MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
*(A constituent unit of MAHE, Manipal)*

# Mini Project Report
of
# Operating Systems Lab (CSE3163)

SUBMITTED
BY

| Name | Registration number | Email address | Roll Number |
|---|---|---|---|
| Disha Agarwal | 210905412 | disha23.official@gmail.com | 62 |
| Soumya Sahu | 210905196 | sahusoumya4164@gmail.com | 35 |
| Aditi Shrivastava | 210905244 | aditi12303@gmail.com | 44 |
| Vrindavaneshwari Subramanian | 210905198 | vrinda1903@gmail.com | 36 |

Department of Computer Science and Engineering
Manipal Institute of Technology, Manipal.
November 2023

## TABLE OF CONTENTS

## ABSTRACT

The "Resource Allocator Monitor" is a user-friendly system focused on real-time monitoring of essential hardware components like CPU, memory, disk, and GPU. Its primary goal is to provide users with a clear understanding of how their computer resources are being used. This information empowers users to make informed decisions about resource allocation, optimize performance, and maintain their systems effectively.

With an easy-to-navigate interface, the project caters to users of all levels, ensuring that both beginners and experts can benefit from its insights. In a world where efficient resource use is crucial, this monitoring tool simplifies the complexities of hardware management, contributing to a more streamlined and optimized computing experience.

# PROBLEM STATEMENT & OBJECTIVES

In the ever-evolving landscape of computer systems, the efficient allocation and utilization of hardware resources stand out as critical factors determining overall system performance and user experience. The Resource Allocator Monitor project addresses the multifaceted challenges associated with resource management, providing a comprehensive solution to a range of issues that can impact the functionality and longevity of computer systems.

**Complexity of Resource Interdependencies:**
The core problem lies in the intricate interdependencies among CPU, memory, disk, and GPU resources. Modern applications, especially those designed for multitasking and resource-intensive tasks, require a delicate balance of these components. Identifying how each resource influences the others and understanding the implications of their allocation is a complex task. Without an effective monitoring system, system administrators and users are left in the dark about the dynamics of these interdependencies, leading to suboptimal resource allocation.

**Identification of Resource Bottlenecks:**
In the absence of a dedicated monitoring tool, identifying bottlenecks and resource constraints becomes a challenging undertaking. Resource bottlenecks, where one or more components are operating at or near full capacity, can severely impede system performance. The Resource Allocator Monitor addresses this problem by providing real-time data on resource usage, enabling users to pinpoint bottlenecks and take corrective actions promptly.

**Lack of User-Friendly Resource Insights:**
For both system administrators and individual users, accessing and interpreting resource usage data is often a daunting task. Traditional monitoring tools might provide raw data, but translating this information into actionable insights is a considerable challenge. The Resource Allocator Monitor bridges this gap by presenting data in an intuitive graphical user interface. This user-friendly approach

ensures that even non-technical users can grasp the nuances of resource utilization and make informed decisions.

**Resource Optimization for Sustainability:**
In the context of increasing concerns about environmental sustainability, resource optimization goes beyond performance considerations. Inefficient resource usage contributes to electronic waste and unnecessary energy consumption. The Resource Allocator Monitor aligns with the modern ethos of responsible computing by empowering users to optimize their resource usage, reducing electronic waste and contributing to a more sustainable digital ecosystem.

**The Need for Real-time Monitoring:**
Real-time updates on resource usage are crucial for scenarios where rapid changes in resource demands, such as spikes in CPU usage or sudden increases in memory requirements, require immediate attention.

In essence, the problem statement revolves around the overarching challenge of efficiently managing hardware resources in an era of increasingly sophisticated applications and dynamic workloads. The Resource Allocator Monitor emerges as a solution that not only tackles these challenges head-on but also brings resource management into the realm of accessibility for users across various technical backgrounds.

# ALGORITHM

**1. Data Collection:**
The algorithm initiates by collecting data from various hardware components, including the CPU, RAM, GPU, and disk drives. Specialized classes, such as 'CPU', 'RAM', 'GPU', and 'Disk', encapsulate the logic for querying system-level information. These classes leverage external libraries like 'psutil' to ensure cross-platform compatibility.

**2. Real-time Updates:**
To capture the dynamic nature of resource usage, the algorithm continuously updates the collected data at regular intervals. The 'update' methods within each hardware-specific class are pivotal in this process. For instance, the 'update' function in the 'CPU' class queries the CPU usage, temperature, and other relevant metrics at each iteration.

**3. Historical Data Storage:**
The algorithm incorporates a historical data storage mechanism to maintain a record of resource usage over time. This is crucial for generating meaningful insights and trends. Deques, such as 'plot_data' in the 'RAM' and 'GPU' classes, store historical data points with a sliding window approach. This ensures that only the most recent data is retained, preventing memory overflow.

**4. Brand-Based GPU Handling:**
The algorithm intelligently handles GPU information, considering the brand-specific libraries available. The 'GPU' class utilizes a factory-like design, dynamically selecting the appropriate class ('_AMD' or '_NVIDIA') based on the availability of the 'pyadl' module. This ensures that GPU data is accurately retrieved regardless of the underlying hardware.

**5. Cross-Platform Compatibility:**
Ensuring cross-platform compatibility is a key design principle. The algorithm employs conditional checks to determine the operating system and selects the

appropriate methods for data retrieval. The project as of now supports only Linux but it makes use of Generics so that the other operating systems can easily be integrated.

**6. User-Friendly Unit Conversion:**
To enhance user understanding, the algorithm includes a utility function, 'to_units', for converting raw numerical data into human-readable units. This function dynamically determines the appropriate unit (e.g., kilobytes, megabytes) based on the magnitude of the data, ensuring that users receive information in a comprehensible format.

**7. Object-Oriented Design:**
The algorithm embraces an object-oriented design, encapsulating related functionalities within specialized classes. This modular structure enhances maintainability and extensibility, allowing for easy integration of additional hardware monitoring capabilities in the future.

**8. Error Handling and Exception Management:**
Robust error handling and exception management were integrated into the implementation to enhance the tool's reliability. The system could gracefully handle scenarios such as unavailability of hardware data or unexpected errors during data retrieval. Informative error messages and logs were implemented to aid users and developers in diagnosing issues.

# METHODOLOGY

**1. Requirement Analysis:**
The methodology commenced with a thorough analysis of the project requirements. Understanding the need for a resource monitoring tool that provides real-time insights into CPU, RAM, GPU, and disk utilization was crucial.

**2. Design Phase:**
The design phase focused on creating a blueprint for the software architecture and user interface. The decision to adopt an object-oriented design was made to encapsulate hardware-specific functionalities within modular and reusable classes. Class hierarchies such as 'CPU', 'RAM', 'GPU', and 'Disk' were defined, each responsible for monitoring a specific hardware component.

**3. Technology Stack Selection:**
Choosing the right technology stack was pivotal for achieving cross-platform compatibility and seamless integration with hardware-level APIs. Python was selected as the primary programming language due to its versatility, extensive library support, and ease of integration with system-level APIs. External libraries like 'psutil', 'pyadl', and 'GPUtil' were incorporated for hardware-specific data retrieval.

**4. Development:**
Each hardware monitoring module (CPU, RAM, GPU, Disk) was developed incrementally and tested in isolation before integration into the main system.

**5. Cross-Platform Considerations**:
Ensuring compatibility across different operating systems was a key consideration. Therefore, the decision to use Generics was made.

**6. User Interface Design:**
The design of the user interface (UI) was approached with a focus on user-friendliness and information clarity. The GUI folder, which houses the

graphical user interface components, was developed concurrently with the backend. The UI design aimed to present real-time and historical data in an intuitive and visually appealing manner, enhancing the overall user experience.

**7. Continuous Testing:**
The methodology prioritized continuous testing throughout the development lifecycle. Unit testing, integration testing, and system testing were conducted at each iteration to identify and rectify bugs.

# IMPLEMENTATION

## HARDWARE MONITOR

### 1. Worker and Worker_Thread Classes:

Worker_Signals Class:
-'Worker_Signals' is a subclass of 'QObject' that defines two signals: 'tick' and 'error'.
- These signals are used for communication between the worker thread and the main application.

Worker Class:
- 'Worker' is a subclass of 'QRunnable', representing a worker thread.
- It is responsible for running a specified function ('fn') in a separate thread.
- The 'run' method catches exceptions during the execution of the function, emits the 'error' signal in case of an exception, and then emits the 'tick' signal to notify the completion of the task.

Worker_Thread Class:
- 'Worker_Thread' is a subclass of 'QThreadPool' that manages worker threads.
- It initializes a timer ('_timer') to periodically execute a job ('__execute') at a set interval.
- The timer is started upon initialization.

### 2. Monitor Class:

Monitor Class:
- The 'Monitor' class is designed as a singleton to ensure only one instance exists.

- It contains a 'data' attribute, an instance of the 'Data' class from the 'HardwareData' module.
- It also has a 'worker' attribute, an instance of the 'Worker_Thread' class, which continuously updates the data using the 'update' method of the 'Data' class.

**3. Overall Flow:**

i. Worker and Worker_Thread Classes:
- These classes establish a framework for concurrent execution of tasks and handling errors using signals.

ii. Monitor Class:
- The 'Monitor' class initializes a single instance of the 'Data' class for collecting hardware information.
- It also creates a 'Worker_Thread' instance, which periodically updates the data using the 'update' method of the 'Data' class.

iii. Usage:
- The 'Monitor' class can be instantiated to create a single instance of the hardware monitor.

iv. Threading and Timer Usage:

- Threading is employed to ensure that the data updates in the background without freezing the main application.
- A timer ('_timer') triggers the execution of the '__execute' method at regular intervals.
- The '__execute' method creates a new worker ('work') and starts it, connecting its 'tick' signal to the callback provided in 'self.callback'.

## HARDWARE DATA

**Common Base Class '_Generic':**
The '_Generic' class serves as the foundational base for various hardware data collection classes, containing common attributes and initialization logic. It includes a deque ('plot_data') to store historical data, and variables to capture system architecture, CPU core information, and temperature. This class sets the groundwork for platform-specific implementations.

**CPU Information Implementation ('_Linux' ):**
- '_Linux' classes extend '_Generic' to provide platform-specific CPU information.
- They gather details such as CPU brand, base clock frequency, package clock frequency, CPU usage, the number of threads, and temperature.
- The '__thread_count' method calculates the total number of threads by iterating through the active processes.
- For Linux, it utilizes the 'psutil' library to fetch CPU frequency and temperature data.

**Disk Information Implementation ('Disk'):**
- The 'Disk' class manages the collection of disk-related data, including partition information, disk usage, and disk I/O counters.
- It utilizes the 'psutil' library to fetch partition and disk usage data. Additionally, it handles disk I/O counters to track read and write bytes.

**GPU Information Implementation ('_AMD' and '_NVIDIA'):**
- The '_AMD' and '_NVIDIA' classes extend '_Generic' to collect GPU information for AMD and NVIDIA GPUs, respectively.
- They capture data such as GPU brand, name, load, and temperature. The 'pyadl' library is used for AMD GPUs, while the 'GPUtil' library is used for NVIDIA GPUs.

**RAM Information Implementation ('_Linux'):**
- '_Linux' classes under 'RAM' handle platform-specific RAM information.
- They gather data like installed RAM, total RAM, used RAM, RAM usage, available RAM, free RAM, standby RAM, and modified RAM.
- For Linux, it relies on 'psutil' for virtual memory data.

**System Information Implementation ('_Linux'):**
- '_Linux'  classes under 'System' provide platform-specific system information.
- They fetch data such as boot time, OS release, OS version, hostname, and a dictionary of active processes with their names, PIDs, and executable paths.

**Utility Function ('to_units'):**
- The 'to_units' function converts numerical values to human-readable units, allowing for a more user-friendly presentation of data. It includes options for setting offset, decimals, and suffix.

## GUI IMPLEMENTATION OVERVIEW:

The provided code represents a graphical user interface (GUI) for a hardware monitoring application. The GUI is built using the PySide6 library. The application has different tabs for monitoring various hardware components, such as CPU, RAM, Disk, GPU, and History.

**'main.py':**

1. Main Window:
   - The main window is initialized with a fixed size of 960x540 pixels.
   - It contains a central widget, which, in turn, includes a horizontal layout.
   - The central widget consists of a tab container ('QTabWidget') for displaying different hardware information tabs.

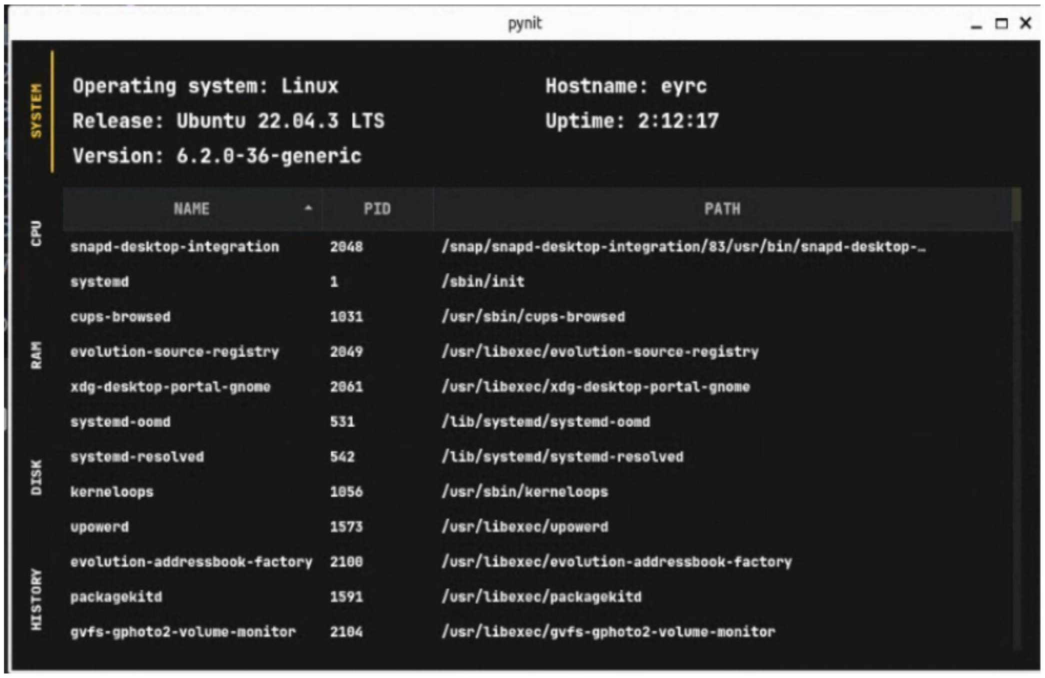
2. Tabs:
   - Tabs are created for System, CPU, RAM, Disk, GPU, and History.
   - Each tab is implemented using separate classes ('system_tab', 'cpu_tab', etc.).
   - Each tab includes a 'Plot' widget for real-time plotting and a 'QGroupBox' for displaying various hardware details.

3. Dynamic GUI Components:
   - The GUI dynamically adapts based on the availability of GPU information. If GPU data is available, the GPU tab is added; otherwise, it is excluded.

4. Signals and Slots:
   - Signals and slots are connected to update the GUI components with real-time data from the hardware monitor.

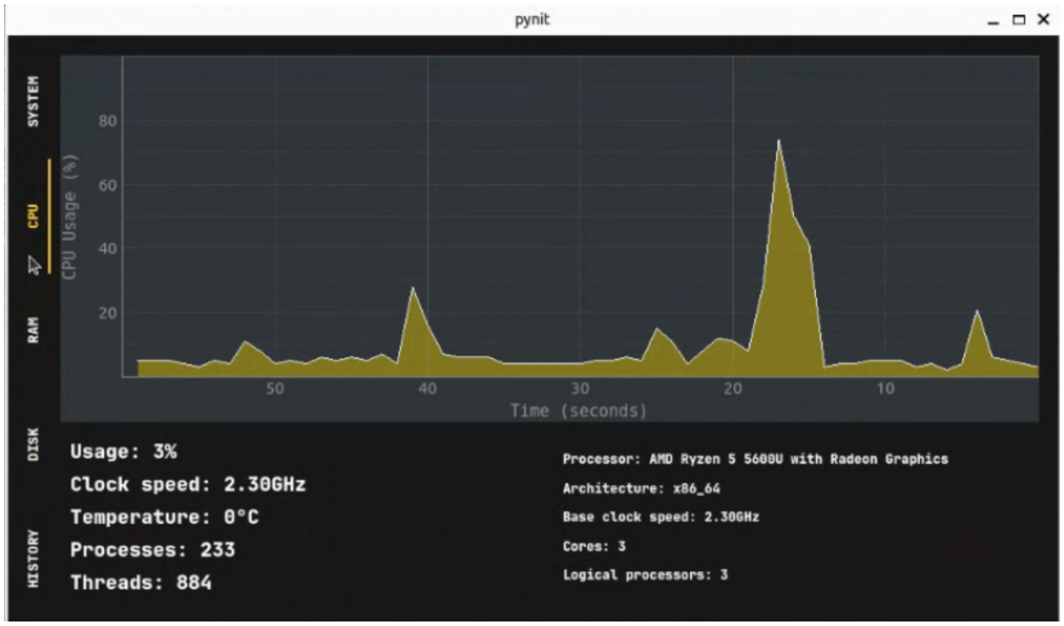
The graphical user interface (GUI) implementation follows a modular design, with separate tabs for different hardware components such as system, CPU, RAM, disk, GPU, and historical data. The main window structure is defined in the 'main.ui' file, and each tab is implemented as a distinct class. The 'cpu.ui' file, for example, defines the layout for the CPU tab. The 'main_window' class dynamically adjusts the UI based on the system's GPU availability and connects real-time updates through a hardware monitor's worker callback. The 'Plot' class customizes the plotting of historical data. The 'load_theme' function applies a dark amber theme and a custom font. This organized structure enhances code readability, maintainability, and facilitates easy expansion of the hardware monitoring application.

## SCREENSHOTS:

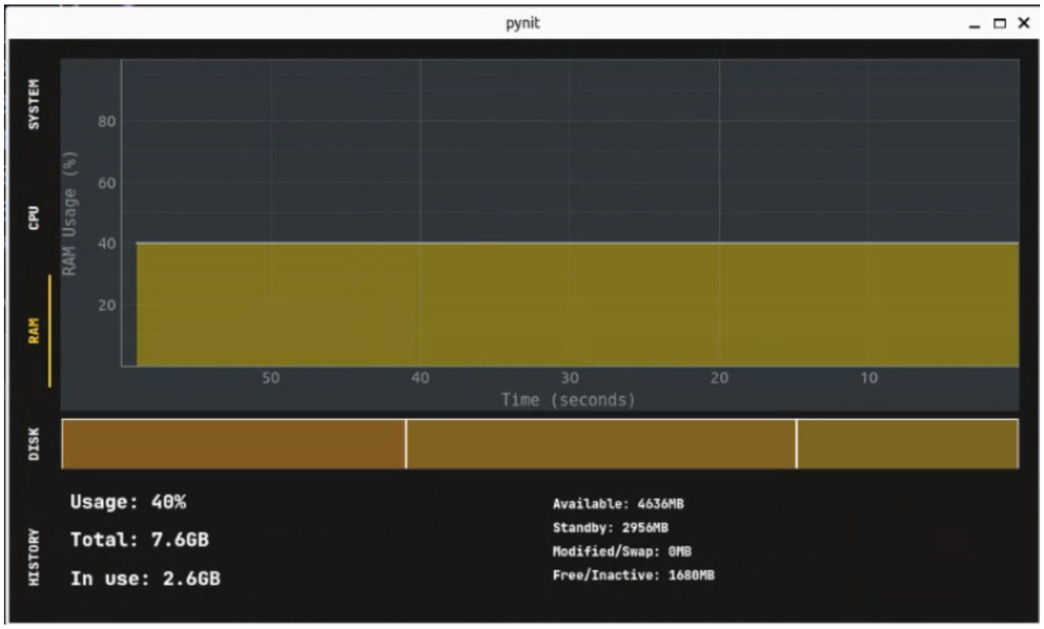


```
Operating system: Linux              Hostname: eyrc
Release: Ubuntu 22.04.3 LTS          Uptime: 2:12:17
Version: 6.2.0-36-generic
```

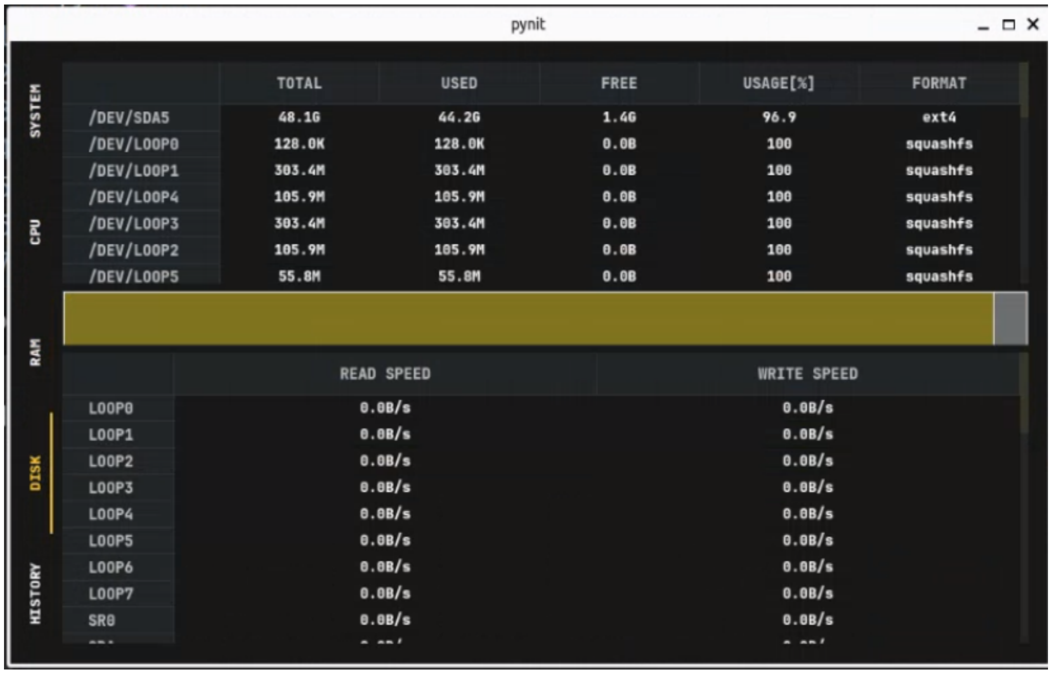
| NAME | PID | PATH |
|---|---|---|
| snapd-desktop-integration | 2048 | /snap/snapd-desktop-integration/83/usr/bin/snapd-desktop-… |
| systemd | 1 | /sbin/init |
| cups-browsed | 1031 | /usr/sbin/cups-browsed |
| evolution-source-registry | 2049 | /usr/libexec/evolution-source-registry |
| xdg-desktop-portal-gnome | 2061 | /usr/libexec/xdg-desktop-portal-gnome |
| systemd-oomd | 531 | /lib/systemd/systemd-oomd |
| systemd-resolved | 542 | /lib/systemd/systemd-resolved |
| kerneloops | 1056 | /usr/sbin/kerneloops |
| upowerd | 1573 | /usr/libexec/upowerd |
| evolution-addressbook-factory | 2100 | /usr/libexec/evolution-addressbook-factory |
| packagekitd | 1591 | /usr/libexec/packagekitd |
| gvfs-gphoto2-volume-monitor | 2104 | /usr/libexec/gvfs-gphoto2-volume-monitor |



```
Usage: 3%                      Processor: AMD Ryzen 5 5600U with Radeon Graphics
Clock speed: 2.30GHz           Architecture: x86_64
Temperature: 0°C               Base clock speed: 2.30GHz
Processes: 233                 Cores: 3
Threads: 884                   Logical processors: 3
```

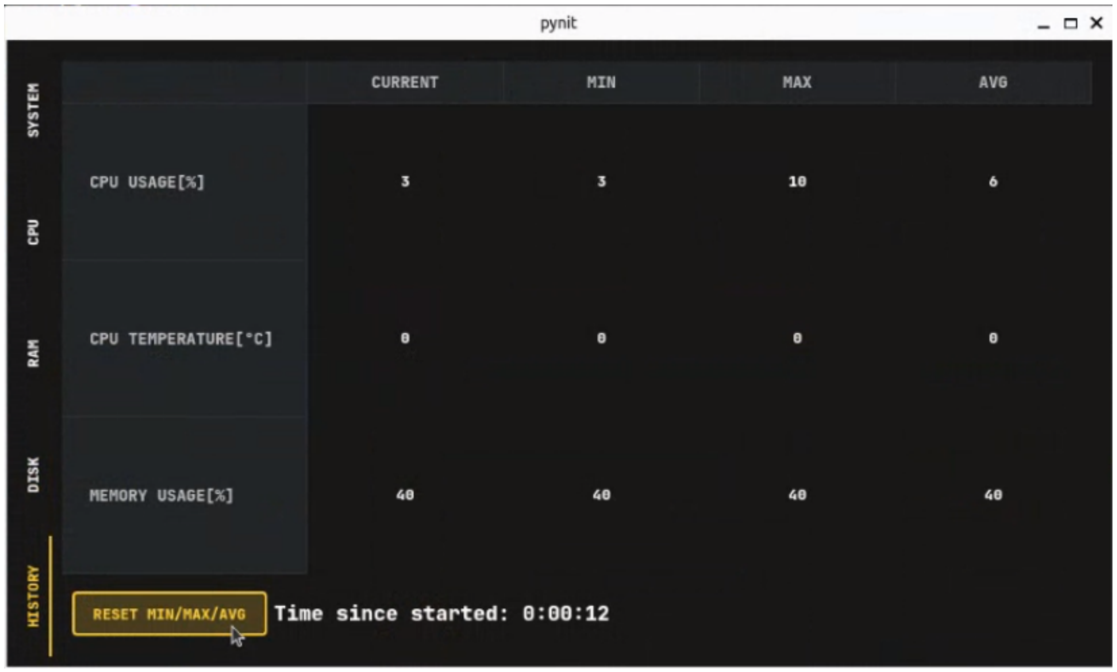


```
Usage: 40%                          Available: 4636MB
Total: 7.6GB                        Standby: 2956MB
In use: 2.6GB                       Modified/Swap: 0MB
                                    Free/Inactive: 1680MB
```



| | TOTAL | USED | FREE | USAGE[%] | FORMAT |
|---|---|---|---|---|---|
| /DEV/SDA5 | 48.1G | 44.2G | 1.4G | 96.9 | ext4 |
| /DEV/LOOP0 | 128.0K | 128.0K | 0.0B | 100 | squashfs |
| /DEV/LOOP1 | 303.4M | 303.4M | 0.0B | 100 | squashfs |
| /DEV/LOOP4 | 105.9M | 105.9M | 0.0B | 100 | squashfs |
| /DEV/LOOP3 | 303.4M | 303.4M | 0.0B | 100 | squashfs |
| /DEV/LOOP2 | 105.9M | 105.9M | 0.0B | 100 | squashfs |
| /DEV/LOOP5 | 55.8M | 55.8M | 0.0B | 100 | squashfs |

| | READ SPEED | WRITE SPEED |
|---|---|---|
| LOOP0 | 0.0B/s | 0.0B/s |
| LOOP1 | 0.0B/s | 0.0B/s |
| LOOP2 | 0.0B/s | 0.0B/s |
| LOOP3 | 0.0B/s | 0.0B/s |
| LOOP4 | 0.0B/s | 0.0B/s |
| LOOP5 | 0.0B/s | 0.0B/s |
| LOOP6 | 0.0B/s | 0.0B/s |
| LOOP7 | 0.0B/s | 0.0B/s |
| SR0 | 0.0B/s | 0.0B/s |



| | CURRENT | MIN | MAX | AVG |
|---|---|---|---|---|
| CPU USAGE[%] | 3 | 3 | 10 | 6 |
| CPU TEMPERATURE[°C] | 0 | 0 | 0 | 0 |
| MEMORY USAGE[%] | 40 | 40 | 40 | 40 |

Time since started: 0:00:12

# REFERENCES

[1] How to use pyqtgraph
https://www.pythonguis.com/tutorials/plotting-pyqtgraph/

[2] cpuinfo.py (License: BSD)
https://github.com/pydata/numexpr/blob/master/numexpr/cpuinfo.py
[3] PySide6 documentation
PySide6.QtWidgets - Qt for Python

[4] qt_material documentation
Qt-Material — Qt Material documentation

[5] pyadl documentation
piwheels - pyadl

[6] GPUtil documentation
GPUtil · PyPI

[7] itertools documentation
itertools — Functions creating iterators for efficient looping — Python 3.12.0 documentation

[8] collections documentation
collections — Container datatypes — Python 3.12.0 documentation

[9] using traceback module
Traceback in Python - GeeksforGeeks

[10] using sys module
Python sys Module - GeeksforGeeks

.