# OS Project Report: Activity Monitor Handler

Severin Memmishofer, Yash Trivedi

08.09.2023

Operating Systems Course - University of Basel

# Introduction

Our project aims to create an efficient tool for monitoring system processes, alerting users whenever excessive CPU consumption in one of the logical CPU cores is detected, and giving them the ability to terminate specific processes. In today's computing environments, processes play a fundamental role in supporting a wide range of applications and services. However, the simultaneous execution of numerous processes can lead to sudden spikes in CPU usage, resulting in performance decline. To address this issue, constant monitoring and control of CPU usage are essential.

In this project, we have developed a process monitoring solution using the C programming language and the Linux operating system. By accurately tracking and analyzing various processes, our tool enables real-time notifications to be sent to users whenever a process surpasses the previously determined CPU usage threshold. This pop-up window gives users the ability to respond to excessive hogging of CPU resources and prevent any potential adverse effects, such as overheating the system. Our project proves particularly useful for power-users, who manage multiple processes concurrently, ensuring their system operates optimally.

# Background

The project was developed entirely on the Ubuntu VM, although the end result also functions on Linux outside of a VM. Our product comprises a C program and Bash scripts, which were created using the Linux text editor.
To test our implementation, we used the *stress* tool on Linux to create a process with an artificially high CPU usage, which our program should, if it works correctly, detect and then give the option to ignore, suspend or kill this process. The tool allows us to create several CPU hogs, which use 100% of the CPU, until terminated.

# Implementation

## Defining Threshold

At the initial stage of our project we considered using the CPU temperature as the threshold, however we encountered difficulties in obtaining the temperature data on Ubuntu Linux using a virtual machine (VM). The lack of access to hardware sensors in a VM prevented us from accessing the thermal file in the sys directory or using the system monitor's sensor option. Therefore, we concluded that it would be more practical to utilize CPU usage as the threshold parameter. This approach would also make our project more accessible to a wider range of users, including those working with virtual machines.

## Obtaining the data

Now that we decided on using the CPU utilization instead of the temperature, we had several ideas on how to achieve this. Our first approach was to research on where the top-command gets it's data from. We found out that there is the /proc directory on linux, where there exists a directory for each process, named by the PID, which includes a stat file. In this stat-file several details about the process are written, like the time the process spent running in user mode, or the time it spent running in kernel mode. Additionally there exists a file which records the total cpu time, meaning the time the cpu spent running since the system booted, measured in clock ticks. We partly succeeded with this approach, however we encountered some difficulties when calculating the correct (current) CPU utilization of a process, as the times recorded are all total values, and for measuring the current utilization, we would need to record the values at the beginning and the end of some time interval and calculate the usage with the difference between them. As we couldn't figure out the correct way to do this and ran into a lot of inconsistencies we searched for different solutions.

Our next idea was, instead of calculating the CPU Usage all by ourselves, there already exists the top-command in Linux, which already does all the relevant work of computing the current values of CPU usage correctly. As this command is well proven, we thought it would be better to use it as input for our program, as then we can rely on having consistent and correct data, which is essential for a correctly functioning program. *Note: We define 100% CPU utilization as maxing out one logical core, as implemented by the top-command, NOT maxing out the whole CPU*.

Additionally we tried to use the „perf" profiling tool to measure system events; but we also encountered several problems. First of all, it requires the installation of linux tools, which are specific to the installed kernel version. And even this installation didn't work on some of our test systems. Additionally, on the systems where it did work, the perf tool only had very limited access to data, and was much less powerful than shown in the lecture. Unfortunately this is again a side effect of running Linux in the VM, and it also depends on the type of virtualization software used, whether you have access to some specific hardware events or not.
Because of this inflexibility, we opted out of this method, as it requires system-specific setup, and we ideally want our application to work universally, without much user input required in the installation process.

---

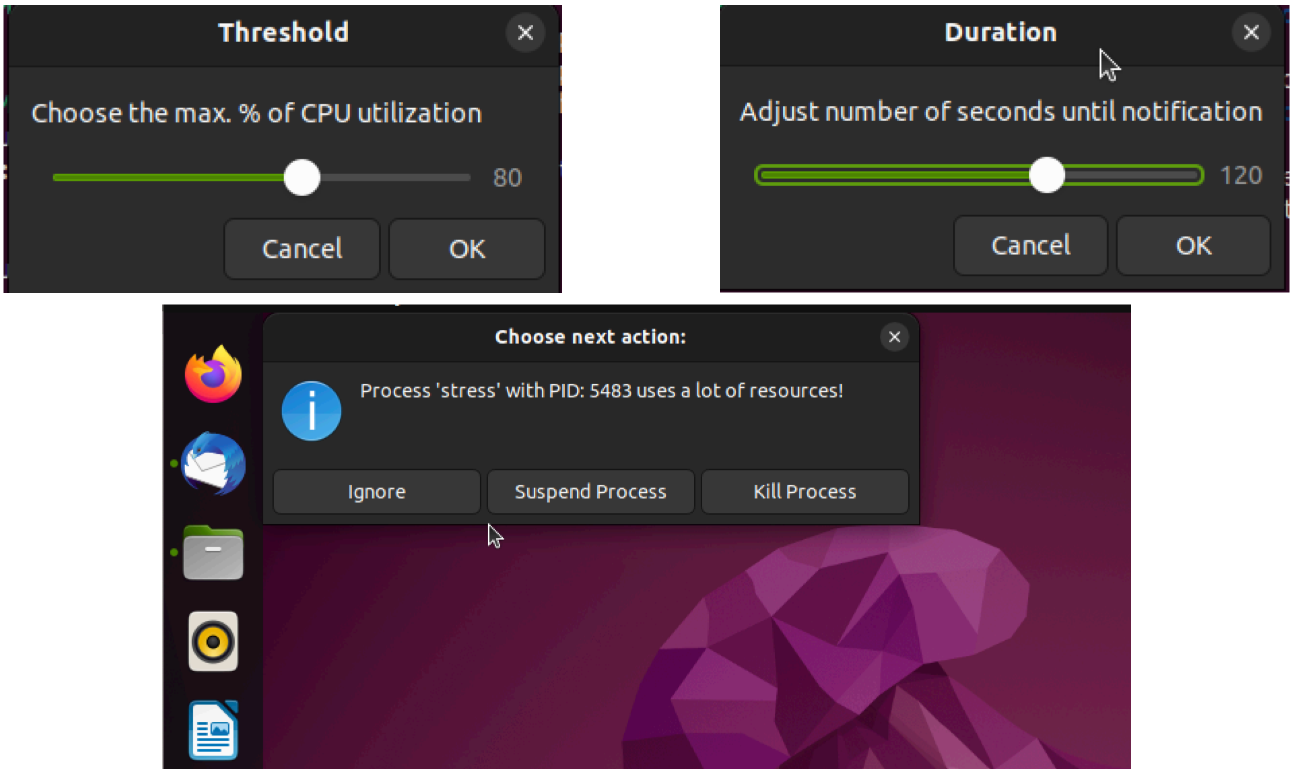
### Approach used for Implementation

**Back-end:** We implemented our solution using a combination of bash scripting and C programming. Firstly, we created a bash script that utilizes the *top* command to retrieve relevant data such as PID, CPU usage percentage, and command name. This data is then written to a file. To integrate this script into our C program, we periodically invoke the bash script using the ***system("sh bashscript.sh")*** function, which executes the script as if it were run in the console. This invocation is placed at the beginning of each loop iteration.
After calling the script, we introduce a one-second delay to ensure the file is updated. Subsequently, we open the file in our C program and read its contents. Since we are interested in displaying the top 5 processes consuming the most CPU, we store their information in an array indexed by their respective PIDs. It is important to note that in Linux, PIDs are limited to 4194304. Within this array we keep track of the number of times a process exceeds the threshold defined by the user (Default is 80%). If a process drops below this threshold, we reset its count.

**Front-end:** We needed a graphical user interface to alert the user whenever the threshold is exceeded, allowing them to choose one of the following actions for the problematic process: Ignore, Suspend, Kill. These three options are integrated into our code using bash scripts. To create the notification pop-up, we utilized Zenity, a tool for generating pop-ups with buttons on Linux. Additionally, Zenity was used to create another pop-up at the beginning to specify the threshold percentage and duration. A simple example of Zenity for an error pop-up would be:
***system("zenity --error --text='Error occurred: File top_output.txt could not be opened!'");***

## Result

In the screenshots below, you can see the user-friendly graphical user interface (GUI) that we made. The GUI is divided into three sections: defining the threshold, specifying the duration until receiving an alert, and an alert pop-up with three available actions. Our primary objective was to design the program to be minimalistic, allowing all necessary actions to be performed directly through the graphical user interface.



Choose the max. % of CPU utilization
80

Adjust number of seconds until notification
120

Process 'stress' with PID: 5483 uses a lot of resources!

Figure 1: Showing the GUI of the product

## Discussion

Our end product is a fully functional process monitor, which can generate alerts. Users have the ability to define their preferred parameters for when to receive alerts. When an alert is triggered, the user can choose to ignore, kill, or suspend the respective process using a single button. Our program operates in the background, constantly monitoring and tracking the top 5 processes that consume the most resources.
During our testing phase, we observed that our own monitoring process has minimal impact on CPU usage. We have carefully set the default threshold parameter to ensure that our program never mistakenly terminates itself, which was a slight concern raised during our initial presentation.

## Conclusion

In conclusion, we successfully accomplished most of the things we set out to do during our initial presentation and met our project goals. We reported the challenge with accessing the temperature sensor in the virtual machine, which was a possible bonus feature, so we decided to use CPU usage as the threshold parameter instead, making our product more accessible to a wider range of users.

To ensure compatibility with our own devices, we tested the program on our Macbook, which runs macOS. There only seems to be one issue on macOS, namely the top-command is structured differently and the -b option, to print the top-output into a file, doesn't work. As a result, the processes in the *top-output.txt* file are wrong. We sadly didn't have any more time before the deadline to work out a solution; but we suspect it should be possible without much alteration of our code. Additionally, we needed to install „zenity" separately. As a future improvement, we would like to finish the program to work also on MacOS, which would further increase the possible user base, including ourselves. Initially, our intention was to develop the program specifically for macOS, but we followed recommendations and opted for Linux development, which was ultimately a good decision as Linux is more development friendly.

## Lessons Learned

Throughout this project, we gained good insights into two main areas. Firstly, we explored the interaction between a C program and a Bash script, realizing the potential to simplify our program with this combination. Secondly, we explored the interaction between users, processes, and the management of processes by the operating system. This allowed us to extend the capabilities of the operating system to meet our needs, such as suspending or killing processes based on user preferences. In summary, we now feel more confident in dealing with various tasks related to the operating system and have become comfortable working within the Linux environment.

The hands-on experience gained from this project has provided us with a good foundation for future projects in similar domains. In addition to realizing our project goals, we also learned a lot about how the interaction between processes and the operating system works internally.

## References

We used following libraries: *<stdio.h>, <stdlib.h>, <time.h>, <string.h>*.

Grammar, spelling mistakes and sentence structure were occasionally checked by Chat GPT; we didn't use it to generate any new content; all content was originally written by us.

***Declaration of Independent Authorship***
*We attest with our individual signatures that we have written this report independently and without outside help. We also attest that the information concerning the sources used in this work is true and complete in every respect. All sources that have been quoted or paraphrased have been marked 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 report may be checked for plagiarism and use of AI-supported technology using the appropriate software. We understand that unethical conduct may lead to a grade of 1 or "fail" or expulsion from the study program.*

S. Memmishofer Yash

*Severin Memmishofer* *Yash Trivedi*