

Custom Linux Task Manager

Project Report
Operating Systems Lecture Spring Semester 2025

University of Basel
Faculty of Science
Department of Mathematics and Computer Science

Fabian Götschi
Jiri Käser
Manuel Buser
Valerio Job

June 10th, 2025



Contents

1	Introduction	1
1.1	Problem	1
1.2	Background	1
1.3	Methodology	3
1.3.1	System Architecture	3
1.3.2	C-core Architecture	3
1.3.3	General Statistics	3
1.3.4	Process Data Collection	4
1.3.5	Sleeper Process Detection	4
1.3.6	Background Daemon and IPC	4
1.3.7	Web API and Front End	6
1.3.8	Configuration and Setup	7
2	Results	8
2.1	Capabilities	8
2.1.1	Process Monitoring see Figure 2	8
2.1.2	Sleeper Detection see Figure 2	8
2.1.3	Process Control see Figure 2	8
2.1.4	API see Figures 1a and 1b	8
2.1.5	System Statistics Dashboard see Figures 3, 4a, 4b, 5a, 5b	9
2.2	Discussion	10
2.2.1	Limitations	10
2.2.2	Achievements	10
3	Conclusion	12
3.1	Summary	12
3.2	Lessons Learned	12
	References	16
	Appendix A: Declaration of Independent Authorship	17
	Appendix B: List of Aids Used for Completing this Project and declaration of AI usage	18
	Appendix C: *	19

1 Introduction

1.1 Problem

Process monitoring and management are important for any operating system [1].

System administrators, developers, and in general users often rely on tools that can provide real-time information regarding processes and system resources. Ubuntu as well as other Linux operating systems per default include `top`, `htop` and the `System Monitor` as system monitoring and process management tools. In `top` one can for example kill a process by typing `k` followed by the PID[1, 2].

This project is set to rebuild the basic system monitoring and management functionalities already present in Ubuntu to really understand how the `/proc` system works and extend on them with sleeper detection and resource management.

The goal is to build the back-end from scratch in C, using system calls for process management and file reading. To do that, we are going to query the `/proc` file-system to receive the raw statistical data on processes and general statistics. Finally to display all the relevant information, Java Spring-Boot will be used for our front-end. This setup was chosen to extend the scope of the project, making it firstly more difficult to complete but more importantly a useful learning experience and something that could actually be used in practice [3, 4, 5].

Working on this project helps us get comfortable working with operating systems. It teaches us how system calls work, how processes communicate, how `/proc` is used to store data, how to understand and parse files inside the `/proc` folder. It also gives us a chance to see and use what we've learned in theory in practice.

1.2 Background

Process monitoring is a fundamental feature of any operating system. It allows users and administrators to observe the state of system processes, their resource usage, and interact with them for maintenance, debugging, or performance optimization. On Linux systems for example `top`, `htop`, and `ps` offer command-line interfaces for monitoring and management of processes. Each process, drive, CPU core, memory, ... is all represented in the `/proc` virtual file system. `/proc` is contained in the Linux kernel. It provides an interface to kernel data structures, exposed as files. Since we will have to access a lot of data in a short amount of time, we were worried, that the i/o times will slow our Linux task manager down. But we were worried without reason since the `/proc` Pseudo File System only exists in memory, which reduces the i/o time significantly. Also note that the files in `/proc` are generated dynamically. This means contents of `/proc` are generated on-the-fly by the kernel when accessed[1, 6].

Due to `/proc` holding all of this data and most of it being available without requiring privileged rights, the `/proc` file-system has become one of the most important components of any monitoring tool that runs on a Linux operating system. Files like `/proc/[pid]/stat`, `/proc/[pid]/status`, and `/proc/meminfo` provide detailed statistics e.g. process state, memory consumption, CPU time, total memory available, time a CPU spent in idle, time CPU spent on system processes, time CPU spent on user processes, and so on. Having all this data nicely organized and fast accessible makes it a very strong option to use when setting up a process monitor[7].

For example one could now use the `/proc` file system to calculate the CPU usage in percent.[8]

First one calculates the deltas in CPU time retrieved from `/proc/stat` by taking two measurements, one at time t_0 and the next at t_1 and subtracting the values at t_1 from the values at t_0 . The deltas of interest are e.g. time CPU spent on user and system processes $T_{user+system}$ and the total time the CPU has been running T_{total} . To then get the CPU usage is % in the specific time interval, one uses the simple equation:

$$\frac{T_{user+system}}{T_{total}} * 100 = CPU_{usage \ percent}$$

Next we need to introduce POSIX. POSIX (Portable Operating System Interface) is a standard, which defines a collection of APIs for Unix-like systems. These APIs include system calls for process management (`fork()`, `exec()`, `kill()`), file operations (`open()`, `read()`, `close()`), and resource control (`setrlimit()`, `getrusage()`) which will be useful when trying to implement this functionality in C. POSIX adds a nice abstraction layer allowing developers to not worry about how exactly a file will be opened or a process will be forked but simply call the function, and it works across all operating systems that implement POSIX. [9].

In addition to data retrieval, controlling processes is another key aspect of task managers. This is usually done via system calls. A good example is the `kill()` system call which takes a PID and kills the process.[8].

For this project, inter-process communication (IPC), specifically sockets, were used to send data from the C back-end to the Java front-end. We chose a socket since it allows for multiple connections, and we expected to be needing that. Since the focus of our project was not to build a http server but to get a running process monitor, we decided to use Microhttpd. That way, we would be able to focus more on building the process statistics queries and front-end. Microhttpd is an open source C library that allows to create a simple http server, which is perfect since we needed the server to act as a simple API between C and Java[10, 11].

To be able to build a process monitoring and management device, a solid understanding of all the themes mentioned above is required.

1.3 Methodology

In the beginning of the project we started by outlining our desired system architecture and the approximate data structure. Then we separated it into the modules described below, distributed the work and made a development plan. Note that all files relevant to this project are available on GitHub under

<https://github.com/S1ntax3rr0r/LinuxTaskManager> [12]

We implemented our Linux Task Manager using a layered and modular architecture. It separates parsing data into the correct format and calculates metrics, provides data via a WebSocket, retrieves the data in another technology stack to make it available in a front-end, and additionally enables bidirectional communication back to the c-core in order to send POST requests, such as kill signals, renice commands, and others.

The technologies used are mentioned more in detail in the following subsections.

1.3.1 System Architecture

The main components of the task manager are:

- A **C core module** that handles, manages, and transforms the data from the `/proc`[7] file system.
- A **background daemon** that collects data using the C core module and serves it to external clients our http server based on `microhttpd`[11].
- A **Java REST API** built with Spring Boot that exposes process data over HTTP.
- A **web front end** that displays the information in a browser using Thymeleaf[13] templates.

This split design allows for a parallel distribution of the workload and eases upgrade ability.

1.3.2 C-core Architecture

In the c-core the data collection happens in the `core.interface`. Firstly it uses the help of the `general_stat_query` for the general information, which needs the `memory_stats`. Secondly, it uses `proc_stat` as well as `memory_stats` and `cpu_usage` for the processes. Further for the sleeper detection of the processes it uses the `sleeper_detection` file.

1.3.3 General Statistics

The general statistics in the c file `general_stat_query` we collected are:

- CPU Info with `/proc/cpuinfo`
- Memory info with `/proc/meminfo`
- Disk statistics with `/proc/diskstats`

- Network with `/proc/net/dev`
- Nvidia GPU with

```
nvidia-smi --query-gpu=memory.used, utilization.gpu --format=csv,noheader,nounits"
```

We read the data with the previous commands and store it using a structure called **general_stat.h**. In the structure are the most important parsed statistics as well as calculations for simpler interpretation. The statistics gets collected in c-daemon through the **core_interface**.

1.3.4 Process Data Collection

For processes, we read the data of each dictionary with only numbers as the name, because these are all processes, with `/proc/[pid]/stat`. The information gets parsed, and after additional calculating, we reduce the info of each structure **proc_stat.h** by transposing the information to a simpler structure **trimmed_info.h**. The trimmed info will be collected in the c-daemon through the **core_interface**.

1.3.5 Sleeper Process Detection

Our custom sleeper process detection uses the `proc_stat` to indicate if a process is a potential "Sleeper". A Process gets marked as sleeper if:

- It has no CPU usage for a 30 second interval,
- And its memory usage remains above our defined threshold of 1%.

This helps identify inefficient processes that use resources unnecessarily.

1.3.6 Background Daemon and IPC

The background daemon (c-daemon) is the process that is constantly running in the background waiting for the frontend to request data. On request from the front-end it then uses our C core to poll process data. The main API calls it can handle are:

- GET on **`http://localhost:9000/api/processes`** which returns a list of all processes with all specific data.
- GET on **`http://localhost:9000/api/stats/all`** which returns all the general statistic information relevant for the front-end. More specifically it returns data on the CPU, its cores, network statistics, disk/ssd statistics, load averages and memory.
- more specific get calls like `*/api/stats/[network—disk—general]` or `GET */api/cpu_mem` if only specific parts are required.
- POST `/api/processes/pid/signal` will kill process with specified PID.
- POST `/api/processes/pid/renice` is used to renice the process with specified PID.

- POST /api/processes/pid/limit/[cpu—ram] will limit the CPU or RAM usage of the specified process.

When using POST, some data needs to be added in the body of the http request. For example renice needs to also be given a new nice value. If one wants to use renice to give processes a higher priority or limit ram/cpu for any process or kill system processes, then the http server needs to be started with privileged rights. I.e. **sudo ./http_server**. The c-daemon sends the data in JSON format. To easily parse c data we used the cJson library.[11, 14, 15]

```

{
  "cpu": {
    "name": "cpu",
    "nice": 295,
    "system": 73937,
    "..."
  },
  "proc_stats": {
    "intr": 38232249,
    "ctxt": 78013721,
    "btime": 1749570147,
    "..."
  },
  "cores": (8) [ (-), (-), (-), (-), (-), (-), (-), (-) ],
  "net_stats": {
    "total_download_MB": 99.541267395019531,
    "total_upload_MB": 13.649529457092285,
    "timestamp_ms": 1749583458464
  },
  "disc_stats": {
    "name": "nvme0n1p2",
    "read_MB": 2216.8291015625,
    "write_MB": 2180.7109375,
    "total_read_MB": 2222.21875,
    "total_write_MB": 2180.7119140625
  },
  "load_stats": {
    "loadavg1": 1.12,
    "loadavg5": 0.62,
    "loadavg15": 0.44,
    "tasks_total": 1602,
    "tasks_running": 1,
    "cpu_util_percent": 4.0175614356994629
  },
  "memory": {
    "total_MB": 31824.28125,
    "free_MB": 22795.25390625,
    "available_MB": 26091.62109375,
    "buffers_MB": 205.4453125,
    "cached_MB": 4812.03125,
    "swap_total_MB": 2047.99609375,
    "swap_free_MB": 2047.99609375
  }
}

```

(a) Shows an example of how GET api/stats/all can look like

```

[
  {
    "pid": 1,
    "cmd": "/sbin/init splash",
    "comm": "systemd",
    "state": "S",
    "cpuPercent": 0,
    "ramPercent": 0.045231284681409731,
    "nice": 0,
    "username": "root",
    "prio": 20,
    "virt": 24272,
    "res": 14740,
    "shared": 9236,
    "upTime": 12977.6,
    "is_sleeper": 0
  },
  {
    "pid": 2,
    "cmd": "",
    "comm": "kthreadd",
    "state": "S",
    "cpuPercent": 0,
    "ramPercent": 0,
    "nice": 0,
    "username": "root",
    "prio": 20,
    "virt": 0,
    "res": 0,
    "shared": 0,
    "upTime": 12977.6,
    "is_sleeper": 0
  }
]

```

(b) Shows an example of how GET api/processes can look like

Figure 1: Get api/stats/all vs GET api/processes

The c-daemon is structured rather simple. The source code contains one make file for the c-daemon specifically and /src and /include folders. The /include folder contains all .h files declaring the structures, functions and types used. Our /src folder contains all our .c files for the c-daemon. **main.c** simply starts the http server defined in **server.c**. The server then listens for incoming traffic and processes the http requests written above by running them through the **routes.c** file. If the request matches a given pattern, **routes.c** calls the functions assigned to each http request. The function then calls the c-daemon to pull data and sends it back to the process that sent the request, or it calls the c-core to do an action like kill a process.

1.3.7 Web API and Front End

Our frontend is a full-stack web application implemented using Java Spring Boot[4], Thymeleaf[13], JavaScript, Chart.js, Bootstrap, and HTML/CSS. It combines these several key technologies to deliver a responsive, interactive web page.

Java Spring Boot, which is a popular Framework often used in Web development, serves as the basis and handles HTTP routing, REST API definitions, dependency injections and application configurations. It automatically runs an embedded Tomcat Server and provides a clean structure to implement the model view controller architecture (MVC). The classes in the **model package** are representing the underlying data structures, each class being one entity. The Lombok library from Java together with the `@data` annotation gives us automatic getters and setters for each entity and the “JsonProperty” helps us parsing the JSON into the model class to use this data inside the java-service package. DTO is just a naming convention and stands for Data Transfer Object.

The classes in the **controller package** are the HTTP endpoints for the backend services but also for rendering the frontend html. They are using helper methods that have been implemented in the **service package**. The overall flow is that once a user goes on the defined localhost port in a web browser, the endpoint to render the HTML (inclusive CSS styling) gets activated. The HTML itself then activates the JavaScript code to fetch data into the frontend which then triggers the backend endpoints from the controller classes. The JavaScript can also handle incoming POST requests from the frontend which are being send back over a PostMapping in the Controller Class to the c-daemon respectively the c-core backend to handle user input and not only fetch data to the frontend. The class in the **config package** defines a configuration class that gives a RestTemplate bean, which is being used by Spring Boot to manage dependencies.

Thymeleaf is a template engine for Java web apps and helps to connect frontend fields directly with backend data. The **Chart.js** is a JavaScript Library that simplifies creating responsive charts like the one that we used to showcase the history stats. **Bootstrap** is a front-end toolkit offering pre-designed components and styling to create a modern looking front-end without writing many lines of code.

1.3.8 Configuration and Setup

The system should be launched in the following order:

1. Inside the root folder, the following command should be run to initiate all make files:

```
make all
```

2. After that one should navigate to the c-daemon folder and start the server with the following command:

```
[sudo] ./http\_server
```

Sudo should only be added if more powerful resource management is required.

3. The last step is to launch the front-end. To do this it is important to first navigate to the package java-service and make sure that one has Maven installed on the machine. Then one should run the following command in the terminal:[16]

```
mvn spring-boot:run -Dspring-boot.run.profiles=api
```

The `mvn` command invokes the Maven framework which uses the projects `pom.xml` to figure out what plugins and dependencies we are using. The `spring-boot:run` section compiles the code, sets up a classpath including all dependencies and launches the application. The `-Dspring-boot.run.profiles=api` tells it which profile to run which is configured in the `application.api.yml` file.

Dependencies include a standard C toolchain, cJson[14], microhttpd[11] and a Java runtime with Maven[16] support.

2 Results

2.1 Capabilities

Our Linux Task Manager provides process monitoring and some process management features.

2.1.1 Process Monitoring see Figure 2

- The web interface lists all currently active processes. Each process shows its PID, User, Prio[rity], Nice, VIRT[ual memory] (KiB), RES[ident memory] (KiB), SHR[ared memory] (KiB), COMMAND[used to start u], Up Time (s), Name[process name], State, % CPU, % RAM
- Sorting Options: Processes can be sorted by CPU, RAM usage.

2.1.2 Sleeper Detection see Figure 2

- Detects processes with low CPU but high memory usage over a time interval.
- A checkbox allows filtering for only sleeper processes which makes it easy to identify background processes that can be killed for more memory.

2.1.3 Process Control see Figure 2

- In the UI one can click the **Kill** button next to each process to kill it. To be able to kill a system process the c-daemon needs to be started with sudo.
- In the UI one can enter a PID and a nice value next to the **Renice** button. Now when clicking the **Renice** button the process that belongs to the PID entered will get assigned the new nice value. If one wants to renice a system process or give a process a lower nice value than it currently has, the the c-deamon needs to be started with sudo.
- In the UI one can set CPU and RAM usage caps using by entering the PID and desired cap next to the according limit button. For RAM one can limit a processes memory to a desired limit. For CPU one can enter how many more seconds the process is allowed to use the CPU. For RAM one can enter how many MB's the process should be allowed to use at maximum.

2.1.4 API see Figures 1a and 1b

- Exposes system statistics and process data over HTTP using MicroHTTPD and JSON via cJSON.
- Example endpoints:
 - GET /api/processes
 - POST /api/processes/{pid}/signal

2.1.5 System Statistics Dashboard see Figures 3, 4a, 4b, 5a, 5b

This is the second tab of our web UI which shows the most important general system statistics at the top. Down below it contains plots that show the network usage, disk r/w, CPU usage and memory usage over the last 60 seconds in which the monitor was active.

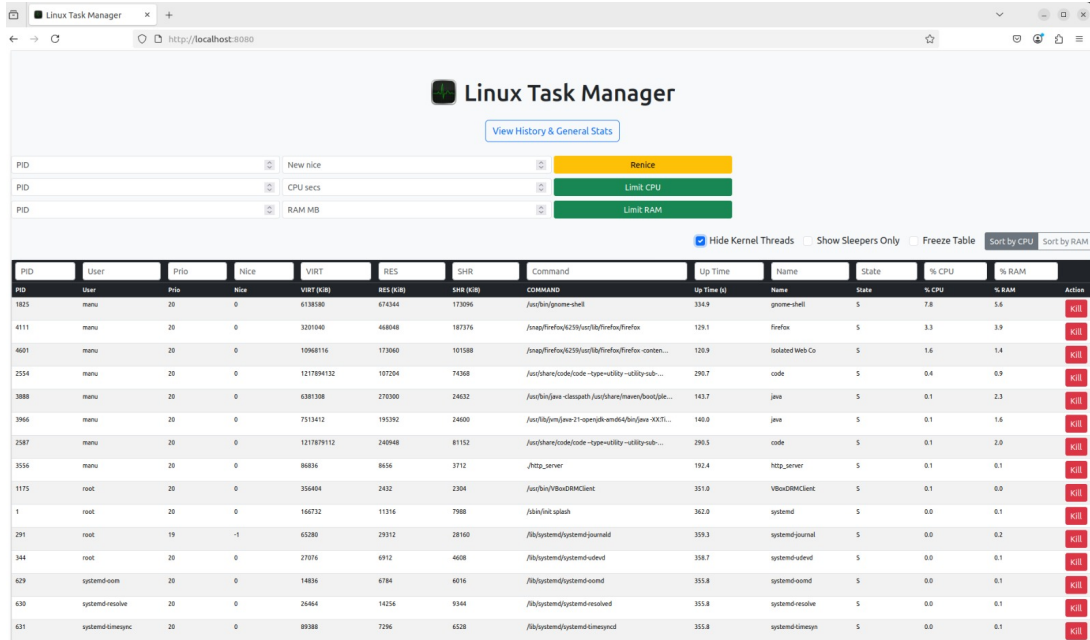


Figure 2: Web-based GUI: Main Page

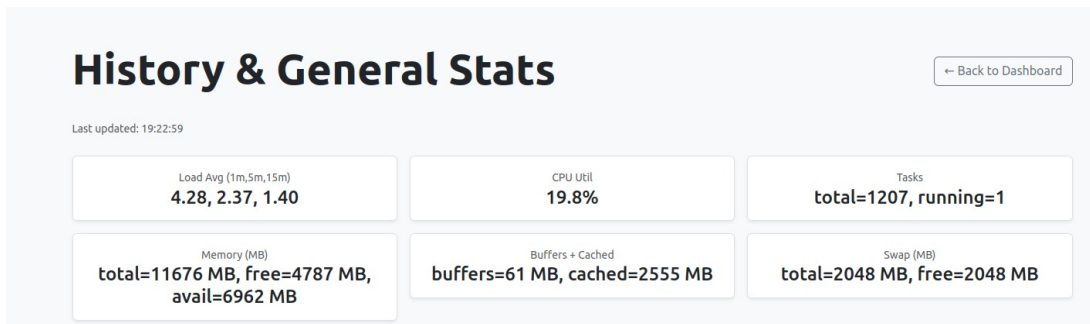


Figure 3: Web-based GUI: General Stats



Figure 4: Network and Disk plots

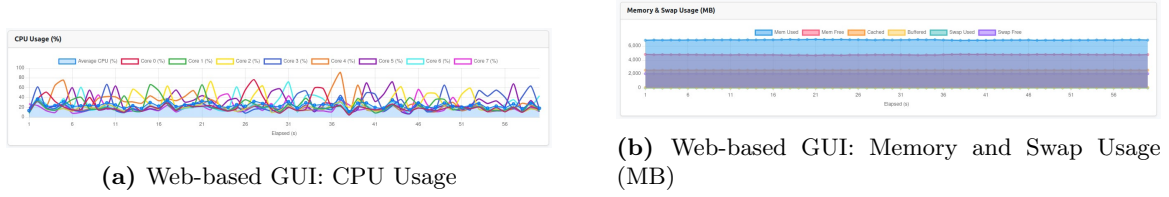


Figure 5: CPU and RAM plots

2.2 Discussion

Our Linux Task Manager contains all core features that were outlined at the beginning of the semester. The initial plan to create a terminal based UI has been swapped out for the UI that is run via Springboot. The process and system data we wanted to implement is all available via the web UI. Some detailed statistics like **wa** that show how much time the CPU spent waiting on I/O operations were explicitly excluded to unclutter the UI. The sleeper detection system that we wanted to build is working fine, but it could use some parameter optimization. Right now it shows all processes that have been sleeping for longer than 30 intervals and use $> 1\%$ memory. Resource management was also implemented. We can limit the RAM available to any process, limit the CPU time a process has left, and change the priorities of processes by changing their nice values. There are many more resource limitations that we could implement but we decided to go with these three due to time constraints.

2.2.1 Limitations

Since we only had limited time and we didn't try to port our tool to Windows. It should however run on any distribution that is able to run Java and has POSIX implemented.

While we tried to test our application to the fullest, we are quite sure that some bugs went undetected since there is a lot of data that needs to be converted in very specific ways. Once found it is an easy fix but we are aware that we have become used to how our UI looks and therefore might not see small errors like conversions from MBytes to Mbit or similar.

Some operations, such as killing system processes or setting resource limits, require elevated permissions (e.g., **sudo**), which may not always be available in user environments.

When running the `c-daemon` using the `sudo` command, it exposes operations that require root rights via HTTP without authentication. Of course, as long as we run it only locally this is not a huge problem but it is still a risk that one needs to be aware of.

We have not found a good way to filter out all disks on a system reliable. Right now we simply filter for `sda` or `nvme` secondary storage devices which works fine on all of our systems but this could potentially be a problem.

GPU support exists in the backend for Nvidia GPU's but this feature never made it into the UI.

2.2.2 Achievements

While there are for sure are a lot of ways in which this project can be further improved there are also a lot of things that are built in very solid. The `/proc/stat` and `/proc/[pid]/stats` file parsers have been written in the first month of the project and since then never needed to be

adjusted. They also allow for dynamic extension, which made it not easy but straight forward to implement the memory, disk and network information.

The front-end turned out to look great compared to what we had first in mind and we even managed to show plots of the last 60 seconds that are dynamically updated and can be customized in the UI by disabling whatever line you don't want to have drawn or enabling it again. Also at least for us as computer science / computational science students the UI seems very intuitive to understand. When we gave it to people not directly related to computer science, they still managed to understand how to navigate the UI quite quickly.

The communication between the front-end and the back-end for sure isn't the prettiest but it does its job reliably and most importantly, we can update the data in the front-end within a quarter of a second, which is four times faster than we need.

3 Conclusion

3.1 Summary

Over the course of the spring semester 2025, we successfully developed a working Linux Task Manager. It allows for process and general resource monitoring and includes a very user friendly interface. The main achievements of this project are:

- The **C-core** that parses the `/proc` file-system and gathers detailed statistics on processes, CPU, memory, disk, network, and GPU usage.
- The **C-daemon** which allows for API calls to retrieve data.
- The **Java Spring Boot web application** which uses the c-daemon to get the data and displays it in the UI and allows for resource management.
- Support for features like **sleeping process detection**, **process killing**, **resources limiting**, and **priority adjustment**.

There are several elements that are certainly not perfect and can or even should be improved in the future:

- The system assumes a Linux environment with access to the `/proc` file-system, and it can thus not be applied to other operating systems without adaptation.
- Since this system already works with API's, it wouldn't be a lot of effort to open up the ports and allow for remote process monitoring. But since there are no security measures in place right now we did not try to do this. And to implement the security features would far exceed the scope of this project.
- The c-core could be cleaned up, since there are a couple of unused functions that got rewritten in the process but never removed again.
- The c-daemon also needs a clean up, since two different structures of the API have been implemented simultaneously and there are still some leftovers from there.
- Finalizing the GPU support
- Adding another dashboard that shows even more detailed information on system statistics.
- Verify that the disk support is working with any kind of storage device

We can conclude that we managed to build a working prototype that achieves our primary goal of allowing easy Linux system monitoring and management. It offers a solid foundation for further refinement and practical application.

3.2 Lessons Learned

Working on this project gave us practical experience with systems programming and teamwork. Applying the knowledge we received from the Operating Systems course to a larger project showed us how useful and fun it can be to tinker with operating systems but also made us

realize how difficult debugging a C program is especially if the error message you receive is simply "core dumped".

Developer improvement:

Working closely with C and the Linux `/proc` file-system gave us a much deeper understanding of how systems interact at a low level. We learned how to:

- Parse real-time process data from `/proc` while applying POSIX-compliant APIs and system calls,
- Calculate resource metrics like CPU usage and memory consumption using system tick and page values,
- Design and implement inter-process communication using `mhttpd`[11],
- Build a RESTful API using Java Spring Boot[4] to bridge system-level data with a web interface,
- Structure a modular, multi-component system that integrates `c-core`, `daemon`, and web components.

Project and Teamwork Skills

Beyond the technical aspects, this project helped us grow as a software development team. We practiced:

- Version control and collaboration using Git and GitHub,
- Modular code design and interface definition across components and languages,
- Coordinated parallel development, where team members worked on distinct layers of the architecture,
- Writing maintainable code with proper documentation and error handling,
- Communicating effectively within the team to align on design decisions and debug cross-component issues.

Individual Reflections

Each team member took on responsibilities that reflected their strengths and allowed them to explore new areas:

- Fabian Götschi and Jiri Käser focused on systems-level programming and mastering the `/proc` structure and maintaining the `c-daemon`.
- Valerio Job and Manuel Buuser mainly focused on developing the front-end as well as the REST API in Java while also initially setting up the `c-daemon` to then give it to the back-end team for maintenance.

These assignments overlapped often. It was not uncommon for the front-end developers to add something small to the `c-core` or `c-daemon` when it was simply quicker to do it by himself than waiting for the backend developers to do it for him. Also the other way around when for example

some data was used incorrectly in the front-end, the back-end developers fixed it themselves if it wasn't something larger.

Overall, the project strengthened both our technical knowledge and our ability to work effectively in a real-world software development scenario.

References

- [1] André Machado, “A Guide to Linux System Monitoring: top, htop, btop, and glances.” [Online]. Available: <https://machaddr.substack.com/p/a-guide-to-linux-system-monitoring>. Last Accessed 4 Jun. 2025.
- [2] Robert Roth, “Gnome Monitor.” [Online]. Available: <https://apps.gnome.org/en/SystemMonitor/>. Last Accessed 10 Jun. 2025.
- [3] Ankush Das, “Better Than Top: 9 System Monitoring Tools for Linux to Keep an Eye on Vital System Stats.” [Online]. Available: <https://itsfoss.com/linux-system-monitoring-tools/>. Last Accessed 2 Jun. 2025.
- [4] “Spring Boot.” [Online]. Available: <https://spring.io/projects/spring-boot>. Last Accessed 10 Jun. 2025.
- [5] “The Linux Kernel.” [Online]. Available: <https://docs.kernel.org/>. Last Accessed 10 Jun. 2025.
- [6] D. O. S. Prof. Dr. F. Ciorba, “Operating Systems Spring Semester 2025 Ch06: File Systems b. Implementation,” in *Operating Systems Spring Semester 2025 Ch06: File Systems b. Implementation*, p. 8, 2025.
- [7] Senthilkumar Palani, “Understanding the Linux /proc Filesystem: A Beginners Guide.” [Online]. Available: <https://ostechnix.com/linux-proc-filesystem/>. Last Accessed 8 Jun. 2025.
- [8] OpsDash by RapidLoop, “Understanding CPU Usage in Linux.” [Online]. Available: <https://www.opsdash.com/blog/cpu-usage-linux.html>. Last Accessed 2 Jun. 2025.
- [9] IEEE SA Open Group, “IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7.” [Online]. Available: <https://standards.ieee.org/ieee/1003.1/7101>. Last Accessed 10 Jun. 2025.
- [10] Brian “Beej Jorgensen” Hall, “Beej’s Guide to Network Programming.” [Online]. Available: <https://beej.us/guide/bgnet/html/split/>. Last Accessed 6 Jun. 2025.
- [11] “Microhttpd.” [Online]. Available: <https://www.gnu.org/software/libmicrohttpd/>. Last Accessed 10 Jun. 2025.
- [12] “Our Github repository.” [Online]. Available: <https://github.com/S1ntax3rr0r/LinuxTaskManager>. Last Accessed 10 Jun. 2025.
- [13] “thymeleaf.” [Online]. Available: <https://www.thymeleaf.org/>. Last Accessed 10 Jun. 2025.
- [14] “cJSON - JSON File Write/Read/Modify in C.” [Online]. Available: <https://www.geeksforgeeks.org/cjson-json-file-write-read-modify-in-c/>. Last Accessed 10 Jun. 2025.
- [15] GeeksforGeeks, “Inter Process Communication (IPC).” [Online]. Available: <https://www.geeksforgeeks.org/inter-process-communication-ipc/>. Last Accessed 2 Jun. 2025.
- [16] “Apache maven.” [Online]. Available: <https://maven.apache.org/>. Last Accessed 10 Jun. 2025.
- [17] “overleaf.” [Online]. Available: <https://www.overleaf.com/>. Last Accessed 10 Jun. 2025.
- [18] “chatgpt.” [Online]. Available: <https://chatgpt.com/>. Last Accessed 10 Jun. 2025.

- [19] “github.” [Online]. Available: <https://github.com/>. Last Accessed 10 Jun. 2025.
- [20] “visualstudio code.” [Online]. Available: <https://code.visualstudio.com/>. Last Accessed 10 Jun. 2025.
- [21] “Chart.js: Simple yet flexible JavaScript charting library for the modern web.” [Online]. Available: <https://github.com/chartjs/Chart.js>. Last Accessed 10 Jun. 2025.
- [22] “Bootstrap: Build fast, responsive sites with Bootstrap.” [Online]. Available: <https://getbootstrap.com/docs/5.3/getting-started/introduction/>. Last Accessed 10 Jun. 2025.
- [23] “UHASH.” [Online]. Available: <https://troydhanson.github.io/uthash/>. Last Accessed 10 Jun. 2025.
- [24] “Quillbot.” [Online]. Available: <https://quillbot.com/>. Last Accessed 10 Jun. 2025.
- [25] “Stackoverflow.” [Online]. Available: <https://stackoverflow.com/>. Last Accessed 10 Jun. 2025.

Appendix A: Declaration of Independent Authorship

We attest with our signature that we have completed this paper independently and without any assistance from third parties and that the information concerning the sources used in this paper is true and complete in every respect. All sources that have been quoted or paraphrased have been referenced accordingly. Additionally, we affirm that any text passages written with the help of AI-supported technology are marked as such, including a reference to the AI-supported program used. This paper may be checked for plagiarism and use of AI-supported technology using appropriate software. We understand that unethical conduct may lead to a grade of 1 or “fail” or to expulsion from the course of studies. I have taken note of the fact that in the event of a justified suspicion of the unauthorized or undisclosed use of AI in written performance assessments, I am upon request obligated to cooperate in confirming or ruling out the suspicion, for example by attending an interview.

Signatures of all authors

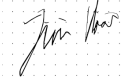
Valerio Job



Manuel Buser



Jiri Käser



Fabian Götschi



Appendix B: List of Aids Used for Completing this Project and declaration of AI usage

Tool	Applied To
Overleaf	Writing and compiling the LaTeX report [17]
ChatGPT Code Assistant	Debugging and writing many lines of print statements as well as verifying code structure, applied to c-core, c-daemon and java-service codebases and sometimes writing comments for functions that others wrote to understand them more quickly [18]
ChatGPT Text Assistant	Improving and refining specific paragraphs in the report; applied in chapter 1.2, and chapter 3; all used prompts looked like this: "Improve the following text", "write this more understandably", "make this text shorter", etc. [18]
GitHub	Version control, collaborative development, and access to project code [19]
Our GitHub repository	https://github.com/S1ntax3rror/LinuxTaskManager [12]
VS Code	Writing and editing source code (C and Java) [20]
Spring Boot Docs	Configuring and integrating the web-based GUI service [4]
Spring Boot Main	Configuring and integrating the web-based GUI service [4]
Thymleaf	Connecting Front-end Components to Back-end Data [13]
Chart.js	Easier deployment of responsive charts [21]
Bootstrap	Pre-built front-end components for modern design [22]
UTHASH	Used for Hash Table [23]
libmicrohttpd	Implementing the HTTP server in the C-daemon [11]
cJSON Library	Serializing and deserializing JSON data in C [14]
Linux man pages	Understanding syscalls and interpreting <code>/proc</code> entries [5]
Quillbot	English Grammer Check for Text Passages; Applied throughout the entire report [24]
Stack Overflow	Debugging specific implementation errors across all codebases [25]

Appendix C: *

List of Figures

1	Get api/stats/all vs GET api/processes	5
2	Web-based GUI: Main Page	9
3	Web-based GUI: General Stats	9
4	Network and Disk plots	9
5	CPU and RAM plots	10