Team members: Felipe Monteiro, Garrett Bosecker, Yaling Wu

1. Description of Overall Design

*Note:* our module has been tested on Ubuntu 14.04, Linux kernel version 3.19.

An array of struct kprobes is created. For each entry in the array, there is a corresponding entry in the character array symbol\_names; each entry’s symbol\_name points to a unique entry in symbol\_names. This way, the OS resolves the address of each system call. In addition, three proc files are created via proc\_create(), one for sysmon\_toggle, one for sysmon\_uid, and one for sysmon\_log. The final data structure we use is a character array called log, which is dynamically allocated a chunk of memory via kmalloc. The proc file sysmon\_log outputs log whenever the former is opened. This method requires no kernel file read or write operations, which easily invite system crashes. To retrieve the data (uid and toggle) inputted by the user, we use uid\_write and toggle\_write functions for /proc to monitor any change from the user space (i.e. echo command on terminal to modify uid and toggle logs). After obtaining user’s commands, we interpret them and change our program status accordingly.

Initially, sysmon\_uid is initialized to 1000 in the module, and hence no logging will take place until the user sets sysmon\_uid to a valid uid. This is a deliberate design choice, since if insmod is called on our module, all actions by all uids will be logged, which will quickly bloat the log. However, sysmon\_toggle is by default set to 1, so that when a valid uid is set in sysmon\_uid, logging automatically begins.

As mentioned above, sysmon\_log outputs the contents of log, a character array residing in memory. Since this array is limited in size, earliest entries are overwritten by latest entries, whenever the array runs out of size. While it would be possible to double the amount of memory allocated to the array each time it runs out of memory, this presents the problem of memory allocation failing, which would require proper error handling, lest the system crash (besides, it is only a matter of time before memory runs out). In any case, printk() “pipes” all the logging done by the module to /var/log/kern.log which will (presumably) hold the entire log history for the module, even after the module is terminated via rmmod.

1. Description of Experiment Configuration

Our good user space program is run.sh. This script iterates forever, and is responsible for instantiating, setting up, and communicating with the module as well as transferring text from sysmon\_log to a user-space program called to\_file.c, which simply appends the text to a file called sysmon.backup--a hard copy, so to speak, of the log.

Executing ps aux provided the following printout for run.sh:

USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND

root 2660 27.6 0.1 16888 3412 pts/12 R+ 23:12 0:02 /bin/bash ./run.sh

This script provides a user-friendly way of interacting with sysmon, and without it being able to behave normally, our project would be less polished and our logging would be less organized and effective.

Before running our script, vmstat provides the following system statistics:

root@osboxes:/proc# vmstat

procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----

r b swpd free buff cache si so bi bo in cs us sy id wa st

4 0 10312 224688 5976 201280 0 2 132 33 181 1214 16 1 81 1 0

While running run.sh, vmstat reports the following:

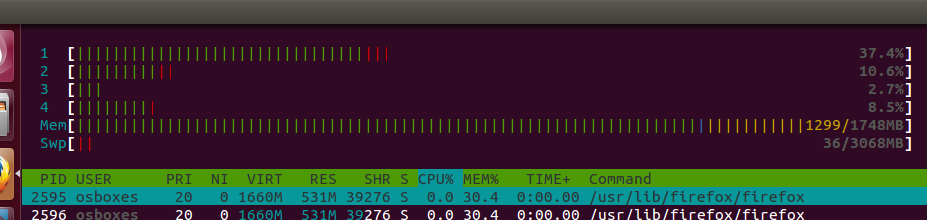
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----

r b swpd free buff cache si so bi bo in cs us sy id wa st

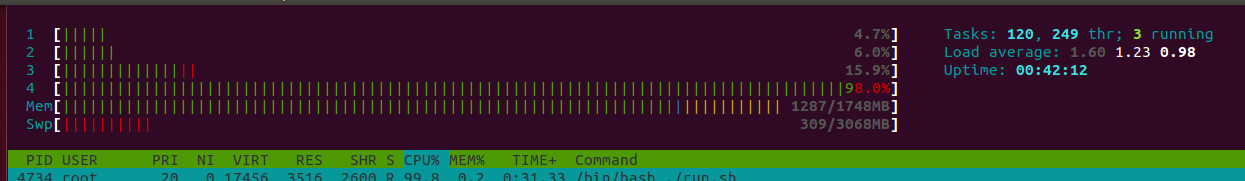
2 0 22056 234272 10820 188088 0 5 128 35 182 1224 16 1 81 1 0

This is not bad at all. No other processes are in the blocked queue. Only two processes are waiting for run time. A lot of memory is free, meaning the module and script are not memory hogs. Since the overall statistics are very similar to the statistics before running the script and module, this test indicates little resource consumption by our script/module.

Before running the script/module, htop shows the following data:



Since the system on which this is being run has 4 cores, the amount of work being done by each one is shown above. While running the script/module, htop shows the following data:



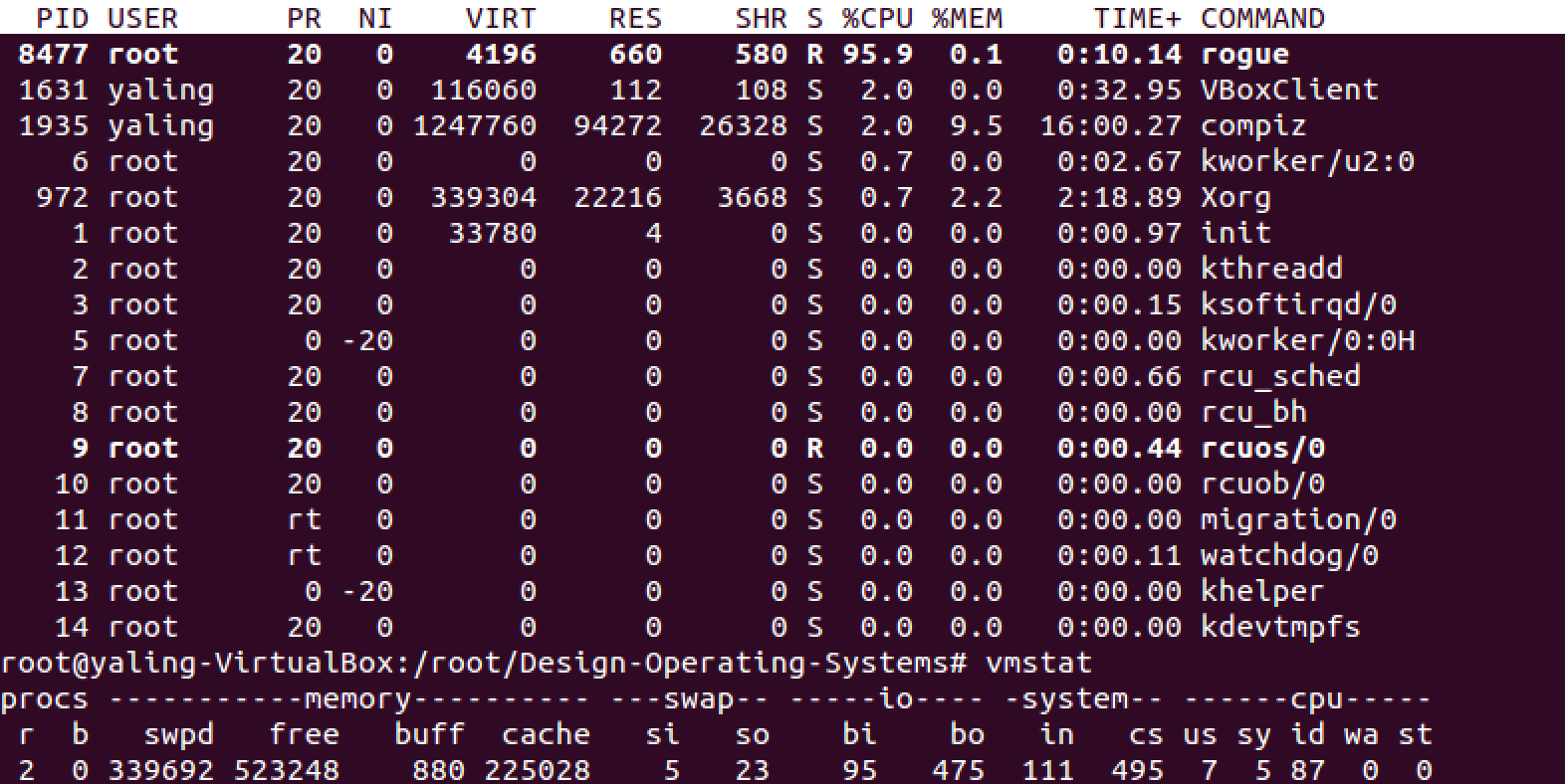
Clearly, memory usage and swap usage have not worsened by much, though one core is maxed out completely while the script/module is running, indicating its performance burden.

**Rogue program**

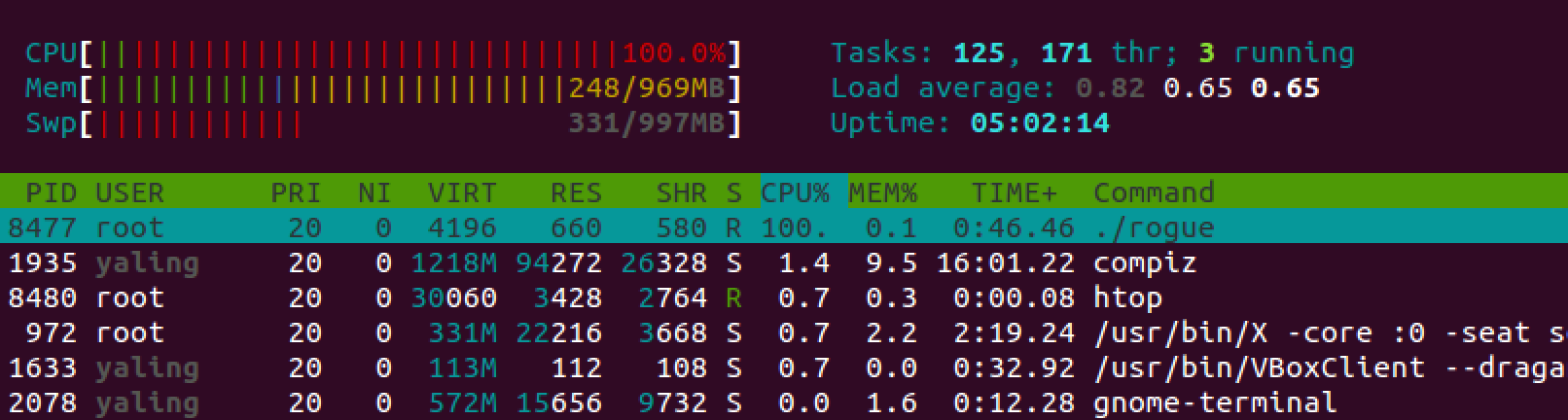
-> rogue.c : a simple user (user is root whose user id is 0) program that opens a “test.dat” file and write “Something” into it many times. The system calls include open(), write(), and close().

Running rogue without interposer

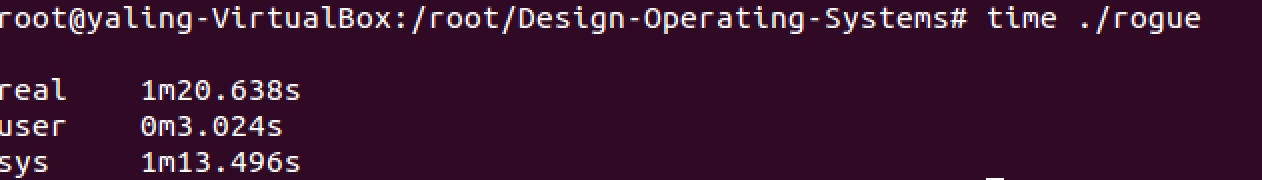
top and vmstat command returns:



htop command shows:

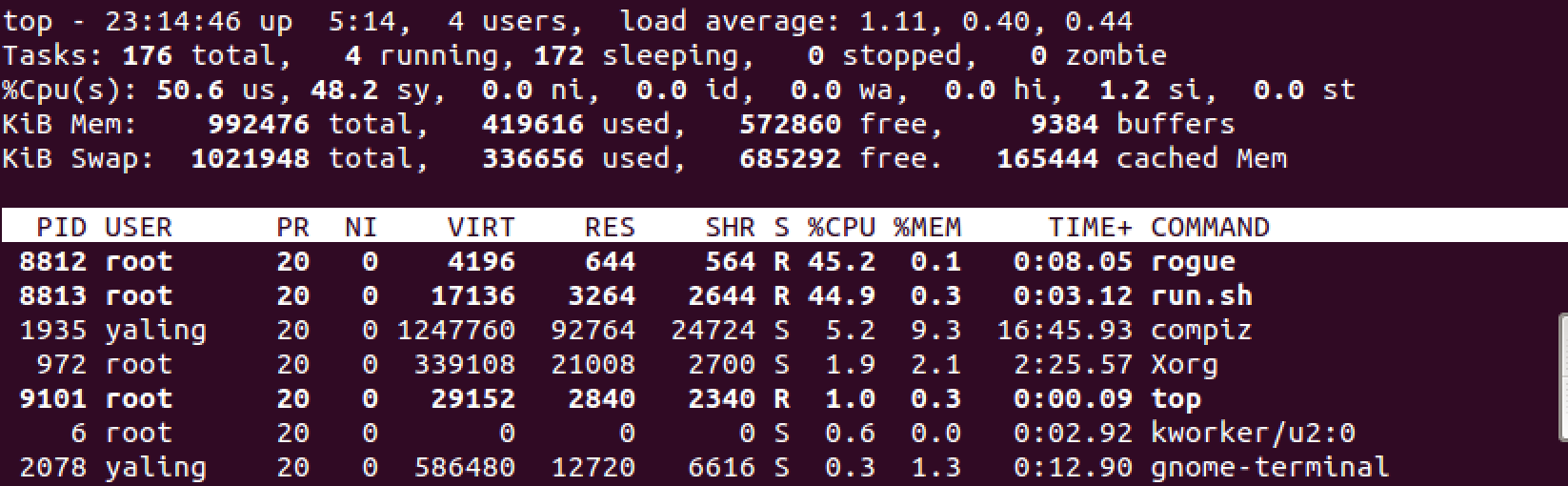


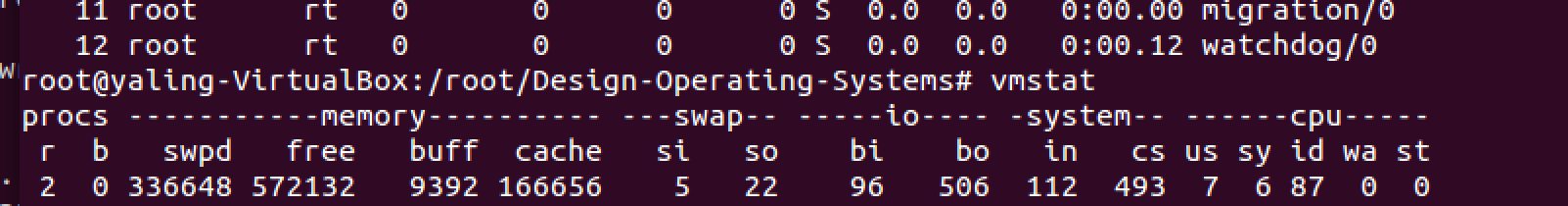
The time rogue.c took to execute is:



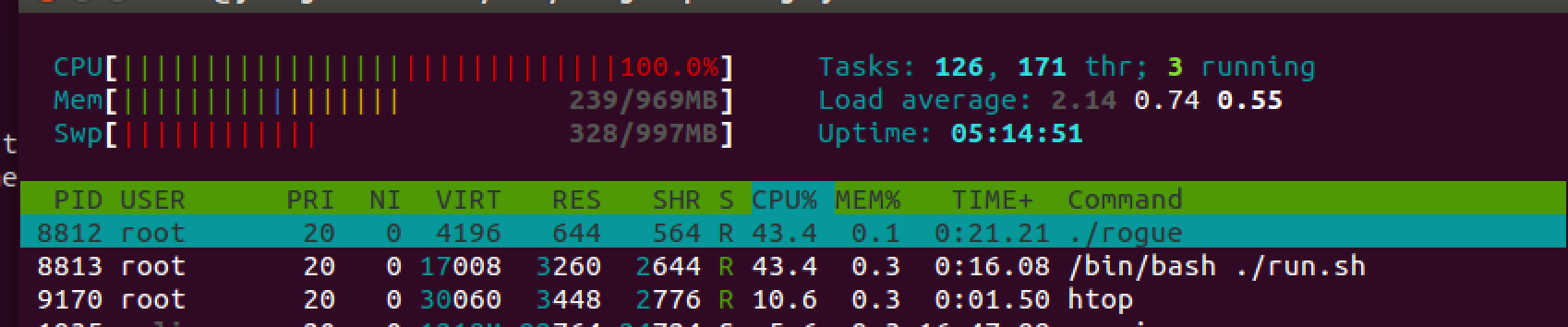
After running run.sh to execute our interposer program

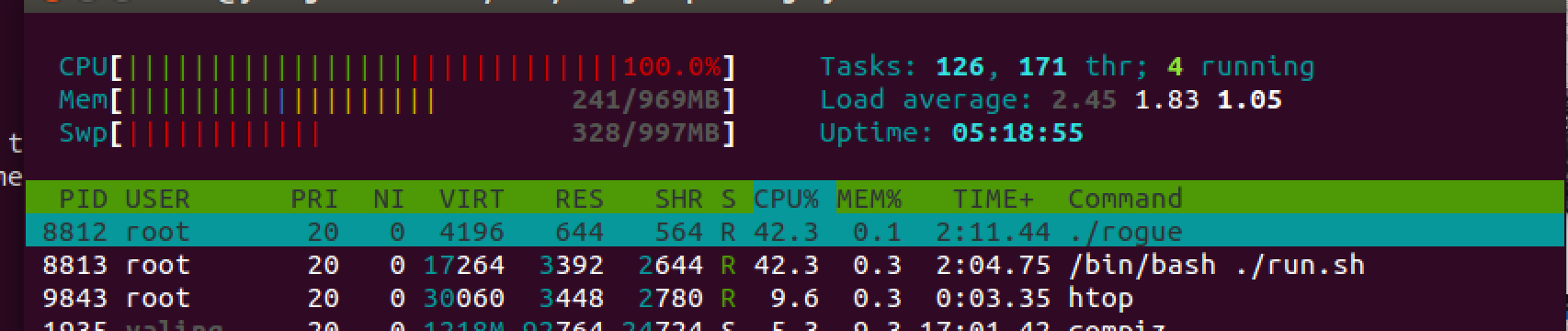
top and vmstat command returns:





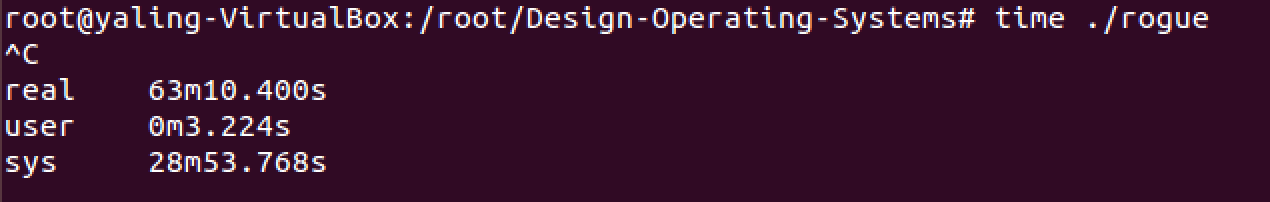
htop command shows:





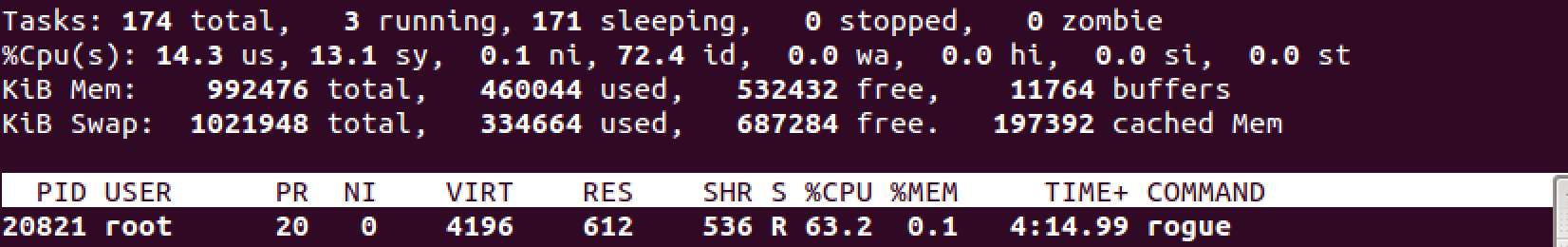
As shown above, this time system resources were not concentrated at rogue program. Our interposer also took a lot of them.

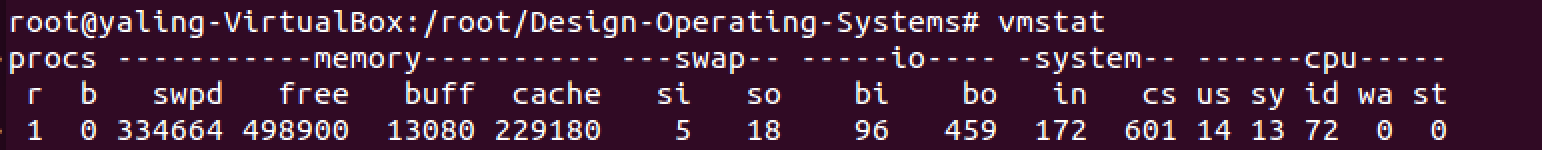
The time rogue.c took to execute was over 63 minutes, which was way over original 1m20s.



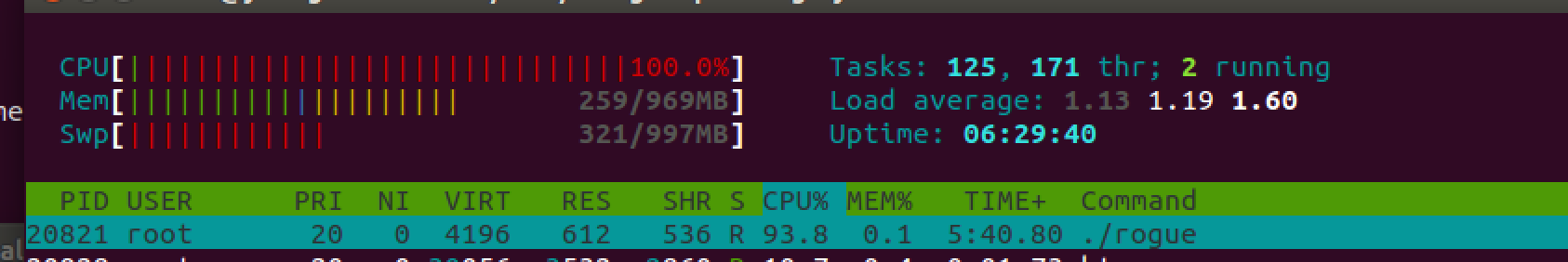
After inserting sysmon.ko to execute our interposer program but not run.sh in the play

top and vmstat command returns:



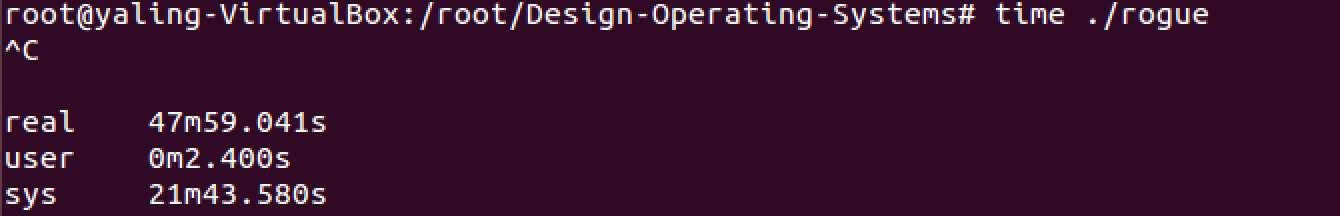


htop command shows:

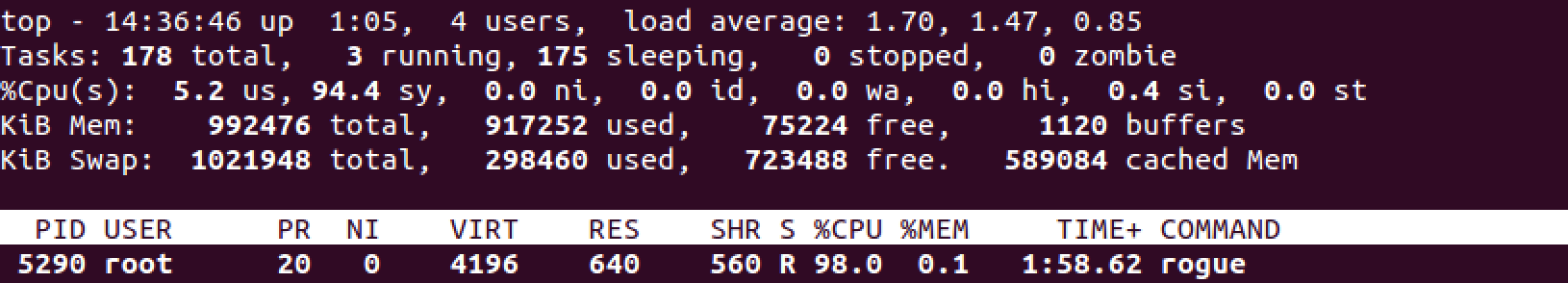


Inserting the bare module caused that system resources were somewhat allocated to our interposer as well.

The time the module itself took to execute was longer than 48 minutes, which was way over original 1m20s. All the above indicates that our interposer affects rogue significantly.

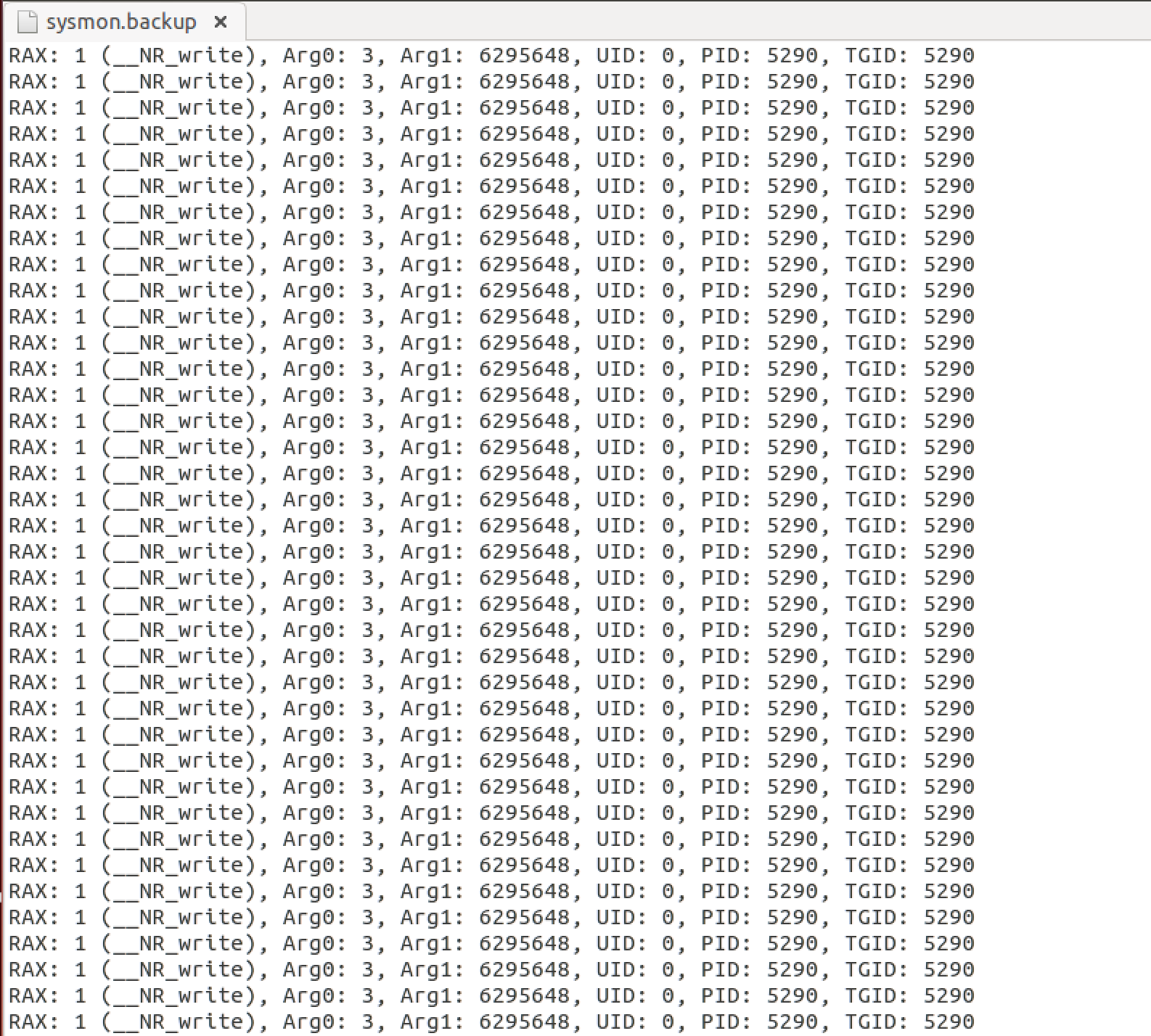


A test of our interposer with rogue:



This time PID was 5290, UID is 0 for the rogue process.

Below is part of the log contents during rogue execution. It shows the system calls called by rogue:



1. Challenges Faced

There were numerous challenges faced. They are as follows:

* *Login challenges:* We could not access factor-login.cc.gatech.edu, so we tried to get the right kernel version on our local machine. We successfully installed Ubuntu 14.04 and Linux kernel version 3.13, but when we rebooted the virtual machine, it threw an error and broke down.
* */proc-related challenges:* We tried updating the integer in /proc/sysmon\_uid and /proc/sysmon\_toggle by directly writing to them (i.e. opening up the file via gedit or vim and changing the integer). However, it would not allow us to save the change, so we came up with a workaround: piping the new integer value from the command line. For example, echo 1 > sysmon\_toggle enables logging by writing a 1 to sysmon\_toggle, which is then detected by our module, after which it updates its behavior accordingly.
* *Creating log backup challenges:* Every attempt at creating a persistent footprint of our log failed, since that entailed writing the log in memory to a file. This is because reading/writing from/to a file in kernel is dangerous, easily resulting in the system crashing: this is what happened repeatedly. One possible workaround is to heavily rely upon printk(). However, after further research, it was discovered that, while “It is useful for debugging and reporting errors,” it should be “use[d] with caution: a machine which has its console flooded with printk messages is unusable” (source: http://www.linuxgrill.com/anonymous/fire/netfilter/kernel-hacking-HOWTO-4.html).
* *Kernel-related challenges:* For a variety of reasons, the system kept crashing. Unless the code was impeccable, the system would crash. Even then, it would crash for impossible-to-understand reasons as well. For example, even though kernel information is directly logged to /var/log/kern.log (which can also be reviewed using the command dmesg), if a crash occurs, the crash propagates throughout the entire kernel, meaning that the log is never even written to, thereby preventing the diagnosis of the crash. Trial and error and logical deduction are needed in place of conventional error diagnosis tools.
* *Documentation quips:* Very little or poor documentation. This is perhaps the worst issue of all. Writing kernel code is already very difficult, especially for students coming from a user-space coding background. It would be helpful if, at least for the first project, students could be pointed to reliable, up-to-date kernel documentation. Since Linux has so many kernel versions, a lot of online documentation is obsolete.
* *Project requirements/feedback quip:* Vague/ambiguous/seemingly contradictory project requirements. Limited feedback from professors and TAs on Piazza to reconcile student concerns did not help either. For example, the requirement that our logs be maintained directly contradicts the warning on Piazza to not write files from within the kernel: how else should a log be maintained, so as to not write out of memory, other than by dumping the contents of the buffer in memory to disk, which of course requires writing to a file? Of course, there might be a kernel tool that helps with this, but it is a matter of sheer luck (or dogged persistence) whether students encounter the tool through their research.

1. Suggested Improvements

* For now, directly opening sysmon\_uid and sysmon\_toggle will not display the correct information to user. Instead, the user has to use ‘dmesg’ on terminal to keep track of changes in uid and toggle. We could improve the program by refining read function call of both proc files.
* Adding error checking everywhere (for example, whenever memory allocation fails, or something else fails). Note that a lot of error checking is already included.
* Breaking up the source code for the module into logically separate files. This was not done because efforts at doing this were frustrated by lack of documentation on how to do this and the general abstruseness of Makefile.
* Due to the frequent system calls for some users, sometimes data in sysmon\_log (limited size) will be replaced by new incoming data before transfer of the log data to sysmon.backup. Expanding the log array could not really solve the problem since it has a limited maximum size, which can still not hold all the flooding data.