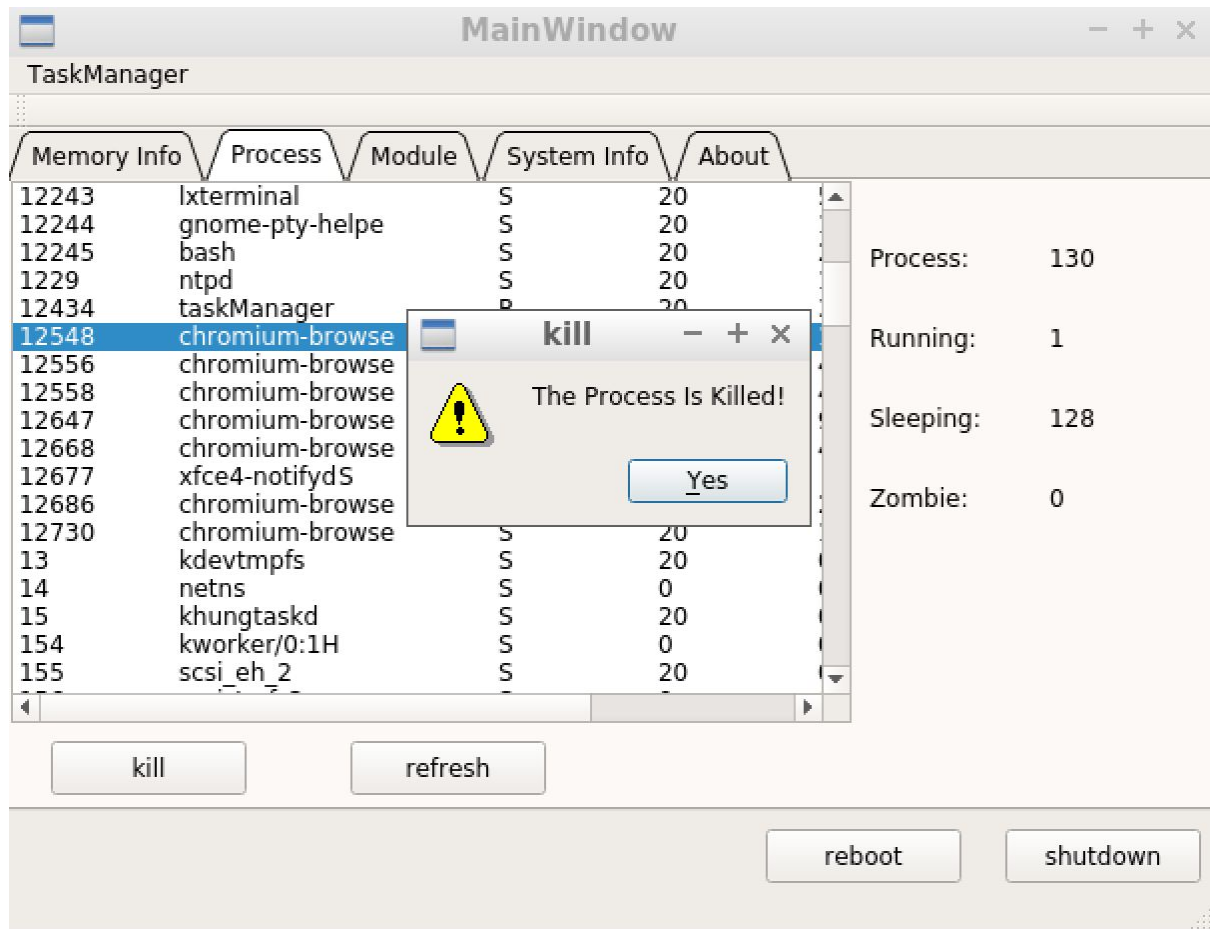


# Linux Task Manager



TianCheng Zhao

Yinfeng Hu  
100981891

April 11, 2018

# Table of Contents

1. Introduction .....	3
1.1 Context.....	3
1.2 Problem Statement.....	3
1.3 Result.....	3
1.4 Outline .....	3
2. Background Information .....	4
3. Result.....	4
4. Evaluation and Quality Assurance	
5. Conclusion .....	8
5.1 Summary	
5.2 Relevance	
5.3 Future Work	
6. Contributions of Team Members	
.....	8
7.References.....	

# 1 Introduction

## 1.1 Context

When was the first programmable computer invented? During World War II. Computer at that time was used to solving the problem like missile route calculation, decode, decryption etc. The appearance of computers in World War II make industries realize that computers calculate much more efficiently and effectively than human beings. With technology speeding up after World War II, computers become more compact. And with lower production costs, computers successfully enter the industry from the military field. As industry practitioners' further contact with computers, they recognized computers can not only calculate, but also has many interesting features including office and other human-interactive function. With the further reduction of the computers' size and cost, computers are brought into consumers' level. People found that computers can also brought entertainment. We can use computers to play video games or watching movie online. These basically conclude the three stages(Military, Industry, Consumer) of computers' history.

Operating system is the main part acts like a brain in programmable computers. It creates a environment allows IO(Input, Output), human-interactive etc. The commonly used operating systems these days like Windows, Mac os usually come with a task manager. These task managers have the same feature, using a user friendly graphical interface allow users to manage the tasks which under processing efficiently and effectively. Most task manager allows users to kill or limit the tasks, and shows the information about CPU load and memory usage.

## 1.2 Problem Statement

There exists several operating systems developed based on linux kernel. For example, Ubuntu, CentOS, Fedora, Debian etc. A large amount of these linux-based operating systems have feature which they specialize in. But some of the basic feature like the built-in task manager are very basic or with GUI(Graphical User Interface) which are already out-of-date. Most of the developers are still using the command line to interact with the task manager function.(*Figure 1*) The command line is reliable but reduce the efficiency at the same time. The GUI with scroll & click is in some way more Intuitive than the traditional command line interface. This is a major concern when common users or developers using these operating systems' environment.

```
student@ldnel: /proc
File Edit Tabs Help
top - 19:26:36 up 25 min, 1 user, load average: 0.03, 0.04, 0.06
Tasks: 133 total, 1 running, 132 sleeping, 0 stopped, 0 zombie
%Cpu(s): 3.7 us, 0.3 sy, 0.0 ni, 96.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 2045948 total, 1322560 free, 195696 used, 527692 buff/cache
KiB Swap: 4194300 total, 4194300 free, 0 used. 1658496 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM     TIME+ COMMAND
    1 root        20   0  119640   5924  4140 S   0.0   0.3   0:00.95 systemd
    2 root        20   0       0       0       0 S   0.0   0.0   0:00.00 kthreadd
    4 root         0 -20       0       0       0 S   0.0   0.0   0:00.00 kworker/0:0H
    6 root        20   0       0       0       0 S   0.0   0.0   0:00.07 ksoftirqd/0
    7 root        20   0       0       0       0 S   0.0   0.0   0:00.21 rcu_sched
    8 root        20   0       0       0       0 S   0.0   0.0   0:00.00 rcu_bh
    9 root        rt    0       0       0       0 S   0.0   0.0   0:00.00 migration/0
   10 root         0 -20       0       0       0 S   0.0   0.0   0:00.00 lru-add-drain
   11 root        rt    0       0       0       0 S   0.0   0.0   0:00.00 watchdog/0
   12 root        20   0       0       0       0 S   0.0   0.0   0:00.00 cpuhp/0
   13 root        20   0       0       0       0 S   0.0   0.0   0:00.00 kdevtmpfs
   14 root         0 -20       0       0       0 S   0.0   0.0   0:00.00 netns
   15 root        20   0       0       0       0 S   0.0   0.0   0:00.00 khungtaskd
   16 root        20   0       0       0       0 S   0.0   0.0   0:00.00 oom_reaper
   17 root         0 -20       0       0       0 S   0.0   0.0   0:00.00 writeback
   18 root        20   0       0       0       0 S   0.0   0.0   0:00.00 kcompactd0
   19 root        25   5       0       0       0 S   0.0   0.0   0:00.00 ksmd
   20 root        39  19       0       0       0 S   0.0   0.0   0:00.01 khugepaged
   21 root         0 -20       0       0       0 S   0.0   0.0   0:00.00 crypto
```

**Figure 1:** traditional taskmanager by command \$top

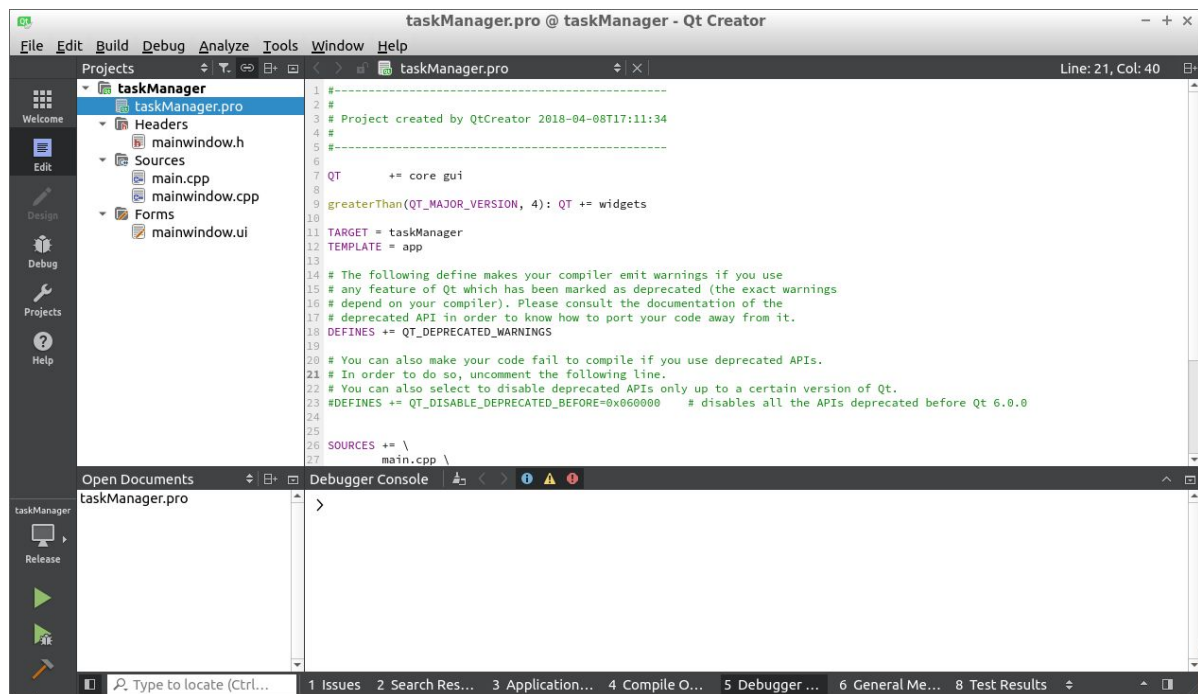
Our goal is to design a better task manager, which is compatible with linux based systems. The software has a user-friendly graphical interface to show CPU load, memory usage, processes' information, modules' information, hardwares' information etc. The software allows users to install modules or start/kill the processes. The software developed based on a principle, which must guarantee the efficiency and effective related to the traditional built-in task managers. The software which has similar feature as the tradition built-in module and doesn't work reliable or efficient, effective will eventually obsolete in some way.

### 1.3 Result

We decided to build the task manager based on Qt which is a cross-platform application framework with intuitive user interface. We want the task manager works efficiently and effectively. The GUI of the task manager may looks similar as the task managers in Windows operating system. Somehow it may reduce the learning cost for the users move from Windows to Linux based systems. And it should bring a simple intuitive user-friendly graphical interface to users.

It is necessary to introduce the Qt framework we use to develop the software briefly. Qt is a cross-platform application framework with intuitive user interface.(Figure 2) The framework can be easily downloaded from its website(<https://www.qt.io/>). Setting up is pretty easy: gives permission(chmod +x ./\*.run), run the installation file(./\*.run) and following the installation steps on screen. Qt is great for us who are the beginners of developing software

interacting with operating system kernel stuff.



**Figure 2: Interface of Qt framework**

## 1.4 Outline

The rest of this report is structured as follows. Section 2 provides more details on the background of this software product. Section 3 providing more information and explain the result with screenshots. The result is evaluated in Section 4 with explanations and data analysis . We conclude with Section 5 with summarizes of the document. Section 6 provides the contributions of each team member.

# 2 Background Information

We divide the feature of our task manager into five main parts: CPU & Memory, Processes, Module, System Information, About. Each feature can be reached by clicking on corresponding tabs on the graphical interface. Details design will be introduced in the following paragraphs.

Linux kernel has already provided a mechanism to access internal kernel data structure and modify kernel runtime parameters through the /proc file system. The /proc is actually a pseudo file system as depicted in *Figure 3*. It exists only in RAM(Random Access Memory) and does not occupy any external storage space. It provides an interface for accessing operating system kernel data as a file system. Each process currently running in the system has a corresponding directory under /proc. The directory is named by the

PID(Process Identifier) number of the process. The directories act as the interface for reading process detail information. And the 'self' directory is the link of the information interface of the read process itself.

```

student@ldnel:~/Desktop$ cd /proc
student@ldnel:/proc$ ls
1      1267  1441  214  33   76   95      kcore      softirqs
10     1279  1446  22   34   77   acpi     keys       stat
1035   1284  1452  23   35   773   asound   key-users  swaps
1050   1295  1457  24   36   774   buddyinfo kmsg       sys
1061   1299  1463  248  3889 78   bus      kpagecgroup sysrq-trigger
11     13    1470  25   4    783   cgroups  kpagecount sysvipc
1120   1300  15    255  433  785   cmdline  kpageflags thread-self
1151   1305  16    26   471  79   consoles loadavg     timer_list
1152   1310  165   260  472  8    cpuinfo  locks       timer_stats
1155   1312  168   2621 481  80   crypto   mdstat      tty
116    1316  17    2676 5    81   devices  meminfo     uptime
117    1322  170   2682 6    82   diskstats misc         version
1170   1335  171   2683 604  83   dma      modules     version_signature
1171   1337  172   269  609  84   driver   mounts       vmallocinfo
1181   1378  173   27   616  841  execdomains mtrr         vmstat
1182   1381  18    278  617  85   fb        net          zoneinfo
1192   1389  186   28   622  86   filesystems pagetypeinfo
1193   1390  187   29   623  87   fs        partitions
1198   1394  19    3    629  88   interrupts sched_debug
1199   14    2    30   640  89   iomem     schedstat
12     1400  20    319  642  9    ioports   scsi
1263   1417  21    32   698  90   irq       self
1266   1423  213   320  7    91   kallsyms  slabinfo

```

**Figure 3:** /proc directory in linux system

Users and applications can read the system information by /proc file system, and modify some of the kernel parameters. As the system information changes dynamically, /proc file system reads the required information from the system kernel dynamically and submits it when a user or application reads a /proc file.

## CPU & Memory

In Linux/Unix based operating systems, CPU utilization can be divided into three mode: user mode, system mode and idle mode. They indicate the time when the CPU is in user mode, the time when the system kernel is executed and the time when idle system processes are executed. The CPU utilization in the task manager generally refers to the formula:

$$CPU\ utilization = \frac{CPU\ execution\ time\ of\ non-system\ idle\ process}{CPU\ total\ execution\ time}$$

In the Linux kernel, there is a global variable: Jiffies, which represents time. The unit of jiffies varies with different hardware platform. The system defines a constant: Hz, which represents the number of minimum intervals per second. In this way, the unit of jiffies is 1/Hz. The unit of jiffies on Intel platform is 1/100 second. This is the smallest time interval the system can distinguish. So the detail calculation of CPU utilization will be the formula:

$$CPU\ utilization = \frac{Jiffies\ executed\ in\ user\ mode + Jiffies\ executed\ in\ system\ mode}{Total\ jiffies}$$

The CPU load is calculated in another way. CPU load is determined by the length of processes queue rather than CPU utilization. When the host system is overloaded, its CPU utilization will be close to 100%, which can not show the CPU load status accurately, and only the length of processes queue can reflect the CPU load directly. For example, there are two systems in runtime. One has 3 processes in queue, and the other has 6 processes in queue. If the CPU utilization is used to indicate the CPU load level, they may both be close to 100 percent. But using length of CPU queue, the result of CPU load will be completely different. In Linux/Unix, CPU load can be determined as the CPU average load. In most multi-processor system, CPU load average is based on number of cores.

In Linux, there is no direct way to get CPU utilization and memory usage through system call. But we can use the /proc pseudo file to implement the feature. Directory /proc/stat includes the CPU status in real time. If we type command: cat /proc/stat, we will see the output like the contents in *Figure 4*. The parameters of “cpu” line indicates the CPU execution time in user mode, nice(scheduling priority), system mode, idle, iowait, irq(interrupt request), softirq. CPU total execution time is the sum of all these execution time.

```
student@ldnel: ~  
File Edit Tabs Help  
student@ldnel:~$ cat /proc/stat  
cpu 584 0 433 1909 99 0 22 0 0 0  
cpu0 584 0 433 1909 99 0 22 0 0 0  
intr 16169 31 59 0 0 0 0 0 0 0 0 0 0 180 0 0 116 35 0 0 285 2231 6896 25 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
cxtx 547274  
btime 1523586684  
processes 2278  
procs_running 3  
procs_blocked 0  
softirq 27245 2 3289 7 360 6380 0 95 0 0 17112  
student@ldnel:~$
```

**Figure 4:** /proc/stat pseudo file indicate the CPU stat info in real time

RAM is the random access memory, which stores the data but will lose them after the system shut down. That is one of the reason that we can not let all the data store in RAM. As



we can not store all the data in RAM, some of extra data and procedures can not always be occupied in RAM. When there is no more available space in RAM, it is necessary to remove procedures which are not frequently used in RAM. Here the SWAP partition comes up. When RAM is close to full, the procedures which are removed from RAM will be stored in SWAP partition temporarily. And when you need the procedure later, it will be reloaded from SWAP to RAM, otherwise, the procedure will not be actively swapped into RAM.

## Processes

Under `/proc` directory, there exists several directory named by number. They are the processes directory, and named by each process ID. *Figure 3* indicates the `/proc` pseudo file system. We can see that there is a lot of PID named directories on the left side of the command line window, which also colored in dark blue. We can find several detail pseudo files under each process directory. In *Figure 7*, we access the PID 1467 as an example. All the files under the PID directory are temporarily stored, and information in each file may change in real time. The contents of the files are listed in the form below.

<code>/proc/pid/cmdline</code>	process startup command
<code>/proc/pid/cwd</code>	link to current process working directory
<code>/proc/pid/envIRON</code>	list of process environment parameter
<code>/proc/pid/exe</code>	link to process execution command file
<code>/proc/pid/fd</code>	all file descriptors associated with the process
<code>/proc/pid/maps</code>	process-related memory mapping information
<code>/proc/pid/mem</code>	memory held by the process, not readable
<code>/proc/pid/root</code>	link to the root directory of the process
<code>/proc/pid/stat</code>	status of the process
<code>/proc/pid/statm</code>	state of the memory used by the process
<code>/proc/pid/status</code>	more state and status information of the process
<code>/proc/pid/self</code>	link to the current running process

What we need the most important is the `stat` file which contains mainly all the information we need. As we see in *Figure 7*, `stat` file includes the process name, state of the



process, memory used by the process etc, but all expressed in numbers. We searched online and found out each expression of the numbers separated by space.

```

student@ldnel: /proc/1467
File Edit Tabs Help
12 1436 21399 33 77 fb net
1236 1439 22 332 78 filesystems pagetypeinfo
1239 1451 2205 34 79 fs partitions
1240 1456 2208 4 8 interrupts sched_debug
student@ldnel:/proc$ cd 1467
student@ldnel:/proc/1467$ sudo ls
[sudo] password for student:
attr          cwd          map_files    oom_adj       schedstat     task
autogroup     environ     maps         oom_score     sessionid     timers
auxv          exe         mem          oom_score_adj setgroups     timerslack_ns
cgroup        fd          mountinfo    pagemap       smaps         uid_map
clear_refs    fdinfo      mounts       patch_state    stack         wchan
cmdline       gid_map     mountstats   personality     stat
comm          io          net          projid_map     statm
coredump_filter limits      ns           root           status
cpuset        loginuid    numa_maps    sched          syscall
student@ldnel:/proc/1467$ cat statm
64882 1114 1015 18 0 6390 0
student@ldnel:/proc/1467$ cat stat
1467 (gvfs-mtp-volume) S 1 1240 1240 0 -1 4194304 269 0 1 0 0 0 0 20 0 3 0 185
3 265756672 1114 18446744073709551615 4194304 4267020 140723959174400 0 0 0 0 40
96 0 0 0 17 0 0 11 0 0 6367464 6379296 33665024 140723959179693 140723959179
731 140723959179731 140723959181266 0
student@ldnel:/proc/1467$

```

**Figure 7:** /proc/1467, access PID 1467 process file.

*statm* shows the memory the process used.

*status* shows more info about the process.

## Module

A LKM(loadable kernel module) is an object file that contains code to extend the running kernel of an operating system. LKMs are typically used to add support for new hardware (as device drivers) or file systems, or for adding system calls. When the functionality provided by a LKM is no longer required, it can be unloaded in order to free memory and other resources.

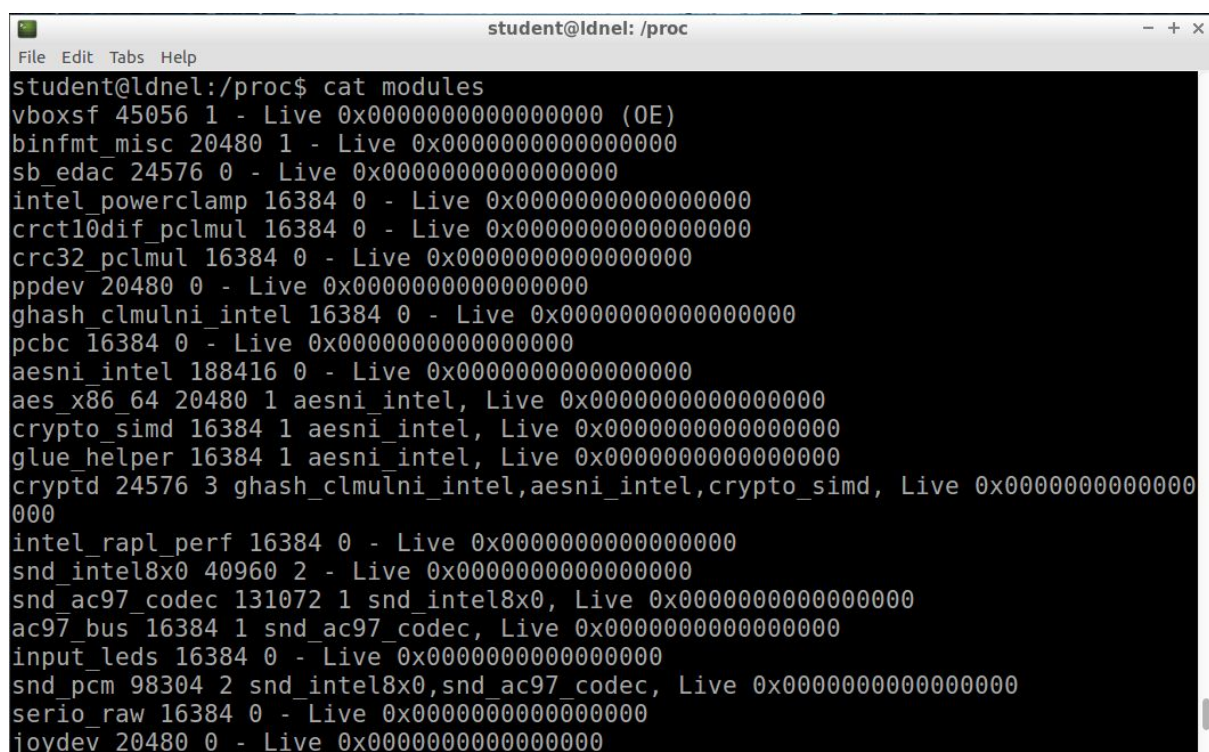
The Linux kernel framework is huge. If you compile all the required functions into the Linux kernel, there will be two problems. Firstly, the generated kernel is very large. Secondly, adding or deleting functions requires recompilation of the kernel. In order to solve the above problems, the LKM is introduced so that the compiled kernel does not need to contain all the functions, and when these functions need to be used, the corresponding code is loaded and loaded into the kernel dynamically.

The latest Linux distribution kernel takes only a relatively small "built-in modules", and the rest of the specific hardware drivers or custom functions are used as "loadable modules" to allow you to load or uninstall them selectively. The built-in modules are statically compiled into the kernel. Unlike loadable kernel modules that can dynamically load, uninstall,

and query modules, such as modprobe, insmod, rmmod, modinfo, or lsmod, the built-in modules are always loaded into the kernel at startup and will not be managed by these commands.

In order to look up the parameters of built-in modules and their values, you can check their contents under /sys/module directory. In the /sys/module directory, The information of all kernel modules, including built-in and loadable, subdirectories. In every module directory. There is a parameters directory, which lists all parameters of this module.

The file /proc/modules displays a list of all modules loaded into the kernel. The contents vary based on the configuration and use of different systems. Here is a sample /proc/modules file output:



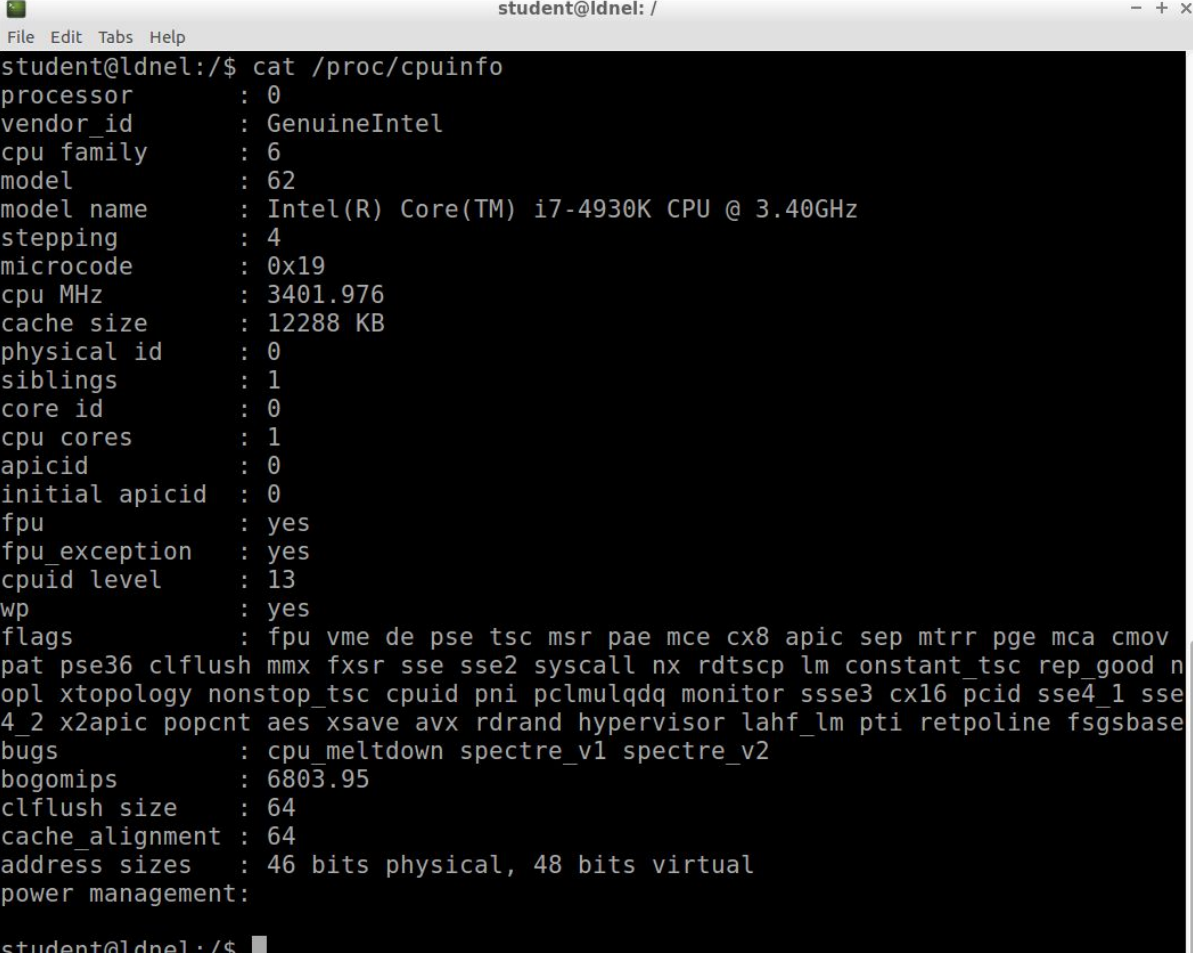
```
student@ldnel:/proc$ cat modules
vboxsf 45056 1 - Live 0x0000000000000000 (OE)
binfmt_misc 20480 1 - Live 0x0000000000000000
sb_edac 24576 0 - Live 0x0000000000000000
intel_powerclamp 16384 0 - Live 0x0000000000000000
crct10dif_pclmul 16384 0 - Live 0x0000000000000000
crc32_pclmul 16384 0 - Live 0x0000000000000000
ppdev 20480 0 - Live 0x0000000000000000
ghash_clmulni_intel 16384 0 - Live 0x0000000000000000
pcbc 16384 0 - Live 0x0000000000000000
aesni_intel 188416 0 - Live 0x0000000000000000
aes_x86_64 20480 1 aesni_intel, Live 0x0000000000000000
crypto_simd 16384 1 aesni_intel, Live 0x0000000000000000
glue_helper 16384 1 aesni_intel, Live 0x0000000000000000
cryptd 24576 3 ghash_clmulni_intel,aesni_intel,crypto_simd, Live 0x0000000000000000
intel_rapl_perf 16384 0 - Live 0x0000000000000000
snd_intel8x0 40960 2 - Live 0x0000000000000000
snd_ac97_codec 131072 1 snd_intel8x0, Live 0x0000000000000000
ac97_bus 16384 1 snd_ac97_codec, Live 0x0000000000000000
input_leds 16384 0 - Live 0x0000000000000000
snd_pcm 98304 2 snd_intel8x0,snd_ac97_codec, Live 0x0000000000000000
serio_raw 16384 0 - Live 0x0000000000000000
joydev 20480 0 - Live 0x0000000000000000
```

**Figure 9:** Sample /proc/modules file output

The first column is the name of the module. The second column shows the memory size of the module, in bytes. The third column lists how many times of the module are currently loaded. A value of zero represents an unloaded module. The fourth column states if the module depends upon another module to be present in order to function, and lists those other modules. The fifth column lists what load state the module is in: Live, Loading, or Unloading are the only possible values. The sixth column lists the current kernel memory offset for the loaded module.

## System Information

This `/proc/cpuinfo` file identifies the type of processor used by your system. Here is a sample `/proc/cpuinfo` file output:



```
student@ldnel:/$ cat /proc/cpuinfo
processor       : 0
vendor_id     : GenuineIntel
cpu family    : 6
model         : 62
model name    : Intel(R) Core(TM) i7-4930K CPU @ 3.40GHz
stepping      : 4
microcode     : 0x19
cpu MHz       : 3401.976
cache size    : 12288 KB
physical id   : 0
siblings      : 1
core id       : 0
cpu cores     : 1
apicid        : 0
initial apicid : 0
fpu           : yes
fpu_exception : yes
cpuid level   : 13
wp            : yes
flags         : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush mmx fxsr sse sse2 syscall nx rdtscp lm constant_tsc rep_good n
opl xtopology nonstop_tsc cpuid pni pclmulqdq monitor ssse3 cx16 pcid sse4_1 sse
4_2 x2apic popcnt aes xsave avx rdrand hypervisor lahf_lm pti retpoline fsgsbase
bugs          : cpu_meltdown spectre_v1 spectre_v2
bogomips      : 6803.95
clflush size  : 64
cache_alignm  : 64
address sizes : 46 bits physical, 48 bits virtual
power managem :
student@ldnel:/$
```

**Figure 10:** Sample `/proc/cpuinfo` file output

**processor** — Provides each processor with an identifying number. On systems that have one processor, only a **0** is present.

**cpu family** — Authoritatively identifies the type of processor in the system. For an Intel-based system, place the number in front of "86" to determine the value. This is particularly helpful for those attempting to identify the architecture of an older system such as a 586, 486, or 386. Because some RPM packages are compiled for each of these particular architectures, this value also helps users determine which packages to install.

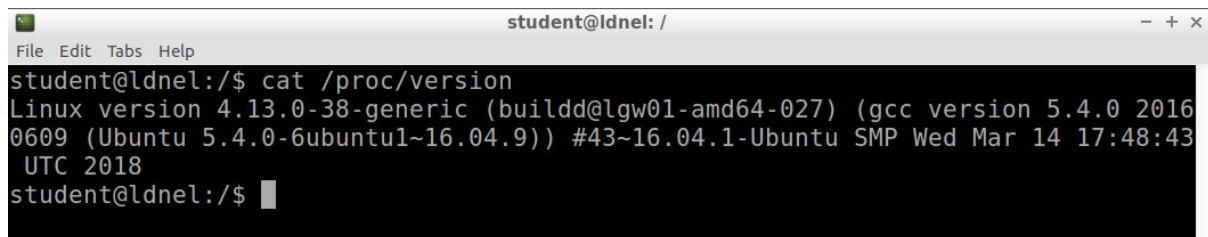
**model name** — Displays the common name of the processor, including its project name.

**cpu MHz** — Shows the precise speed in megahertz for the processor to the thousandths decimal place.

**cache size** — Displays the amount of level 2 memory cache available to the processor.

**siblings** — Displays the number of sibling CPUs on the same physical CPU for architectures which use hyper-threading.

This /proc/version file specifies the version of the Linux kernel and gcc in use. Here is a sample /proc/version file output:

A screenshot of a terminal window titled 'student@ldnel: /'. The window has a menu bar with 'File', 'Edit', 'Tabs', and 'Help'. The terminal shows the command 'student@ldnel:/\$ cat /proc/version' and its output: 'Linux version 4.13.0-38-generic (buildd@lgw01-amd64-027) (gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.9)) #43~16.04.1-Ubuntu SMP Wed Mar 14 17:48:43 UTC 2018'. The prompt 'student@ldnel:/\$' is shown again at the bottom with a cursor.

```
student@ldnel:/$ cat /proc/version
Linux version 4.13.0-38-generic (buildd@lgw01-amd64-027) (gcc version 5.4.0 2016
0609 (Ubuntu 5.4.0-6ubuntu1~16.04.9)) #43~16.04.1-Ubuntu SMP Wed Mar 14 17:48:43
UTC 2018
student@ldnel:/$
```

**Figure 11:** Sample /proc/version file output

### 3 Result

Our main goal when we began the project was to create a simple Linux task manager for users who needs it and for users who transfer from other operating systems. As the development progressing, we come up with new module display windows and cut off unnecessary display option for Linux system. Also, we make it support most popular Linux systems.

Here comes to the part that how to gather the information from the Linux system and display it to the users. The proc file system is a pseudo file system that exists only in memory and does not occupy external memory space. It provides an interface for accessing system kernel data in the form of a file system. Users and applications can obtain system information through proc and can change some parameters of the kernel.

To implement the CPU utilization, we read the stat file under /proc directory. The file is read line by line, and we takes 2 points sampling to calculate cpu current utilization. CPU total execution time can be sum up by looping each line to get execution time in every mode. So that after the calculation, we can get the CPU utilization and print it to the GUI which linked by Qt framework.

```

121     tempFile.setFileName("/proc/stat"); //open cpu status file
122     if ( !tempFile.open(QIODevice::ReadOnly) )
123     {
124         QMessageBox::warning(this, tr("warning"), tr("The stat file can not open!"), QMessageBox::Yes);
125         return;
126     }
127     tempStr = tempFile.readLine();
128     for (int i = 0; i < 7; i++)
129     {
130         cpuInfo[2-tt][i] = tempStr.section(" ", i+1, i+1).toInt();
131         cpuTotal[1][2-tt] += cpuInfo[2-tt][i];
132         if (i == 3)
133         {
134             cpuTotal[0][2-tt] += cpuInfo[2-tt][i];
135         }
136     }
137     tt--;
138     tempFile.close(); //close cpu status file

```

**Figure 5: CPU utilization implemented in code**  
(mainwindow.cpp:121-138)

RAM usage and SWAP partition implementation are similar as the CPU utilization. Pseudo file under /proc/meminfo includes all the information about the RAM usage in real time. So we read the meminfo file under /proc directory line by line.(Figure 6) We store the memory information string into corresponding variables according to their location in the meminfo file.The RAM used can be calculated by the formula:

$$RAM\ used = Total\ memory - Free\ memory$$

The swap partition can be calculated by the formula:

$$Swap\ partition\ used = Total\ swap - Free\ Swap$$

```

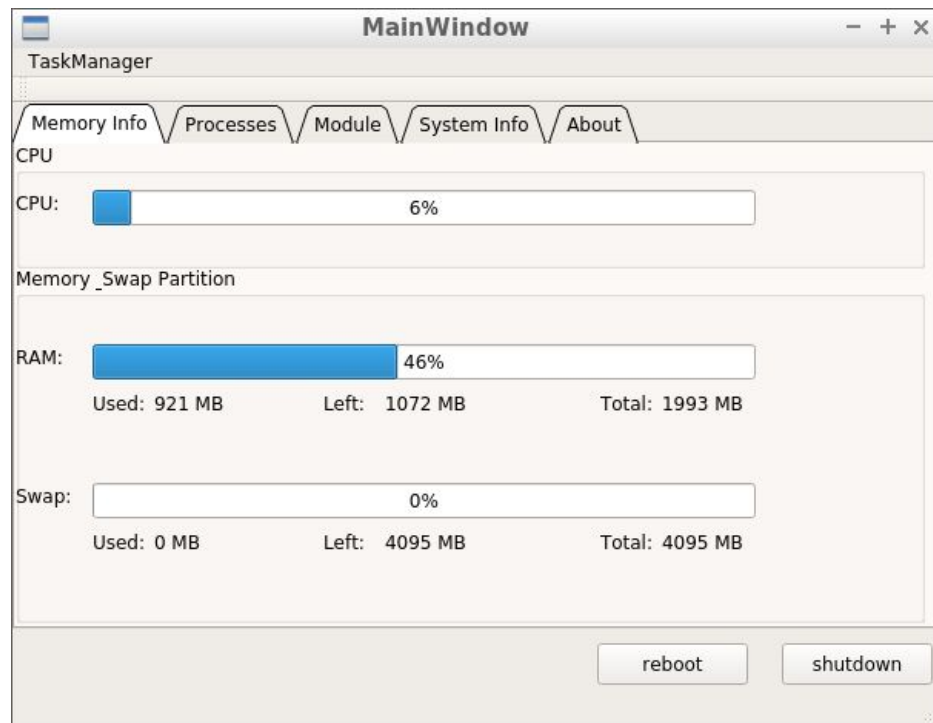
67     tempStr = tempFile.readLine();
68     pos = tempStr.indexOf("MemTotal");
69     if (pos != -1)
70     {
71         memTotal = tempStr.mid(pos+10, tempStr.length()-13);
72         memTotal = memTotal.trimmed();
73         nMemTotal = memTotal.toInt()/1024;
74     }
75     else if (pos = tempStr.indexOf("MemFree"), pos != -1)
76     {
77         memFree = tempStr.mid(pos+9, tempStr.length()-12);
78         memFree = memFree.trimmed();
79         nMemFree = memFree.toInt()/1024;
80     }
81     else if (pos = tempStr.indexOf("SwapTotal"), pos != -1)
82     {
83         swapTotal = tempStr.mid(pos+11, tempStr.length()-14);
84         swapTotal = swapTotal.trimmed();
85         nSwapTotal = swapTotal.toInt()/1024;
86     }
87     else if (pos = tempStr.indexOf("SwapFree"), pos != -1)
88     {
89         swapFree = tempStr.mid(pos+10, tempStr.length()-13);
90
91         swapFree = swapFree.trimmed();
92         nSwapFree = swapFree.toInt()/1024;
93         break;
94     }
95 }
96 nMemUsed = nMemTotal - nMemFree;
97 nSwapUsed = nSwapTotal - nSwapFree;

```



**Figure 6:** Memory usage implemented in code  
(mainwindow.cpp:67-97)

The CPU utilization and RAM usage linked through the Qt framework and print all the information to the graphical user interface as the *Figure 7*.



**Figure 7:** CPU & Memory info

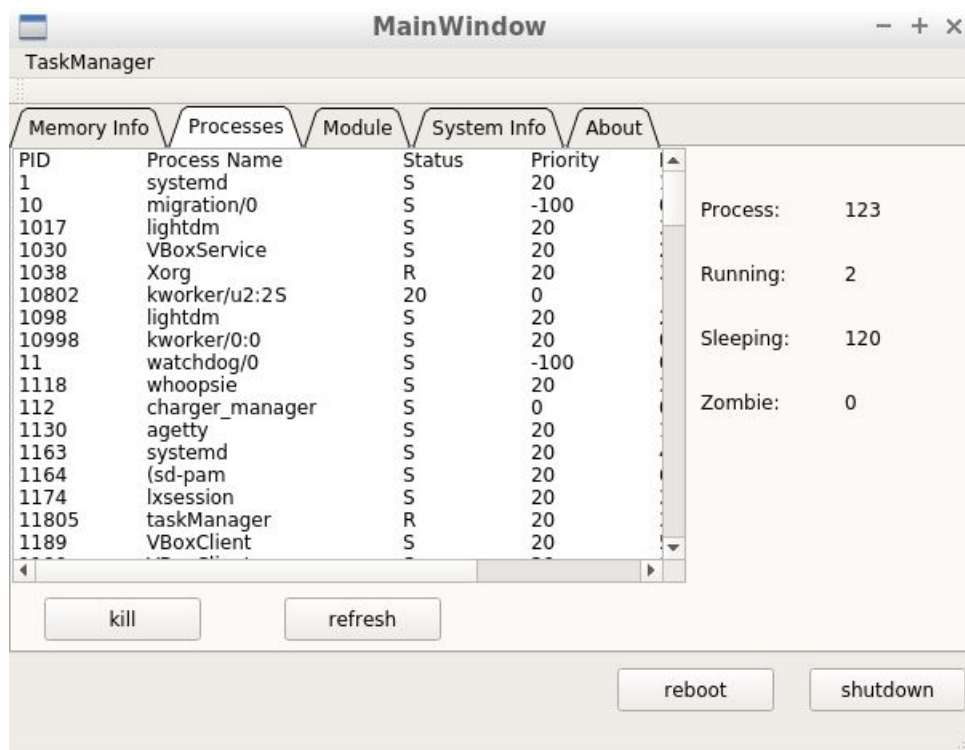
The implementation of process list is to loop all the process files. And for each process file, we read its stat file, and convert all the numbers to corresponding status expression or the unit expression. The data stored in the variables link with the GUI text by Qt framework.

```

223 //open the process file according to PID
224 tempFile.setFileName("/proc/" + id_of_pro + "/stat");
225 if ( !tempFile.open(QIODevice::ReadOnly) )
226 {
227     QMessageBox::warning(this, tr("warning"), tr("The pid stat file can not open!"), QMessageBox::Yes);
228     return;
229 }
230 tempStr = tempFile.readLine();
231 if (tempStr.length() == 0)
232 {
233     break;
234 }
235 a = tempStr.indexOf(" ");
236 b = tempStr.indexOf(" ");
237 proName = tempStr.mid(a+1, b-a-1);
238 proName.trimmed();
239 proState = tempStr.section(" ", 2, 2);
240 proPri = tempStr.section(" ", 17, 17);
241 proMem = tempStr.section(" ", 22, 22);
242 switch ( proState.at(0).toLatin1() )
243 {
244     case 'S': number_of_sleep++; break; //Sleep
245     case 'R': number_of_run++; break; //Running
246     case 'Z': number_of_zombie++; break; //Zombie
247     default: break;
248 }

```

**Figure 8: Processes list implementation in code**  
(mainwindow.cpp:223-248)



**Figure 9: Processes implemented in GUI**

The module information is also stored in the proc file system directory. The module information generally includes: dynamically loaded link libraries and executable file information. The process information that can be obtained by traversing the module includes: name of the module, the memory size of the module, how many times of the module are currently loaded, load state the module etc. Users and applications can get system information through /proc and can change some parameters of the kernel. Since the system



information is dynamically changed, when the user or application reads the proc file, the proc file system dynamically reads out the required information from the system kernel and submits it. Here is the code that we implement the module display window in *Figure 12*.

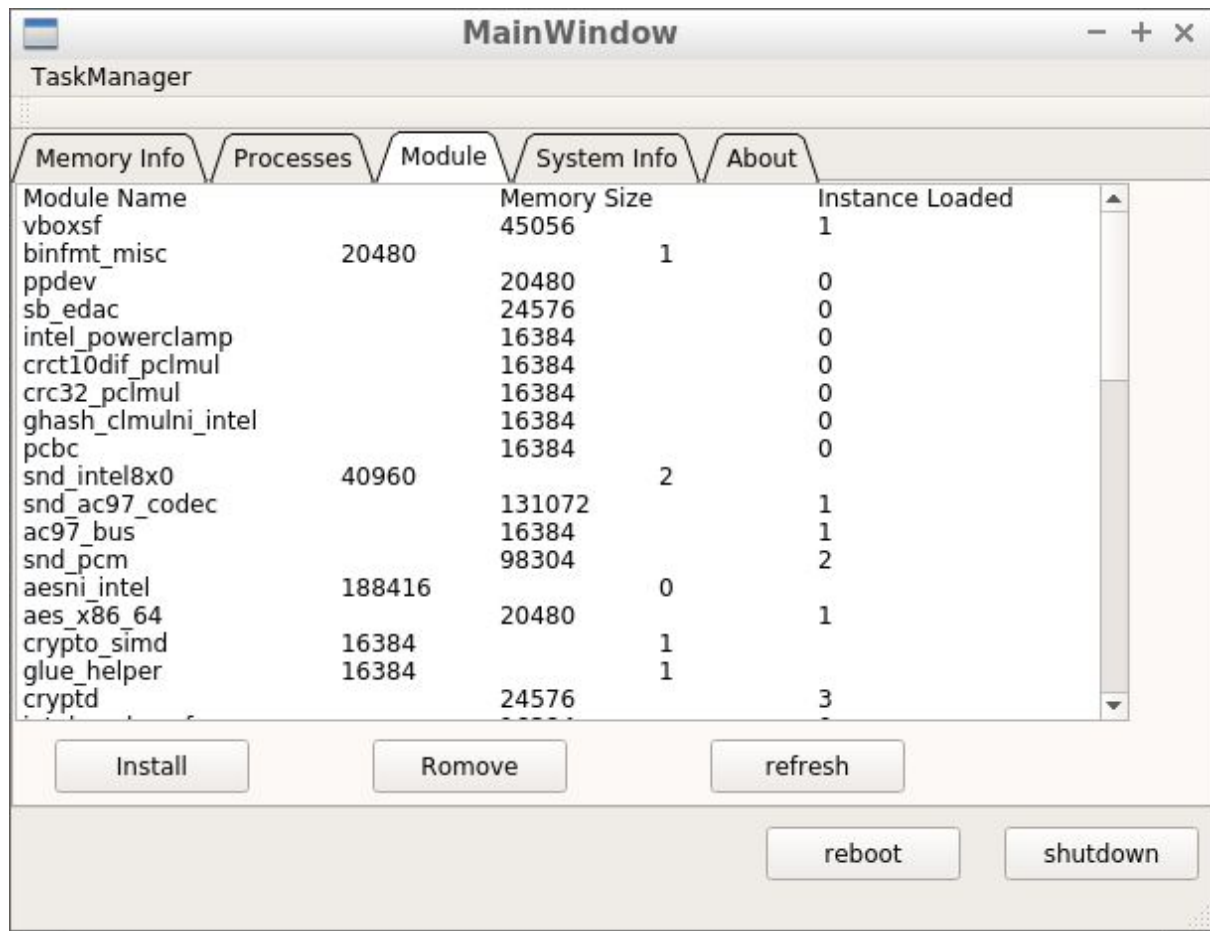
```

281 tempFile.setFileName("/proc/modules"); //open module file
282 if ( !tempFile.open(QIODevice::ReadOnly) )
283 {
284     QMessageBox::warning(this, tr("warning"), tr("The modules file can not open!"), QMessageBox::Yes);
285     return ;
286 }
287 //setting module tab title
288 QListWidgetItem *title = new QListWidgetItem( QString::fromUtf8("Module Name") + "\t\t" +
289     QString::fromUtf8("Memory Size") + "\t\t" +
290     QString::fromUtf8("Instance Loaded"), ui->listWidget_model);
291 QString mod_Name, mod_Mem, mod_Num;
292 //loop the file and search for the info
293 while (1)
294 {
295     tempStr = tempFile.readLine();
296     if (tempStr.length() == 0)
297     {
298         break;
299     }
300     mod_Name = tempStr.section(" ", 0, 0);
301     mod_Mem = tempStr.section(" ", 1, 1);
302     mod_Num = tempStr.section(" ", 2, 2);
303     if (mod_Name.length() > 10)
304     {
305         QListWidgetItem *item = new QListWidgetItem(mod_Name + "\t\t" +
306             mod_Mem + "\t\t" +
307             mod_Num, ui->listWidget_model);
308     }
309     else
310     {
311         QListWidgetItem *item = new QListWidgetItem(mod_Name + "\t\t\t" +
312             mod_Mem + "\t\t" +
313             mod_Num, ui->listWidget_model);
314     }

```

**Figure 12:** Module display window implement code  
(mainwindow.cpp:281-314)

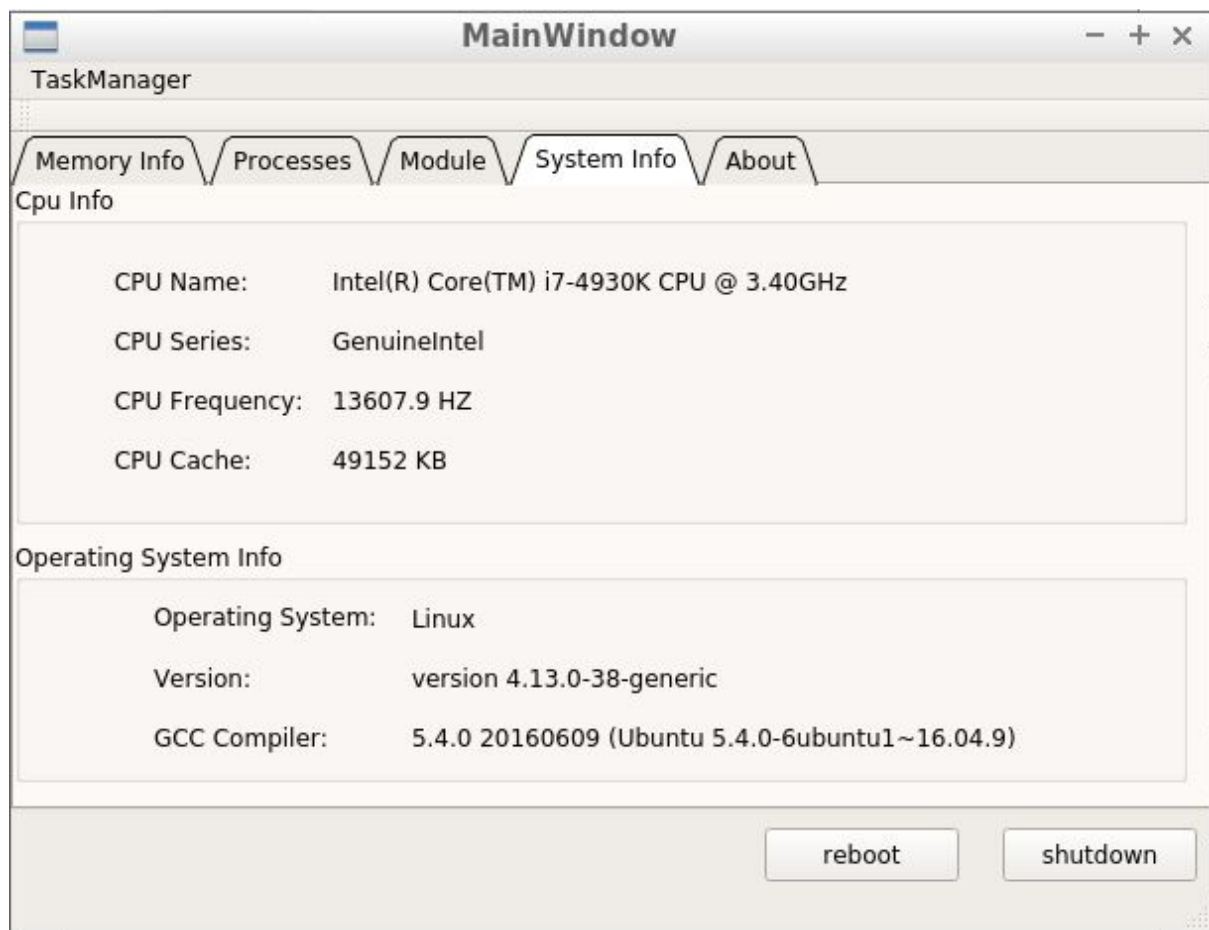
To programmatically read this information, we first need to understand the /proc structure, understand its arrangement of various kinds of information, and then filter it out. So, there are three key points when we code the program. First, we need to know where the information is. Second, we need to see how the information is arranged. Third, we need to design an algorithm removes useless information. Here is the final virtual effect page of the module display window:



**Figure 13:** Module display window

If we want to display system information, we only need to perform the corresponding file operation. First open the corresponding file, read the required information, write it into a buffer, and then add the contents of the buffer to the corresponding control of Qt, and finally

display the control combination. Here is the system information display window:



**Figure 14:** System information display window

*The virtual file read for this display window are /proc/cpuinfo and /proc/version.*

## 4 Evaluation and Quality Assurance

Critical evaluations were applied to this project in order to reflect the true Quality of the software. Performance, usability, and visual effect are the three main domains that we select as the benchmarks to evaluate our software. We test the software on different Linux kernel systems. Create a questionnaire for users who used our software in order to get the feedbacks.

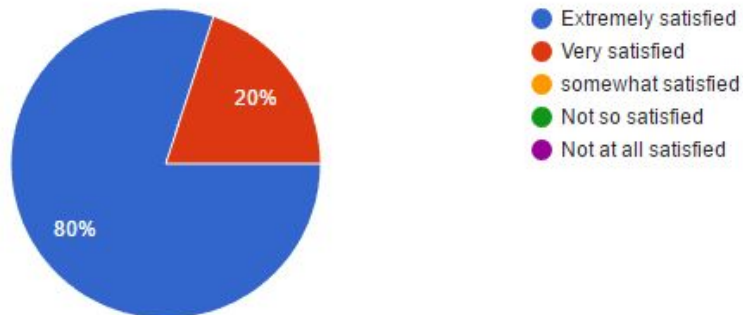
As a Linux task manager, we wish it to support most Linux systems. We installed some popular Linux distributions such as Ubuntu, Deepin, Debian, Lubantu (system image for COMP 2404). After compile and running it on these systems, the software is functional on these systems. However, for Debian, it could not display the system information properly.

We update the code so it can correctly read the system information. Therefore, we can confirm that our software should be working on most Linux distributions.

In order to test the performance and reliability, we designed some typical test cases for our software such as startup time, response time and running time. To be Comparative, we compare our Linux task manager to the Windows task manager and the Mac system monitor. First, we measure the startup time for every management tool on the same computer and record the time. They are all very close under normal cases, but it takes longer time for our Linux task manager pop up under heavy load. Similarly, we find the close records for the response times of these three tools. However, the running time for our software is very different from other two. We find out that after running the Linux task manager for few hours, it sometimes crushes when the user tries to use it again. The reason why leads to the crushed was not determined. But we will make it more stable in the future work. Based on these test cases, we can say that our Linux task manager is good in performance aspect but needs some improvement on the reliability. It generally meets our design purpose because user often not keep running the management tools for hours, they use it more like a onetime deal for few minutes.

In order to evaluate the usability, and visual effect aspects of the software, we provided a questionnaire that asked user questions about it. There are ten questions about this software. We choose the most typical ones to analysis for the visuals and usability. The Linux task manager was developed with Qt. We decided it was our best choice for programming the task manager because it had built-in user interface components. Everything related to the GUI was coded in a separate file. When we build the UI of the software, we design it as user friendly as we can. We take a reference at the Window task manager and Mac OS as well. Then we come up with a module information display window which also allow user to install or uninstall modules. It is specifically designed for Linux systems. The other display options such as users, history, and performers were cut out because we want to keep the software simple and easy to use. The following chart show our decision is right.

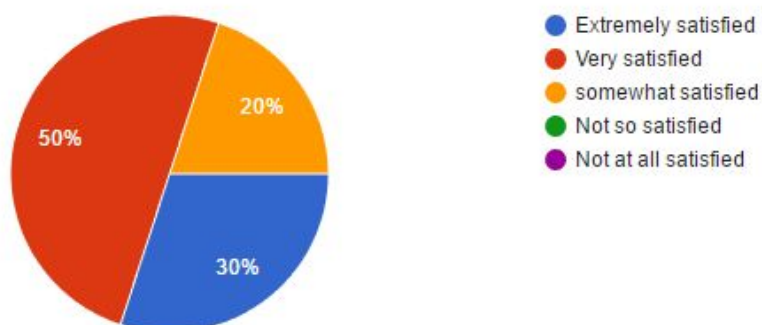
How satisfied are you with the look and feel of this software?



As the chart shows, most people were satisfied with the UI design of the software. There were none of them think it was a bad design. We simply try keep the UI clean, delete the unnecessary options and add a module information window for Linux systems. People satisfied because this Linux task manager has every basic function they need. Therefore, the virtual effects of this software meet our requirements.

Another important requirement is make users who transfer from other operating system feel familiar with this task manager. There are people who complain about there are no build in task manager for some Linux systems. Also, users want in control of their computers, and the ability to manage it. We design this software to lower the learning cost when user transfer from different systems.

How satisfied are you with this software if you transfer from Windows or Mac?



As we can see from this chart, most user feel satisfied when they use the Linux manager. It has most functions they needed for a Linux system and successfully lower the learning cost. In this case, we can say that usability and visuals meet the requirement of the design.

Our Linux task manager archived design purpose on performance, usability, and visual effect aspects. Although the reliability of this software can be improved. And more functions could be implement as well. Right now, it is a simple but powerful Linux task manager that support most popular Linux systems.

## **5 Conclusion**

### **5.1 Summary**

Overall, the main goal of this project is achieved. This simple Linux task manager is able to let most Linux users feel familiar when they transfer from system like Windows or Mac. Also, it displays the system information clearly in a user-friendly interface which let the user easily see and manage the system by themselves. Throughout the development process, many plans come up to conquer challenges like bugs, design prototypes, and make user-friendly UI logic. Windows system task manager and Mac system task manager were taken as a reference when this Linux system task manager were design. The Linux system task manager not only has most features that user familiar with but also it includes a module system manager as it designed specifically for the different Linux systems. The module manager not only allow user to see the running modules but also give the user the ability to install new modules or uninstall them. However, solution for install and uninstall modules were included in the future work with the user-friendly UI designed finished in this example. For most Linux systems, there is no such a thing like a task manager. It increased the learning cost for user who transfer from Windows and other system. However, people can customize their own system, and this is the starting point of our project. On the other hand, it is a great experience for us to design and finish this project. This project has enabled a better understanding for us on how Linux filesystem works and what is a user-friendly program. As a task manager for users who transfer from other operating system or need a manager for Linux, it has succeeded for its job.

### **5.2 Relevance**

For academic purpose, this Linux task manager is a combination use of file system, system call, and module. These three topics are all included in the course topics. Specifically, we use the /proc filesystem to get the real time information of the system usage

and display it to the user. It is special because it is a virtual file system, also refer as a process information pseudo-file system. It does not contain 'real' files but runtime system information. Simply put, it is the control and information center of the kernel. The task manager uses it to read the runtime system info and interact with the kernel. Through the development, we learned how the modules that interact with the kernel in order to perform actions within the user mode. The GUI component is also deals with how user mode is used to interact with the kernel.

### **5.3 Future Work**

Although we reached most purpose of our project design, there is some room for further improvement. For Example, the implementation of install and uninstall modules function in the task manager. Currently, we only allow user to kill the process, and display all running processes and modules. The Linux system is a kernel-based system, the kernel modules can be loaded and unloaded into the kernel upon demand. When people customer their system, they want to know what modules are loaded and uninstall unnecessary modules.

### **Contributions of Team Members**

#### **TianCheng Zhao:**

Responsible for writing the Introduction, Background information and result part of the report, building and implementing the runtime usage of CPU, process and system information functions. Also, analysis the test data. His Contributed to project proposal, conclusion of report and GUI design for the project.

#### **Yinfeng Hu:**

Responsible for writing the Evaluation and Quality Assurance and Conclusion part of the report and building and implementing the runtime usage of RAM and module information functions. His also create the project proposal, design the GUI for the project and gather the test data. His also Contributed to result of report, GUI design for the project.

### **References**

- [1] Both Feed, David. *A User's Guide to Links in the Linux Filesystem*.  
[opensource.com/article/17/6/linking-linux-filesystem](https://opensource.com/article/17/6/linking-linux-filesystem). Accessed 23 March. 2018.
- [2] "Discover the Possibilities of the /Proc Directory." *Linux.com | The Source for Linux Information*, 15 Feb. 2008, [www.linux.com/news/discover-possibilities-proc-directory](http://www.linux.com/news/discover-possibilities-proc-directory).



[3] *Linux Filesystem Hierarchy*, 30 July 2004,

[www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/index.html](http://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/index.html).

[4] “Writing a Linux Kernel Module - Part 1: Introduction.” *Derekmolloy.ie*,

[derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction/](http://derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction/). Accessed 1 Apr. 2018.

[5] “Red Hat Customer Portal.” 5.2. *Top-Level Files within the Proc File System*,

[access.redhat.com/documentation/en-us/red\\_hat\\_enterprise\\_linux/5/html/deployment\\_guide/s1-proc-topfiles](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/5/html/deployment_guide/s1-proc-topfiles). Accessed 5 Apr. 2018.