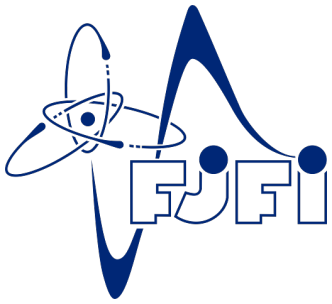
Hardware System Monitor

Jan Vyhnánek

May 2025



Contents

1 Introduction

This is documentation for the *HWSM* Python application. *HWSM* is a python script-based multiplatform universal open source application for professional use. The main scope of the application's functionalities is focused on system resource monitoring and maintenance. Originally this project began as an idea from need as there was very little good system monitor applications for Linux systems, especially those with highly-customizable desktop environments such as I3WM, which this all started on. Later on, this progressed to the point of semastral work for the Faculty of Nuclear Sciences on the CTU in Prague. These two factors together pushed this project from an idea into creation, and also explain why this documentation is released as an official CTU document.
It is vital to answer the question of the choice of Python. Indeed Python, as an interpreted programming language, is in no way optimal for such jobs. On the other hand Python is widely known, used and simple to debug and maintain. The primary aim is for sure on the GNU/Linux based system, where Python is nowadays always pre-installed with the kernel. Also Python applications are easy to carry around as they can run from a single script and will always run on any machine with Python installed. Therefore we don't have to worry about operating system, CPU architecture or instruction set. However this still doesn't tip the scales completely, so in the future there might be a rework into C.

2 Primary features

This section addresses the most important features of the back-end code of the application. There is still room for discussion whether the utilities and libraries used were the best choice, but for now the documentation concerns only the present libraries.

2.1 Matplotlib

Matplotlib is a widely-known library for Python that provides an interface for graphs figures. Despite being great for plotting graphs, it can also handle user interfaces fairly flawlessly. There should be a disclaimer before the continuation, and that is about the fact that Matplotlib was not designed to handle entire application GUI like this. *HWSM* is simply a clever construction based on Matplotlib that makes it possible, but there are many non-ideal features due to it.
Matplotlib is utilized in this project in basically every feature. It handles the entire GUI, including buttons, screen handling, and theme handling. Naturally, it also covers the plotting of every single graph and text in the application. More about screen and theme handling in Sections 3 and 5.

2.2 Multiprocessing

Multiprocessing is a library that provides an easy way to handle multithreading using Python. In this application, it is used as a way to monitor the usage of RAM and CPU without disrupting the GUI or handlers in any way. More about resource monitoring in Section 4.1. In short, the data are being read every 200 milliseconds, and this process would temporarily interrupt any process posted by the GUI or any handler of the application. This would cause an unstable and sluggish program that would lack usability. Therefore, multiprocessing is the way to maximize the speed and usability of the application by maintaining separate threads for these individual processes.

2.3 Psutils

Psutils is a library for Python using which you can easily get a vast multitude of system information. For us, most notable intel will be RAM, CPU and disk statistics. Nearly all measurements in the application are realized via this library and, therefore, it is a vital part in its core.

2.4 Other modules

3 Screen handling

HWSM is sectioned into so-called screens. Essentially, they are just logically partitioned segments that use their own elements and are contorted by calling a

function for the given screen. The basic algorithm for each screen is always the same for modularity reasons. You can see this "blank" algorithm in picture 1.

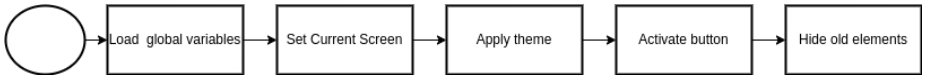


Figure 1: General screen algorithm

4 Subdirectories

Subdirectories is a way of refering to the individual screens that user can interact with. The main source of navigation between the given subdirectories is the main panel in the top of the application. The panel consists of interactive buttons that transform the screen into the selected one. As it was stated before, there are no instances of screens, instead there are sets of elements that switch on call by the given screen function. This is a way of overriding the original design of Matplotlib. It isn't ideal but when optimized correctly, it can work smoothly. However, there are few thigns to keep in mind.
First and possibly the most important part of the issue that can occur is duplicate elements. This is an issue with creating or calling an element which is not accessible anymore. This can occur for instance when user switches between screens but the program still calls on an element that is no longer in the view. Similar to this, if a user changes screen, but some element fails to unload, it can lead to overwriting the old element. Since most of the elements are referred to as functions, this can cause a crash or at the very least a bug.
In this application, a function "hide_all" is utilized to deal with this issue. This destructor-styled function is just a collection of removing and hiding all elements that are present in the screen switching mechanism. By calling this function, every element will be switched off or removed completely, to prevent overwriting or duplication of elements. It also consists of multitude of other "child" functions, that are defined all over the application. These functions are descendants of individual screens, helping to unify destructor features and keep the hide_all function clear. Keep in mind that every time the hide_all function is called, all of the function are also called. This can come out as redundant, but in most cases no actions are caused as the switched features will already be in correct state. Therefore it can be taken as positive redundancy instead, keeping the code clear from accidental bugs.

4.1 Resource Monitor

The resource monitor is the default screen that is displayed when the application starts. It is a simple screen with a couple of graphs plotting RAM and CPU usage over short periods of time. This screen is aimed at monitoring the current state of the resources, not the long-term statistics.
Both RAM usage and CPU usage are monitored by their functions, which are

ran on separate threads. Multi-threading is handled as stated in chapter 2.2. The inner working of both functions is basically identical, with the exception of time handling which is appended in the RAM monitoring function.
"monitor_ram" function takes two positional arguments: ram_list and time_list. These are both list structures containing data about the measurements. They're both initialized and passed down by the main thread and are only modified in the separate threads, therefore we won't run into much synchronization troubles (however they can occur).
The core of the monitor function is psutils function $psutil.virtual\_memory().used$ which return the used virtual memory in bytes. For the sake of human-readable output, we'll modify the output by simple mathematical conversion into gigabytes instead. Therefore the final monitor function will be
$psutil.virtual\_memory().used/(1024 * *3)$
As it was mentioned, after getting virtual memory usage, the timestamp is memorized. This is done by appending a value of the length of the current time list. This serves a purpose of synchronization as we'll use this single list for both CPU and RAM values and they will not run completely simultaneously.
The function "cpu_monitor" work similarly to the "ram_monitor" with the exception of different psutils function and the exception of not creating the timestamp as we'll use the same timestamp from RAM monitor. The psutils function $cpu_percent$ takes an argument of an interval in milliseconds. This is the value of interval field in which we monitor CPU usage. To simply answer the question of why we need this, we could get into the speeds of modern processors and that we can hardly get precise value in languages like python. For more information refer to the Psutils manual.

4.2 Process Monitor

The process monitor is second in the order of navigation panel. The main functionality is viewing the running processes and information about them. Secondary to that is process management, which is utilized only by the ability to "kill" the individual processes. In fact, on linux devices, the signal "Terminnate" will be used instead as it is the default for the "kill" command.