title: “Operating Systems Homework 1 Report” author: [Andres Ponce, 0616110] date: “2020-10-24” titlepage: true, titlepage-text-color: “FFFFFF” titlepage-rule-color: “360049” titlepage-rule-height: 0 titlepage-background: “background.pdf” —

## Discussion Questions

1. **What is a kernel function? What is a system call?**

A kernel function allows us to interact directly with the kernel. The kernel allows us to write functions that implement some of the kernel utilities on the system. For example, if we want our program to make a new directory, we can use the provided kernel functions on our operating system of choice. These functions will then execute the required system calls.

A system call requests something directly from the kernel. However, a kernel function might just be a wrapper around the system call that directly executes the responsible code.

2. **what is KASLR? What is it for?**

KASLR stands for **K**ernel **A**ddress **S**pace **L**ayout **R**andomizer. This utility will load the kernel in a random place in memory during boot time. If the kernel code were loaded in the same memory location every time, then we could exploit the location of the kernel functions for some nefarious use. We would just need to know the code structure and then find a way to insert malicious code into that address space. By loading the kernel at a random location in memory, an attack of this sort is made much harder.

We can turn this setting off during boot time by using the `nokaslr` option.

3. **What are GDB’s non-stop and all-stop modes?**

In GDB, the all-stop and non-stop modes refer to how the program stops execution. In the former, *all* the currently executing threads stop. This allows us to view the entire state of the program at a certain point. The latter mode refers to only stopping certain threads while allowing other currently executing threads to continue.

One might be more useful for isolating the behavior of a single thread, while the other might be more useful for viewing the entire state of the program at a given point.

4. **Explain what the command `echo g > /proc/sysrq-trigger` does.**

The `/proc/sysrq-trigger` file allows us to issue instructions directly to the kernel. In the Linux file system, the `proc` directory contains information about the currently executing processes. The file `sysrq-trigger` *triggers* something to happen in the kernel. The `g` that we echo into the file is specifically used by kgdb. This is why kgdb regains control after we echo it into this file. Besides this, we can also crash the system or immediately restart the system.

## Questions From Do It Yourself 2

5. **Perf also has the report command. Explain:**
   - **What is it for?**
   - **For fileCopyTest, show and interpret the results.**

The `perf` program measures the performance of commands on Linux. We can get some other statistics on how our program is performing, and a complete trace of the system calls that are executing.
**Your Screenshot Here**

6. **Perf has more commands. Select another command (besides report, trace and record), and explain what it is for and show how to use it.**

If we look at the documentation for `perf`, we see that there are many commands available.

```
perf

 usage: perf [--version] [--help] COMMAND [ARGS]

 The most commonly used perf commands are:
  annotate        Read perf.data (created by perf record) and display annotated code
  archive         Create archive with object files with build-ids found in perf.data file
  bench           General framework for benchmark suites
  buildid-cache   Manage <tt>build-id</tt> cache.
  buildid-list    List the buildids in a perf.data file
  diff            Read two perf.data files and display the differential profile
  inject          Filter to augment the events stream with additional information
  kmem            Tool to trace/measure kernel memory(slab) properties
  kvm             Tool to trace/measure kvm guest os
  list            List all symbolic event types
  lock            Analyze lock events
  probe           Define new dynamic tracepoints
  record          Run a command and record its profile into perf.data
  report          Read perf.data (created by perf record) and display the profile
  sched           Tool to trace/measure scheduler properties (latencies)
  script          Read perf.data (created by perf record) and display trace output
  stat            Run a command and gather performance counter statistics
  test            Runs sanity tests.
  timechart       Tool to visualize total system behavior during a workload
  top             System profiling tool.

 See 'perf help COMMAND' for more information on a specific command.
```

**Figure 1:** Some `perf` commands.

For example, we have the `perf stat` command, which will gather performance counter statistics on our progam. To use it we can run some shell command or execute some program, and it can relay information such as memory usage, cache misses, and other potentially useful information. To run it we just type

```
sudo perf stat [OPTIONS] command
```

where *command* is a shell command.

## Screenshot Discussion

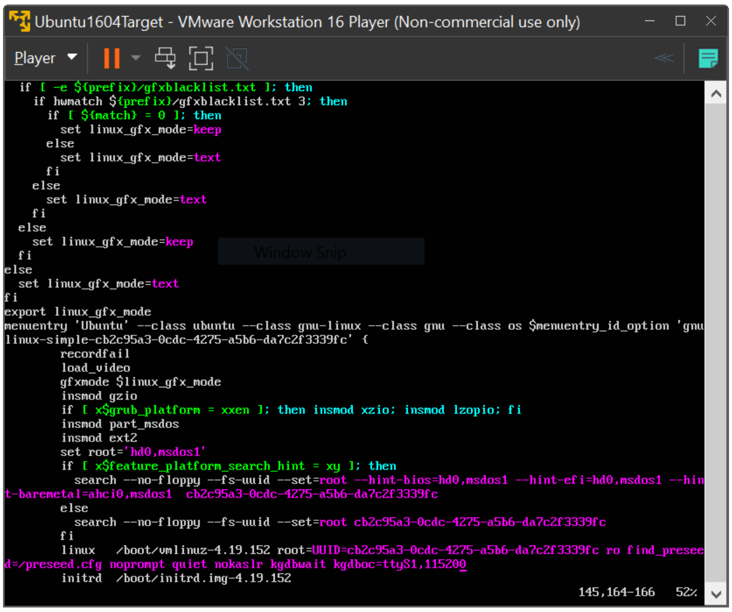



**Figure 2:** Updating Grub

The first step in our homework assignment is to update the GRUB. We disable KASLR, which loads the kernel at random locations in memory to avoid potential attacks on the kernel's memory space. Since this is enabled by default, we have to adjust it on the kernel's command line parameters. After we do so, we have to run the `sudo update-grub` command, which generates `grub.cfg`. However, we have to add again our own kernel parameters which we added in 1.A. These parameters are the `kgdbwait` and the `kgdboc=ttyS1,112500`.

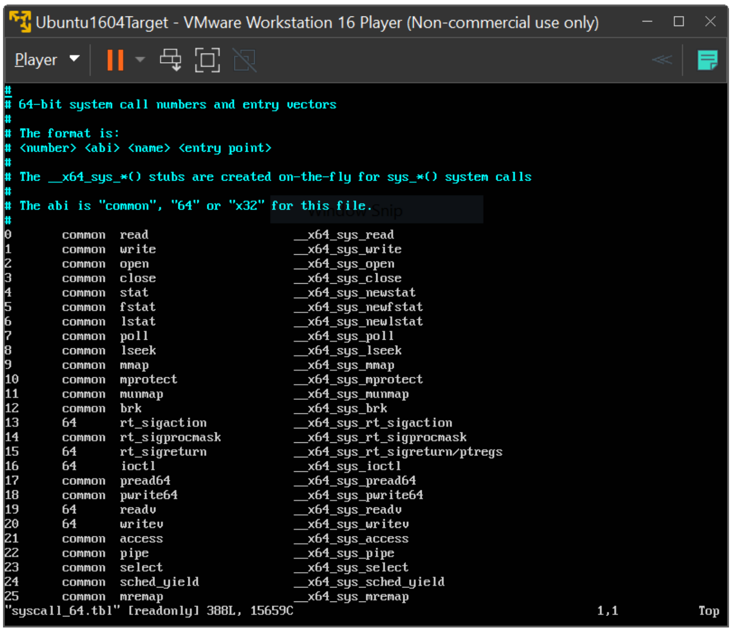
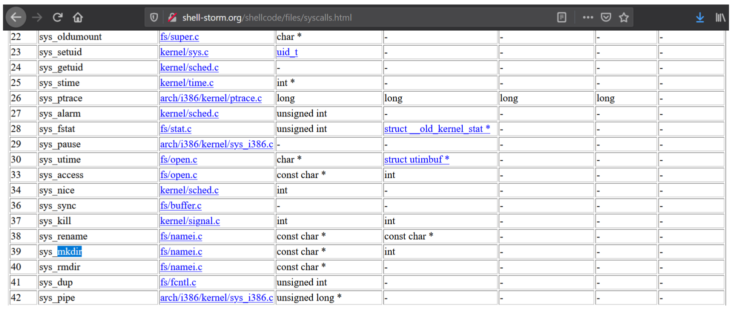
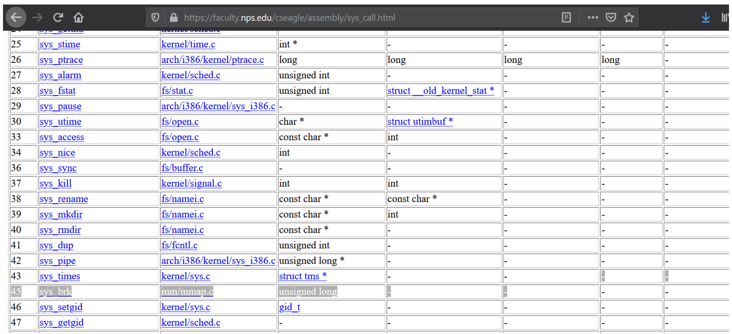



**Figure 3:** Syscall Table




| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 22 | sys_oldumount | fs/super.c | char * | - | - | - | - |
| 23 | sys_setuid | kernel/sys.c | uid_t | - | - | - | - |
| 24 | sys_getuid | kernel/sched.c | - | - | - | - | - |
| 25 | sys_stime | kernel/time.c | int * | - | - | - | - |
| 26 | sys_ptrace | arch/i386/kernel/ptrace.c | long | long | long | long | - |
| 27 | sys_alarm | kernel/sched.c | unsigned int | - | - | - | - |
| 28 | sys_fstat | fs/stat.c | unsigned int | struct __old_kernel_stat * | - | - | - |
| 29 | sys_pause | arch/i386/kernel/sys_i386.c | - | - | - | - | - |
| 30 | sys_utime | fs/open.c | char * | struct utimbuf * | - | - | - |
| 33 | sys_access | fs/open.c | const char * | int | - | - | - |
| 34 | sys_nice | kernel/sched.c | int | - | - | - | - |
| 36 | sys_sync | fs/buffer.c | - | - | - | - | - |
| 37 | sys_kill | kernel/signal.c | int | int | - | - | - |
| 38 | sys_rename | fs/namei.c | const char * | const char * | - | - | - |
| 39 | sys_mkdir | fs/namei.c | const char * | int | - | - | - |
| 40 | sys_rmdir | fs/namei.c | const char * | - | - | - | - |
| 41 | sys_dup | fs/fcntl.c | unsigned int | - | - | - | - |
| 42 | sys_pipe | arch/i386/kernel/sys_i386.c | unsigned long * | - | - | - | - |

**Figure 4:** Online reference




| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 25 | sys_stime | kernel/time.c | int * | - | - | - | - |
| 26 | sys_ptrace | arch/i386/kernel/ptrace.c | long | long | long | long | - |
| 27 | sys_alarm | kernel/sched.c | unsigned int | - | - | - | - |
| 28 | sys_fstat | fs/stat.c | unsigned int | struct __old_kernel_stat * | - | - | - |
| 29 | sys_pause | arch/i386/kernel/sys_i386.c | - | - | - | - | - |
| 30 | sys_utime | fs/open.c | char * | struct utimbuf * | - | - | - |
| 33 | sys_access | fs/open.c | const char * | int | - | - | - |
| 34 | sys_nice | kernel/sched.c | int | - | - | - | - |
| 36 | sys_sync | fs/buffer.c | - | - | - | - | - |
| 37 | sys_kill | kernel/signal.c | int | int | - | - | - |
| 38 | sys_rename | fs/namei.c | const char * | const char * | - | - | - |
| 39 | sys_mkdir | fs/namei.c | const char * | int | - | - | - |
| 40 | sys_rmdir | fs/namei.c | const char * | - | - | - | - |
| 41 | sys_dup | fs/fcntl.c | unsigned int | - | - | - | - |
| 42 | sys_pipe | arch/i386/kernel/sys_i386.c | unsigned long * | - | - | - | - |
| 43 | sys_times | kernel/sys.c | struct tms * | - | - | - | - |
| 45 | sys_brk | mm/mmap.c | unsigned long | - | - | - | - |
| 46 | sys_setgid | kernel/sys.c | gid_t | - | - | - | - |
| 47 | sys_getgid | kernel/sched.c | - | - | - | - | - |

**Figure 5:** Online reference 2

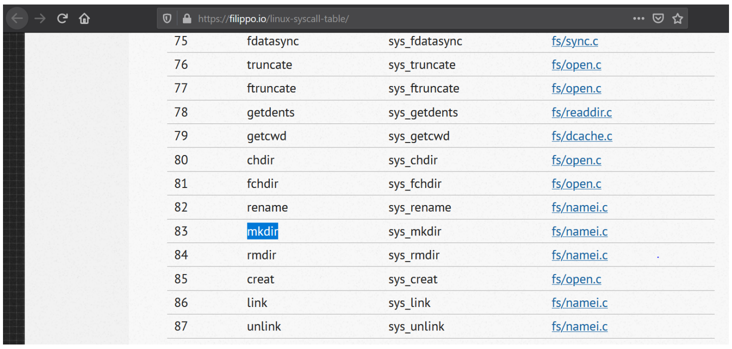



**Figure 6:** Online reference 3

The next screenshots show in general how we find the definition of the system calls. First, we look at the reference of the system calls in teh kernel source code. Then, we can look for online references and see which file the system call is in. We can see that the `mkdir` is in the `fs/namei.c` file. We used multiple references in order to make sure that the kernel version does not inlfuence the location of the file we need.

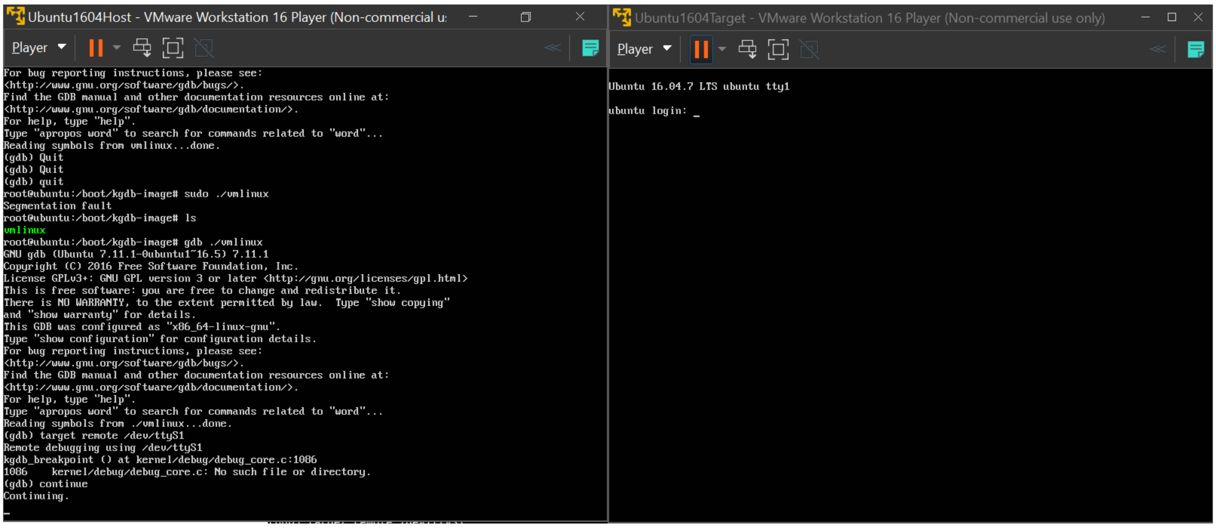



**Figure 7:** Connecting GDB

At this point, we need to start doing the performance measure on the functions, which means we need to connect our host machine's gdb to our target machine. Similar to the previous assignment, we need the gdb to remotely execute the debugging on the target machine. To do this, we use the `(gdb)target remote /dev/ttyS1`, which will send our commands to the `/dev/ttyS1` and thus to our target machine.

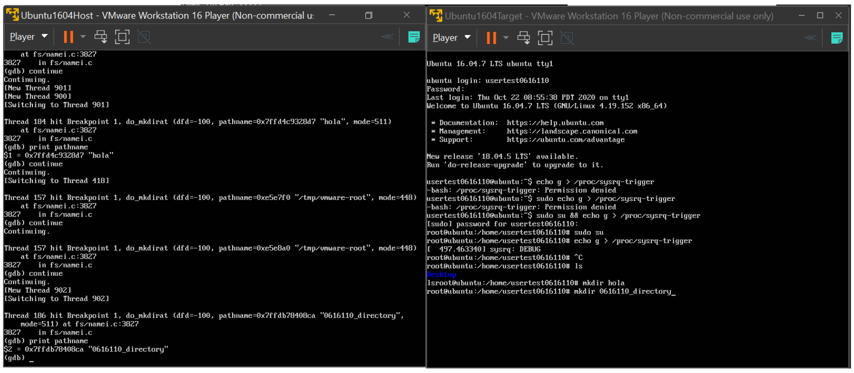


**Figure 8:** Pathname parameter of `mkdir`

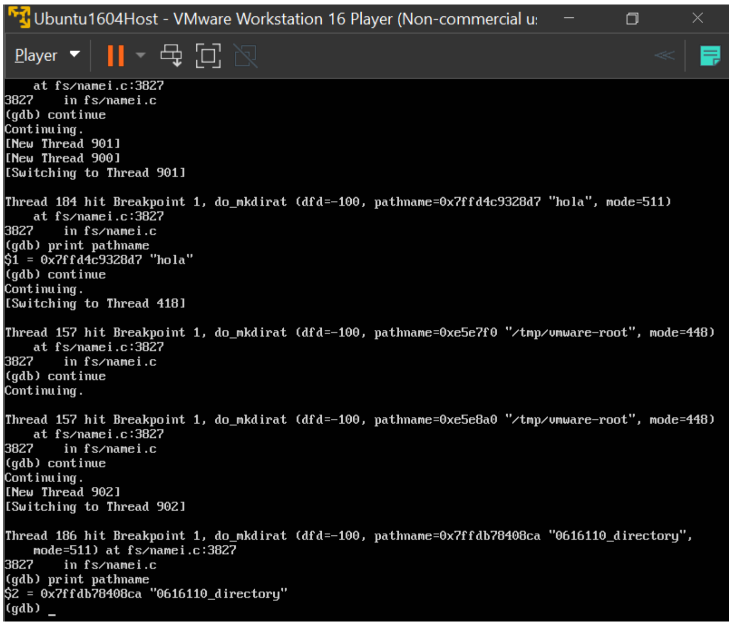



**Figure 9:** Host machine showing newly created directory

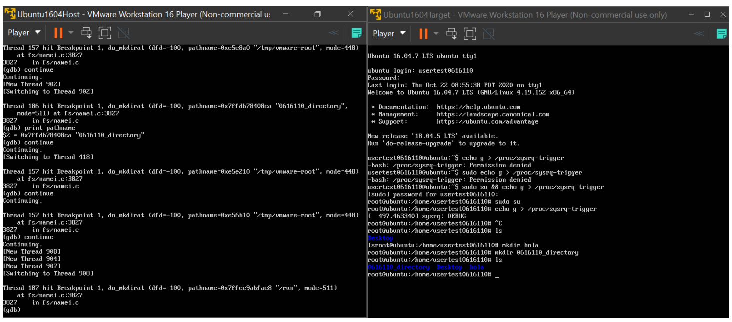


**Figure 10:** Created directory

The next pair of screenshots show the process of debugging the kernel `mkdir` function. After looking up the system call definition in `fs/namei.c`, we created a breakpoint at the corresponding system function `do_mkdirat`. Then, we test it by creating the `0616110_directory`. The gdb output is shown in the second screenshot. In the very last screenshot, the target machine has the newly created directory which we saw in the previous screenshot. In the meantime, since there was a breakpoint at the function, the target machine did not continue executing after the directory had been created and triggered the breakpoint. We had to constantly tell gdb to continue execution on the target machine.

### Do It Yourself 1

For the first DIY section of the assignment, I chose the common `rmdir` function. The reason I chose this function is becuase the debugging process was quite similar. From the implementation, it was also quite similar to the `mkir` command. It was also being implemented in the `fs/namei.c` file, and the naming convection of the system call was also quite similar.

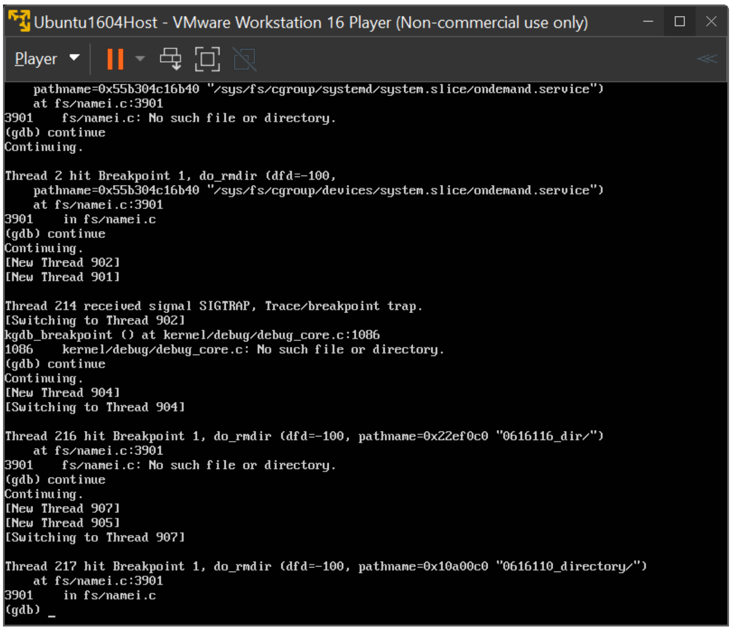



**Figure 11:** Image On And Responsive

In the first image, the gdb program already created the breakpoint at the `rmdir` execution, which is called by the `do_rmdir(...)` function call in the `fs/namei.c` file. We see the trigger caused by that function in the image, where I just typed **`continue`** to reclaim control of the target machine.

The way we trigger this breakpoint is similar to the one in the given example. In our Target machine, we remove a directory using the `rmdir` command in the terminal and this systemcall is then invoked. The `do_rmdir()` function gets called in the kernel which triggers our gdb instance to emit some information.

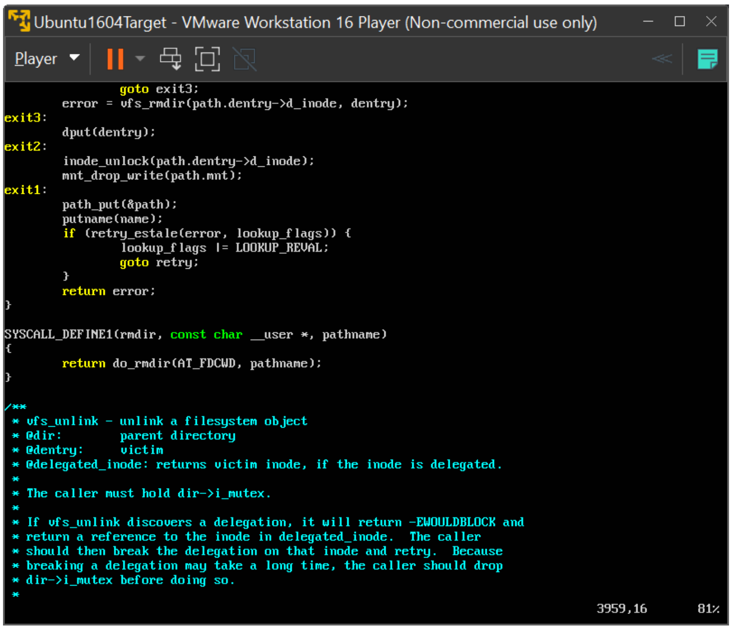



**Figure 12:** Creating the Breakpoint

For the breakpoint, we use the **`break`** `do_rmdir` command in GDB so that the Target machine stops executing. In the file where the systemcall is defined in the kernel (`fs/namei.c`) in this case, we see the actual C function being called, the `do_rmdir(...)` function. We then tell GDB to make the breakpoint when this function executes.

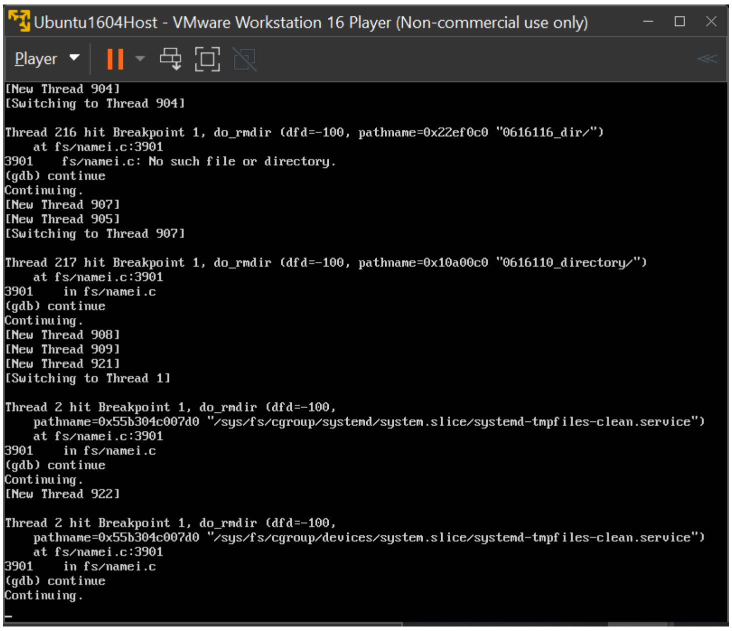



**Figure 13:** How to trigger

After we create the break point in GDB, we actually go ahead and execute the trigger. The result in the above screenshot is the Host machine, which tells us in which file the command executes, and in the topmost section of the screenshot, we can see the value of the parameters with which this function was called.

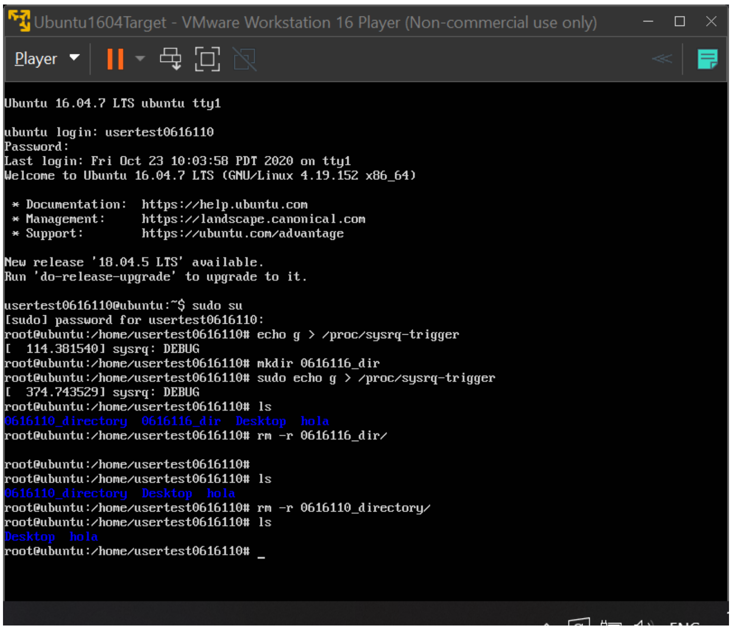



**Figure 14:** Done

In the final screenshot, we can just see the action is executed in our Target machine. The directory we wanted removed was deleted with the command and we were able to analyze the output in the GDB console on the Host machine.

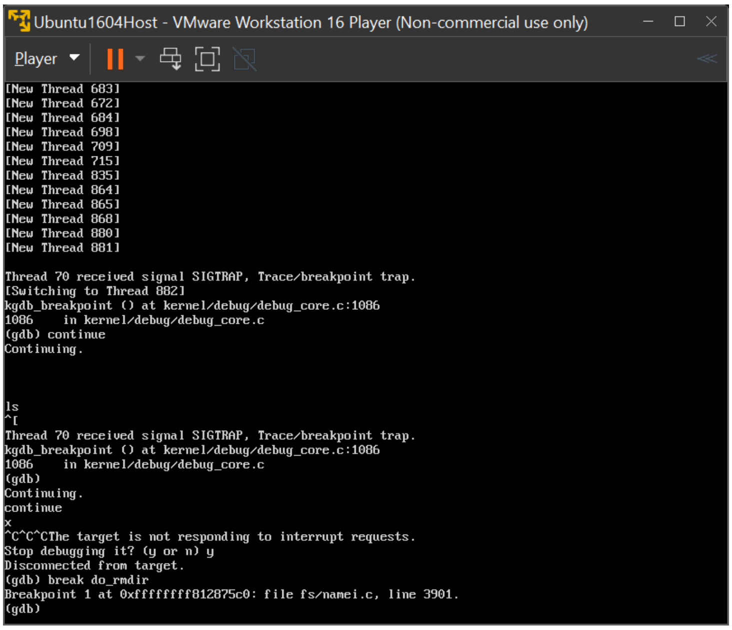



**Figure 15:** Triggering custom breakpoint

The above screenshot shows the gdb triggering the breakpoint when we execute the `rmdir` system call, or the `do_rmdir` call in C.o

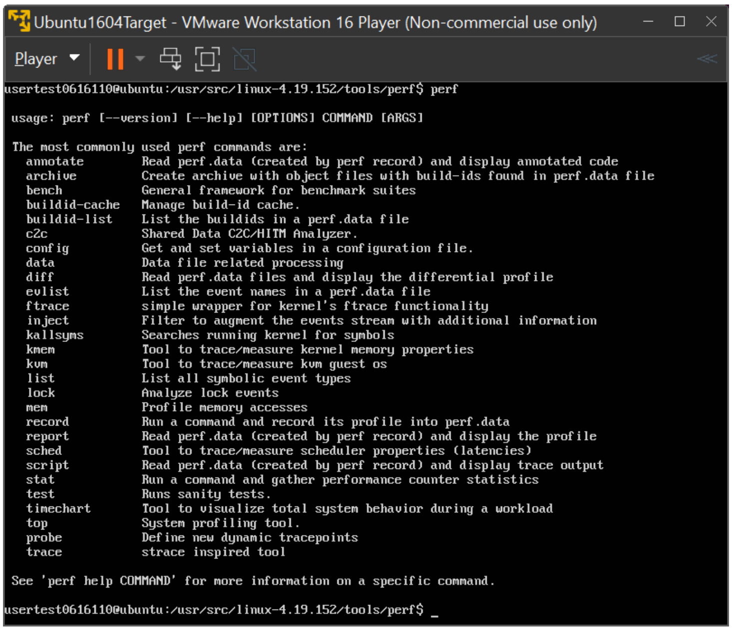



**Figure 16:** Perf command help

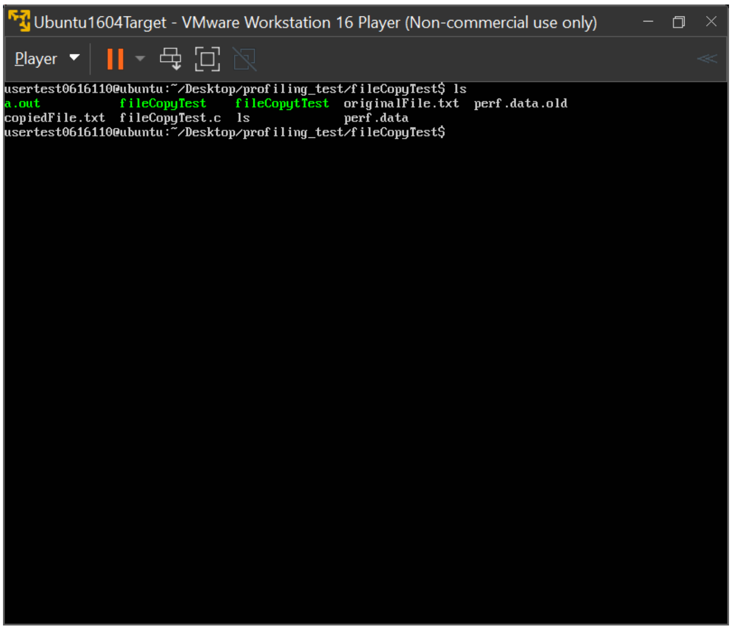



**Figure 17:** Copied file outputs

For the final section of the assignment, we have to measure the performance of an empty program and the program that copies the files, and compare which sections of code will execute regardless of program content. When we copy the file, after running the specified commands, we see the copied files with the same contents. The files and the copied files are seen in the second screenshot.

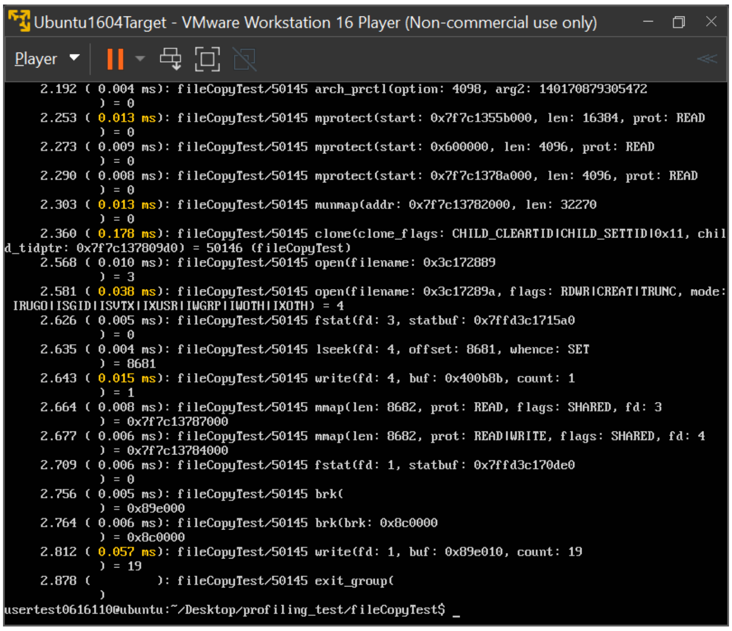



**Figure 18:** Perf program output

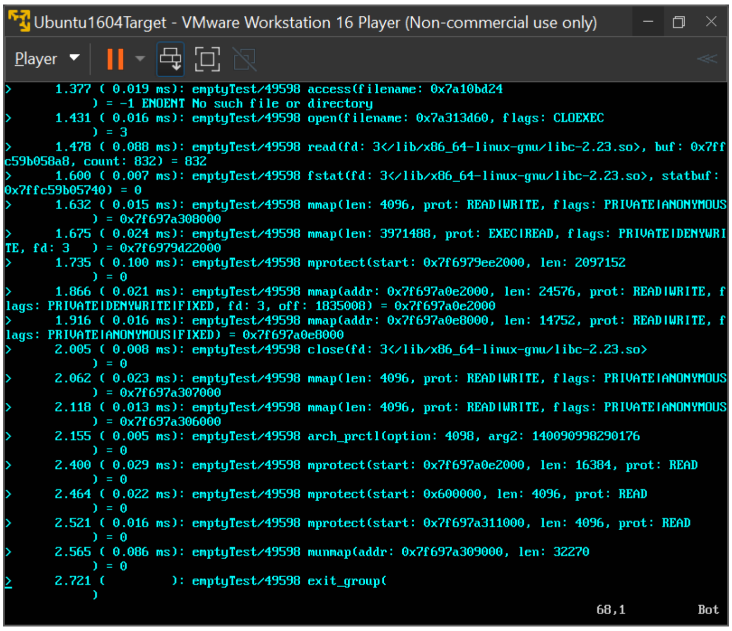



**Figure 19:** Diff program output

The `perf` program will measure all the system calls and the time it took to execute them. We can see the different system calls related to memory, writing, reading, opening files etc.. The results are shown in miliseconds, and we can profile the amount of time it takes to execute.

Once we execute the perf program on the empty C file and the C program that copies the programs, we see the difference between the two files using the `diff` program in Linux. This program will point the common and different lines in two files. When we use it on the two `perf` outputs, we can see the lines that they have in common, i.e. the calls that are common to the two files, and the lines that differ among them.

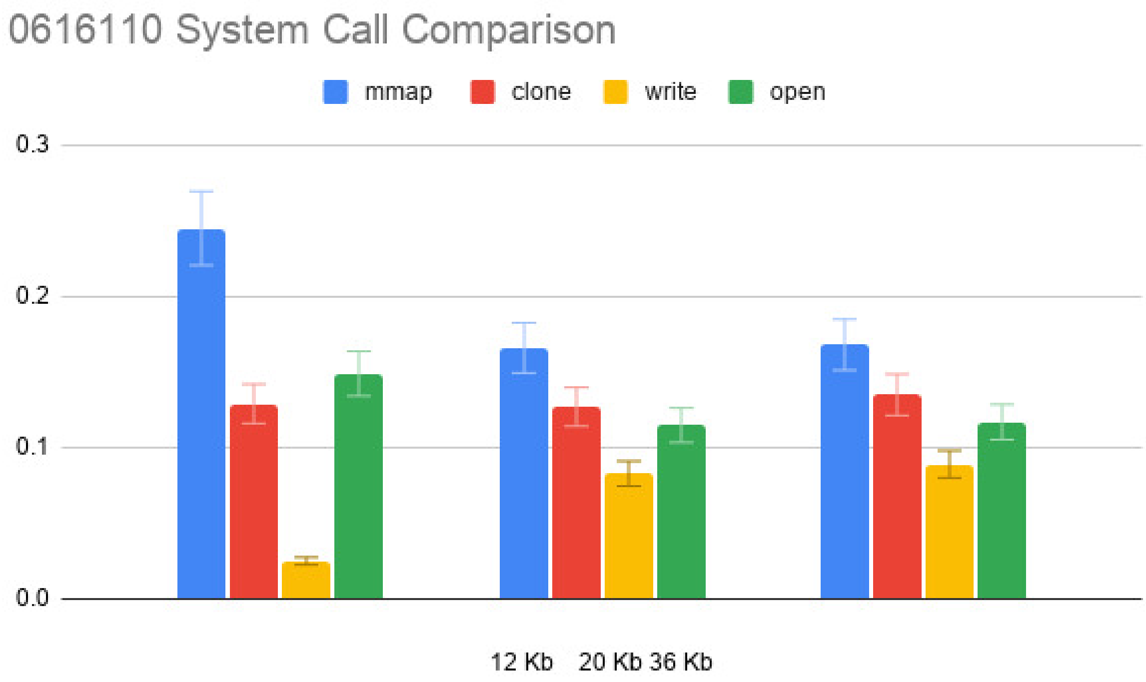



**Figure 20:** Impact of file sizes on system call time