

# Monitoring Memory and CPU Usage for Processes in Linux

Jatczak, David      McNabb, Trent

December 9, 2016

## 1 Introduction

### 1.1 Context

Addressing this issue requires knowledge of what information is relevant to the calculation of memory and CPU (Central Processing Unit) resources, and how to get that information. Specifically, knowledge of the Linux `/proc/` file is needed.

### 1.2 Problem Statement

Computer users who are used to Windows might use Ctrl+Alt+Delete when they wish to see which processes are running and how much of the system's resources those processes are consuming. They may not be comfortable using the terminal to run commands that would display that information, and they might prefer to see the information in a Graphical User Interface (GUI). We would like to write a program that we can make accessible via the Ctrl-Alt-Delete keyboard shortcut that resembles the Windows Task Manager. It should allow for a clear and easy view of which processes are running and what resources each process is using. It should be easy to terminate a running process.

### 1.3 Result

We have written a program that is called on our system using the Ctrl-Alt-Delete keyboard shortcut. The program uses a GUI to display the running

processes and the percentage of CPU resources and memory used by each process. The program allows users to easily select and terminate processes.

## 1.4 Outline

The rest of this report is structured as follows: Section 2 presents background information, including information about calculating system resource usage by specific processes using the Linux `/proc/` file; Section 3 describes in detail the program that we have written, which is the result of this project; Section 4 evaluates the result of the project, primarily through a usability study; Section 5 is the conclusion.

## 2 Background Information

The Linux `/proc/` file system stores information about processes and also other system information. Each process has a directory in the `/proc/` file system[3, p. 792]. The directory is created at `/proc/[pid]/`, where pid is the process identification number. Each of these directories include a `'stat'` file, a `'statm'` file, and a `'status'` file. In addition to the information about each process, there is also system information, i.e. the `/proc/cpuinfo` file and the `/proc/meminfo` file.

The Resident Set Size is the number of pages that the system currently has in real memory. If we multiply that by the page size, that gives the amount of RAM currently used by the process. If we divide that by the total amount of memory used by the system, that will give us the percentage of system memory used by the process. The calculation is:

$$(\text{Resident Set Size} * \text{Page Size}) / (\text{Total System Memory})$$

Information about the resident set size of a process can be found in `'/proc/status/statm'` [2]. Information about the total system memory can be found in `'/proc/meminfo'` [2].

A CPU can either be running or idle. If it is running, it can be running a user space program, or running the kernel [4]. If you add the time it is running the kernel, plus the time it is running a user space program, plus the amount of time it is idle, you get what we will call the Total CPU availability. The percentage of CPU usage by a process is the amount of time that the CPU is running that specific process, divided by the Total CPU availability

multiplied by 100. The percentage of total CPU usage is the amount of usage, divided the total CPU amount, multiplied by 100, where the amount of usage is the time running user space programs plus the time running kernel space programs.

The amount of time that a process has been running in user mode can be found in `/proc/pid/stat`. [2] The total time spent by the CPU running user space processes and the amount of time running the kernel can be found in `/proc/stat` [2].

Shortcuts, also known as hotkeys, that give functionality to applications are typically handled by an event handler. Whenever a user presses a key on an operating system, an event is made and sent out for the event handlers to handle. There are many more types of events than key presses and key releases, such as mouse movements, and GUI events. Event handlers are in event loops. Event loops are embedded in applications themselves and/or in the operating system. Simple processes such as shell tools do not typically have event handlers, but many modern applications do. The typical structure of an event loop is that goes through the queue of events it has, does what it needs and then chooses to dispatch the event or not, afterwards waiting for more events.

The Qt framework offers a library that gives GUI functionality and is based in C++, but it also used in other languages such as Python where it has been ported. The Qt framework has an event loop and receives its events from windowing systems such as X11 or Wayland. If you are interested in this topic, it is explained further by Thiago Macieira in his powerpoint on the Qt event loop[1].

For this text, we use X11 as our window system. X11 is the base of the GUI system for Linux. It handles all the communication between the user input and other windows because it is the framework that other applications are working off. X11 has its own event loop, and is basically at the top of the hierarchy of event loops.

To interact with X11's event loop, you require a library such as Xlib or XCB. XLib was made by the same company that created X11. XCB is a faster alternative to Xlib and is asynchronous. Both of these libraries communicate with X11 via binary because X11 uses a client-server design for its system.

Unity is the desktop environment for Ubuntu, it operates with X11 (by default). Unity is able to use `.desktop` files to launch applications. These `.desk-`

top files are in plaintext and assign key-value pairs such as what command to execute. An option to launch the files upon startup is done by putting them in `~/.config/autostart`. Unity also has custom shortcuts available that can execute commands. The configuration of shortcuts are separate for each user and are located in the user's `/path /.config/dconf`. The configuration files are not plaintext, so the only way to change the configurations by terminal is through the command `'gsettings'`.

### 3 Result

The GUI layout was created in QT Designer, which outputs a `'ui'` file, which is an file in XML (Extensible Markup Language). That `'ui'` file is compiled by PyQt5 into the files that run the GUI. For our program, those files were `'mainwindow.cpp'`, `'mainwindow.h'`, and `'mainwindow.cpp'`, and the python class that allows our program, `'Overseer.py'`, to interface with the gui, `'Umainwindow.py'`. `'Umainwindow.py'` has a class called `Ui_MainWindow` that allows the creation of `Ui_MainWindow` objects. These files are all generated by the PyQt5 compiler, and any changes in them are made by changing and recompiling the `'ui'` file. When the main function is run in `'Overseer.py'`, it creates an `Ui_MainWindow` object, runs the `OverseerMainWindow` function and tells the `Ui_MainWindow` object to show on the screen.

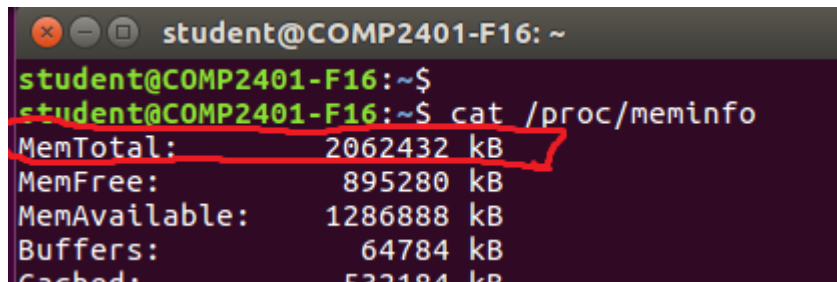
The `OverseerMainWindow` function creates the the model for the table that holds the data so that the UI can display it. It creates an instance of the `Proc` class from `'proc.py'`. The `OverseerMainWindow` function calls the `update` function, which updates the data by calling the `readData` function on the `Proc` object and then updates the view by calling the `updateView` function. It repeats this once every second.

The `Proc` object stores the following data: a dictionary called `processList`, a `UserList` object called `userList`, from `'userlist.py'`, an integer called `totalMem`, and two lists called `cpu` and `openWindows`.

The `readData` function runs the functions that populate each of those data members. `readTotalMem` gets the total memory usage from `'/proc/meminfo/'`. See Figure 1. The circled data is the data stored as `totalMem` and the `freeMem` in the `Proc` object.

`readCupTimes` gets the CPU infomation from `'/proc/stat'`. See Figure 2. The circled data are the numbers put in the list `cpu`. The circled data are the numbers put in the list `cpu`. This in the combined data for all processors

### Total Memory

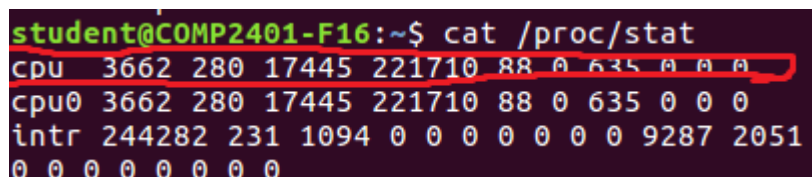


```
student@COMP2401-F16: ~  
student@COMP2401-F16:~$  
student@COMP2401-F16:~$ cat /proc/meminfo  
MemTotal:      2062432 kB  
MemFree:       895280 kB  
MemAvailable:  1286888 kB  
Buffers:       64784 kB  
Cached:       532184 kB
```

Figure 1: Output of 'cat /proc/meminfo'

in the system. The first number is the time spent in user mode, and it is stored in `cpu[0]` The third number is the time spent in system mode, and it is stored in `cpu[3]` The fourth number is the time spent in idle, and it is stored in `cpu[4]`. None of the other numbers are relevant.

### CPU Information



```
student@COMP2401-F16:~$ cat /proc/stat  
cpu 3662 280 17445 221710 88 0 635 0 0 0  
cpu0 3662 280 17445 221710 88 0 635 0 0 0  
intr 244282 231 1094 0 0 0 0 0 0 0 9287 2051  
0 0 0 0 0 0 0 0
```

Figure 2: Output of 'cat /proc/stat'

Overseer is able to calculate its CPU usage and RAM usage through polling. The interval for polling is every second. The data that is used for polling is describing running processes and the operating system and is stored on `/proc`. Reading these files may seem like a strain, but there should not be any overhead from IO blocking since these files in `/proc` are virtual files and not actual files on a disk. Virtual files are stored only on RAM(Random-access Memory), also known as the main memory. In theory you should be able to read all the info from `/proc` almost instantly. Thus having 1 second of polling should not be too bad for a single threaded program.

When all this data is collected from `/proc`, we also perform our CPU and RAM usage calculations as well as collecting a list of users. The list of users is collected at the start of the poll cycle. Collecting a list of users gives overhead as there is possible IO blocking from reading `/etc/passwd` because

this is not a virtual file but an actual file on the disk. Information on open windows in the window manager is collected through `wmctrl` and might give overhead as well depending on how `wmctrl` operates; window information is also collected in each poll cycle.

The original goal was to have Overseer launch upon startup and it would sit in the system tray, where it would show the window when the shortcuts *Ctrl+Alt+Delete* or *Ctrl+Shift+Escape* was pressed. Overseer would have used an event handler for detecting the shortcuts (key press events). If a key release event was detected for either key in a shortcut before all 3 keys having a key press event, the check for the shortcut would fail. Implementation for putting a `.desktop` file in `~/.config/autostart` did manage to get the program to launch on startup. But if the user were to log out and log back in, it would not launch again.

However, the main crux is the issue of reading events from X11 within the main thread of the application, which Qt operated off. The Qt framework only provided ways to have working shortcuts if the Qt application itself is in focus, meaning it is unable to read key presses when the Qt application is minimized or selected to receive user input. So another library was needed to read key press events, which led to Xlib.

Utilizing Xlib to read key presses with Python 3.5.2 is possible. There is an example at `/usr/share/doc/python3-xlib/examples/record_demo.py` that illustrates reading events. However, due to our lack of knowledge with Xlib and the poor documentation [5] for the Python version of Xlib, the event loop could not be changed to our liking.

Essentially the call back loop handles all the events coming from X11, then keeps calling itself to handle more events. This call back loop was not possible to implement in our application's main thread because Qt operates off the main thread. For example if `time.sleep(5)` were to be executed in the main thread, the GUI would stop working all together and look frozen for 5 seconds. This call back loop was the only thing executing so Qt did not operate.

It was thought it would be possible to implement the Xlib call back loop in another thread, a `QThread`, where this 2nd thread would check for key press events with its callback loop while the main Qt GUI thread would run. However, it is incredibly dangerous when Xlib is used out of the main thread, as was explained to us by the Qt Core Maintainer Thiago Macieira over IRC. Also to add, multithreading is not a wise idea with Qt in general. The reason is you could have more overhead to Qt's main event loop if your

threads were sending more events to the main thread's event loop. This is explained further by Qt's web page[6].

Thus XCB was turned to instead, as another way to read events. A derived class of `QAbstractNativeEventFilter`, a class in the Qt framework, was created to handle the events coming from X11. The type `xcb_generic_event_t` is sent to `QAbstractNativeEventFilter` by Qt. Unfortunately, Python 3.5.2 does not recognize that type of data, and the PyQt5 library does not have an implementation of this data type either. An XCB library was needed to understand the `xcb_generic_event_t` data.

Due to our entire application being programmed in Python 3.5.2, there are no decent libraries of XCB available. Most libraries for XCB on Python are in the 2.x version of Python, e.g. <https://xcb.freedesktop.org/XcbPythonBinding/>. Search engine results led to other XCB implementations in Python 3.x such as <https://github.com/samurai-x/samurai-x/tree/master/pyxcb/pyxcb> or <https://github.com/tych0/xcffib>. Both were unsuccessful for deciphering key press events from `xcb_generic_event_t` data. There was some progress in deciphering the data, but there are many missing functions and constant values such as `XCB_MOUSE_PRESS`. The work can be seen from <http://github.com/Davidj361/Comp3000Group5/tree/global-shortcuts> and <http://github.com/Davidj361/Comp3000Group5/tree/global-shortcuts-method2>

Finally shortcut functionality was achieved with Unity's custom shortcuts. Our Overseer does not deal with any events in the code, it merely interacts with Unity's custom shortcuts system. Our application does not launch no longer on startup with a `.desktop` file, but instead launches from a bash file when a shortcut is pressed which in turn opens our Overseer application. *Ctrl+Alt+Delete* or *Ctrl+Shift+Escape* Unfortunately the *Ctrl+Shift+Escape* does not work. The reason is because Ubuntu's shortcut manager does not work with *Escape* as part of a shortcut, as illustrated in this bug report <https://bugs.launchpad.net/ubuntu/+source/compiz/+bug/158855>.

The applications tab in Overseer utilizes `wmctrl`, a 3rd party tool, to interact with windows in the desktop environment. It collects information on windows showing in desktop environment and also gives the functionality for switching to a window. Unfortunately `wmctrl` also collects information on junk windows such as 'Hud', 'unity-dash', 'unity-panel', 'unity-launcher', and 'XdndCollectionWindowImp', even if it's not on the desktop environment's taskbar.

Selections on the *QtTableViews* in the process tab and applications tab

has issues because Python does not have pointers. It is hard to utilize Model View Controller design, which is the basis of *QtTableView*, because we cannot link our data straight from Overseer's Proc class as a model to a *QtTableView*. Instead, there's a poll cycle and new data is grabbed, all of the entries in each *QtTableView* is erased and reinserted. However, this gets rid of our selection every poll cycle. So Overseer remembers a *PID* if there was a selection before the erase and reselects by iterating over the entire list in the *QtTableView* looking for a matching *PID*, otherwise leaves no selection if not found. Sadly this adds overhead to the main thread.

## 4 Evaluation

Evaluation goes here.

## 5 Conclusion

### 5.1 Summary

Summary and highlights.

### 5.2 Relevance

Relevance with respect to the course topic.

### 5.3 Future Work

Question for TA – future work in the field of this project, or ways that the project application itself could be continued on.

## Contributions of Team Members

## References

- [1] Thiago Macieira - Qt Core Maintainer [http://github.com/boostcon/cppnow\\_presentations\\_2013/blob/master/mon/qt\\_event\\_loop.pdf?raw=true](http://github.com/boostcon/cppnow_presentations_2013/blob/master/mon/qt_event_loop.pdf?raw=true)



- [2] Proc(5) - Linux Manual Page <http://man7.org/linux/man-pages/man5/proc.5.html>
- [3] Tanenbaum, A. S. (2015). *Modern operating systems*.
- [4] Understanding Linux CPU stats <http://blog.scoutapp.com/articles/2015/02/24/understanding-linuxs-cpu-stats>
- [5] [/usr/share/doc/python3-xlib/html](#) & <http://python-xlib.sourceforge.net/doc/html/index.html>
- [6] [https://wiki.qt.io/Threads\\_Events\\_QObjects](https://wiki.qt.io/Threads_Events_QObjects)