CS1007

A03- Scripting and Process

240021654

Table of Contents

Introduction

The purpose of that lab was to solidify my understanding of process management in Unix-like systems through the creation of Bash scripts mimicking the functionalities of ps, top, and htop. The scripts I wrote are : pso-which mimics the ps command and supports flags such as -u, -a, and --p -toppo, which extends pso with dynamic features similar to top, supporting flags like -d, -p, and -U and htoppo: Builds on toppo, providing interactive process management while supporting the -d and -p flags.

This practical offered me first-hand experience with the /proc filesystem, enabling me to extract and manipulate process data effectively. The assignment concluded with the submission of the scripts, accompanied by a write-up describing their implementation and capabilities, which helped reinforce my knowledge of shell scripting and process monitoring.

Research and Understanding of the various commands

The ps command

The ps command, known as "process status," provides an overview of what is happening on the computer, such as running different programs or apps. You can give the ps command special instructions, called flags, to customize the information displayed, such as checking what a specific user is doing or how much power a particular program is consuming. Some flags include -u, -a, -p. -u displays processes that belong to a specific user along with information like CPU and memory usage. -a displays all processes except session leaders and processes not associated with a terminal. -p displays information for a specific process id i.e. it lets you focus on a single process to check its status.

The top command

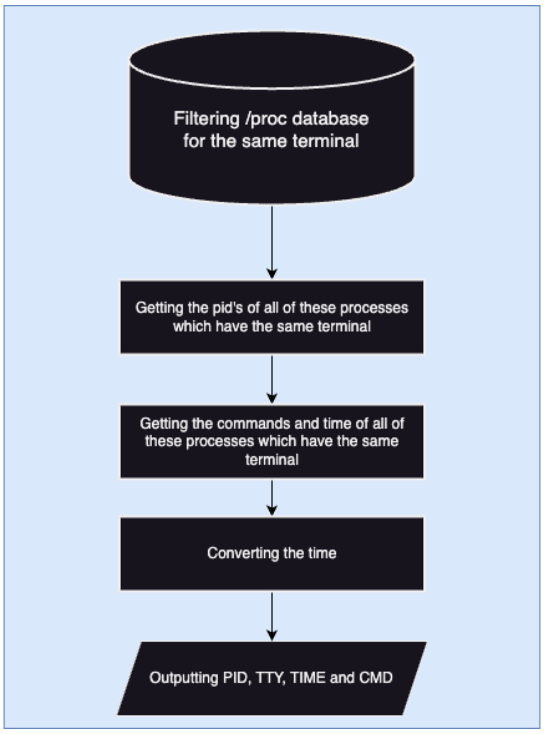
The top command displays real-time data of system performance, providing a detailed overview of system processes and resource usage. This view keeps itself up-to-date, displaying the most CPU-demanding tasks and information on resources being used. It is usually sorted by %cpu and the process which use the most amout of cpu are displayed at the top and it goes down in decending order. Some flags with top include -p, -d, -U. The -p looks particularly into 1 process ID and displays the information of that specific process id. The -d flag takes a second numeric argument which suggests the update time to delay by the user input. The update time is 2 seconds usually, but with the -d flag, it can be altered. The -U flag is the user flag and it implies that the information displayed will be all the processes executed by the user. The header of the top page displays system information relevant to the states of the tasks and the memory usage, swap memory and %cpu.

The htop command

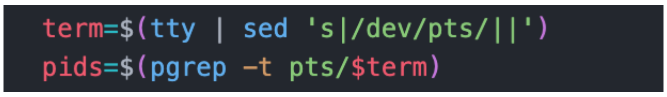
htop is an interactive process viewer that improves upon the functionality of top. It provides a more user-friendly interface with colored output, allowing users to easily navigate and manage processes, view detailed information, and perform actions such as killing processes directly from the interface. Some of the flags I worked on and researched were -p and -d. These flags have functionality similar to those in the top command. The -p flag displays information for a specific process ID, while the -d flag takes a numeric argument from the user and sets it as the sleep time.In addition, htop displays a graphical representation of all CPU usage percentages and provides statistics such as kernel threads, threads, load average, uptime, and the number of tasks. Like top, all processes are dynamically updated every two seconds. However, a significant advantage of htop is its ability to display tasks in any desired order. Users can sort them in ascending or descending order based on various parameters such as SHR, %CPU, RES etc.

Design of each command and their respective flags

ps (no flag)


Filtering /proc database for the same terminal
Getting the pid's of all of these processes which have the same terminal
Getting the commands and time of all of these processes which have the same terminal
Converting the time
Outputting PID, TTY, TIME and CMD

As per my research and understanding, the ps command outputs the PIDs of processes that share the same terminal as the user. The process I used to achieve this involved first obtaining the user's terminal using the following command:

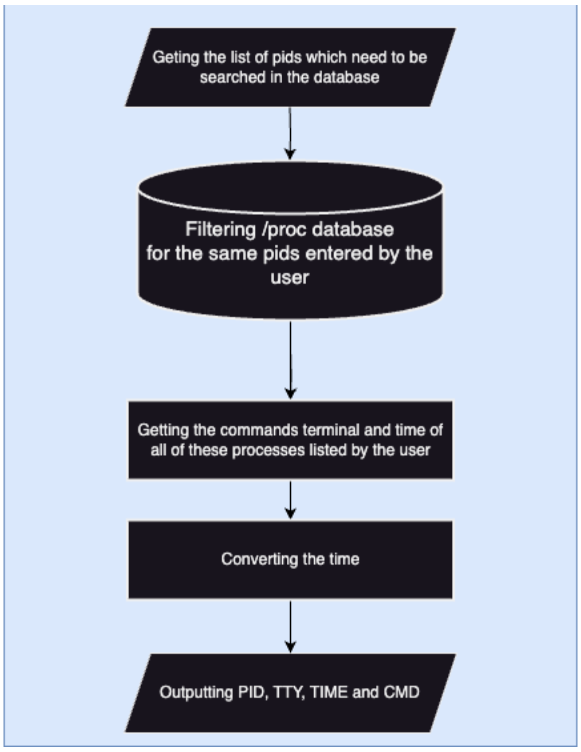


```
term=$(tty | sed 's|/dev/pts/||')
pids=$(pgrep -t pts/$term)
```

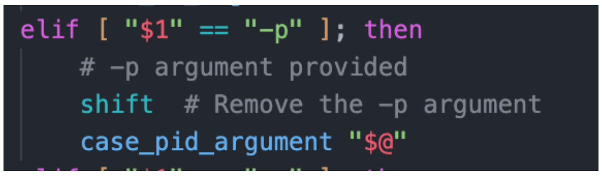
With pgrep, I was able to search for processes associated with this specific terminal. I then iterated through all the PIDs in a for loop to retrieve the corresponding commands and performed a few calculations to convert the time format.

For the output, I had two choices: using echo or printf. I preferred printf because it allowed me to achieve precise spacing, which would have been more challenging with echo. Although I tested both echo -e and printf, I ultimately found printf to be more suitable for achieving the desired formatting.

ps -p

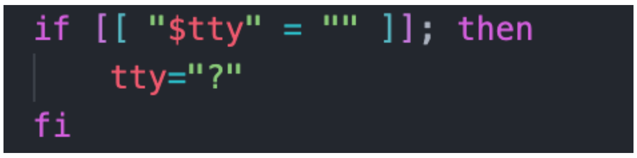

Geting the list of pids which need to be searched in the database
Filtering /proc database for the same pids entered by the user
Getting the commands terminal and time of all of these processes listed by the user
Converting the time
Outputting PID, TTY, TIME and CMD

I used the shift command to remove the -p flag, leaving me with only a list of PIDs that I



```
elif [ "$1" == "-p" ]; then
    # -p argument provided
    shift  # Remove the -p argument
    case_pid_argument "$@"
```

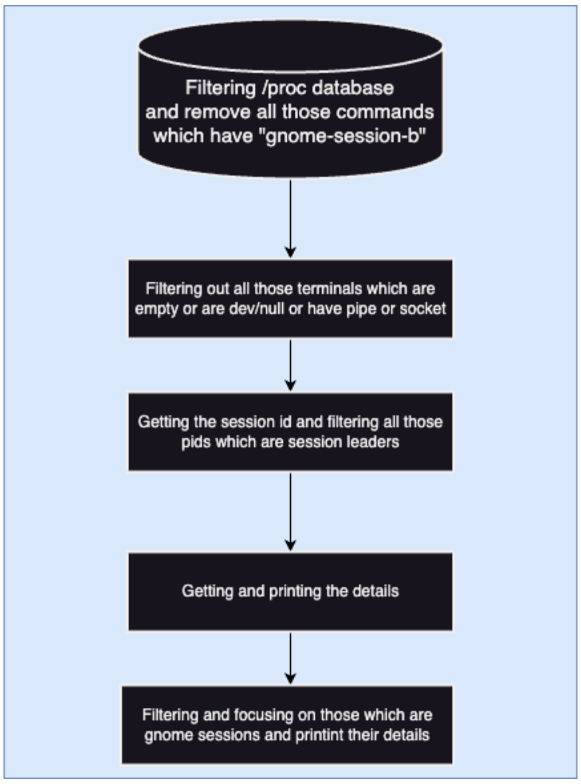
needed to iterate through in the for loop. I essentially replicated the for loop I had used for ps, but to improve error handling, I added the following if statement:



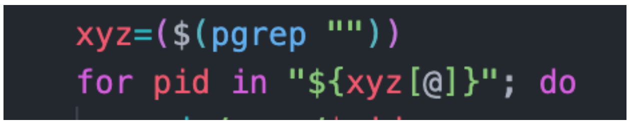
```
if [[ "$tty" = "" ]]; then
    tty="?"
fi
```

This ensured that if a process did not have a terminal, it would not appear as blank but would instead be represented by a question mark.

ps -a

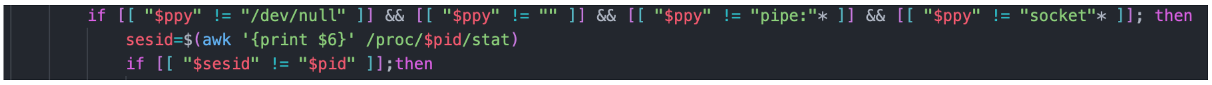

Filtering /proc database and remove all those commands which have "gnome-session-b"
Filtering out all those terminals which are empty or are dev/null or have pipe or socket
Getting the session id and filtering all those pids which are session leaders
Getting and printing the details
Filtering and focusing on those which are gnome sessions and printint their details

ps -a usually filters out those pids which are assocated to a terminal and are not session leaders. When I running my scripts I noticed that I was not getting the gnome sessions (I gave mentioned this in the limitations in detail) so I filtered them out. I filtered out all those process ids which do not have a terminal and if it process id and session id didn't match, I outputted them.



```
    xyz=($(pgrep ""))
    for pid in "${xyz[@]}"; do
```

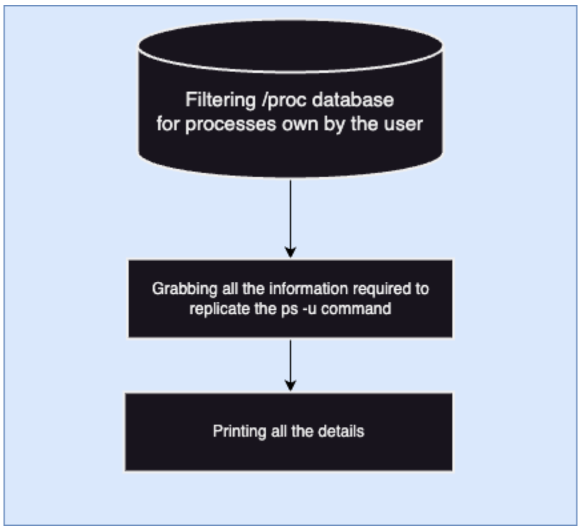
I was experimenting using an array in this script and it turned out that the array managed to efficiently store and manage all the pids without the need of separate variables.



```
        if [[ "$ppy" != "/dev/null" ]] && [[ "$ppy" != "" ]] && [[ "$ppy" != "pipe:"* ]] && [[ "$ppy" != "socket"* ]]; then
            sesid=$(awk '{print $6}' /proc/$pid/stat)
            if [[ "$sesid" != "$pid" ]];then
```

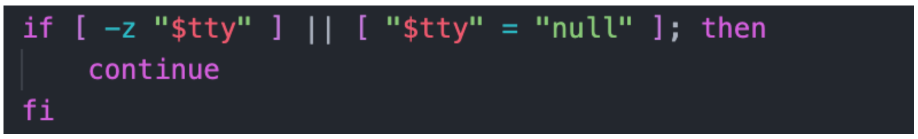
This is a snapshot of my filtering process.

ps -u


Filtering /proc database for processes own by the user
Grabbing all the information required to replicate the ps -u command
Printing all the details

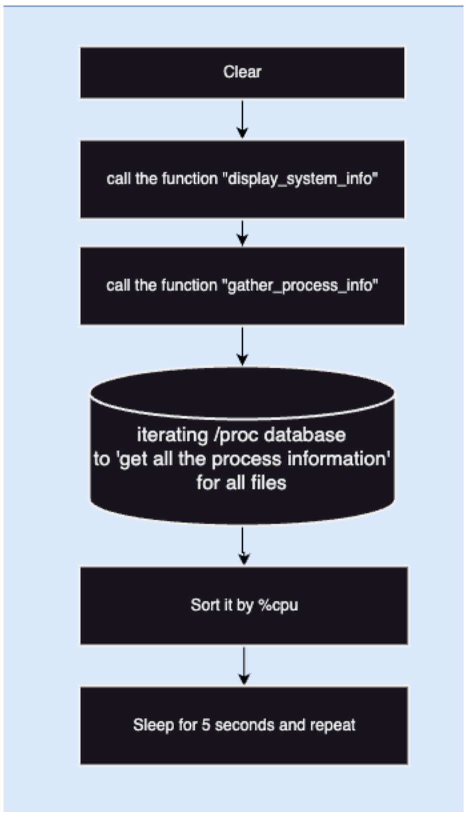
The ps -u command is typically used to filter processes by user. It is designed to display all the processes owned by the specified user. Additionally, it provides more detailed information compared to other ps command variations.

In this script, I preferred using echo -e to better align all the columns. However, I missed including a column called stat, which I have detailed in the limitations section. Additionally, my decimal printing was not as precise as I would have liked, which is something I aim to improve as an extension to this project. I also filtered out processes with null terminals, as ps is intended to display only processes owned by the user that are associated with a terminal.

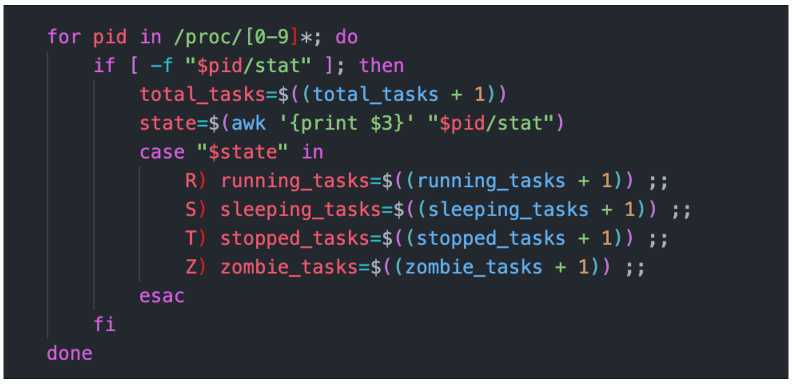


```
if [ -z "$tty" ] || [ "$tty" = "null" ]; then
    continue
fi
```

top (no flag)


Clear
call the function "display_system_info"
call the function "gather_process_info"
iterating /proc database to 'get all the process information' for all files
Sort it by %cpu
Sleep for 5 seconds and repeat

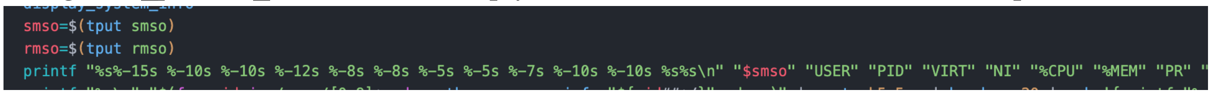
The top command displayed a dynamic view of running processes, including their resource usage and system load. In my implementation, I first cleared the screen and then called the display_system_info function, which displayed the header section of the top command.



```
for pid in /proc/[0-9]*; do
    if [ -f "$pid/stat" ]; then
        total_tasks=$((total_tasks + 1))
        state=$(awk '{print $3}' "$pid/stat")
        case "$state" in
            R) running_tasks=$((running_tasks + 1)) ;;
            S) sleeping_tasks=$((sleeping_tasks + 1)) ;;
            T) stopped_tasks=$((stopped_tasks + 1)) ;;
            Z) zombie_tasks=$((zombie_tasks + 1)) ;;
        esac
    fi
done
```

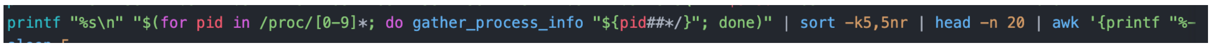
I used a switch statement instead of an if statement here, as the latter would have made the code longer and increased repetition. After obtaining the state of a task, the variable was incremented by 1.

The gather_process_info function simply fetched all the information from the /proc files.

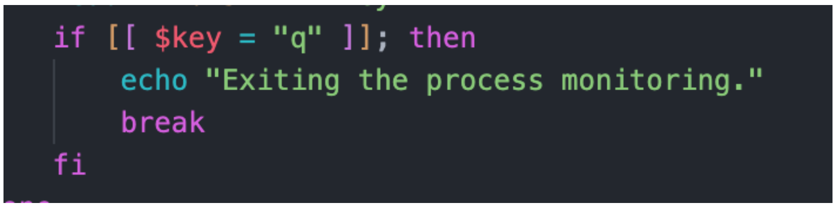


```
smso=$(tput smso)
rmso=$(tput rmso)
printf "%s%-15s %-10s %-10s %-12s %-8s %-8s %-5s %-5s %-7s %-10s %-10s %s%s\n" "$smso" "USER" "PID" "VIRT" "NI" "%CPU" "%MEM" "PR" "
```

To replicate the background of the header, I used 2 variables to store the terminal control sequences for enabling and disabling standout mode and getting a white background. Moreover, I added the feature where once the q key is clicked, the program ends.



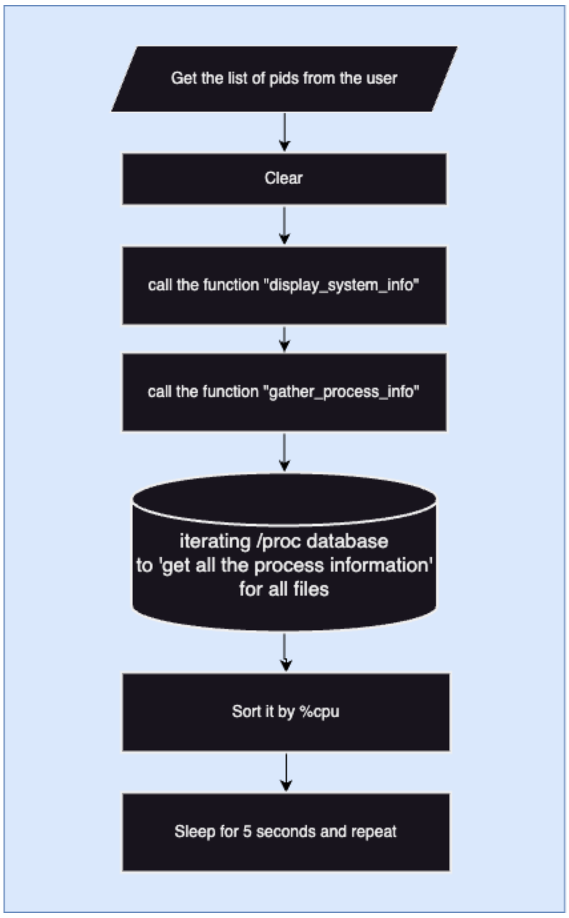
```
printf "%s\n" "$(for pid in /proc/[0-9]*; do gather_process_info "${pid##*/}"; done)" | sort -k5,5nr | head -n 20 | awk '{printf "%-
```



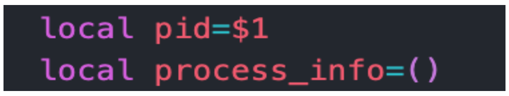
```
if [[ $key = "q" ]]; then
    echo "Exiting the process monitoring."
    break
fi
```

Additionally, I sorted the processes by %CPU, where -k5,5 represented the 5th column, and nr denoted numeric reverse order—hence, displaying the processes in descending order. Moreover, I decided to output only the top 20 processes, as there were many processes listed. When running top, we typically see only the top few processes. By using head -n 20, I ensured that only the top 20 processes were shown, sorted by %CPU.

top -p


Get the list of pids from the user
Clear
call the function "display_system_info"
call the function "gather_process_info"
iterating /proc database to 'get all the process information' for all files
Sort it by %cpu
Sleep for 5 seconds and repeat

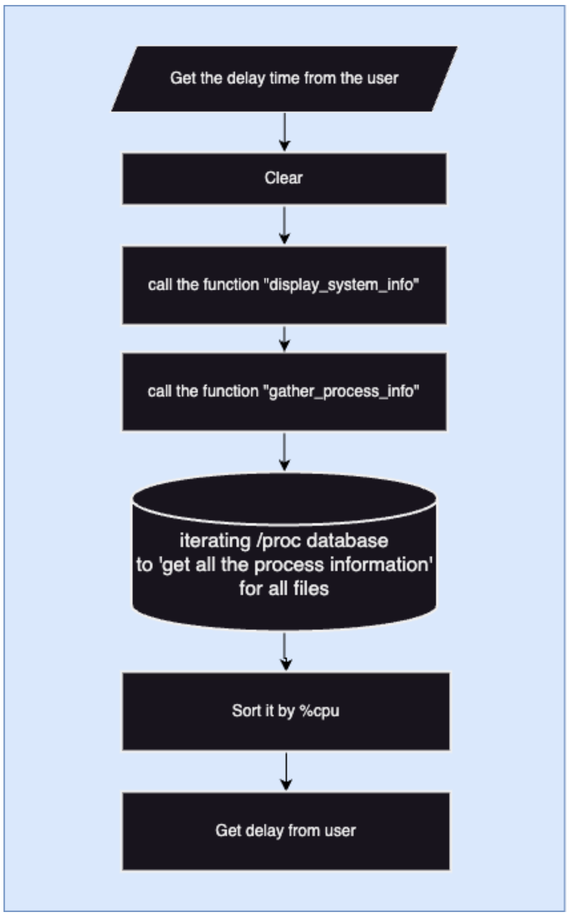
There are not many differences between top and top -p besides that the user enters arguments-which are PID's- to search for. I made use of local variables during the implementation.



```
local pid=$1
local process_info=()
```

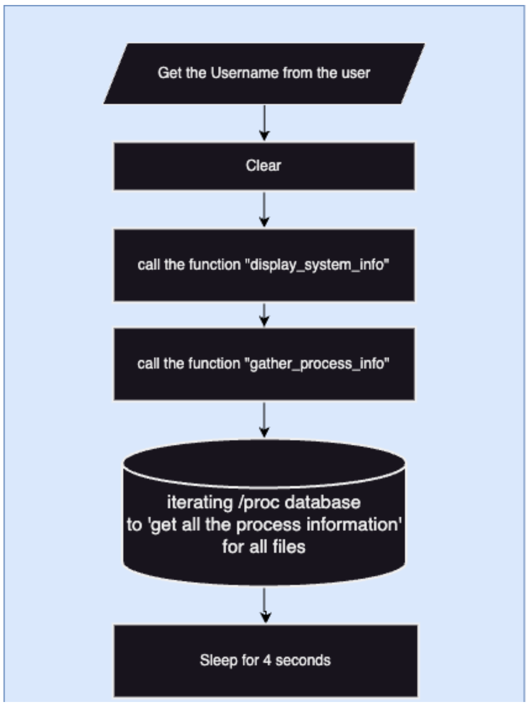
The reason for using this local variable was that I did not need to use it outside of this function. Therefore, I created a local variable specifically for this function.

top -d

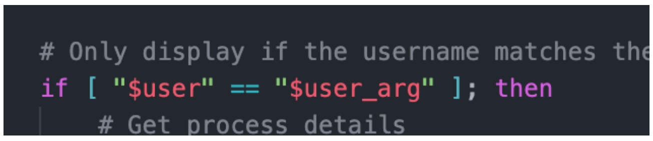

Get the delay time from the user
Clear
call the function "display_system_info"
call the function "gather_process_info"
iterating /proc database to 'get all the process information' for all files
Sort it by %cpu
Get delay from user

There are not many differences in top and top -d as well except that the user enters arguments which is the delay time to extent the refresh time.

top -U

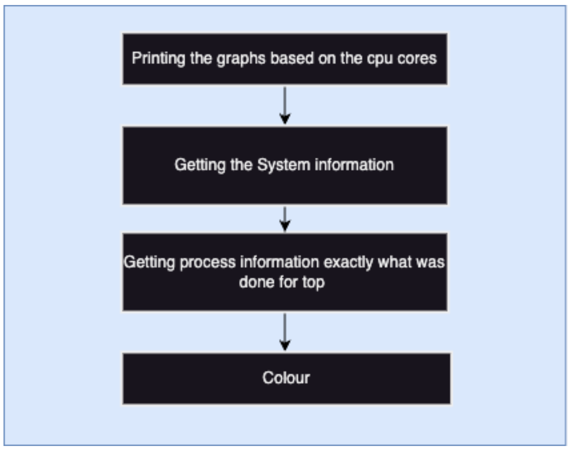

Get the Username from the user
Clear
call the function "display_system_info"
call the function "gather_process_info"
iterating /proc database to 'get all the process information' for all files
Sleep for 4 seconds

There are not many differences in top and top -d as well except that the user enters arguments which is the delay time to extent the refresh time. All I did was filter out the outputs.



```
# Only display if the username matches the
if [ "$user" == "$user_arg" ]; then
    # Get process details
```

htop


Printing the graphs based on the cpu cores
Getting the System information
Getting process information exactly what was done for top
Colour

The function (get_cpu_usage_snapshot) captures two snapshots of CPU statistics from the /proc/stat file, which contains metrics related to CPU usage, and computes the CPU usage percentage for a specified CPU core. It takes an integer specifying the core as an argument and invokes grep to fill an array with raw statistics about time spent by the kernel and other system tasks between the idle, hardware interrupts handled, and CPU usage time spent in user mode, nice mode, and system mode among processes of the specified core. It first takes a snapshot, and after waiting for a user-defined period for practical comparison purposes, it will repeat the procedure to collect another snapshot. Using the elapsed time between both snapshots for each of the metrics captured in the two snapshots, it now calculated total and active CPU time during the interval, yielding the CPU usage percentage.

The print_graph was basically to get a visual representation of the CPU usage percentage calculated by the function get_cpu_usage_snapshot by taking usage percentage as an input parameter. It defines a maximum length of 50 characters for the graphical representation, calculates exactly how many should be printed based on the usage percentage, and fits that within the maximum length. The function assigns colors to the graph based on the usage percentage: blue for low usage (below 20%), green for moderate usage (from 20% to 40%), red for high usage (40% to 60%), and back to blue for very high usage (above 60%). It prints the values using printf, outputting the CPU usage percentage and the colorized bar graph made up of the specified number of character symbols while resetting the terminal color back to default after this output.

The flags and the display_process_info function are the same as top, so I will not be analysing more of that.

Here I defined a stat function which is a replica of the htop stats, something new I learned about was kthreads- also known as kernal threads. Kernal threads are those threads, which have a state other than zombie. To get the number of tasks, I filter out the number of running tasks and store them in a variable.

htop -d and -p

I will not be repeating for -d and -p as they are similar to top and htop.

Testing

I have tested all my scripts, and they work really well. Screenshots have been provided below. I had all my scripts peer tested as well.

ps

Command	PASS/FAIL	Comment
ps	PASS	
ps -a	Slight pass	The gnome-session-b terminals are not visible
ps -u	Slight pass	Didnt include the stat column. The %cpu is not rounded of correctly
ps -p	PASS	

top

Command	PASS/FAIL	Comment
top	PASS	
top -U	pass	
top- d	Slight pass	Due to the number of process id’s it takes more time/
top-p	PASS	

htop

Command	PASS/FAIL	Comment
htop	PASS	The threads are incorrect
top- d	Slight pass	Due to the number of process id’s, it takes more time.
top-p	PASS	

Limitations

There were quite a few limitations in this practical. Firstly, when I ran my script alongside the ps -a command, there were certain outputs I was not receiving, specifically the "gnome-session-b" commands. I tried filtering out those specific commands, only to discover that they did not have a terminal. I checked the /proc file for those specific PIDs as well, but it showed /dev/null. Secondly, to create an htop replica, user interaction was required, and for that, I needed to install libraries like dialog. However, it required sudo access, which I didn’t have, so I couldn’t achieve my goal. Moreover, in the ps -u script, there is a stat column that I was attempting to replicate. Since it was not present in the provided example, I decided to skip it. Additionally, there was a feature called "threads" in htop that I was trying to understand. Despite trying multiple variations, the thread information I obtained was incorrect. Finally, my scripts took too long to load due to the large number of PIDs they had to process, which is a limitation, as it resulted in delays when updating the output.

Conclusion

In conclusion, this practical has deepened my understanding of process management in Linux through the implementation of Bash scripts that mimic some of the standard Unix utilities like ps, top, and htop. It has given me a clearer view of how to manage and monitor processes in a Linux environment by interacting with the pseudo-filesystem /proc. Alongside developing these scripts, I have honed my Bash scripting skills and gained a deeper understanding of data handling and process management principles. I thoroughly enjoyed this practical and learning more about Bash, and I would certainly like to continue exploring it in the future.

References

[1] https://studres.cs.st-andrews.ac.uk/CS1007/

[2] https://www.cyberciti.biz/faq/show-all-running-processes-in-linux/

[3] https://www.javatpoint.com/linux-top

[4] https://www.geeksforgeeks.org/ps-command-in-linux-with-examples/#what-is-a-process-in-linux

[5] https://www.ionos.co.uk/digitalguide/server/tools/htop-the-task-manager-for-linux-mac-os-x-and-bsd/

[6] https://www.cyberciti.biz/faq/check-how-many-cpus-are-there-in-linux-system/

[7] https://superuser.com/questions/818605/interactive-menu-autocomplete-in-bash-with-dialog1

[8] https://tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html

[9] https://man7.org/linux/man-pages/man5/proc.5.html

[10] https://unix.stackexchange.com/questions/224015/memory-usage-of-a-given-process-using-linux-proc-filesystem

[11] https://neoserver.site/help/how-check-memory-utilization-process-linux#:~:text=In%20Linux%2C%20process%20memory%20utilization,stack%20of%20physically%20allocated%20memory.

[12] https://medium.com/@razika28/inside-proc-a-journey-through-linuxs-process-file-system-5362f2414740

[13] I used the man files and unix stackexchange mostly