# Algorithms and Operating Systems

Project Report

## Linux System Monitor

Team001
Siddharth Gaur (20171198)
Ayush Singhania (20171031)
Kunal Lahoti (2018122002)

### Index



### Introduction:

A **system monitor** is a hardware or software component used to monitor system resources and performance in a computer system. For Linux, we most commonly use **top** or **htop** commands for process monitoring and process management.

**htop** is an interactive system-monitor, process-viewer and process-manager. It is designed as an alternative to the Unix program top. It shows a frequently updated list of the processes running on a computer, normally ordered by the amount of CPU usage. Unlike top, htop provides a full list of processes running, instead of the top resource-consuming processes. htop uses color and gives visual information about processor, swap and memory status.

Since htop is written in **C** using **ncurses** library, we try to implement a similar process monitor using **C++** and **ncurses** library.

## Running the Project:

For compiling the files and execution:

1. Run **./compile.sh** this makes a directory, “build” which contains all the files required inorder to run the project.
2. Run **./build/monitor** this displays the system monitor on your terminal.
3. Use **ctrl+c** to stop the system monitor.
4. Run **./clean.sh**, this deletes the “build” directory, hence deleting the executable file, monitor.

## File structure:

Below mentioned is the file structure of the project directory:

**Cpp-System-Monitor-Project**

- /Team1/include
  - /include/format.h
  - /include/linux_parser.h
  - /include/ncurses_display.h
  - /include/process.h
  - /include/processor.h
  - /include/system.h
- /Team1/src
  - /src/format.cpp
  - /src/linux_parser.cpp
  - /src/main.cpp
  - /src/ncurses_display.cpp
  - /src/process.cpp
  - /src/processor.cpp
  - /src/system.cpp
- /Team1/CMakeLists.txt
- /Team1/clean.sh
- /Team1/compile.sh

## /include:

Studying the contents of the include directory, used to declare functions and variables that are used in the main code of the program.

### 1. format.h

Declares a method **ElapsedTime()** , used to convert time in second to HH:MM:SS.

### 2. linux_praser.h

This gets information of different files and directory, used to get the path of various files and directory, which in turn are used to get information of different cpu processes.
Directories/files that are used in this parser are as follows:

- ➔ /proc
- ➔ /cmdline
- ➔ /cpuinfo
- ➔ /status
- ➔ /stat
- ➔ /uptime
- ➔ /meminfo
- ➔ /version
- ➔ /etc/os-release
- ➔ /etc/passwd

Description of these files and directories:

**/proc directory:**

procfs is a pseudo file system that lets users and programs get information from the kernel by reading files. It is usually mounted at /proc/ and to us it looks like a regular directory that we can browse with ls and cd.

**/proc/[pid]/cmdline** will give the command that was used to launch the process.

**/proc/[pid]/status** will give uid, user name, memory used etc for the given process id.

The operating system stores the CPU utilization of a process in the **/proc/[pid]/stat** file. Linux stores memory utilization for the process in /proc/[pid]/status.

**/proc/uptime**, this file contains two numbers: the uptime of the system (seconds), and the amount of time spent in the idle process (seconds).

**/proc/meminfo** gives information about the memory used by the system such as total memory, free memory, cached memory etc.

**/proc/version** stores the Linux version that is installed on the machine.

**/etc/os-release** contains OS identification data.

**/etc/passwd** contains the information regarding user, associated user id etc.

In order to facilitate display, we convert the memory utilization into megabytes.

a. Declared various functions to get various values such as MemoryUtilization(), UpTime(), Pids(), RunningProcesses(), OperatingSystem(), Kernel().
b. We are using an enum(user declared data type) CPUstates to declare various states kUser, kNice, kSystem, kIdle etc.
c. To measure total cpu processing time we are using Jiffies(); and active cpu processing time ActiveJiffies().
d. We are calling methods Command(pid), Ram(pid), Uid(pid), User(pid) and UpTime(pid) for each pid.

**3. ncurses_display.h**

Display functions such as DisplaySystem, DisplayProcesses, ProgressBar are declared here.

**4. process.h**

a. This file makes a Process object for the given pid as its private attribute.
b. It then declares methods like, Pid(), Command(), Ram(), User(), UpTime(), CpuUtilization() and an operator for sorting the processes according to their cpu utilization.
c. Other than Pid() and the operator, all other methods when used in “process.cpp” file, make use of the methods that are already declared in “linux_parser.h” file hence this “process.h” doesn’t require any other attributes other than pid which is used to get all the other related information.

**5. processor.h**

a. Declare a method float Utilization(), which is then used to find the CPU utilization. Cpu utilisation is the fraction of time with respect to that interval that the CPU has been executing instructions corresponding to each of the processes , only running processes are considered ,not those that are waiting ;whether they are in queue or asleep but not interruptible are considered.

**6. system.h**

a. A class, System is declared here which contains different attributes, mentioned below:
    i. A Processor, Cpu(),
    ii. A vector of processes, Processes(), for storing data of different processes,
    iii. float variable, MemoryUtilization(), gives system memory utilization,
    iv. long int variable, UpTime(), gives system uptime in seconds,
    v. int variable, RunningProcesses(), gives the number of total running processes,
    vi. string, Kernel(), which contains the system kernel and
    vii. string, OperatingSystem(), containing the information regarding the os running on the machine.

## /src:

Studying the contents of src directory which contains different files as follows:

### 1. ncurses_display.cpp:

This is the file that is responsible for presenting the data on the terminal to the user. To compile our C/C++ programs using ncurses/curses library we need to include the curses header file <curses.h>. For ncurses, we may include either <curses.h> or <ncurses.h>.

This file uses ncurses library to present our data with a background, in tables, using text of different colors and with a progress bar. The working of this file is explained below:

1. **Display()** method initializes the ncurses library and sets the background color. It also makes two windows, system window and process window which contains information regarding our system and about individual processes respectively. The methods, DisplaySystem() and DisplayProcesses() are also called in this method.
2. **DisplaySystem()** method uses mvwprintw to display different information about the system on the window. mvwprintw is analogous with printf. In effect, the string that would be output by printf is output instead as though waddstr (printing output format in curses window) were used on the given window. All the values of the system such as operating system version, kernel, cpu utilization, memory utilization, running process and uptime are displayed on this window using mvwprintw .
3. **DisplayProcess()** method takes the process vector as input and displays the value of Pid, user name, cpu utilization, Ram usage and command for all the elements of the process vector on the display box.
4. **ProgressBar()** method takes the percentage of various inputs and displays the progress bar depending on the percentage of the input. The maximum being 50 bars which represent 100%. This method is used for displaying the CPU utilization and Memory utilization.

**2. format.cpp:**

ElapseTime method takes input in second and converts time from second to HH:MM:SS format.

**3. process.cpp:**

**a. PID()** method is used to get the value of the process id.

**b. CpuUtilization()** method is used to get the process's CPU utilization.

**c. Command()** method is used to get the command that generated this process.

**d. Ram()** method is used to get the process's memory utilization.

**e. User()** method is used to get the user (name) that generated this process.

**f. UpTime()** method is used to get the age of this process (in seconds).

**g. Lesser operator** method is used to sort the processes according to their cpu utilization.

**4. main.cpp:**

a. Declares an object “system”, of class System.
b. Call the function Display() which was declared in “ncurses_display.h” with “system” as the input.

**5. System.cpp:**

1. System:: Processes() method :- Creates a vector of pids, stores all the system processes data such as Pid, User, command, CpuUtilization, Ram etc and returns a sorted container of above system processes.
2. System:Kernel() methods used to get the identifier of the kernel.
3. MemoryUtilization() method is used to get the systems' memory utilization.
4. OperatingSystem() method is used to get the name of the operating system.
5. RunningProcesses() method is used to get the count of total running processes in the system.

6. UpTime() method used to get the number of seconds since the system started running

**6. Processor.cpp:-**

This contains an Utilization method which calculates the percentage of Cpu utilization which is the ratio of Active Jiffies and total Jiffies.

**7. Linux_parser.cpp:**

This file uses the object, linux_parser.h, that was created in the /include directory. There are a number of functions defined in this file which serve different purposes. The functions and their functionality is listed below:

- **KeyValParser:** This function takes two string arguments “key” and “path”, and then searches for that keyword in the file located at the given path. This function is used in other functions which require data associated with a specific keyword in order to return results.
- **OperatingSystem:** This function doesn’t take any argument; it searches for the os release in the file “/etc/os-release” which was declared in “linux_parser.h” earlier. This returns a string which contains the OS name and its version.
- **Kernel:** Doesn’t take any argument, searches for the kernel name in “/proc/version” file and returns a string containing the same.
- **Pids:** This function returns an integer vector which contains the process ids of all the processes, these process ids are present in the “/proc” directory.
- **CpuUtilization:** This function returns a vector of strings which contains the cpu utilization of all the processes. It gets the relevant data from the “/proc/stat” file and returns it after putting it in a vector.
- **MemoryUtilization:** This returns the total memory that is being used by the system. This looks for relevant data in “/proc/meminfo”, which contains data for total memory and total free memory, so we subtract later from the former to get the total memory used.

- **RunningProcesses:** Just like TotalProcesses this returns the count of processes, but this time these are the processes which are running currently. These are present in the same file, we search for these running processes using the keyword “procs_running”.
- **UpTime:** Returns system uptime in seconds which is stored in “/proc/uptime”.
- **Jiffies:** Returns the number of jiffies for the system. We get these jiffies by summing the count of idle jiffies and active jiffies.
- **IdleJiffies:** Idle jiffies are the sum of idle jiffies and jiffies that represent the processes that are waiting for any type of input/output hence, are not running at the moment. We get idle jiffies by the same function, CpuUtilization as the columns of “/proc/stat” declare different states, such as user, nice, system etc. and idle and iowait are the ones that we need out of these states for calculating the total idle jiffies, we then divide this figure by “sysconf(_sc_clk_tck)”.
- **ActiveJiffies:** We stream “/proc/stat” file and the columns (states) user, nice, system, irq, softirq and steal contribute to the total active jiffies. We divide this figure by “sysconf(_sc_clk_tck)” which is essentially the number of clock ticks per second in our system.
- **ActiveJiffies(pid):** This function returns the number of active jiffies for a PID. This function streams the “/proc/[pid]/stat” file and takes the “utime”, CPU time spent in user code and “stime”, CPU time spent in kernel code, and adds them to give the total active jiffies for the given process id.
- **Uid(pid):** Returns the user id associated with the given process id. It is stored in “/proc/[pid]/status” and we search it by the keyword “Uid:”.
- **User(pid):** Reads and returns the user associated with the given process id, for this we first find the associated user id for the process id using the Uid(pid) function declared above, then we traverse the contents of “/etc/passwd”, if there is a match of our user id and the user id in the line that we are traversing in, we return the user name associated with this user id.

- **Ram(pid):** This reads and returns the memory used by a process by searching the keyword “VmSize:” in the file “/proc/[pid]/status”.
- **UpTime(pid):** Returns the uptime of a process. We stream “/proc/[pid]/stat” and take out the “starttime” which is the time when the process started, we divide this by “sysconf(_sc_clk_tck)” which is essentially the number of clock ticks per second in our system, so this ratio gives us the uptime of a process.
- **Command(pid):** Reads and returns the command associated with the process with process id “pid”. This value is stored in “/proc/[pid]/cmdline” and is returned as a string.

## Build:

We make a file, **CMakeLists.txt**, this file specifies the version of cmake that is required in order to build our project. This file also lets us give a name to our project which in our case is **monitor**. This then looks for the packages that are required for this project and includes its files and directories in our project, such as the ncurses library in our case. Then we include the header files that we have made and are stored in “/include” so that the compiler looks for these files in the specified path.

After this we add a target where we wish to add all the executable files, which in our case is **monitor** and all the files that are present in “/src” are our source files which are compiled. We then link the Curses library with our project so that it is compiled along with our header files, i.e. 3rd party header files also get compiled with user header files.

We have made a file **compile.sh** which when run, makes a directory **build** which will contain all the MakeFile that are required to build a project and a common executable file for complete project. For making the MakeFile, we can either create a MakeFile which contains all the file names which are to be compiled in order to build the project or we can simply run a command **cmake** which itself creates MakeFile. After making the MakeFile which is essential in order to build the project, we simply run the command **make**, this command searches for the makefile available in our project directory and executes it, in turn

compiling all the relevant files and making **one** executable file for whole project which in our case is present at **./build/monitor**.

We also have a file **clean.sh** which when run deletes the **build** directory and cleans our project.

## Challenges :

The output results of our project were in resonance with the output result of the htop function , the display of information on the htop window was comprehensive and interactive. Making our project comprehensive and easy user interface was one of the major challenges that we faced.

This required reading and exploring ncurses library, although Ncurses display provided a lot of the interactive display methods and other functionality and we were able to study and use a lot of them , still some more of the functionality of the ncurses can be explored and the display window could be made more interactive as a path for future improvement.

While displaying the results in columns, number of characters were limited in the row , these errors needed to be sorted out as they were displaying faulty results even though the calculation done by the program were correct.

The cpu uptime calculator was faulty at times and needed certain modification to meet with actual results.

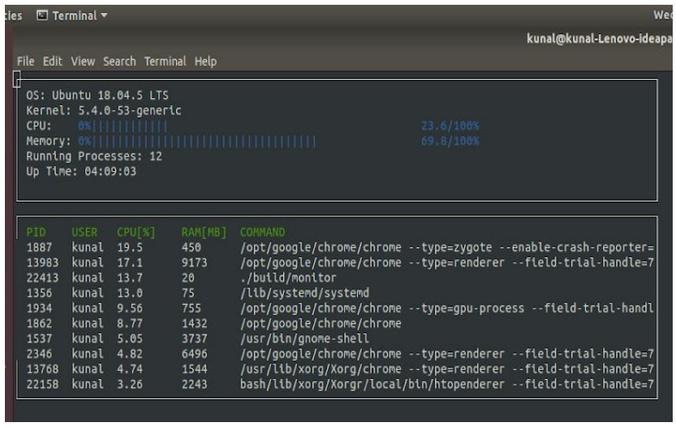
We were required to study documentation of htop to get a better understanding of various calculations being used in it. It helped us a lot as we were able to address the errors in our project.

Htop documentation also helped us in displaying results on the window, many of the compelling ideas such as showing progress bars and other details came from htop documentation.

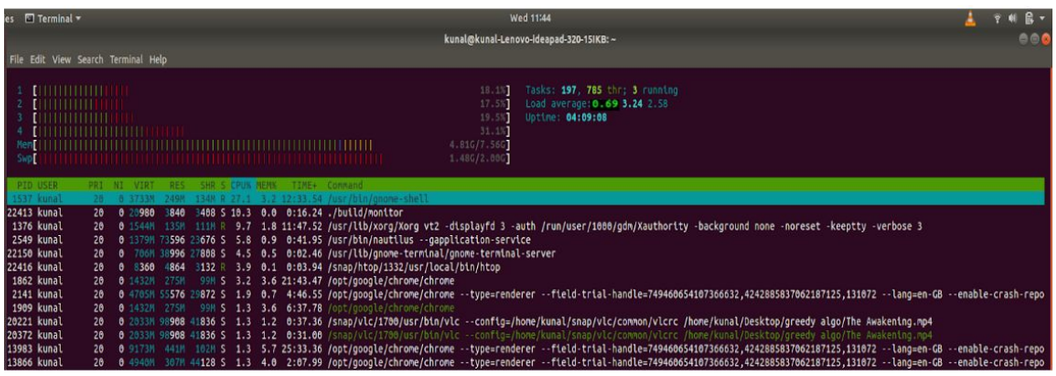
## Results & Conclusion:

Our results are very close to that of htop results but some commands can be made more accurate so that even in the decimal place the result of our code is similar to that of htop.

The results we compare results of the htop and our project




Results of our project




Results of htop

| Value measured | Htop | Our project |
|---|---|---|
| CPU usage | 23% | 20% |
| Memory usage | 69% | 4.8/8 = 60% |
| Running process | 12 | 12 |
| Uptime | 04:09:03 | 04:09:03 |

| | | |
|---|---|---|
| OS name | Ubuntu 18.04..5 | Ubuntu 18.04..5 |
| Kernel Name | 5.4.0-53-generic | 5.4.0-53-generic |

We are using the results of htop to compare the results of our project and verify them we can see that the results of htop are similar to that of our project, the pid value ,user name , memory utilisation, cpu usage ,uptime , command line, running processes are similar in our window to that of the htop window.

The CPU average load value is a metric used to understand the behavior of an operating system, and especially its current and recent past status.

The CPU load value represents in the operating system the average number of jobs (read a set of program instructions in machine language corresponding to a process execution thread) that are running, in runnable state, or, very important, asleep but not interruptible (uninterruptible sleep state). That is to say, to calculate the value of CPU load only the processes that are running or waiting to be assigned CPU time are taken into account. Normal asleep processes (sleep state), zombies, or stopped processes are not considered.

In systems with multiple processors or cores , the meaning of CPU load value varies depending on the number of processors present in the system.

The utilization percentage would represent the fraction of time with respect to that interval that the CPU has been executing instructions corresponding to each of those processes. But for this calculation only running processes, not those that are waiting, whether they are in queue (runnable state) or asleep but not interruptible (for example waiting for the end of an input/output operation) are considered.

So to calculate cpu utilization we can consider the formula: %cpu utilization= Load/cores

All in all, it was a great learning experience and we were able to successfully achieve the desired target results all the while maintaining standard programming practices so that it is easier for us or anyone to add on more features using this project as the base code.

==========