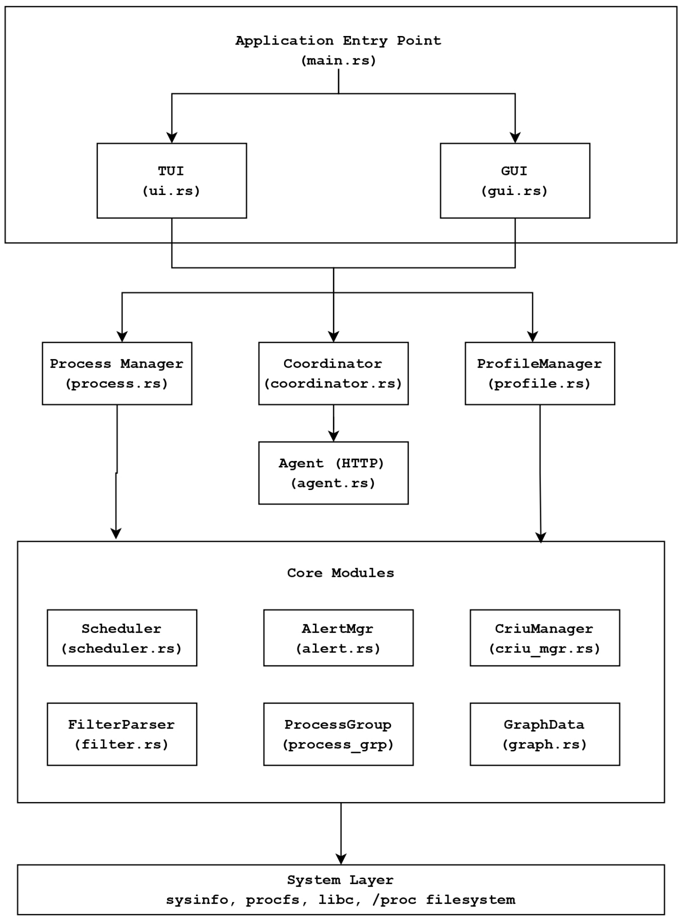


Figure 1: High-level architecture of the Linux Process Manager

The architecture diagram shows that execution begins in `main.rs`, which decides whether to launch the TUI front-end (`ui.rs`) or the GUI front-end (`gui.rs`). Both interfaces talk to shared back-end components: the `ProcessManager` for process monitoring and control, the `Coordinator` (and HTTP `Agent` in `agent.rs`) for multi-host and remote communication which are an additional feature that is not fully developed yet, and the `ProfileManager` for profiles, alerts, and rules. These in turn rely on other core modules such as the scheduler, alert manager, CRIU manager, filter parser, process grouping logic, and graph data handling. At the bottom, all of these layers depend on the system layer, which reads real process and resource information via `sysinfo`, `procfs`, `libc`, and the Linux `/proc` filesystem.

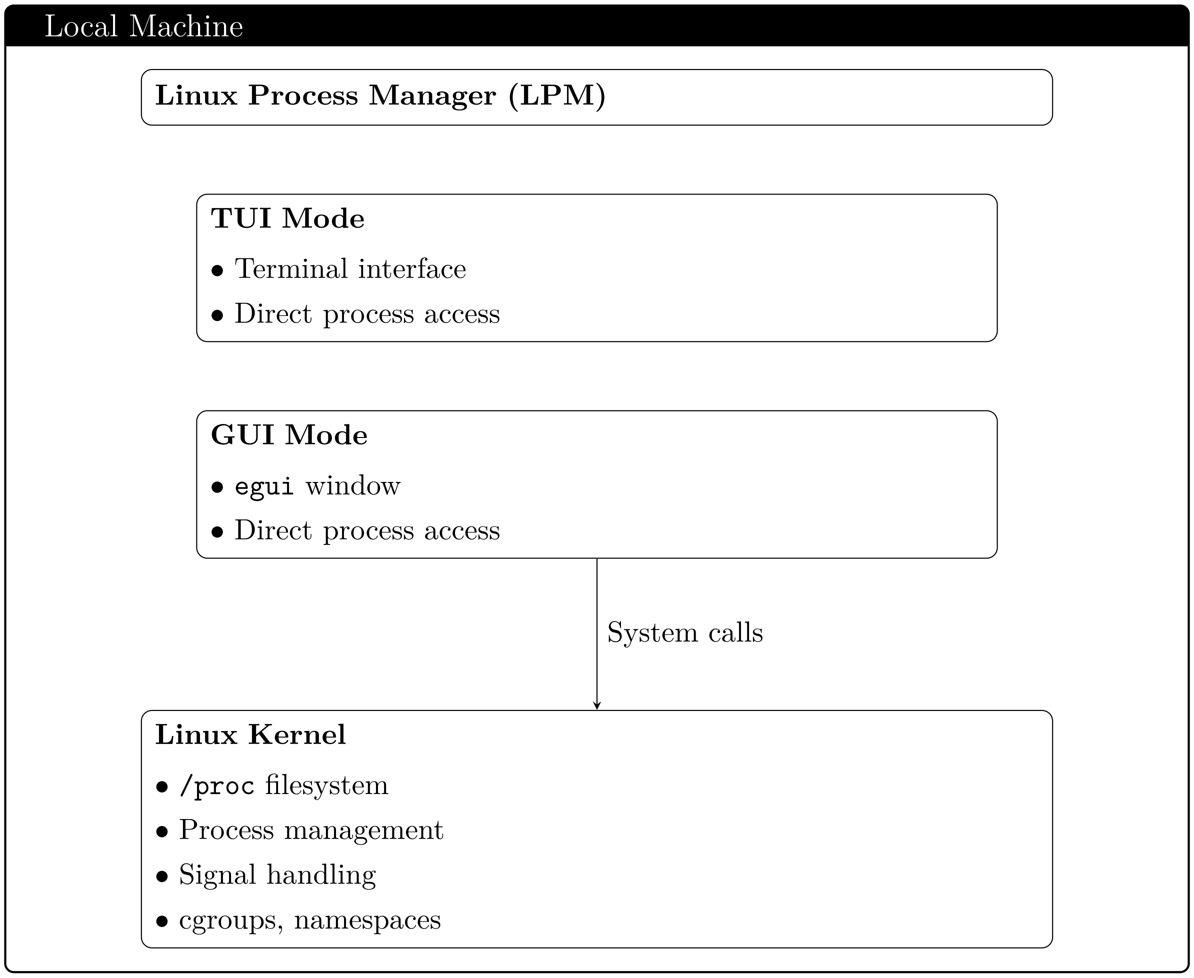
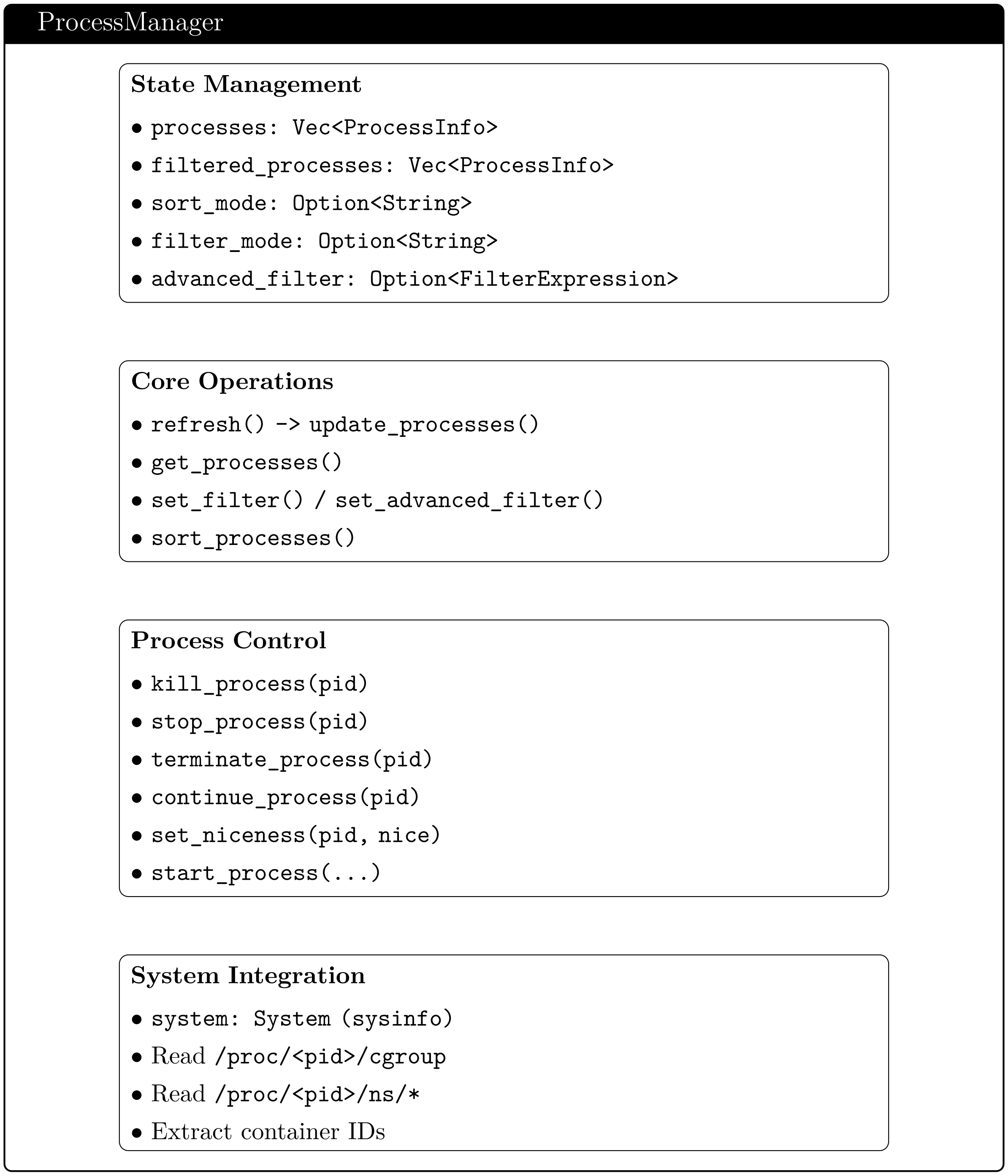


Figure 2: Single-host deployment architecture.

This diagram shows how the Linux Process Manager runs on a single machine: both TUI and GUI modes run in user space and communicate with the Linux kernel via system calls and the `/proc` filesystem.



ProcessManager

State Management

- `processes: Vec<ProcessInfo>`
- `filtered_processes: Vec<ProcessInfo>`
- `sort_mode: Option<String>`
- `filter_mode: Option<String>`
- `advanced_filter: Option<FilterExpression>`

Core Operations

- `refresh() -> update_processes()`
- `get_processes()`
- `set_filter() / set_advanced_filter()`
- `sort_processes()`

Process Control

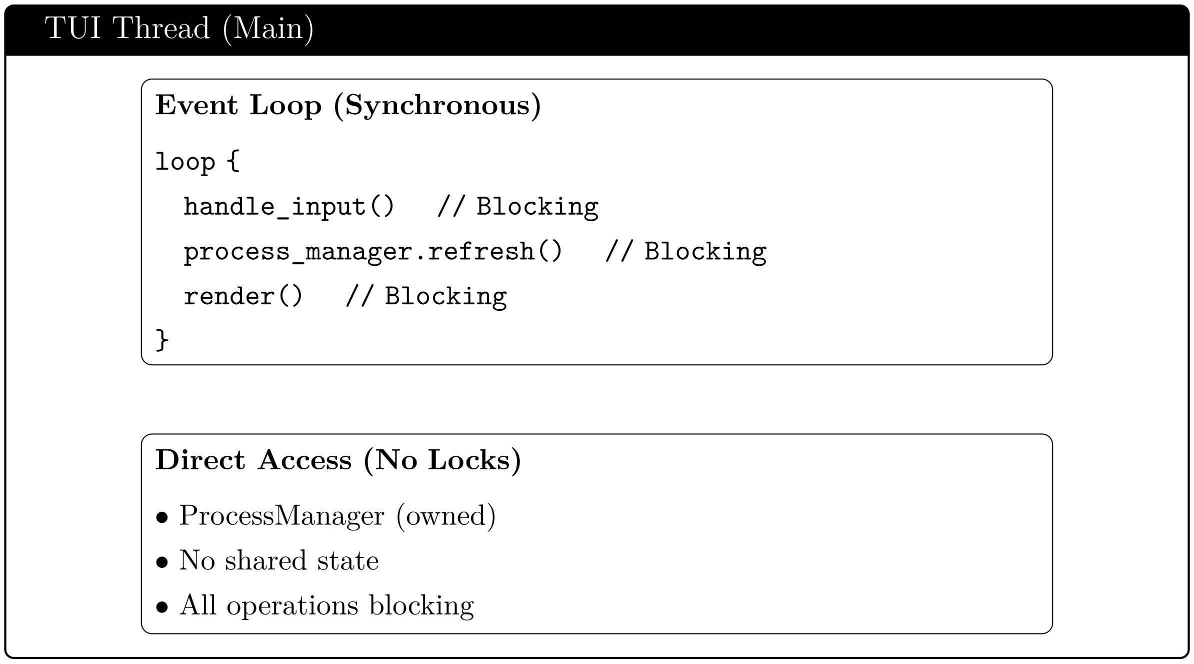
- `kill_process(pid)`
- `stop_process(pid)`
- `terminate_process(pid)`
- `continue_process(pid)`
- `set_niceness(pid, nice)`
- `start_process(...)`

System Integration

- `system: System (sysinfo)`
- Read `/proc/<pid>/cgroup`
- Read `/proc/<pid>/ns/*`
- Extract container IDs

Figure 3: ProcessManager internal structure.

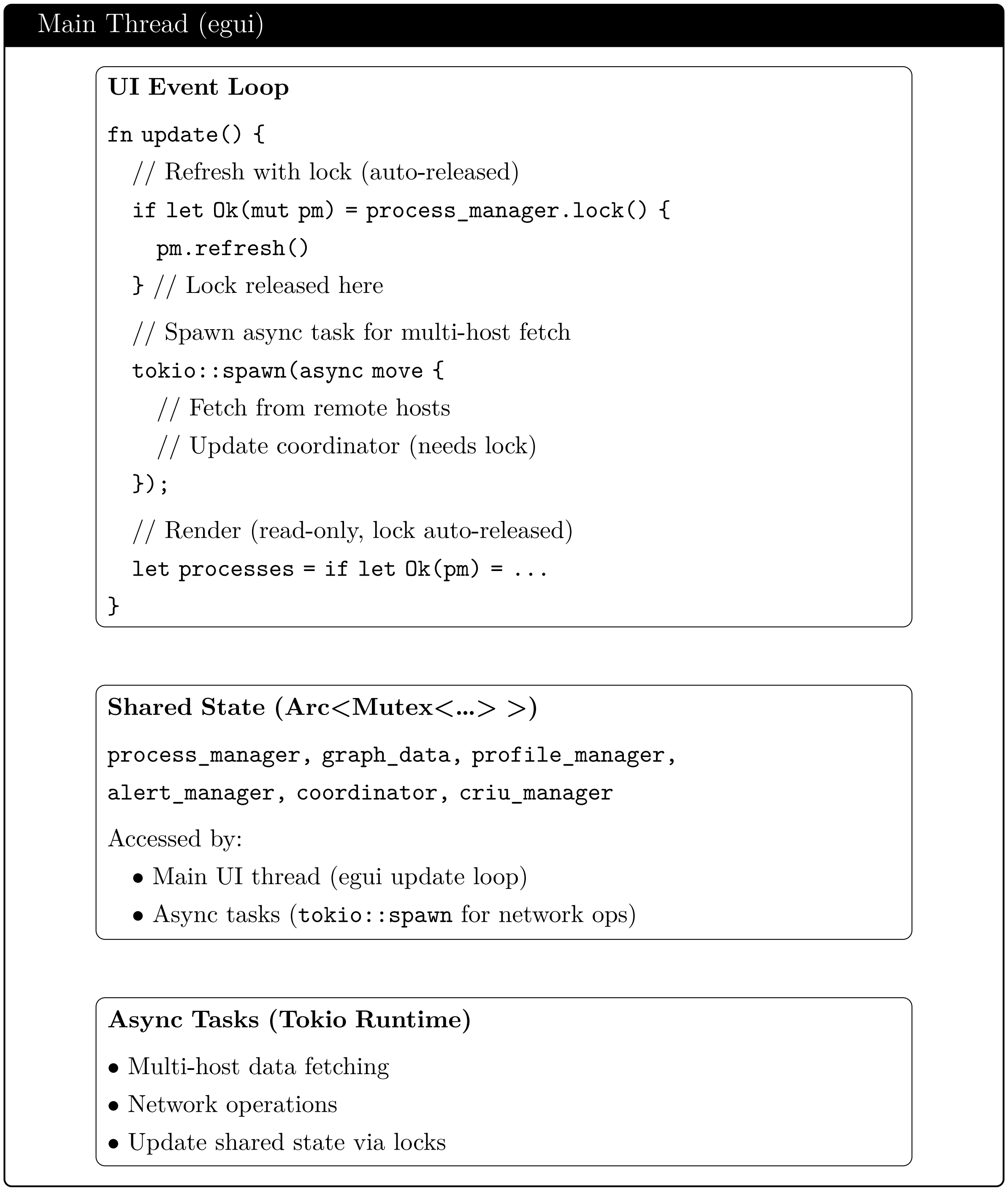
This diagram breaks down the main responsibilities of the `ProcessManager`: maintaining process state, refreshing and filtering the list, issuing control actions (kill, stop, nice, etc.), and integrating with Linux process metadata under `/proc`.



```
loop {
  handle_input()   // Blocking
  process_manager.refresh()   // Blocking
  render()   // Blocking
}
```

Figure 4: TUI single-threaded execution.

Here we show the TUI mode as a single blocking event loop. The main thread owns the `ProcessManager` directly, so no mutexes are needed and all input, refresh, and rendering happen sequentially.



```
fn update() {
  // Refresh with lock (auto-released)
  if let Ok(mut pm) = process_manager.lock() {
    pm.refresh()
  } // Lock released here
  // Spawn async task for multi-host fetch
  tokio::spawn(async move {
    // Fetch from remote hosts
    // Update coordinator (needs lock)
  });
  // Render (read-only, lock auto-released)
  let processes = if let Ok(pm) = ...
}
```

`process_manager`, `graph_data`, `profile_manager`, `alert_manager`, `coordinator`, `criu_manager`

Accessed by:

- Main UI thread (egui update loop)
- Async tasks (`tokio::spawn` for network ops)

- Multi-host data fetching
- Network operations
- Update shared state via locks

Figure 5: GUI mode with shared state and async tasks.

The GUI mode uses `Arc<Mutex<…> >` to share state between the egui update loop and background Tokio tasks. Refreshing and rendering take locks briefly, while asynchronous jobs fetch data and update coordinators.

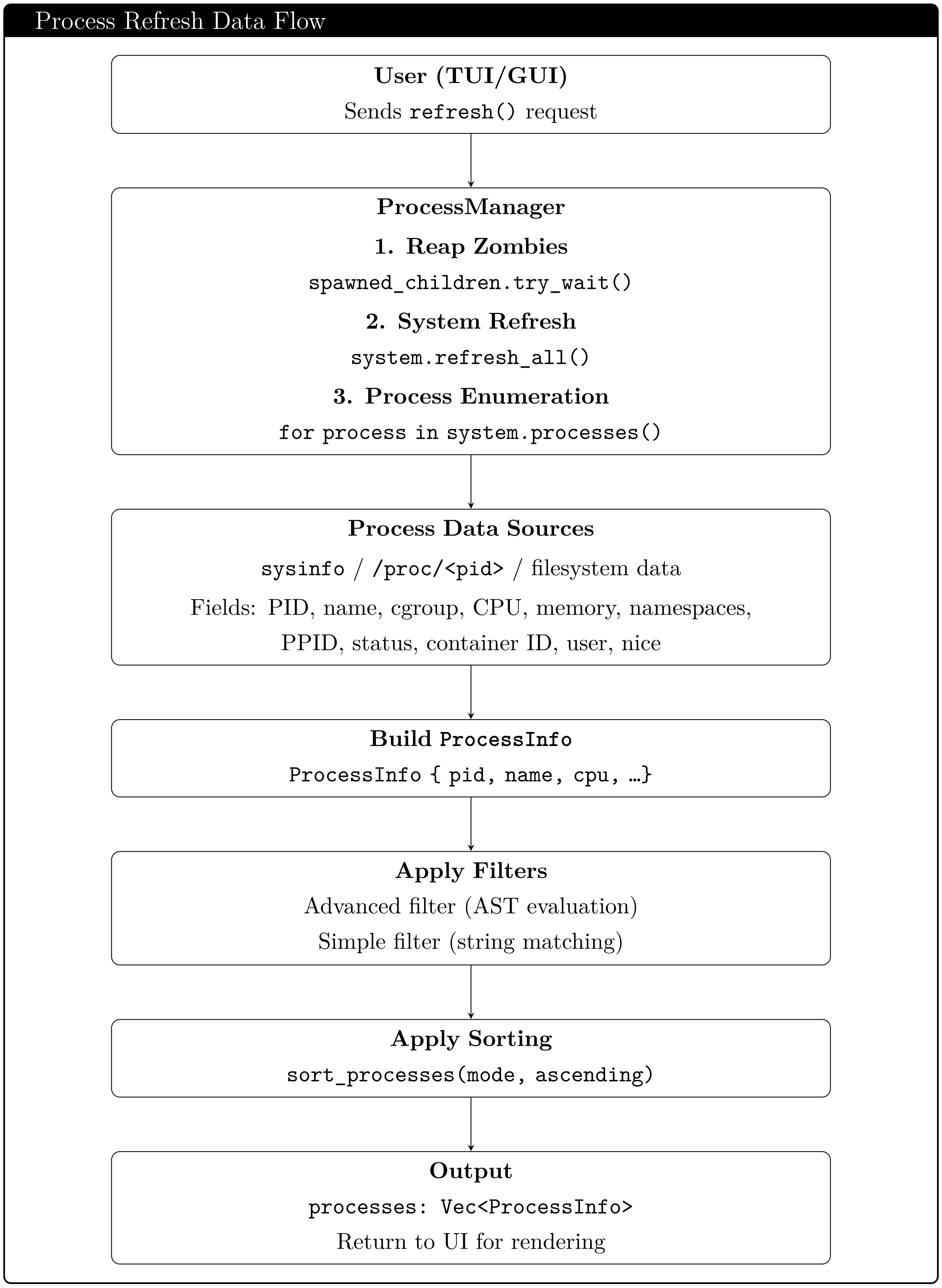

Process Refresh Data Flow
User (TUI/GUI)
Sends refresh() request
ProcessManager
1. Reap Zombies
spawned_children.try_wait()
2. System Refresh
system.refresh_all()
3. Process Enumeration
for process in system.processes()
Process Data Sources
sysinfo / /proc/<pid> / filesystem data
Fields: PID, name, cgroup, CPU, memory, namespaces, PPID, status, container ID, user, nice
Build ProcessInfo
ProcessInfo { pid, name, cpu, …}
Apply Filters
Advanced filter (AST evaluation)
Simple filter (string matching)
Apply Sorting
sort_processes(mode, ascending)
Output
processes: Vec<ProcessInfo>
Return to UI for rendering

Figure 6: End-to-end refresh pipeline.

This data-flow shows what happens when the user triggers a refresh: zombie processes are reaped, the system snapshot is updated, raw data from `/proc` and `sysinfo` is converted into `ProcessInfo` objects, then filtered, sorted, and returned to the UI.

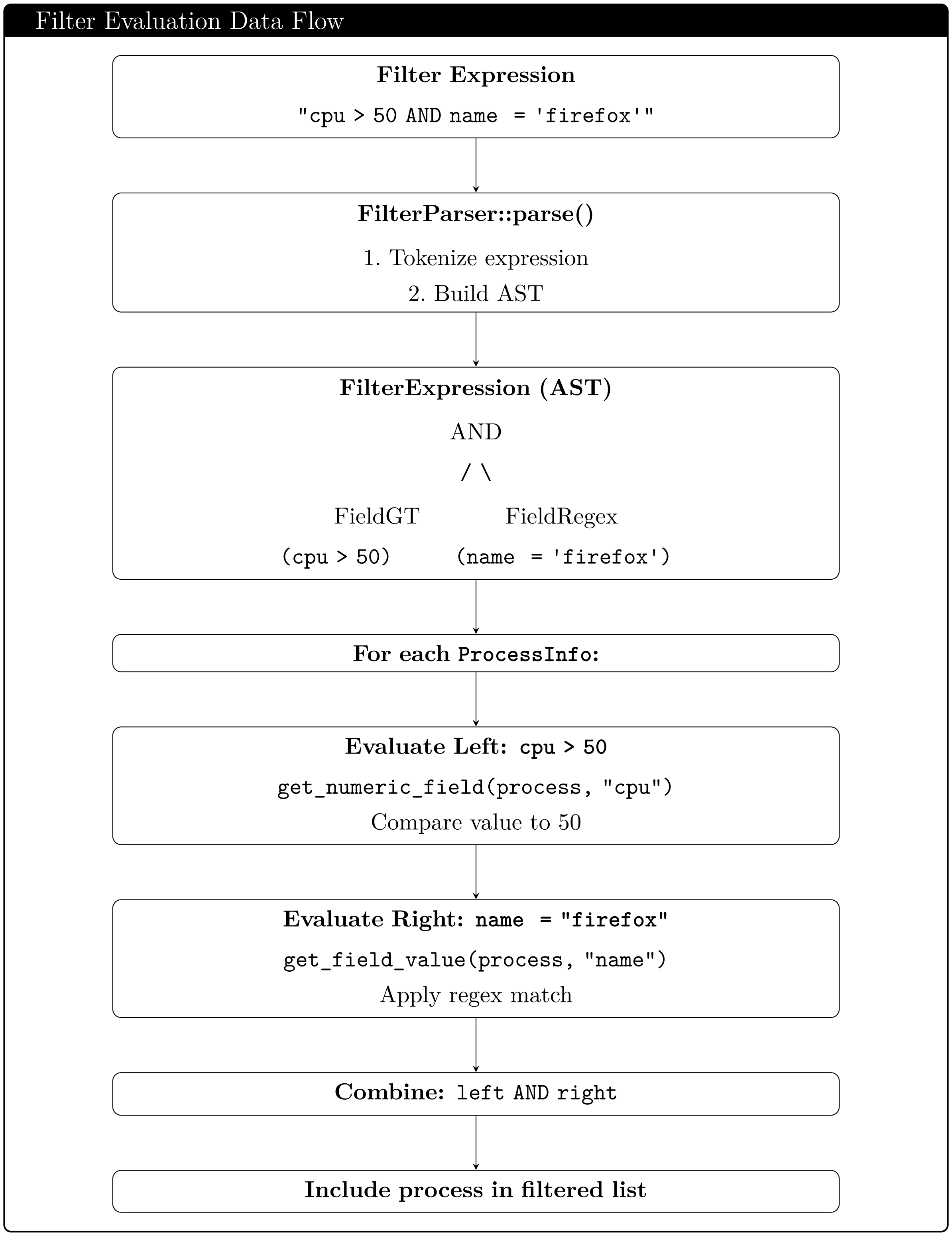


Figure 7: Filter parsing and evaluation.

The filter system parses the user's string into an AST, then evaluates each node (numeric comparisons and regexes) per process. Only processes where the full boolean expression is `true` remain in the filtered list.

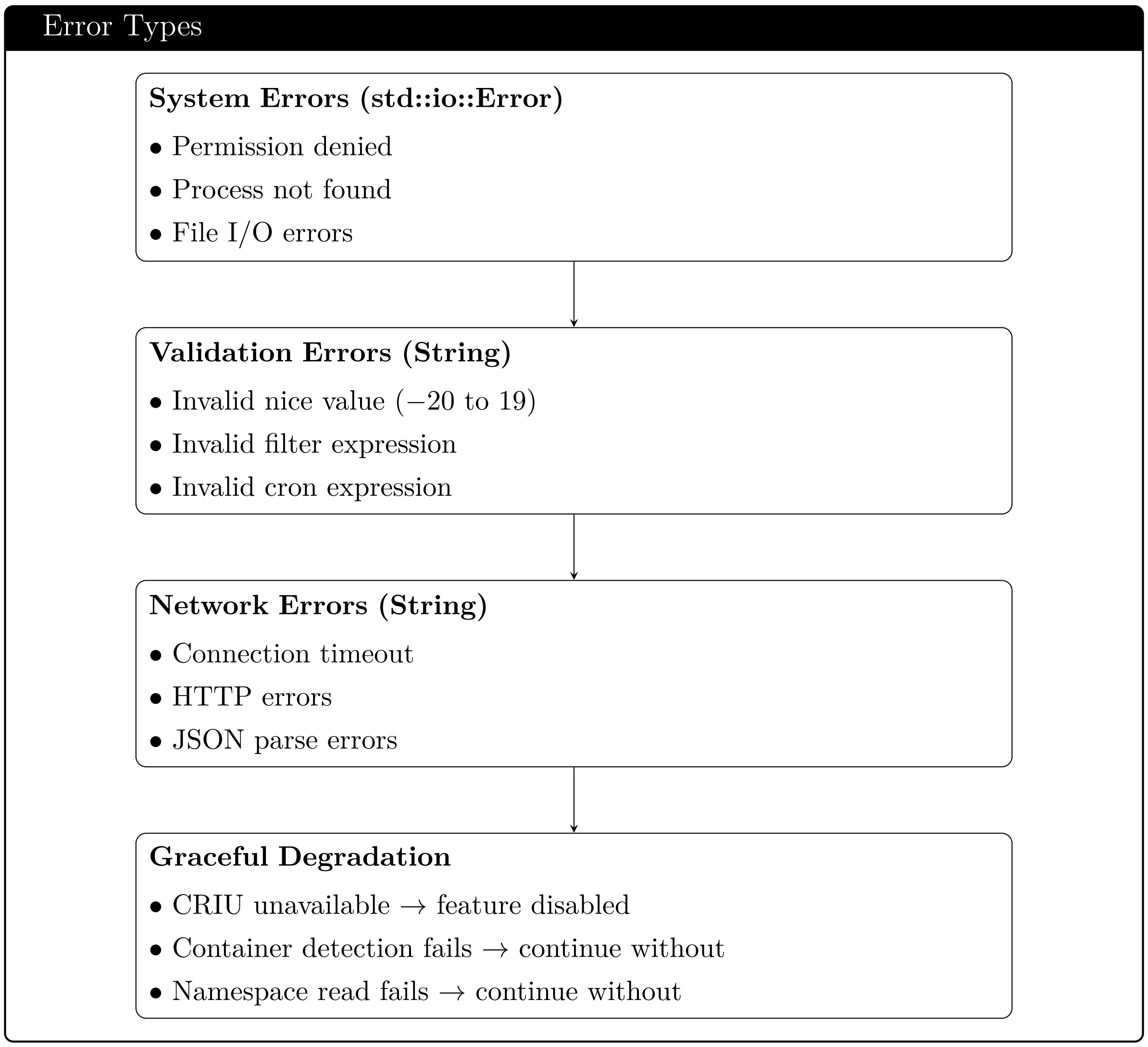

Error Types
System Errors (std::io::Error)
• Permission denied
• Process not found
• File I/O errors
Validation Errors (String)
• Invalid nice value (−20 to 19)
• Invalid filter expression
• Invalid cron expression
Network Errors (String)
• Connection timeout
• HTTP errors
• JSON parse errors
Graceful Degradation
• CRIU unavailable → feature disabled
• Container detection fails → continue without
• Namespace read fails → continue without

Figure 8: Error categories and degradation strategy.

We classify errors into system, validation, and network errors, and show how some failures cause features to be disabled while the rest of the application continues to operate.

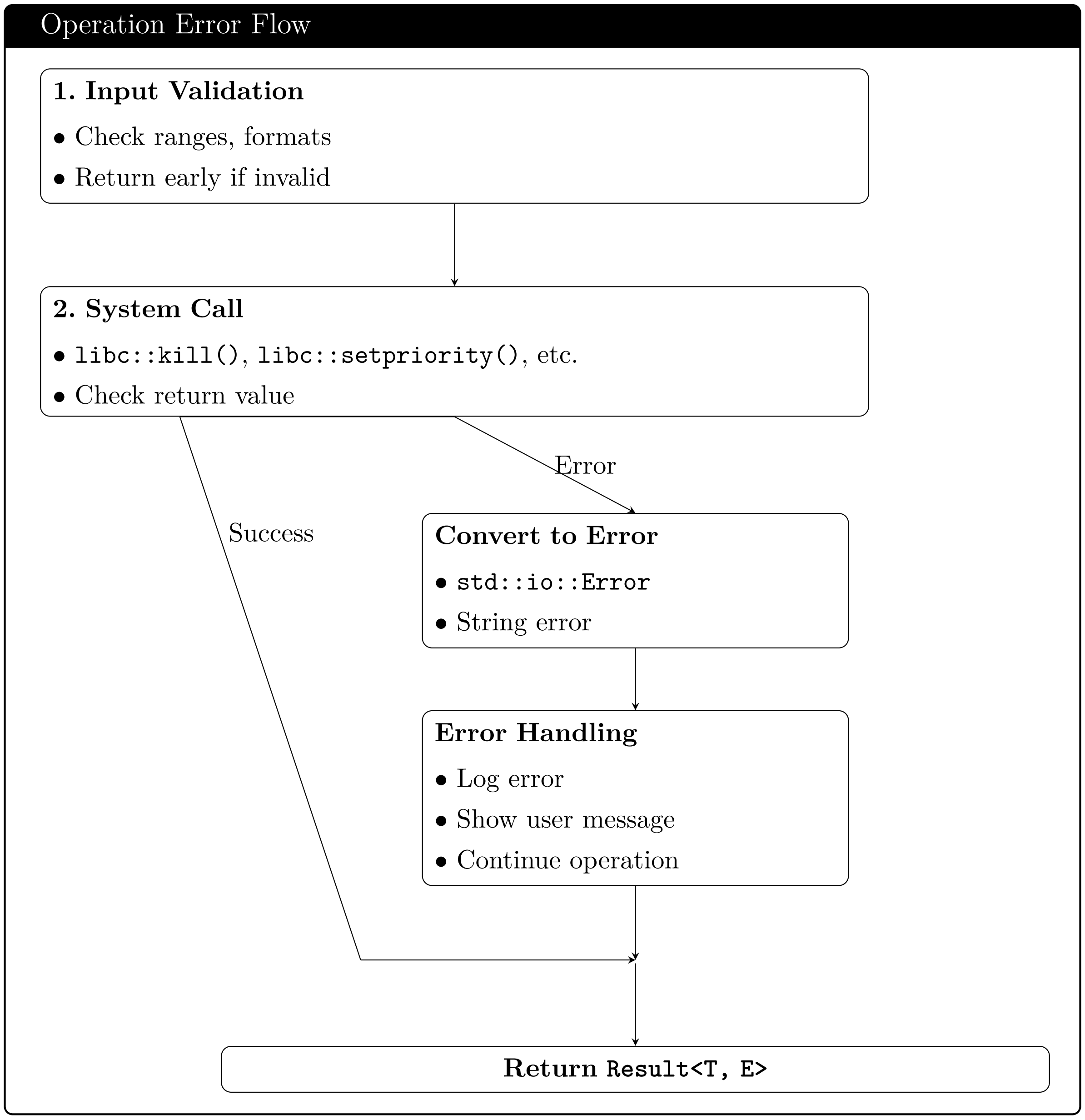


Figure 9: Success and error paths for an operation.

This flow shows how each user operation is validated, then mapped to a system call. On success we return directly, while on failure we convert the error, log it, inform the user, and still return a `Result<T, E>` value.

Configuration Storage

Configuration Files

```
~/.lpm/
  profiles.toml                                   (ProfileManager)
  alerts.toml                                     (AlertManager)
  scheduled_tasks.toml                            (Scheduler)
  checkpoints/
    checkpoint_<id>/
    checkpoints.toml
```

Format: **TOML** (Tom's Obvious Minimal Language)

Example `profiles.toml`

```
[[profiles]]
name = "build"
prioritize_processes = ["gcc", "make"]
hide_processes = ["firefox"]
[profiles.nice_adjustments]
"gcc" = -5
```

Figure 10: Configuration layout and example profile.

We store user configuration under `~/.lpm/` in TOML files: profiles, alerts, scheduled tasks, and checkpoints. The example shows how a profile can prioritise or hide processes and override nice values.

Locking Patterns

1. Short-Lived Locks (Error-Safe)

```
if let Ok(mut pm) = process_manager.lock() {
  pm.refresh();
} // Lock auto-released
// If lock fails, operation is skipped (no panic)
```

2. Read-Only Locks (Auto-Release)

```
let processes = if let Ok(pm) = process_manager.lock() {
  pm.get_processes().clone()
} else { Vec::new() };
// Lock auto-released
```

3. Nested Locks (Sequential)

```
if let Ok(pm) = process_manager.lock() {
  if let Ok(mut gd) = graph_data.lock() {
    gd.update(&pm);
  }
} // Inner then outer lock released
```

4. No Long-Held Locks

- Never hold locks across `await` points
- Never hold locks during UI rendering
- Release immediately after operation

5. RwLock for Read-Heavy Operations

- Use `RwLock` for async read-heavy agents
- Writes occur only on refresh
- Multiple concurrent reads are allowed

Figure 11: Locking guidelines for shared state.

Finally, this diagram summarises the main locking patterns: short critical sections, read-only locks for snapshots, sequential nested locks, and rules to avoid long-held locks, plus when to switch to `RwLock` for read-heavy workloads.

5 Rust Language Choice and Implementation Challenges

5.1 Overview

The aim of this project was to create a Linux Process Manager at the system level that enables real-time monitoring, delivers low-latency updates to the user interface, and communicates securely with the Linux kernel. Rust was chosen because of its guarantees on memory safety, reliable performance, and safe concurrency — qualities that are crucial for a process manager.

5.2 Reasons Behind Selecting Rust

5.2.1 System-Level Requirements

- **Direct system calls.** The tool frequently interacts with `/proc` and relies on system calls such as `kill` and `setpriority`. Rust's `libc` bindings offer typed access, preventing the fragility present in Python's `ctypes`.
- **Memory safety.** Rust's ownership model avoids problems such as use-after-free bugs, race conditions, and buffer overflows, which is especially critical when managing extensive kernel-exposed data.
- **Zero-cost abstractions.** Rust allowed using higher-level features (iterators, pattern matching) without adding runtime overhead.

5.2.2 Performance

- **Real-time monitoring.** The software updates process data multiple times per second. Rust ensures low latency, free from garbage-collection pauses.

5.3 Major Modules – Linux Process Manager (Summary)

The Linux Process Manager is constructed on a framework that clearly segregates responsibilities into individual components. The system includes 17 modules, grouped into process management, user interfaces, filtering, visualization, resource management, automation, and multi-host functionality. The architecture aims to be maintainable, expandable, and reusable, and is compatible with both TUI and GUI environments.

5.3.1 Core Process Management

The primary module, `process.rs`, oversees all operations connected to the process lifecycle and system data collection. It defines the `ProcessInfo` structure, manages sorting, filtering, and state updates, and interacts directly with the filesystem. It provides functions for refreshing process data, sending signals, modifying scheduling priorities, and launching new processes.

Supporting this module, `process_log.rs` stores information on processes to facilitate uptime monitoring and debugging for system admins. Meanwhile, `process_group.rs` categorizes processes further by cgroup, container, namespace, or user, allowing for resource evaluation.

5.3.2 User Interfaces

Access to the core is provided by two UI layers. The TUI (`ui.rs`) was created using `ratatui` and `crossterm`; it offers tables, multiple viewing modes, keyboard navigation, and color-coded system metrics.

The GUI (`gui.rs`) employs `egui` and provides visual displays: it features interactive charts, tabbed navigation, and mouse-driven actions. Both user interfaces function on shared state and interact with the same backend components.

5.3.3 Filtering and Querying

The `filter_parser.rs` module implements an expression-based filtering system that allows users to query processes using flexible conditions. For example, an expression such as:

```
memory > 200 AND name ~= "^fi"
```

returns all processes consuming more than 200 MB of memory and whose names start with "fi". The parser supports boolean operations, numeric and string comparisons, regular expressions, and field-specific predicates through an abstract syntax tree (AST). For instance, a query can also be expressed in a programmatic style as:

```
mem > 200 && name.starts_with("fi")
```

These capabilities enable expressive filtering based on CPU thresholds, memory usage, regex-driven name matching, and other field-level criteria.

5.3.4 Visualization and Monitoring

`graph.rs` and `per_process_graph.rs` handle system and per-process metrics, respectively. Both modules store CPU and memory usage, collecting system-level metrics like CPU, memory, disk I/O, and temperature while supplying time-series data to user interfaces. They support the identification of trends, spike recognition, and long-term monitoring of processes.

5.3.5 Resource Management

The `container_view.rs` and `namespace_view.rs` modules provide visibility into containers and namespaces, recognizing Docker/Podman containers, and Linux namespaces. These components extract metadata from cgroup paths and `/proc` to facilitate inspection of containerized or namespaced processes.

5.3.6 Automation and Intelligence

There exist four modules that provide automation capabilities. `scheduler.rs` runs jobs activated by cron timings or intervals, covering restarts, cleanup operations, and rule execution. `alert.rs` oversees threshold criteria such as CPU and memory utilization or process stoppage, handling alerts accordingly. `profile.rs` applies focus-mode profiles by prioritizing or hiding processes. `scripting_rules.rs` manages automation driven by rules formed from filter expressions.

5.3.7 Multi-Host and Fault Tolerance (Uncomplete additional features)

`coordinator.rs` and `agent.rs` allow for monitoring between hosts via HTTP communication to access process information. `criu_manager.rs` boosts fault tolerance by integrating CRIU to checkpoint and restore processes, enabling recovery using stored snapshots.

5.4 Challenges

5.4.1 Concurrency & State Management

Shared State Complexity. The application extensively utilizes `Arc<Mutex<T>>` to share data between background data-collection threads and the UI rendering thread.

Issue: Prevent deadlocks when several managers need to be accessed.
Solution: Reduce lock contention to maintain UI responsiveness, as 60 FPS, when the system experiences a heavy workload.

Async/Sync Bridge. The project's code combines asynchronous components (Tokio) with synchronous components (`sysinfo`, `ratatui` for rendering).

Challenge: Bridging async and sync worlds without blocking the async runtime nor introducing delays in the UI thread.

Rust GUI Complexity. Rust provides assurances regarding memory safety, yet its comprehensive GUI ecosystem remains divided. Creating a contemporary UI in Rust requires combining more fundamental crates, like `ratatui`, `winit`, and `crossbeam` instead of relying on a single well-developed GUI framework. This approach raises the development complexity, especially when combined with concurrency and real-time data rendering.

5.4.2 System Interaction & Performance

TOCTOU: Race Conditions. Processes are short-lived. A process may exist when listed, but disappear before its CPU/memory stats are queried.

Challenge: Handling `ProcessNotFound`, without any crashes or UI flickering "ghost entries."

Permission Boundaries. Operations like `renice`, or `kill`, checkpointing require elevated privileges.

Challenge: This application usually runs as a non-root user, so it needs to handle "Permission Denied" errors cleanly and propagate that to the user without failing.

6 Usage Guide – Linux Process Manager

6.1 Compilation

To compile the Linux Process Manager, navigate to the project directory and build using Cargo:

```
cd Linux_process_manager
cargo build
```

6.2 Running

6.2.1 TUI Mode (default)

To run the application in Text-based User Interface (TUI) mode:

```
cargo run
```

6.2.2 GUI Mode

To run the application in Graphical User Interface (GUI) mode:

```
cargo run -- --gui
```

6.3 Basic Navigation (TUI)

6.3.1 Main Process List

- ↑/↓ – Navigate processes
- **1** – Filter/Sort menu

- **2** – Change priority (nice value)
- **3** – Kill/Stop/Terminate/Continue
- **4** – Per-process graphs
- **5** – Process log
- **6** – Help
- **g** – Grouped view (cgroups/containers)
- **j** – Job scheduler
- **n** – Start new process
- **p** – Profile management
- **A** – Alert management
- **c** – Checkpoint management
- **m** – Toggle multi-select mode
- **Space/Enter** – Select/deselect process (multi-select)
- **s** – Statistics dashboard
- **q** – Quit

6.3.2 General Navigation

- **Esc** – Go back/exit current view
- **Tab** – Switch between input fields
- **Enter** – Confirm/execute action
- **Backspace** – Delete character

6.3.3 Statistics View

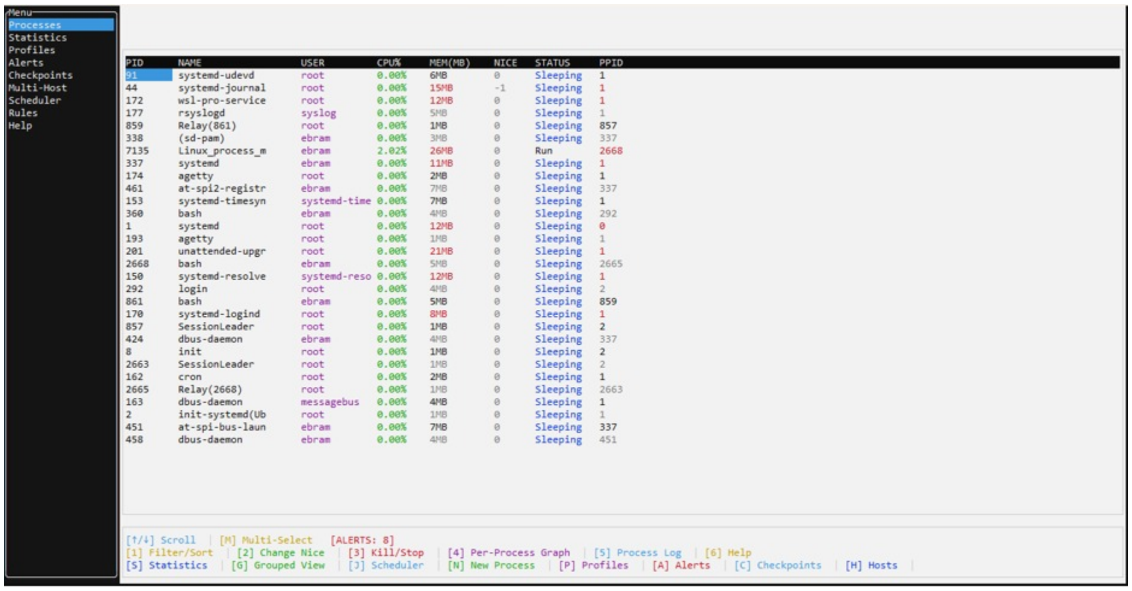
- **1–6** – Switch tabs (Graphs, Overview, CPU, Memory, Disk, Processes)
- **s/q/Esc** – Return to process list

6.3.4 Filter/Sort Menu

- **1** – Sort by field
- **2** – Filter processes
- **3** – Advanced filter
- **x** – Script Filtering
- **Esc** – Back to process list

6.4 Walkthrough Guide

Start screen:

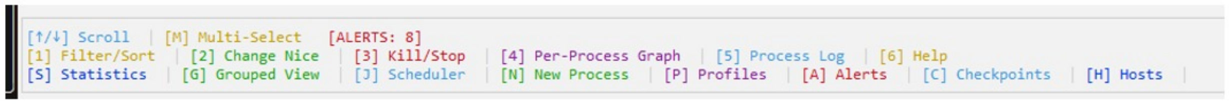


```
Menu
Processes
Statistics
Profiles
Alerts
Checkpoints
Multi-Host
Scheduler
Rules
Help
```

```
PID   NAME              USER         CPU%    MEM(MB)   NICE   STATUS     PPID
91    systemd-udevd     root         0.00%   6MB       0      Sleeping   1
44    systemd-journal   root         0.00%   15MB      -1     Sleeping   1
172   wsl-pro-service   root         0.00%   12MB      0      Sleeping   1
177   rsyslogd          syslog       0.00%   5MB       0      Sleeping   1
859   Relay(861)        root         0.00%   1MB       0      Sleeping   857
338   (sd-pam)          ebram        0.00%   3MB       0      Sleeping   337
7135  Linux_process_m   ebram        2.02%   26MB      0      Run        2668
337   systemd           ebram        0.00%   11MB      0      Sleeping   1
174   agetty            root         0.00%   2MB       0      Sleeping   1
461   at-spi2-registr   ebram        0.00%   7MB       0      Sleeping   337
153   systemd-timesyn   systemd-time 0.00%   7MB       0      Sleeping   1
360   bash              ebram        0.00%   4MB       0      Sleeping   292
1     systemd           root         0.00%   12MB      0      Sleeping   0
193   agetty            root         0.00%   1MB       0      Sleeping   1
201   unattended-upgr   root         0.00%   21MB      0      Sleeping   1
2668  bash              ebram        0.00%   5MB       0      Sleeping   2665
150   systemd-resolve   systemd-reso 0.00%   12MB      0      Sleeping   1
292   login             root         0.00%   4MB       0      Sleeping   2
861   bash              ebram        0.00%   5MB       0      Sleeping   859
170   systemd-logind    root         0.00%   8MB       0      Sleeping   1
857   SessionLeader     root         0.00%   1MB       0      Sleeping   2
424   dbus-daemon       ebram        0.00%   4MB       0      Sleeping   337
8     init              root         0.00%   1MB       0      Sleeping   2
2663  SessionLeader     root         0.00%   1MB       0      Sleeping   2
162   cron              root         0.00%   2MB       0      Sleeping   1
2665  Relay(2668)       root         0.00%   1MB       0      Sleeping   2663
163   dbus-daemon       messagebus   0.00%   4MB       0      Sleeping   1
2     init-systemd(Ub   root         0.00%   1MB       0      Sleeping   1
451   at-spi-bus-laun   ebram        0.00%   7MB       0      Sleeping   337
458   dbus-daemon       ebram        0.00%   4MB       0      Sleeping   451

[↑/↓] Scroll  | [M] Multi-Select   [ALERTS: 8]
[1] Filter/Sort  | [2] Change Nice  | [3] Kill/Stop  | [4] Per-Process Graph  | [5] Process Log  | [6] Help
[S] Statistics  | [G] Grouped View  | [J] Scheduler  | [N] New Process  | [P] Profiles  | [A] Alerts  | [C] Checkpoints  | [H] Hosts  |
```

Figure 12: Start screen.

The options available are below:



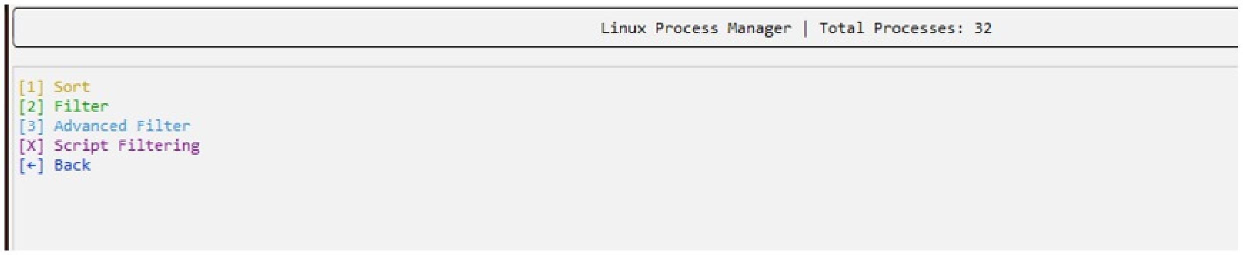
```
[↑/↓] Scroll   | [M] Multi-Select   [ALERTS: 8]
[1] Filter/Sort   | [2] Change Nice   | [3] Kill/Stop   | [4] Per-Process Graph   | [5] Process Log   | [6] Help
[S] Statistics   | [G] Grouped View   | [J] Scheduler   | [N] New Process   | [P] Profiles   | [A] Alerts   | [C] Checkpoints   | [H] Hosts   |
```

Figure 13: The options available in the menu.

Let's take some examples of how to use it:

1- Sorting by memory

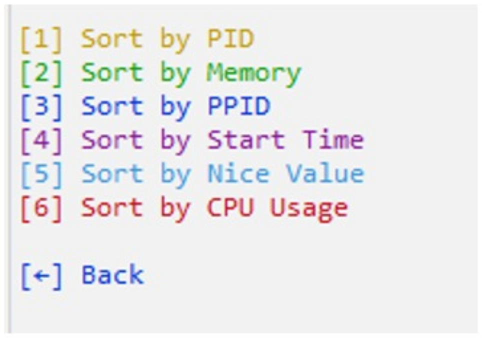
Press 1, then this would appear:



```
Linux Process Manager | Total Processes: 32

[1] Sort
[2] Filter
[3] Advanced Filter
[X] Script Filtering
[←] Back
```

Figure 14: Filter/Sort menu after pressing 1.

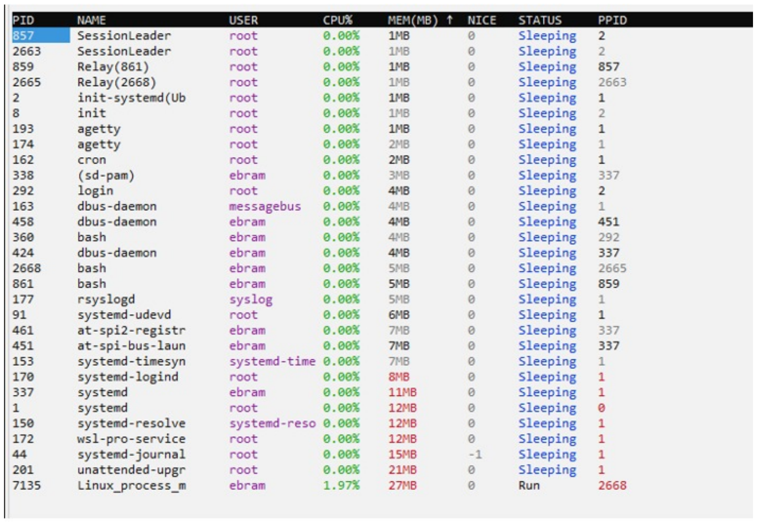
Then, choose 1 again to sort:



```
[1] Sort by PID
[2] Sort by Memory
[3] Sort by PPID
[4] Sort by Start Time
[5] Sort by Nice Value
[6] Sort by CPU Usage

[←] Back
```

Figure 15: Sort menu after choosing 1 again.

Multiple options appear here. Then choose 2.

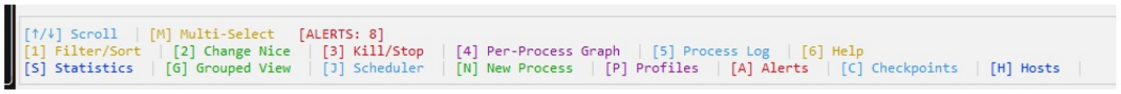


```
PID   NAME             USER          CPU%    MEM(MB) ↑  NICE   STATUS     PPID
857   SessionLeader    root          0.00%   1MB        0      Sleeping   2
2663  SessionLeader    root          0.00%   1MB        0      Sleeping   2
859   Relay(861)       root          0.00%   1MB        0      Sleeping   857
2665  Relay(2668)      root          0.00%   1MB        0      Sleeping   2663
2     init-systemd(Ub  root          0.00%   1MB        0      Sleeping   1
8     init             root          0.00%   1MB        0      Sleeping   2
193   agetty           root          0.00%   1MB        0      Sleeping   1
174   agetty           root          0.00%   2MB        0      Sleeping   1
162   cron             root          0.00%   2MB        0      Sleeping   1
338   (sd-pam)         ebram         0.00%   3MB        0      Sleeping   337
292   login            root          0.00%   4MB        0      Sleeping   2
163   dbus-daemon      messagebus    0.00%   4MB        0      Sleeping   1
458   dbus-daemon      ebram         0.00%   4MB        0      Sleeping   451
360   bash             ebram         0.00%   4MB        0      Sleeping   292
424   dbus-daemon      ebram         0.00%   4MB        0      Sleeping   337
2668  bash             ebram         0.00%   5MB        0      Sleeping   2665
861   bash             ebram         0.00%   5MB        0      Sleeping   859
177   rsyslogd         syslog        0.00%   5MB        0      Sleeping   1
91    systemd-udevd    root          0.00%   6MB        0      Sleeping   1
461   at-spi2-registr  ebram         0.00%   7MB        0      Sleeping   337
451   at-spi-bus-laun  ebram         0.00%   7MB        0      Sleeping   337
153   systemd-timesyn  systemd-time  0.00%   7MB        0      Sleeping   1
170   systemd-logind   root          0.00%   8MB        0      Sleeping   1
337   systemd          ebram         0.00%   11MB       0      Sleeping   1
1     systemd          root          0.00%   12MB       0      Sleeping   0
150   systemd-resolve  systemd-reso  0.00%   12MB       0      Sleeping   1
172   wsl-pro-service  root          0.00%   12MB       0      Sleeping   1
44    systemd-journal  root          0.00%   15MB       -1     Sleeping   1
201   unattended-upgr  root          0.00%   21MB       0      Sleeping   1
7135  Linux_process_m  ebram         1.97%   27MB       0      Run        2668
```

Figure 16: The result appears correctly with processes sorted according to memory.

2- Enabling alerts

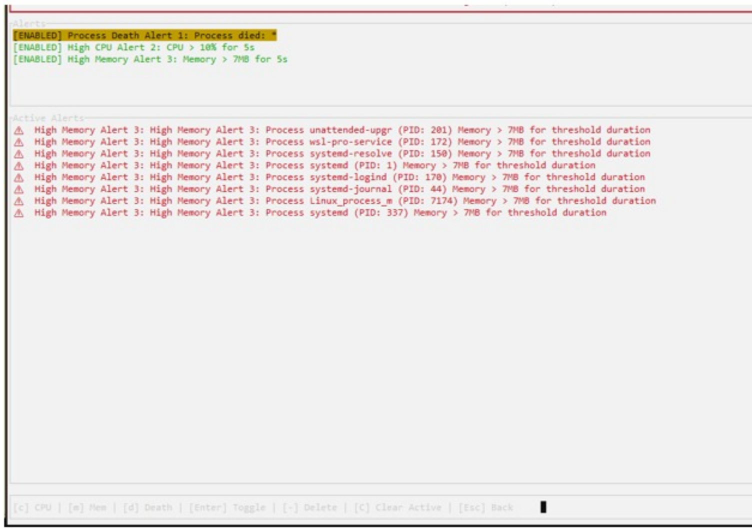
Again, check the menu:



```
[↑/↓] Scroll   | [M] Multi-Select    [ALERTS: 8]
[1] Filter/Sort  | [2] Change Nice  | [3] Kill/Stop  | [4] Per-Process Graph  | [5] Process Log  | [6] Help
[S] Statistics  | [G] Grouped View  | [J] Scheduler  | [N] New Process  | [P] Profiles  | [A] Alerts  | [C] Checkpoints  | [H] Hosts  |
```

Figure 17: Menu showing the Alerts option.

Choose A,



```
Alerts
[ENABLED] Process Death Alert 1: Process died: *
[ENABLED] High CPU Alert 2: CPU > 10% for 5s
[ENABLED] High Memory Alert 3: Memory > 7MB for 5s

Active Alerts
⚠  High Memory Alert 3: High Memory Alert 3: Process unattended-upgr (PID: 201) Memory > 7MB for threshold duration
⚠  High Memory Alert 3: High Memory Alert 3: Process wsl-pro-service (PID: 172) Memory > 7MB for threshold duration
⚠  High Memory Alert 3: High Memory Alert 3: Process systemd-resolve (PID: 150) Memory > 7MB for threshold duration
⚠  High Memory Alert 3: High Memory Alert 3: Process systemd (PID: 1) Memory > 7MB for threshold duration
⚠  High Memory Alert 3: High Memory Alert 3: Process systemd-logind (PID: 170) Memory > 7MB for threshold duration
⚠  High Memory Alert 3: High Memory Alert 3: Process systemd-journal (PID: 44) Memory > 7MB for threshold duration
⚠  High Memory Alert 3: High Memory Alert 3: Process Linux_process_m (PID: 7174) Memory > 7MB for threshold duration
⚠  High Memory Alert 3: High Memory Alert 3: Process systemd (PID: 337) Memory > 7MB for threshold duration

[c] CPU | [m] Mem | [d] Death | [Enter] Toggle | [-] Delete | [C] Clear Active | [Esc] Back
```

Figure 18: Alerts screen.

Here are the whole options

C = creates cpu alert

M = created memory threshold alert

D = creates death alerts

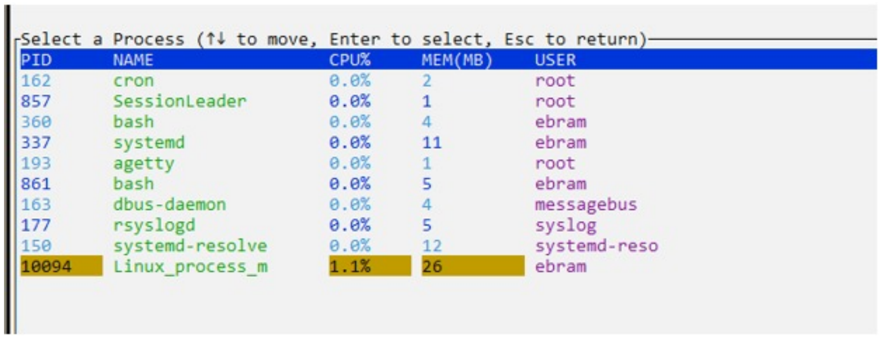
And all active alerts appear in the “active alerts” section.

3- Show per process graphs and statistics

According the options menu, choose 4:

This shows:

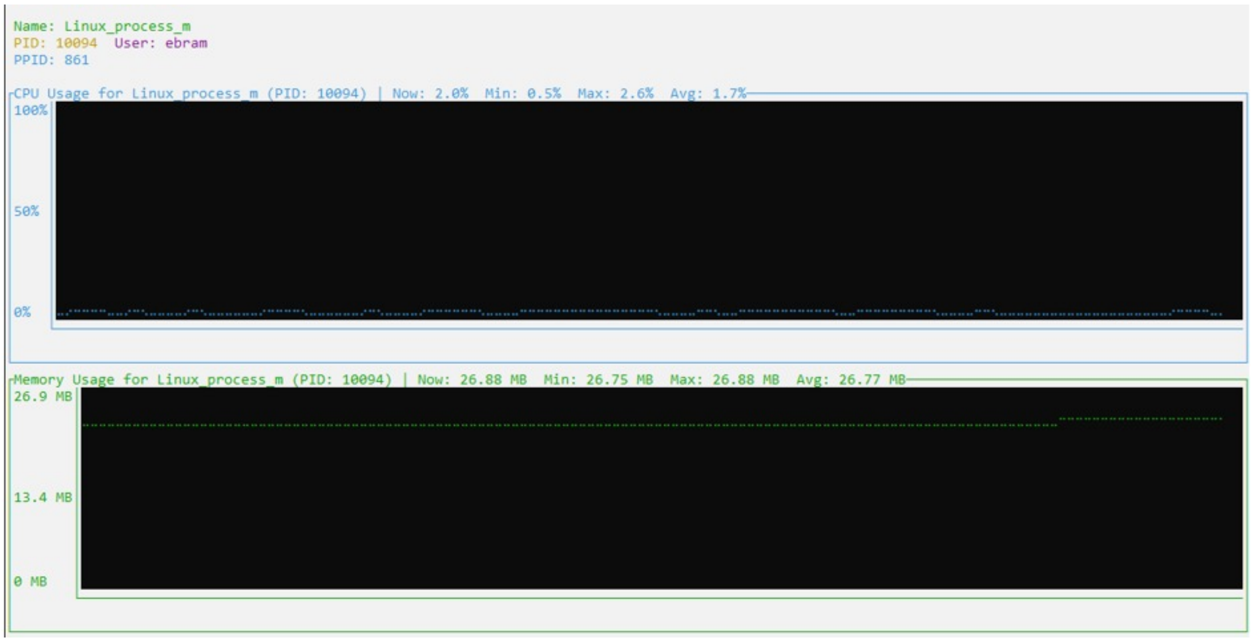
List of processes:



```
Select a Process (↑↓ to move, Enter to select, Esc to return)
PID        NAME                 CPU%       MEM(MB)       USER
162        cron                 0.0%       2             root
857        SessionLeader        0.0%       1             root
360        bash                 0.0%       4             ebram
337        systemd              0.0%       11            ebram
193        agetty               0.0%       1             root
861        bash                 0.0%       5             ebram
163        dbus-daemon          0.0%       4             messagebus
177        rsyslogd             0.0%       5             syslog
150        systemd-resolve      0.0%       12            systemd-reso
10094      Linux_process_m      1.1%       26            ebram
```

Figure 19: List of processes for per-process graphs.

Select one of them

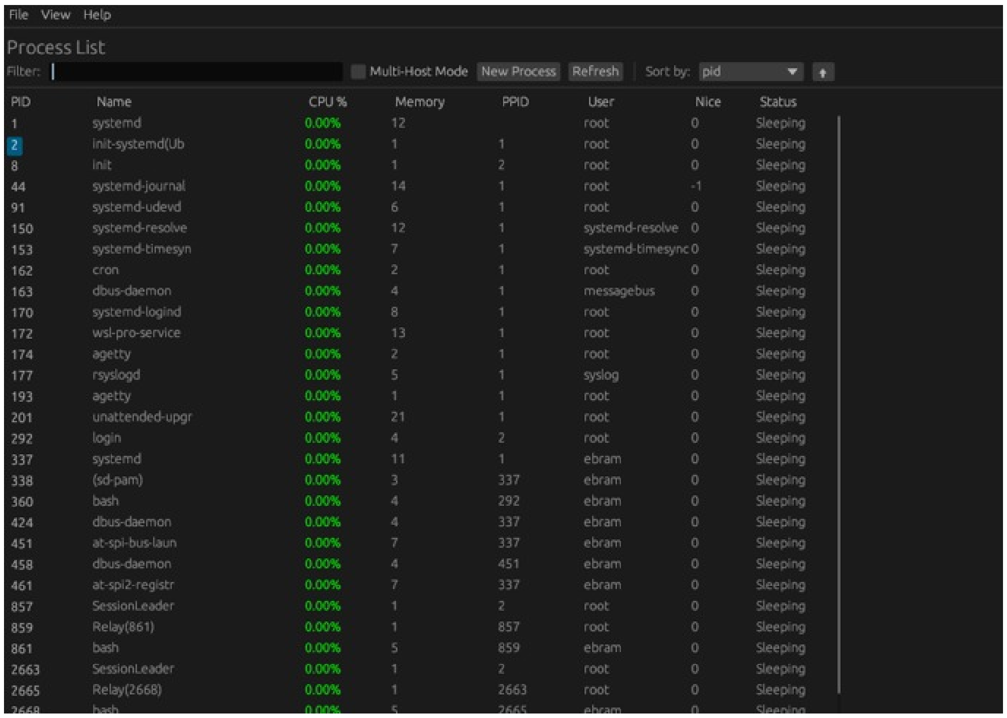


```
Name: Linux_process_m
PID: 10094  User: ebram
PPID: 861

CPU Usage for Linux_process_m (PID: 10094) | Now: 2.0%  Min: 0.5%  Max: 2.6%  Avg: 1.7%

Memory Usage for Linux_process_m (PID: 10094) | Now: 26.88 MB  Min: 26.75 MB  Max: 26.88 MB  Avg: 26.77 MB
```

Figure 20: This shows the graph.

GUI:

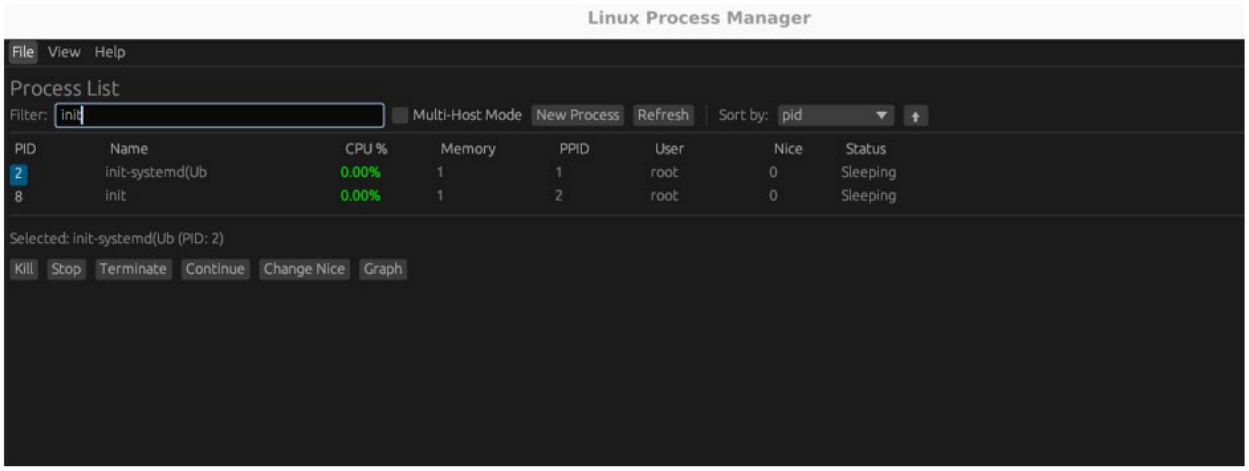
1- Searching for a specific process.
This is the main screen shows when the GUI runs



PID	Name	CPU %	Memory	PPID	User	Nice	Status
1	systemd	0.00%	12		root	0	Sleeping
2	init-systemd(Ub	0.00%	1	1	root	0	Sleeping
8	init	0.00%	1	2	root	0	Sleeping
44	systemd-journal	0.00%	14	1	root	-1	Sleeping
91	systemd-udevd	0.00%	6	1	root	0	Sleeping
150	systemd-resolve	0.00%	12	1	systemd-resolve	0	Sleeping
153	systemd-timesyn	0.00%	7	1	systemd-timesync	0	Sleeping
162	cron	0.00%	2	1	root	0	Sleeping
163	dbus-daemon	0.00%	4	1	messagebus	0	Sleeping
170	systemd-logind	0.00%	8	1	root	0	Sleeping
172	wsl-pro-service	0.00%	13	1	root	0	Sleeping
174	agetty	0.00%	2	1	root	0	Sleeping
177	rsyslogd	0.00%	5	1	syslog	0	Sleeping
193	agetty	0.00%	1	1	root	0	Sleeping
201	unattended-upgr	0.00%	21	1	root	0	Sleeping
292	login	0.00%	4	2	root	0	Sleeping
337	systemd	0.00%	11	1	ebram	0	Sleeping
338	(sd-pam)	0.00%	3	337	ebram	0	Sleeping
360	bash	0.00%	4	292	ebram	0	Sleeping
424	dbus-daemon	0.00%	4	337	ebram	0	Sleeping
451	at-spi-bus-laun	0.00%	7	337	ebram	0	Sleeping
458	dbus-daemon	0.00%	4	451	ebram	0	Sleeping
461	at-spi2-registr	0.00%	7	337	ebram	0	Sleeping
857	SessionLeader	0.00%	1	2	root	0	Sleeping
859	Relay(861)	0.00%	1	857	root	0	Sleeping
861	bash	0.00%	5	859	ebram	0	Sleeping
2663	SessionLeader	0.00%	1	2	root	0	Sleeping
2665	Relay(2668)	0.00%	1	2663	root	0	Sleeping
2668	bash	0.00%	5	2665	ebram	0	Sleeping

Figure 21: Main GUI screen.

Then we can type in the search bar the name of the process needed, for example, “systemd.” As shown:



PID	Name	CPU %	Memory	PPID	User	Nice	Status
2	init-systemd(Ub	0.00%	1	1	root	0	Sleeping
8	init	0.00%	1	2	root	0	Sleeping

Selected: init-systemd(Ub (PID: 2)

Figure 22: Filtering by “systemd” in the GUI.

7 Testing and Evaluation Summary

The Linux Process Manager (LPM) was extensively tested for functionality, stress, robustness, and performance. These tests were conducted on a range of platforms, including Linux, macOS, and WSL through a Docker environment. In total, over 100 test suites were done, besides the stress testing that was done to test the stability of the system over a prolonged period of 24 hours.

7.1 Functional Testing

All 18 key feature areas have been checked for 95–100% pass rate for over 100 test cases. The most important validation techniques, involving real-world testing, include:

- Artificially spawning hundreds of processes within Docker and the container to test the capability to detect containers, rebuild the cgroup hierarchy, and aggregate resources.
- Acting purposefully to bring certain cgroups to 90–100% CPU utilization through `stress-ng`, verifying that the CPU graphs, alerts, and throttling for each cgroup, as well as each container's CPU, had updated properly.
- Creating processes within custom mount, PID, user, and network namespaces using the commands `unshare` and `nsenter` for testing grouping functionality by namespace and drill-down capabilities.
- Directly changing the priorities of processes through terminals using `nice` and `renice` commands during the execution of LPM to ensure the immediate detection of changes in the nice values and policies.
- Creation of processes as non-root, and test permission boundaries, for example negative nice values.
- Handling one-shot, interval, and simplified cron-style tasks for killing or restarting processes based on certain conditions, along with testing the persistence of these processes.
- Activation of CPU, memory, and process death alerts through setting processes above established thresholds of time and duration.

Most features are implmented in both the gui, tui, but some features are unique. Minor feature omissions include simplified cron parsing (capabilities limited to 60-second granularity), read-only multiple host monitoring, incomplete I/O alerts, and lack of pre-restore checkpoint integrity validation.

7.2 Stress and Scalability Testing

LPM was tested under extraordinarily harsh conditions:

- Systems that start from 1,000 to 1,500 processes at the same time.
- High rate of process creations and terminations, approximately 10 per second.
- Maintained 85–95% system CPU utilization and 92–95% memory pressure by using stress and memory-hog tools.

Results consistently exceeded targets:

- Process refresh latency was maintained at 1.2–2.5 s even for 1,500 processes.
- Memory remained under 130 MB even for maximum tested scales.
- UI was fully responsive without any freezing or dropped frames.
- No memory leakage was found after 24-hour continuous operation.

7.3 Robustness and Fault Tolerance

Fault injection testing was done extensively. Handling of improper input, permission denied, network errors, sudden process death for refresh or control, and broken configuration files resulted in prominent error messages. Race conditions from concurrent kills, filter changes, or multi-host fetches were not a problem because of good locking and async practices. Permission boundaries remained firm, and there were no privilege escalation paths.

7.4 Performance Benchmark

Table 1: Performance benchmark.

Metric	**Typical (500 proc)**	**High Load (1,500 proc)**	**Target Met**
CPU overhead	2–5%	3–7%	Yes
Memory usage	110-130 MB	180-220 MB	Yes
Refresh latency	1.2 s	2.5 s	Yes
Complex filter latency	0.4–0.9 s	0.7 – 1. s	Yes
Process control latency	< 0.1 s	< 0.15 s	Yes

7.4.1 Comparison with `htop` / `top`

Compared to traditional tools like `htop` and `top`, LPM uses moderately more system resources. However, this overhead is justified by the significantly richer functionality it provides, including:

- real-time graphs,
- advanced filtering,
- alerting capabilities,
- CRIU integration for checkpoint/restore,
- multi-host coordination, and
- automation features.

These capabilities go far beyond basic process listing and make LPM suitable for more advanced monitoring and control scenarios.

7.4.2 Known Issues That Need Attention

Functional Issues.

- **Minor:** Some mis-groupings occur when building PPID-based process trees.

Safety-Related Issue.

- **Medium (safety):** There is a use of `unsafe static mut` to calculate disk I/O speed, which may introduce a data race under concurrent access.

Feature Gaps.

- No remote control capabilities for processes on remote hosts.
- No checkpoint integrity checks before performing CRIU restores.
- Simplified cron support only (limited cron expression handling).
- Missing I/O alerts (I/O monitoring alert conditions are not fully implemented).

8 Limitations & Future Improvements

8.1 Current Limitations

The existing Linux Process Manager (LPM) code is quite good for monitoring, although its architectural and functional constraints make it difficult to work well for a distributed system, a larger system, or a security-related system.

Task Scheduling

The system only supports basic cron expressions for its scheduling system. It is limited since complex cron expressions, such as month, day-of-week, or ranges, are not supported.

Process Lifecycle Management

The system does not monitor dependents or service definitions. Process trees become simplified, making it difficult to monitor relationships between parents and children.

Security

Security constraints introduce a significant risk. In remote mode, there is a lack of authentication, transport-layer security, Role-Based Access Control, and audit logging. Trivial operations such as CRIU restore and remote-control tasks lack proper permissions.

Data Persistence

Exit logs, graph history, and statistics are maintained only in memory and are lost after each restart. There is no facility for investigation or trend analysis.

Performance & Scalability

Although adequate for 1000+ processes, the system can become less responsive for extremely high numbers of processes (e.g., > 10k processes). Caching mechanisms, indexed queries, or optimizations for long-term data collection are not employed.

User Interface

Theming, shortcut customization, plugins, and better visualizations are absent in the UI. The GUI and TUI toolsets are not fully integrated.

Finally, some added functionalities that are not included in the core functional requirements were started but could not be finished due to lack of time.

Remote Control & Multi-Host Management

Remote monitoring is provided only with read-only capabilities. Process control operations such as kill, stop, restart, etc., on remote systems cannot be performed by administrators. Communication between the agent and coordinator is neither encrypted nor authenticated, making the system vulnerable to eavesdropping and unauthorized access. Host discovery, grouping, and load-balancing capabilities are unavailable.

Fault Tolerance & Recovery

In LPM, a resilience layer is absent. There is no watchdog process, no automatic retries for network errors, and no system reset for background operations that result in a panic. The absence of a circuit breaker, exponential backoff, or consistency checks creates a situation where system failure requires manual intervention, possibly putting the system into a state of partial inconsistency.

Checkpointing

The integration of CRIU is left incomplete. The ability to resume checkpoints requires root privilege, but handling this is left unattended. It merely supplies dummy PIDs, fails to conduct integrity checks prior to resumption, and is incompatible with complex process trees. Checkpointing schedules, UI state, and metadata handling have yet to be implemented.

8.2 Future Improvements

Future development work includes extending LPM to form a secure, distributed, fault-tolerant, and flexible system that is appropriate for a larger-scale environment. These improvements cover remote control, security, persistence, automation, and enhanced monitoring.

Remote Control & Distributed Management

Future releases will bring full remote process control capabilities (kill, stop, restart, priority adjustments) along with secured authentication and TLS encryption. Host discovery, tagging, health checks, and group operations will make configuration easier and increase scalability. Initial development for remote control capabilities has already begun.

Fault Tolerance & Recovery

Adding a Resilience Layer A resilience layer, incorporating watchdog processes, auto-restart of components, state recovery, and failure handling through graceful degradation, will be introduced. Network-related resilience features such as circuit breakers, exponential backoff, and connection health checks will reduce downtime. Fault-tolerance development is already initiated but unfinished.

Task Scheduling

Future versions will support full cron expression strings, time zones, and task actions such as restart, cleanup, and templates. Tasks will be persisted, traced, and able to send failure notifications, among other capabilities.

Comprehensive Alerting

Alerts will be extended to include I/O, network, process, and custom metrics. Alerting channels for notifications will include email, Slack, webhooks, and SMS. Features for handling alerts will include alert history, suppression, deduplication, and escalation paths to accommodate enterprise-grade monitoring.

Process Lifecycle & Service Management

Process templates can be utilized to set up commands, environment variables, and dependencies. LPM can be used for service-like management, offering capabilities such as restart and health checks, along with a dependency graph. The process tree can be viewed graphically.

Security Updates

The plan is to implement RBAC, MFA, OAuth2/OIDC, LDAP integration, audit logging, certificate-based authentication, and rate limiting. Security hardening and audit tooling for regular reviews will be added to ensure that the deployment is safe.

Persistence & Analytics

Historical logs, long-term statistics, and configuration settings will be stored using SQLite or PostgreSQL. Trend analyses, outlier detection, custom report generation, export functions for CSV, JSON, or PDF, and time-range queries will be implemented. Advanced analyses and ML-driven outlier detection will also be added.

Performance & Scalability

Optimizations will target large numbers of processes and servers via caching, parallelism, lazy loading, and indexing. Distributed caching and connection pooling will support multiple hosts.

UI Optimization & Extensibility

Themes, custom hotkeys, enhanced graphing capabilities (process graphs, heat maps), and accessibility tools can be incorporated. Community-driven plugins and custom widgets can be developed through a plugin system.

References

[1] 10 best linux task managers. LinuxAndUbuntu. (n.d.).
Available at: `https://www.linuxandubuntu.com/home/10-best-linux-task-managers`

[2] 15 best linux task managers - Dunebook. (n.d.-a).
Available at: `https://www.dunebook.com/best-linux-task-managers/`

[3] Best task managers for linux – Linux Hint. (n.d.-b).
Available at: `https://linuxhint.com/task-managers-linux/`

[4] Hasan, M. (2025, November 19). *12 best task managers for linux system*. UbuntuPIT.
Available at: `https://www.ubuntupit.com/best-linux-task-managers-reviewed-for-linux-nerds/`

[5] How to launch and use the ubuntu task manager. IONOS Digital Guide. (2023, January 19).
Available at: `https://www.ionos.com/digitalguide/server/configuration/ubuntu-task-manager/`

[6] Kumar, A. (2020, February 9). *6 best task managers for linux*. FOSS Linux.
Available at: `https://www.fosslinux.com/27008/6-best-task-managers-for-linux.htm`

[7] Lecture 8. *Introduction to Linux Process Management*. (n.d.-d).
Available at: `https://www.kaznu.kz/content/files/news/folder23177/Lecture%208.pdf`

[8] Niazi, R. (2022, February 17). *Linux process management: The ultimate guide*. MUO.
Available at: `https://www.makeuseof.com/linux-process-management/`

[9] r00t. (2022, October 28). *The best linux task manager*. idroot.
Available at: `https://idroot.us/best-linux-task-manager/`

[10] Run linux on windows or mac with a virtual machine (VM). (n.d.-e). Microchip Developer.
Available at: `https://microchipdeveloper.com/linux:install-virtual-machine`

[11] Unix / Linux processes management. (n.d.-f). TutorialsPoint.
Available at: `https://www.tutorialspoint.com/unix/pdf/unix-processes.pdf`