INSTITUT SUPERIEUR INFORMATIQUE ISI
المعهد العالي للإعلامية

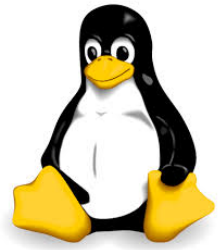
Technical Report

Multi-task Scheduler Internals

The Sudoers Group4 - 1ING3

Presented by:

Firas Kahlaoui - Scrum Master
Oussema Abdelmoumen - Developer
Hamza Bargoug - Developer
Seifeddine Ben Fredj - Developer
Malek Nasri - Developer



December 2025

Table of Contents

1 Introduction

This technical report details the internal architecture, algorithms, and implementation strategies of the Multi-task Scheduler project. The system is a user-space simulator written in C that mimics the behavior of an Operating System's short-term scheduler.

2 System Architecture

The project follows a modular architecture to ensure separation of concerns between the core logic, the UI, and the data management.

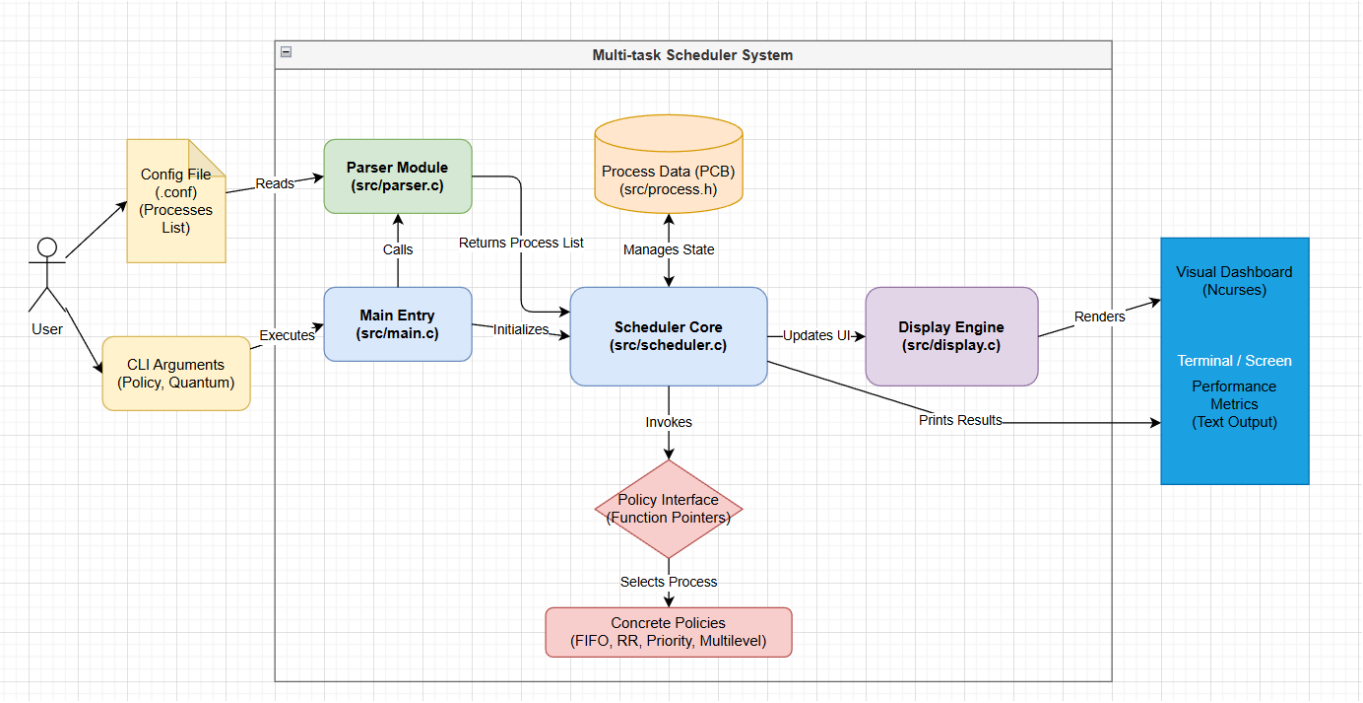


Figure 1: High-Level System Architecture

2.1 Core Modules

- **Scheduler Core (`src/scheduler.c`)**: The central orchestrator. It maintains the global clock, manages the state of the simulation, and invokes the specific scheduling policy.
- **Parser (`src/parser.c`)**: Responsible for reading configuration files and populating the `process_t` array.
- **Display Engine (`src/display.c`)**: Handles all `ncurses` drawing operations. It decouples the logic from the presentation layer.
- **Policies (`policies/*.c`)**: Each scheduling algorithm is isolated in its own file, adhering to a common interface.

2.2 Data Structures: The PCB

The Process Control Block (PCB) is represented by the `process_t` structure in `src/process.h`.

```c
typedef struct {
    char name[32];
    int arrival_time;
    int burst_time;
    int priority;
    int remaining_time;
    int waiting_time;
    int turnaround_time;
    int completion_time;
    int start_time;
    int queue_level;
    process_state_t state;
} process_t;

```

Figure 2: Process Control Block Definition

Key fields include:

- `remaining_time`: Critical for preemptive algorithms to track progress.
- `state`: Enum tracking `READY`, `RUNNING`, `TERMINATED`.
- `priority`: Used by the Priority and Multilevel policies.

3 Detailed Code Documentation

3.1 Scheduler Core (`scheduler.c`)

Overview: `scheduler.c` is the core engine of the application. It manages the lifecycle of the simulation, handles the selection of the active scheduling policy, and coordinates between the logic layer and the visualization layer. It also calculates and displays the final performance metrics.

Key Functionality:

- **Policy Management (Strategy Pattern)**: The scheduler uses a function pointer `current_policy` to switch between different algorithms dynamically.
  - `init_scheduler(const char *policy_name)`: Sets `current_policy` to point to the requested function (e.g., `fifo_schedule`, `multilevel_schedule`).

- **Visualization & Logging**:
  - **Visual Mode**: If enabled, `update_visualization` is called every tick to refresh the Ncurses UI. It introduces a small delay (`usleep`) to make the animation visible.
  - **Text Logging**: If visual mode is off, `log_event` prints execution intervals (e.g., "Executing P1 from time 0 to 3") to the console.
- **Metrics Calculation**: After the simulation completes, `display_results` calculates:
  - **Turnaround Time**: Completion Time - Arrival Time.
  - **Waiting Time**: Turnaround Time - Burst Time.
  - **Averages**: Computes the mean waiting and turnaround times for the batch.
  - **Execution Order**: Sorts processes by completion time to print the final sequence (e.g., P1 → P2 → P3).

3.2 Display Engine (`display.c`)

Overview: `display.c` handles the **Graphical User Interface (GUI)** using the `ncurses` library. It transforms the terminal into a dynamic dashboard showing real-time simulation status.

Key Components:

- **Dashboard Layout**: The screen is divided into several sections:
  - **Header**: Project title and current time.
  - **Process List**: A table showing the state (READY, RUNNING, TERMINATED) and progress of each process.
  - **CPU State**: Shows which process is currently on the CPU.
  - **Gantt Chart**: A scrolling horizontal bar chart at the bottom visualizing the execution history.
- **Visualization Logic**:
  - `init_graphical_display()`: Sets up ncurses mode, hides the cursor, and enables colors.
  - `update_graphical_display(...)`: Clears the screen, draws borders, iterates through processes to draw **Progress Bars**, updates the Gantt chart string, and refreshes the screen.

3.3 Parser (`parser.c`)

Overview: `parser.c` is responsible for reading the input configuration file and converting it into the internal `process_t` structures.

Logic:

1. **File Opening**: Opens the specified file path.
2. **Line Reading**: Reads the file line by line.
3. **Parsing**: Uses `sscanf` or string tokenization to extract the 4 fields (Name, Arrival, Burst, Priority).
4. **Validation**: Skips comments (lines starting with #) and empty lines.
5. **Allocation**: Dynamically allocates an array of `process_t` to store the data.

4 Algorithm Implementation Details

4.1 FIFO (First-In, First-Out)

Overview: `fifo.c` implements the **First-In-First-Out (FIFO)** scheduling algorithm, also known as First-Come-First-Served (FCFS). It is the simplest non-preemptive scheduling policy.

Algorithm Logic:

1. **Sorting**: The processes are conceptually sorted by their `arrival_time`.
2. **Non-Preemptive**: Once a process starts execution, it runs until its `burst_time` is fully consumed. No other process can interrupt it.
3. **Idle Time**: If no process has arrived by the current time, the CPU remains idle until the next arrival.

Data Structure & Design Choice: We use a simple **array of `process_t`** structures. Since FIFO requires processing in arrival order, we sort the array once at the beginning based on `arrival_time`. This avoids the overhead of maintaining a dynamic linked list queue, as the number of processes is fixed after parsing.

4.2 Round-Robin

Overview: `round_robin.c` implements the **Round-Robin (RR)** scheduling algorithm. This is a preemptive policy designed for time-sharing systems, ensuring that no single process monopolizes the CPU.

Algorithm Logic:

1. **Time Quantum**: Each process is given a fixed slice of time (`quantum`) to execute.

2. **Circular Queue**: Processes are treated as a circular queue. The scheduler picks the next available process that has arrived.
3. **Preemption**:
   - If a process finishes within its quantum, it terminates.
   - If it exceeds the quantum, it is paused (preempted), and the scheduler moves to the next process in the queue.
4. **Cycle**: The loop continues until all processes are `TERMINATED`.

Data Structure & Design Choice: The processes are stored in an **array**, which we iterate over cyclically using an index `i` and modulo arithmetic. This effectively simulates a **Circular Queue**. We chose this over a linked list to improve cache locality and simplify the logic for accessing process fields during visualization updates.

```
while(completed < n) {
        bool progress = false;
        for(int i=0; i<n; i++) {
            if(processes[i].state != TERMINATED && processes[i].arrival_time <= current_time) {
                progress = true;
                if(processes[i].start_time == -1) processes[i].start_time = current_time;

                processes[i].state = RUNNING;
                int time_slice = (processes[i].remaining_time > quantum) ? quantum : processes[i].remaining_time;

                log_event(current_time, processes[i].name, time_slice);

                for(int t=0; t<time_slice; t++) {
                    update_visualization(processes, n, current_time, i);
                    processes[i].remaining_time--;
                    current_time++;
                }

                if(processes[i].remaining_time == 0) {
                    processes[i].waiting_time = current_time - processes[i].burst_time - processes[i].arrival_time;
                    processes[i].completion_time = current_time;
                    processes[i].turnaround_time = current_time - processes[i].arrival_time;
                    processes[i].state = TERMINATED;
                    completed++;
                } else {
                    processes[i].state = READY;
                }
                update_visualization(processes, n, current_time, -1);
            }
        }
        if(!progress) {
            update_visualization(processes, n, current_time, -1);
            current_time++;
        }
    }
}
```

Figure 3: Round-Robin Scheduling Loop

4.3 Preemptive Priority

Overview: `priority.c` implements **Preemptive Priority Scheduling**. It ensures that critical tasks are executed as soon as possible.

Algorithm Logic:

1. **Selection**: At every time tick, the scheduler scans all available processes (`arrival <= current_time`).

2. **Comparison**: It selects the process with the **highest priority value**.
3. **Preemption**: Because the check happens at every tick, if a new process arrives with a higher priority than the currently running one, the current process is immediately paused (context switch).
4. **Starvation Risk**: Unlike Multilevel, this basic implementation does *not* include aging, so low-priority processes may starve if high-priority ones keep arriving.

Data Structure & Design Choice: We use an **unsorted array** and perform a linear scan ($O(N)$) at every time tick to find the process with the highest priority. While a **Max-Heap** would offer $O(1)$ access to the maximum, the frequent updates (preemption) and small dataset size make the array approach simpler and sufficiently performant for this simulation.

4.4 Multilevel Queue with Aging

Overview: `multilevel.c` implements a **Preemptive Multilevel Queue Scheduling** algorithm with **Aging**. This is the most complex policy in the project, designed to balance system responsiveness with fairness.

Algorithm Logic:

- **Priority Queues**: The system conceptually maintains multiple priority levels. At any time t, the scheduler selects the process with the **highest priority** that has arrived.
- **Preemption**: The scheduler checks for the highest priority process **at every time tick**.
- **Aging Mechanism (Starvation Prevention)**:
  - **Tracking**: An `aging_counters` array tracks how long each process has been waiting in the READY state.
  - **Rule**: For every **10 ticks** a process waits, its priority is incremented by **+1**.
  - **Reset**: When a process starts running, its aging counter is reset to 0.

Data Structure & Design Choice: We utilize the main `process_t` array alongside a parallel **integer array** `aging_counters`. This separation of concerns allows us to track the "wait time" for aging purposes without cluttering the main PCB structure with simulation-specific temporary data. The "queues" are conceptual; we scan the array to find the highest priority, effectively simulating multiple priority levels.

```c
for (int i = 0; i < n; i++) {
            if (processes[i].state != TERMINATED && processes[i].arrival_time <= current_time) {
                if (i != idx) {
                    aging_counters[i]++;
                    if (aging_counters[i] >= 10) {
                        processes[i].priority++;
                        aging_counters[i] = 0;

                    }
                } else {

                    aging_counters[i] = 0;
                }
            }
        }
```

Figure 4: Aging Mechanism Implementation

5 Compilation and Build Process

The project uses a `Makefile` to automate the compilation process.

5.1 Build Steps

1. **Variables**:
   - `CC = gcc`: Uses the GNU C Compiler.
   - `CFLAGS = -Wall -Wextra -std=c11`: Enables warnings and enforces C11 standard.
   - `LDFLAGS = -lncurses`: Links against the ncurses library.
2. **Targets**:
   - `all`: The default target. It depends on the executable.
   - `$(EXEC)`: Links object files from `obj/` to create the binary in `bin/`.
   - `$(OBJ_DIR)/%.o`: Compiles source files (`.c`) into object files (`.o`).
3. **Commands**:
   - `make`: Compiles the entire project.
   - `make clean`: Removes `obj/` and `bin/` directories.
   - `make install`: Copies the executable to `~/.local/bin`.

6 Licensing

The project is released under the **MIT License**.

6.1 Justification and Validation

The MIT License was chosen for its simplicity and permissiveness. It allows anyone to use, modify, and distribute the code for private or commercial purposes, provided the original copyright notice is preserved.

The license choice was validated by scanning the codebase with `scancode-toolkit`. This ensures that no conflicting proprietary code or incompatible licenses exist within the project.

What is Scancode Toolkit? `scancode-toolkit` is a reliable open-source tool used to scan code for license text, copyright notices, and package manifests. It helps developers ensure license compliance by identifying the origin and licensing of every file in a codebase.

7 Project Management (Scrum Sprints)

The development was conducted in 4 Sprints, tracked via Trello.

7.1 Sprint 1: Infrastructure & Initialization

Task/Story	Description	Label	Due Date	Member
User Story: Project Structure	As a developer, I want a clear project structure to organize source code and documentation.	User Story	2025-11-20	Firas Kahlaoui
User Story: Build System	As a developer, I want an automated build system (Makefile) to compile the project easily.	User Story	2025-11-20	Firas Kahlaoui
User Story: Config Definition	As a user, I want to define processes in a configuration file so I can test different scenarios.	User Story	2025-11-20	Firas Kahlaoui
1.1 Project Setup	Create directory structure (src, policies, bin, obj) and setup Git repository. Related Files: .gitignore	High	2025-11-20	Firas Kahlaoui
1.2 Build System	Create a Makefile to compile source and policy files automatically. Related Files: Makefile	High	2025-11-20	Hamza Bargoug

Task/Story	Description	Label	Due Date	Member
1.3 Data Structures	Define the process_t struct (PCB) with fields for arrival, burst, priority, and state. Related Files: src/process.h	High	2025-11-20	Hamza Bargoug
1.4 Config Parser	Implement a parser to read .conf files and populate the process array. Related Files: src/parser.c	High	2025-11-20	Malek Nasri
1.5 Basic Scheduler	Create the scheduler.c skeleton and a basic main.c to test loading processes. Related Files: src/scheduler.c, src/main.c	High	2025-11-20	Malek Nasri

7.2 Sprint 2: Core Scheduling Algorithms

Task/Story	Description	Label	Due Date	Member
User Story: FIFO Simulation	As a user, I want to run a FIFO simulation to observe processes executing in arrival order.	User Story	2025-11-26	Firas Kahlaoui
User Story: Round-Robin Simulation	As a user, I want to run a Round-Robin simulation to observe time-sharing execution.	User Story	2025-11-26	Firas Kahlaoui
User Story: Performance Metrics	As a user, I want to see performance metrics (waiting time, turnaround time) to analyze algorithm efficiency.	User Story	2025-11-26	Firas Kahlaoui
2.1 FIFO Policy	Implement First-In-First-Out logic. Sort by arrival time and run to completion. Related Files: policies/fifo.c	High	2025-11-26	Hamza Bargoug
2.2 Round-Robin	Implement Round-Robin logic with time quantum handling and circular queue simulation. Related Files: policies/round_robin.c	High	2025-11-26	Oussema Abdelmoumen
2.3 Metrics Calculation	Implement calculation for Waiting Time and Turnaround Time. Related Files: src/scheduler.c	Medium	2025-11-26	Malek Nasri

Task/Story	Description	Label	Due Date	Member
2.4 Context Switch Logging	Implement log_event to print execution intervals to the console (Text Mode). Related Files: src/scheduler.c	Medium	2025-11-26	Seifeddine Ben Fredj
2.5 Unit Tests	Create configuration files to verify FIFO and RR behavior. Related Files: config_examples/test1.conf, config_examples/test2.conf	Medium	2025-11-26	Firas Kahlaoui

7.3 Sprint 3: Advanced Features & Visualization

Task/Story	Description	Label	Due Date	Member
User Story: Priority Simulation	As a user, I want to run a Priority simulation so that higher priority processes execute first.	User Story	2025-12-02	Firas Kahlaoui
User Story: Multilevel Queue	As a user, I want to run a Multilevel Queue simulation to observe aging and dynamic priorities.	User Story	2025-12-02	Firas Kahlaoui
User Story: Visual Dashboard	As a user, I want a visual dashboard (ncurses) to watch the scheduling process in real-time.	User Story	2025-12-02	Firas Kahlaoui
3.1 Priority Policy	Implement Preemptive Priority scheduling (highest priority runs, preemption on arrival). Related Files: policies/priority.c	High	2025-12-02	Seifeddine Ben Fredj
3.2 Multilevel Queue	Implement Multilevel Queue with Aging (priority increases every 10 ticks of waiting). Related Files: policies/multilevel.c	High	2025-12-02	Firas Kahlaoui
3.3 Ncurses Setup	Initialize ncurses library, colors, and window layout. Related Files: src/display.c	Medium	2025-12-02	Seifeddine Ben fredj
3.4 Real-time UI	Implement update_visualization to show Process Table, Progress Bars, and Gantt Chart. Related Files: src/display.c	Medium	2025-12-02	Malek Nasri

Task/Story	Description	Label	Due Date	Member
3.5 Interactive Menu	Update main.c to allow user selection of policies and visual mode at runtime. Related Files: src/main.c	Low	2025-12-02	Firas Kahlaoui

7.4 Sprint 4: Documentation, Polish & Finalization

Task/Story	Description	Label	Due Date	Member
User Story: Help Menu	As a user, I want an interactive help menu to learn available commands.	User Story	2025-12-07	Firas Kahlaoui
User Story: Documentation	As a user, I want comprehensive documentation to understand how to install and use the simulator.	User Story	2025-12-07	Firas Kahlaoui
User Story: Licensing	As a stakeholder, I want to ensure the project is properly licensed and free of copyright issues.	User Story	2025-12-07	Firas Kahlaoui
4.1 Code Cleanup	Refactor code, ensure consistent indentation, and remove debug prints. Related Files: src/*.c, policies/*.c	Low	2025-12-07	Malek Nasri
4.2 Help System	Implement the help and ? commands in the main menu. Related Files: src/main.c	Low	2025-12-07	Hamza Bargoug
4.3 Technical Report	Write the final project report detailing architecture and algorithms. Related Files: doc/rapport.md	Medium	2025-12-07	Seifeddine Ben Fredj
4.4 User Guide	Create an installation and usage guide. Related Files: doc/INSTALL.en.md	Medium	2025-12-07	Hamza Bargoug
4.5 Licensing	Verify license compliance (using scancode-toolkit) and add LICENSE file. Related Files: doc/LICENSE, scan_results.json	High	2025-12-07	Firas Kahlaoui