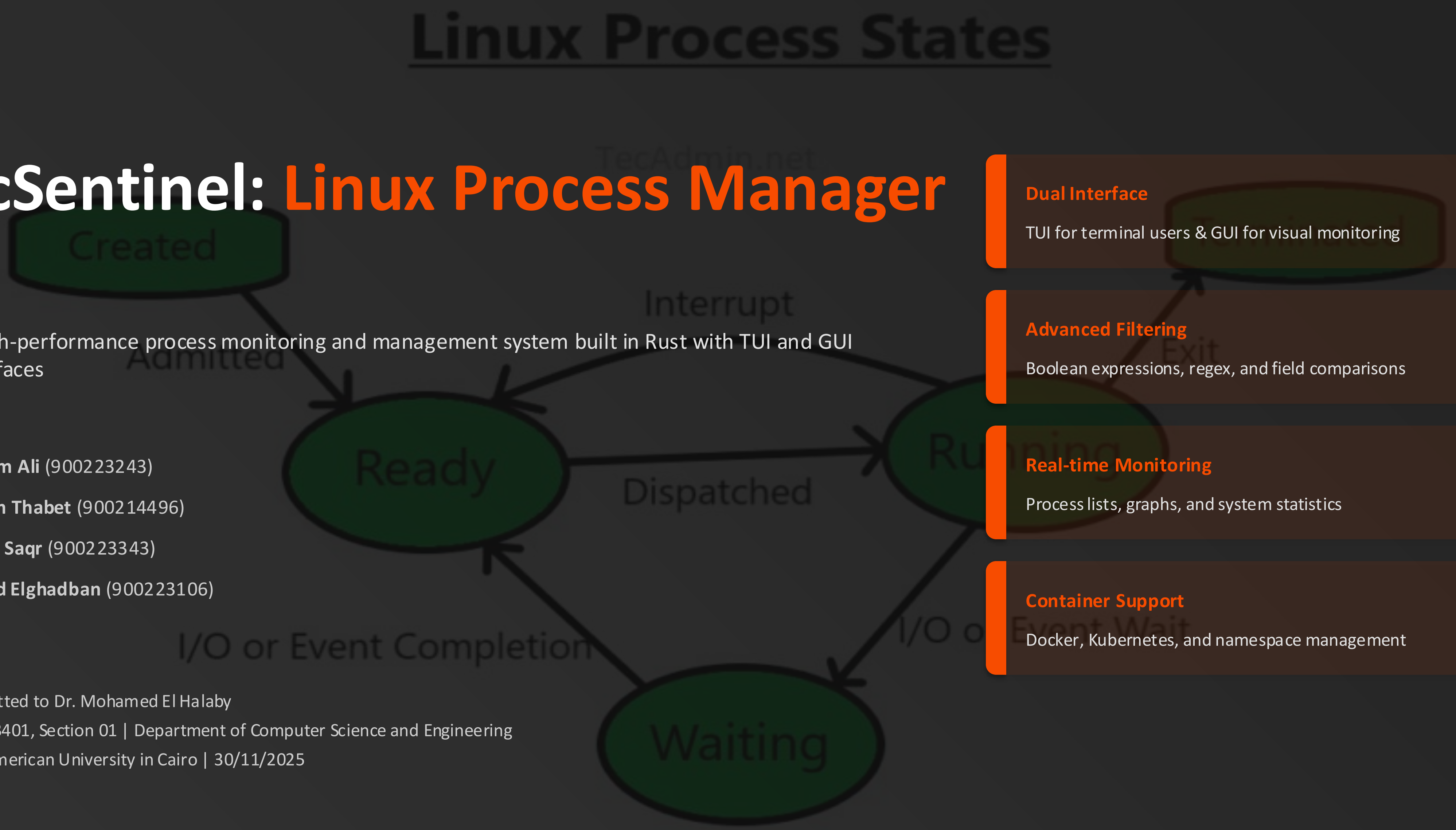


ProcSentinel: Linux Process Manager

A high-performance process monitoring and management system built in Rust with TUI and GUI interfaces

Adham Ali (900223243)

Ebram Thabet (900214496)

Omar Saqr (900223343)

Aabed Elghadban (900223106)

Submitted to Dr. Mohamed El Halaby
CSCE 3401, Section 01 | Department of Computer Science and Engineering
The American University in Cairo | 30/11/2025

Dual Interface

TUI for terminal users & GUI for visual monitoring

Advanced Filtering

Boolean expressions, regex, and field comparisons

Real-time Monitoring

Process lists, graphs, and system statistics

Container Support

Docker, Kubernetes, and namespace management

Team Members

Adham Ali

900223243

- **Real-time process listing** with PID, name, CPU, memory, PPID, start time, nice value, user, status
- **Process sorting** by CPU, memory, PID, PPID, start time, nice value
- **Confirmation prompts** for all process control operations
- **Dependency-aware termination** warnings for parent processes with children

Ebram Thabet

900214496

- **Per-process graphs** with real-time CPU and memory usage charts
- **Global system dashboard** with CPU, memory, swap, system health
- **Advanced filtering system** with multi-field queries, Boolean logic, regex
- **Focus-mode profiles** for workflow-based process management

Omar Saqr

900223343

- **Process control signals** (SIGKILL, SIGTERM, SIGSTOP, SIGCONT)
- **Resource grouping** by cgroups, containers, namespaces
- **Container view module** with per-container metrics
- **Job scheduling** with cron-like tasks for restarts and cleanup

Aabed Elghadban

900223106

- **CRIU integration** for checkpointing and restoring processes
- **Coordinator module** for remote host connections and synchronization
- **Complete GUI** using egui with tabs for all major features
- **TUI-GUI integration** with shared core architecture

Abstract & Summary

ProcSentinel is a high-performance Linux process manager built in Rust, featuring both TUI and GUI interfaces. It provides real-time monitoring, advanced filtering, and comprehensive process control capabilities.

Dual Interface Design

- ✓ Terminal UI for power users
- ✓ Visual GUI for monitoring

Advanced Filtering System

- ✓ Boolean expressions (AND, OR, NOT)
- ✓ Regular expressions support

Container & Namespace Support

- ✓ Docker & Kubernetes integration
- ✓ Linux namespace management

Automation Capabilities

- ✓ Profiles & alerts system
- ✓ Task scheduling & rules

Functional Requirements

Process List

- View processes with **PID, name, CPU, memory**
- Auto-refresh at configurable intervals
- Log process exit events

Process Control

- **Kill, stop, terminate, continue** processes
- Change priority (nice value: -20 to 19)
- Kill process trees recursively

Filtering & Sorting

- Simple filtering by user, name, PID
- **Boolean expressions** with AND, OR, NOT
- Regular expressions & field comparisons

User Interfaces

- **TUI** with keyboard navigation
- **GUI** with tabs for processes, graphs, alerts
- Color-coded information & confirmation dialogs

System Statistics

- Host info: hostname, OS, kernel, uptime
- **CPU, memory, disk** usage statistics
- Real-time graphs of system metrics

Grouping

- Group by **cgroups, containers, namespaces**
- Aggregate CPU & memory per group
- Expand/collapse & drill-down details

Profiles & Alerts

- Highlight or filter processes by profile
- **Alerts** triggered by CPU/memory thresholds
- Persist profiles to configuration file

Task Scheduling

- **Fixed intervals** and cron expressions
- One-shot execution support
- Execution log & timestamp tracking

Automation Rules

- Custom rules using **Rhai scripting**
- Access to process variables (cpu, mem, pid)
- Boolean return for process filtering

Non-Functional Requirements

Performance

- **Non-blocking UI** during data refresh
- Process list refresh: 1 second (default)
- Graphs refresh: 500ms (default)
- Handle **1000+ processes** without degradation

Reliability

- Graceful handling of terminated processes
- Handle kernel read failures & permission errors
- Explicit **mutex poisoning** handling
- Validate user inputs (nice values, PIDs)

Security

- **Root privilege check** for negative nice values
- Clear error messages for permission-denied
- Secure process control operations
- Safe handling of system calls

Usability

- Clear **keyboard shortcuts** in TUI
- Intuitive tabbed interface in GUI
- Color-coded CPU/memory thresholds
- Confirmation dialogs for destructive actions

Maintainability

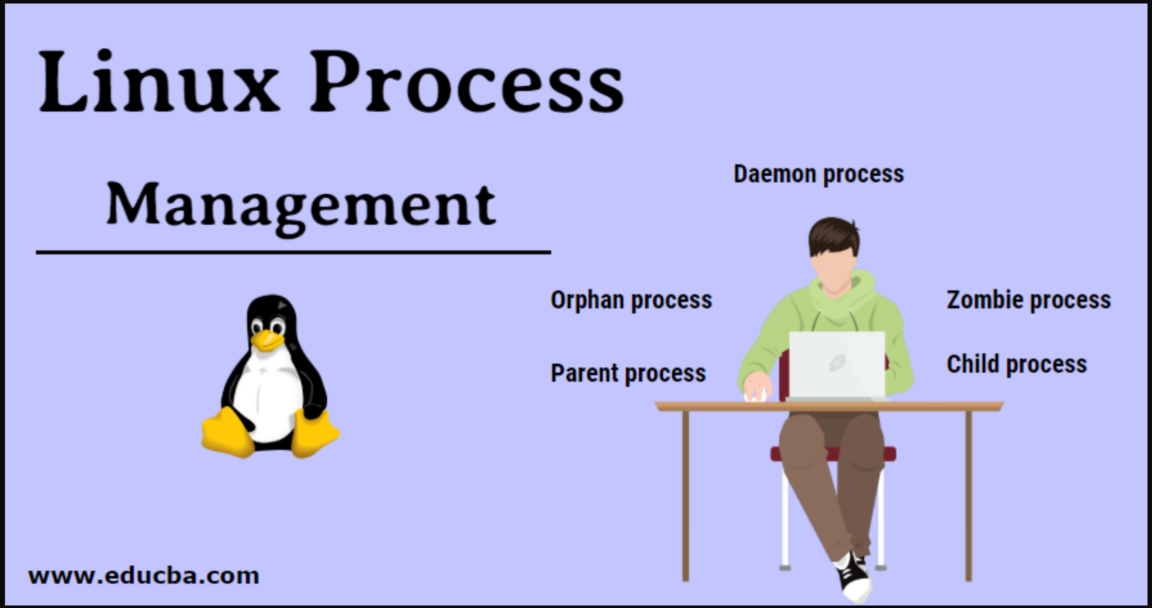
- **Rust** for memory safety
- Ownership model prevents memory issues
- Arc> for shared state
- Modular architecture with clear separation

Compatibility

- Primary target: **Linux**
- Fallbacks for macOS & WSL
- Docker container detection
- Standard Rust dependencies

Architecture

High-Level Architecture



Data Flow

1. **Process Refresh**
   Reap zombies, refresh system snapshot
2. **Data Collection**
   Extract from /proc & sysinfo
3. **Filtering**
   Apply simple & advanced filters
4. **Sorting**
   Order by selected fields
5. **UI Rendering**
   Display in TUI or GUI

Key Components

ProcessManager
Core process monitoring & control

FilterParser
Advanced filtering expressions

UI Layers
TUI & GUI interfaces

Grouping
cgroups, containers, namespaces

Shared State Management

Arc>
Thread-safe shared state

Short-Lived Locks
UI responsiveness

Async/Sync Bridge
Tokio & UI integration

Configuration
~/.lpm/ directory

Rust Implementation

Why Rust?

System-Level Requirements

Direct system calls, **memory safety**, zero-cost abstractions

Performance

Real-time monitoring, no garbage collection pauses

Safety

Ownership model prevents race conditions & buffer overflows

Major Modules

Core Process Management

ProcessInfo, process control, state updates

User Interfaces

TUI (ratatui) & **GUI** (egui)

Filtering & Querying

Expression parser, AST evaluation

Visualization

Graph data, per-process metrics

Resource Management

Container view, namespace view

Automation

Scheduler, alerts, profiles, rules

User Interface

TUI Mode

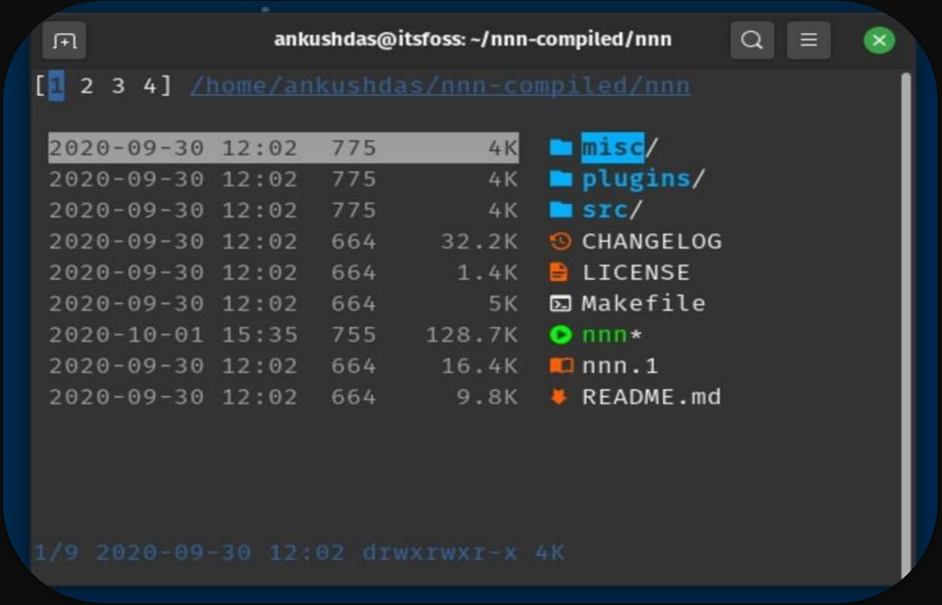
- **Keyboard navigation** with intuitive shortcuts
- Multiple view modes: process list, statistics, graphs
- Advanced filtering & sorting with color-coded metrics
- Grouped views for cgroups, containers, namespaces

```
cargo run
```

GUI Mode

- **Mouse-driven** interaction with visual feedback
- Tabbed interface: processes, graphs, alerts, schedules
- Real-time search & filtering with visual results
- Interactive charts & per-process graphs

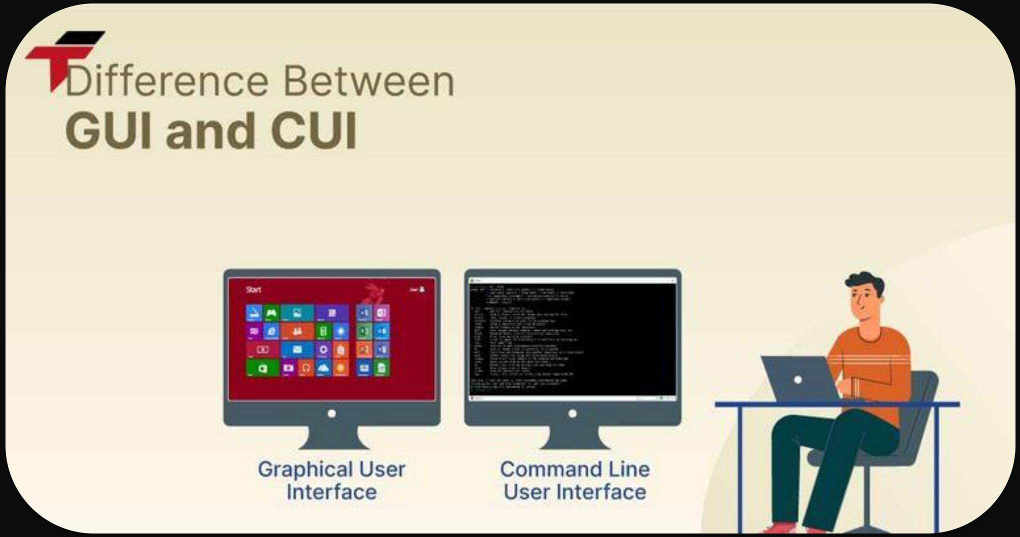
```
cargo run -- --gui
```



```
ankushdas@itsfoss: ~/nnn-compiled/nnn
[1 2 3 4] /home/ankushdas/nnn-compiled/nnn

2020-09-30 12:02  775       4K  misc/
2020-09-30 12:02  775       4K  plugins/
2020-09-30 12:02  775       4K  src/
2020-09-30 12:02  664    32.2K  CHANGELOG
2020-09-30 12:02  664     1.4K  LICENSE
2020-09-30 12:02  664       5K  Makefile
2020-10-01 15:35  755   128.7K  nnn*
2020-09-30 12:02  664    16.4K  nnn.1
2020-09-30 12:02  664     9.8K  README.md

1/9 2020-09-30 12:02 drwxrwxr-x 4K
```



Implementation Challenges

Concurrency & State Management

- ! **Shared State Complexity** with Arc>
- ! Preventing deadlocks with multiple managers
- ! Reducing lock contention for UI responsiveness

Solution

Short-lived locks, sequential nested locks, and avoiding locks across await points

System Interaction & Performance

- ! **TOCTOU Race Conditions** with short-lived processes
- ! Handling permission boundaries gracefully
- ! Avoiding UI flickering with "ghost entries"

Solution

Robust error handling, permission checks, and graceful degradation

Rust GUI Complexity

- ! **Fragmented GUI ecosystem** in Rust
- ! Combining multiple crates (ratatui, winit, crossbeam)
- ! Integration complexity with real-time data rendering

Solution

Shared backend logic for both interfaces, modular UI components

Async/Sync Bridge

- ! **Mixing async & sync** components (Tokio & sysinfo)
- ! Avoiding blocking the async runtime
- ! Preventing UI thread delays during data fetching

Solution

Background tasks with tokio::spawn, brief critical sections for locks

Testing & Evaluation

Functional Testing

- **95-100%** pass rate across 18 key feature areas
- Container detection & cgroup hierarchy validation
- Process priority changes & permission boundaries
- Alert activation for CPU, memory, and process death

Stress & Scalability

- Tested with **1,000-1,500** concurrent processes
- High rate of process creation/termination (10/sec)
- Maintained **85-95%** CPU & 92-95% memory pressure
- No memory leaks after 24-hour continuous operation

Robustness & Fault Tolerance

- Graceful handling of permission denied & network errors
- No race conditions with concurrent operations
- Firm permission boundaries, no privilege escalation
- Automatic recovery from unexpected process termination

Performance Benchmarks

UI Responsiveness 95%

Typical Use Cases

ProcSentinel addresses various Linux system management scenarios through its comprehensive monitoring and control capabilities

System Administration

Managing system resources and processes across multiple users

- ✓ Process priority adjustment
- ✓ Resource allocation monitoring
- ✓ User activity tracking

Container Management

Monitoring and managing Docker, Kubernetes, and other containerized environments

- ✓ Resource usage per container
- ✓ Namespace isolation analysis
- ✓ Cgroup-based monitoring

Performance Tuning

Optimizing system performance through detailed process analysis

- ✓ CPU/Memory bottleneck identification
- ✓ Historical performance graphs
- ✓ Resource usage alerts

Debugging

Troubleshooting system issues and process behavior problems

- ✓ Process exit logging
- ✓ Resource consumption analysis
- ✓ Process dependency mapping

Security Monitoring

Detecting and investigating suspicious process activities

- ✓ Unexpected process detection
- ✓ Resource abuse monitoring
- ✓ Process activity alerts

Automation

Creating automated responses to system events and conditions

- ✓ Threshold-based alerts
- ✓ Scheduled process management
- ✓ Custom automation rules

Limitations & Future Improvements

Current Limitations

Task Scheduling

Basic cron expressions only (no month, day-of-week, ranges)

Security

No authentication, TLS, RBAC, or audit logging in remote mode

Data Persistence

Exit logs, graph history stored only in memory

Performance

Limited responsiveness for >10k processes

Future Improvements

Remote Control

Full remote process control with **TLS encryption** and authentication

Fault Tolerance

Watchdog processes, auto-restart, circuit breakers

Comprehensive Alerting

I/O, network, custom metrics with **multiple channels** (email, Slack)

Persistence & Analytics

Historical logs, SQLite/PostgreSQL storage, ML-driven outlier detection

Report Summary

Key Achievements

- **Dual Interface Design** — TUI for terminal users, GUI for visual monitoring
- **Advanced Filtering** — Boolean expressions, regex, field comparisons
- **Container Support** — Docker, Kubernetes, namespace management
- **Automation Layer** — Profiles, alerts, scheduling, rules

Technical Innovation

- **Rust Implementation** — Memory safety, zero-cost abstractions
- **Concurrency Management** — Arc> for shared state
- **Modular Architecture** — Clear separation of concerns

Value Proposition

- **Bridges Gap** — Terminal users and GUI enthusiasts
- **Advanced Features** — Beyond basic process listing
- **Extensible** — Plugin architecture for future growth

ProcSentinel represents a significant advancement in Linux process management, combining traditional approaches with modern features for enhanced system control and performance optimization

Thank You

Thank you for your attention! We hope you found our **ProcSentinel** Linux Process Manager presentation informative.

Questions?