

# **Operating Systems Homework 1 Report**

Andres Ponce, 0616110

2020-10-24

## Video Link

Click Me :D

## 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.
```

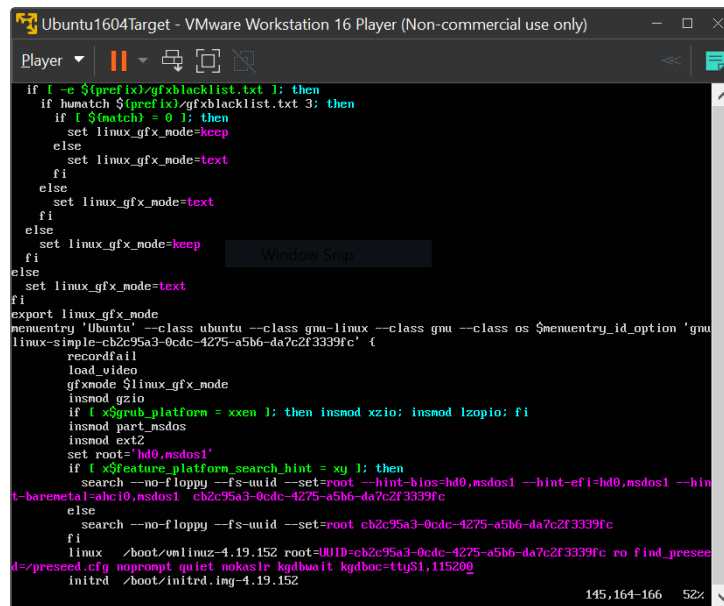
**Figure 1:** Some `perf` commands.

For example, we have the `perf stat` command, which will gather performance counter statistics on our program. 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

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

where *command* is a shell command.

## Screenshot Discussion



```
if [ -e "${prefix}/gfxblacklist.txt ]; then
  if ! grep -q "${prefix}/gfxblacklist.txt 3: then
    if [ $(cat /dev/urandom | fold -n 100 | tr -dc 'a-z0-9' | fold -n 64 | xargs sha1sum | cut -d ' ' -f 1) = 0 ]; then
      set linux_gfx_mode=keep
    else
      set linux_gfx_mode=text
    fi
  else
    set linux_gfx_mode=text
  fi
else
  set linux_gfx_mode=keep
fi
set linux_gfx_mode=text
fi
export linux_gfx_mode
menuentry 'Ubuntu' --class ubuntu --class gnu-linux --class gnu --class os $menuentry_id_option 'gnulinux-simple-cb2c95a3-0cdc-4275-a5b6-da7c2f3339fc' {
  recordfail
  load_video
  gfxmode $linux_gfx_mode
  insmod gzio
  if [ x$grub_platform = xxen ]; then insmod xzio; insmod lzopio; fi
  insmod part_msdos
  insmod ext2
  set root='hd0,msdos1'
  if [ x$feature_platform_search_hint = xy ]; then
    search --no-floppy --fs-uuid --set=root --hint-bios=hd0,msdos1 --hint-efi=hd0,msdos1 --hint-baremetal=ahci0,msdos1 cb2c95a3-0cdc-4275-a5b6-da7c2f3339fc
  else
    search --no-floppy --fs-uuid --set=root cb2c95a3-0cdc-4275-a5b6-da7c2f3339fc
  fi
  linux /boot/vmlinuz-4.19.152 root=UUID=cb2c95a3-0cdc-4275-a5b6-da7c2f3339fc ro find_preseed=/preseed.cfg noprompt quiet nokaslr kgdbwait kgdboc=ttyS1,115200
  initrd /boot/initrd.img-4.19.152
```

**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=ttyS1,115200`.

```

# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The __x64_sys_*() stubs are created on-the-fly for sys_*() system calls
#
# The abi is "common", "64" or "x32" for this file.
#
0      common read      __x64_sys_read
1      common write     __x64_sys_write
2      common open      __x64_sys_open
3      common close     __x64_sys_close
4      common stat       __x64_sys_newstat
5      common fstat      __x64_sys_newstat
6      common lstat      __x64_sys_newstat
7      common poll       __x64_sys_poll
8      common lseek      __x64_sys_lseek
9      common mmap       __x64_sys_mmap
10     common mprotect   __x64_sys_mprotect
11     common munmap     __x64_sys_munmap
12     common brk        __x64_sys_brk
13     64 rt_sigaction   __x64_sys_rt_sigaction
14     common rt_sigprocmask __x64_sys_rt_sigprocmask
15     64 rt_sigreturn   __x64_sys_rt_sigreturn/pregs
16     64 ioctl          __x64_sys_ioctl
17     common pread64     __x64_sys_pread64
18     common pwrite64    __x64_sys_pwrite64
19     64 readv          __x64_sys_readv
20     64 writev          __x64_sys_writev
21     common access     __x64_sys_access
22     common pipe       __x64_sys_pipe
23     common select      __x64_sys_select
24     common sched_yield __x64_sys_sched_yield
25     common mremap     __x64_sys_mremap
"syscall_64.tbl" [readonly] 388L, 15659C
1.1 Top

```

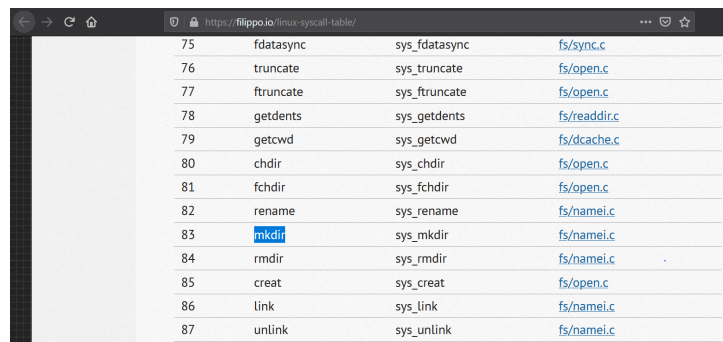
Figure 3: Syscall Table

System Call	Source File	Data Type	...	...	...	...
sys_oldumount	fs/super.c	char *	-	-	-	-
sys_setuid	kernel/sys.c	uid_t	-	-	-	-
sys_getuid	kernel/sched.c	-	-	-	-	-
sys_stime	kernel/time.c	int *	-	-	-	-
sys_ptrace	arch/386/kernel/ptrace.c	long	long	long	long	-
sys_alarm	kernel/sched.c	unsigned int	-	-	-	-
sys_fstat	fs/stat.c	unsigned int	struct __old_kernel_stat *	-	-	-
sys_pause	arch/386/kernel/sys_i386.c	-	-	-	-	-
sys_utime	fs/open.c	char *	struct utimbuf *	-	-	-
sys_access	fs/open.c	const char *	int	-	-	-
sys_nice	kernel/sched.c	int	-	-	-	-
sys_sync	fs/buffer.c	-	-	-	-	-
sys_kill	kernel/signal.c	int	int	-	-	-
sys_rename	fs/namei.c	const char *	const char *	-	-	-
sys_mkdir	fs/namei.c	const char *	int	-	-	-
sys_rmdir	fs/namei.c	const char *	-	-	-	-
sys_dup	fs/fcntl.c	unsigned int	-	-	-	-
sys_pipe	arch/386/kernel/sys_i386.c	unsigned long *	-	-	-	-

Figure 4: Online reference

System Call	Source File	Data Type	...	...	...	...
sys_stime	kernel/time.c	int *	-	-	-	-
sys_ptrace	arch/386/kernel/ptrace.c	long	long	long	long	-
sys_alarm	kernel/sched.c	unsigned int	-	-	-	-
sys_fstat	fs/stat.c	unsigned int	struct __old_kernel_stat *	-	-	-
sys_pause	arch/386/kernel/sys_i386.c	-	-	-	-	-
sys_utime	fs/open.c	char *	struct utimbuf *	-	-	-
sys_access	fs/open.c	const char *	int	-	-	-
sys_nice	kernel/sched.c	int	-	-	-	-
sys_sync	fs/buffer.c	-	-	-	-	-
sys_kill	kernel/signal.c	int	int	-	-	-
sys_rename	fs/namei.c	const char *	const char *	-	-	-
sys_mkdir	fs/namei.c	const char *	int	-	-	-
sys_rmdir	fs/namei.c	const char *	-	-	-	-
sys_dup	fs/fcntl.c	unsigned int	-	-	-	-
sys_pipe	arch/386/kernel/sys_i386.c	unsigned long *	-	-	-	-
sys_times	kernel/sys.c	struct tms *	-	-	-	-
sys_setgid	kernel/sys.c	gid_t	-	-	-	-
sys_getgid	kernel/sched.c	-	-	-	-	-

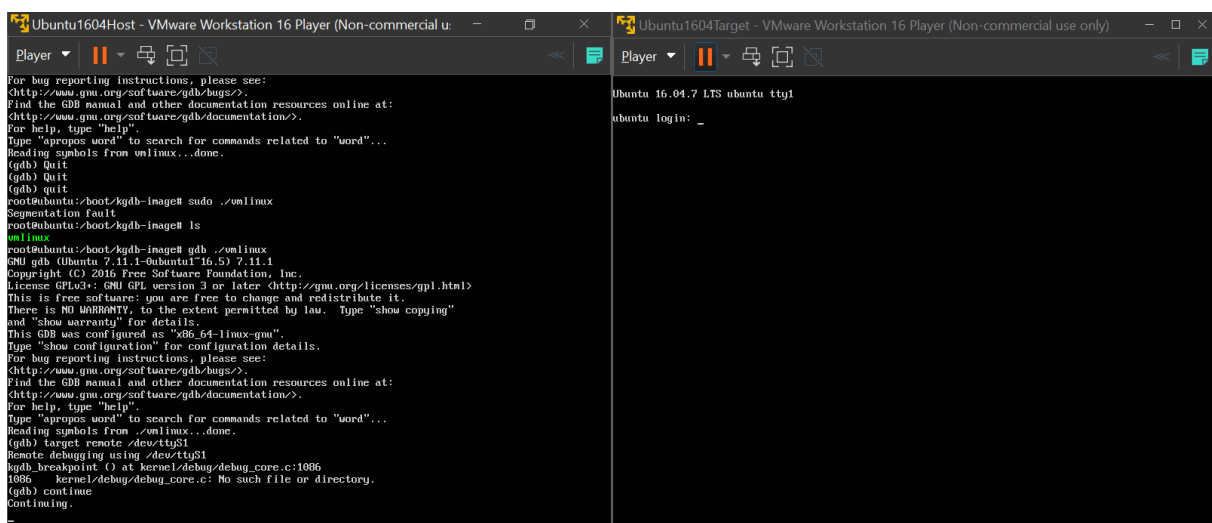
Figure 5: Online reference 2



75	fdatasync	sys_fdatasync	<a href="#">fs/sync.c</a>
76	truncate	sys_truncate	<a href="#">fs/open.c</a>
77	ftruncate	sys_ftruncate	<a href="#">fs/open.c</a>
78	getdents	sys_getdents	<a href="#">fs/readdir.c</a>
79	getcwd	sys_getcwd	<a href="#">fs/dcache.c</a>
80	chdir	sys_chdir	<a href="#">fs/open.c</a>
81	fchdir	sys_fchdir	<a href="#">fs/open.c</a>
82	rename	sys_rename	<a href="#">fs/namei.c</a>
83	<b>mkdir</b>	sys_mkdir	<a href="#">fs/namei.c</a>
84	rmdir	sys_rmdir	<a href="#">fs/namei.c</a>
85	creat	sys_creat	<a href="#">fs/open.c</a>
86	link	sys_link	<a href="#">fs/namei.c</a>
87	unlink	sys_unlink	<a href="#">fs/namei.c</a>

**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 the 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 influence the location of the file we need.



**Figure 7:** Connecting GDB

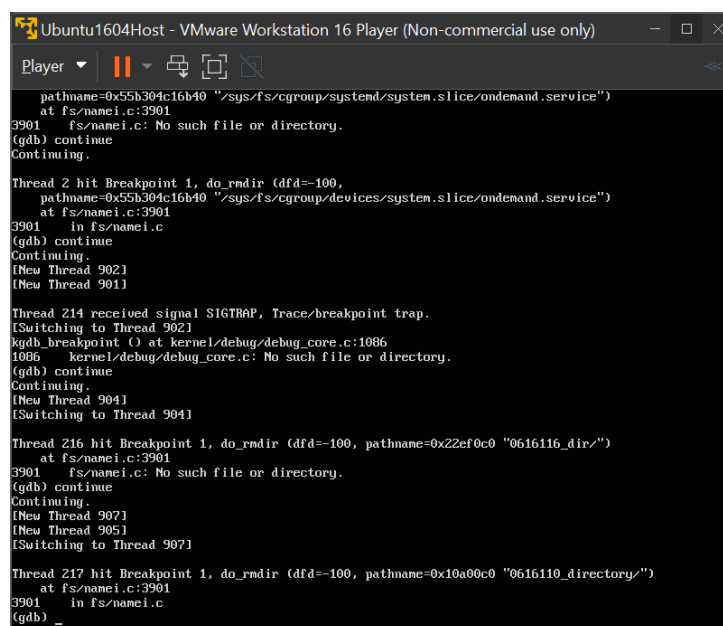
At this point, we need to start doing the performance measure on the functions, which means we need to connect our host machine's gdb to our target machine. Similar to the previous assignment, we need the gdb to remotely execute the debugging on the target machine. To do this, we use the `(gdb)target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



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 because 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 convention of the system call was also quite similar.



```

pathname=0x55b304c16b40 "/sys/fs/cgroup/systemd/system.slice/ondemand.service")
at fs/namei.c:3901
3901 fs/namei.c: No such file or directory.
(gdb) continue
Continuing.

Thread 2 hit Breakpoint 1, do_rmdir (dfd=-100,
pathname=0x55b304c16b40 "/sys/fs/cgroup/devices/system.slice/ondemand.service")
at fs/namei.c:3901
3901 in fs/namei.c
(gdb) continue
Continuing.
[New Thread 902]
[New Thread 901]

Thread 214 received signal SIGTRAP, Trace/breakpoint trap.
[Switching to Thread 902]
kgdb_breakpoint () at kernel/debug/debug_core.c:1086
1086 kernel/debug/debug_core.c: No such file or directory.
(gdb) continue
Continuing.
[New Thread 904]
[Switching to Thread 904]

Thread 216 hit Breakpoint 1, do_rmdir (dfd=-100, pathname=0x22ef0c0 "0616116_dir/")
at fs/namei.c:3901
3901 fs/namei.c: No such file or directory.
(gdb) continue
Continuing.
[New Thread 907]
[New Thread 905]
[Switching to Thread 907]

Thread 217 hit Breakpoint 1, do_rmdir (dfd=-100, pathname=0x10a00c0 "0616110_directory/")
at fs/namei.c:3901
3901 in fs/namei.c
(gdb) _

```

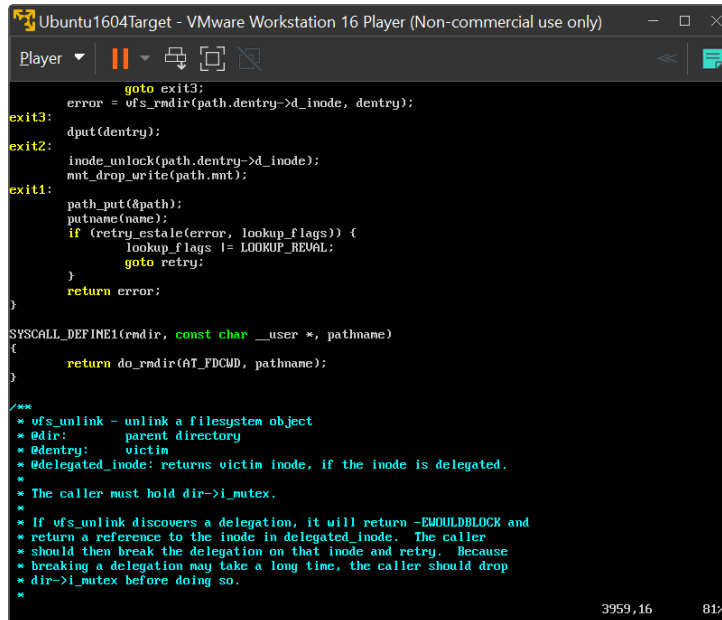
**Figure 11:** Image On And Responsive

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 syscall is then invoked. The `do_rmdir()` function gets called in the kernel which triggers our gdb instance to emit some information.



```

goto exit3;
error = vfs_rmdir(path.dentry->d_inode, dentry);
exit3:
dput(dentry);
exit2:
inode_unlock(path.dentry->d_inode);
mnt_drop_write(path.mnt);
exit1:
path_put(&path);
putname(name);
if (retry_estale(error, lookup_flags)) {
    lookup_flags |= LOOKUP_REVAL;
    goto retry;
}
return error;
}

SYSCALL_DEFINE1(rmdir, const char __user *, pathname)
{
    return do_rmdir(AT_FDCWD, pathname);
}

/**
 * vfs_unlink - unlink a filesystem object
 * @dir:      parent directory
 * @dentry:   victim
 * @delegated_inode: returns victim inode, if the inode is delegated.
 *
 * The caller must hold dir->i_mutex.
 *
 * If vfs_unlink discovers a delegation, it will return -EWOULDBLOCK and
 * return a reference to the inode in delegated_inode. The caller
 * should then break the delegation on that inode and retry. Because
 * breaking a delegation may take a long time, the caller should drop
 * dir->i_mutex before doing so.
 */

```

**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 syscall 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.

```

[New Thread 904]
[Switching to Thread 904]

Thread 216 hit Breakpoint 1, do_rmdir (dfd=-100, pathname=0x22ef0c0 "0616116_dir/")
  at fs/namei.c:3901
3901  fs/namei.c: No such file or directory.
(gdb) continue
Continuing.
[New Thread 907]
[New Thread 905]
[Switching to Thread 907]

Thread 217 hit Breakpoint 1, do_rmdir (dfd=-100, pathname=0x10a00c0 "0616110_directory/")
  at fs/namei.c:3901
3901  in fs/namei.c
(gdb) continue
Continuing.
[New Thread 908]
[New Thread 909]
[New Thread 921]
[Switching to Thread 1]

Thread 2 hit Breakpoint 1, do_rmdir (dfd=-100,
  pathname=0x55b304c007d0 "/sys/fs/cgroup/systemd/system.slice/systemd-tmpfiles-clean.service")
  at fs/namei.c:3901
3901  in fs/namei.c
(gdb) continue
Continuing.
[New Thread 922]

Thread 2 hit Breakpoint 1, do_rmdir (dfd=-100,
  pathname=0x55b304c007d0 "/sys/fs/cgroup/devices/system.slice/systemd-tmpfiles-clean.service")
  at fs/namei.c:3901
3901  in fs/namei.c
(gdb) continue
Continuing.

```

**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.

```

Ubuntu 16.04.7 LTS ubuntu tty1
ubuntu login: usertest0616110
Password:
Last login: Fri Oct 23 10:03:58 PDT 2020 on tty1
Welcome to Ubuntu 16.04.7 LTS (GNU/Linux 4.19.152 x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

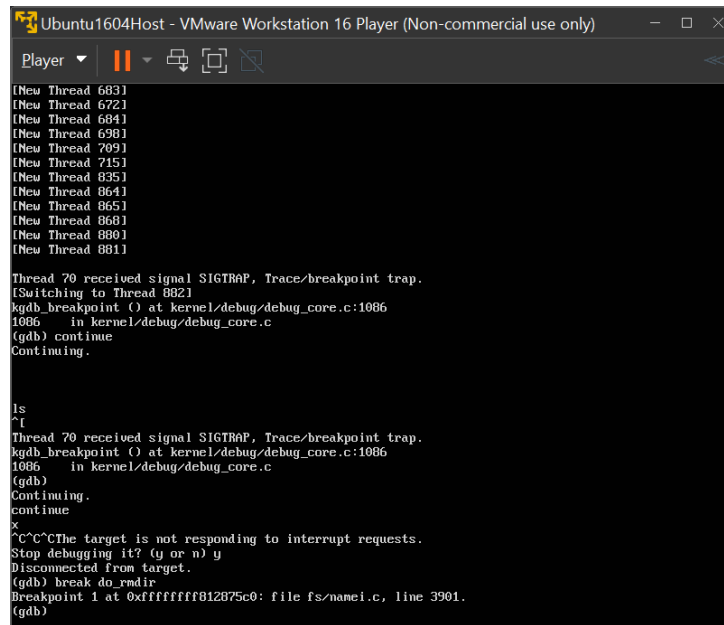
New release '18.04.5 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

usertest0616110@ubuntu:~$ sudo su
[sudo] password for usertest0616110:
root@ubuntu:/home/usertest0616110# echo g > /proc/sysrq-trigger
[ 114.381540] sysrq: DEBUG
root@ubuntu:/home/usertest0616110# mkdir 0616116_dir
root@ubuntu:/home/usertest0616110# sudo echo g > /proc/sysrq-trigger
[ 374.743529] sysrq: DEBUG
root@ubuntu:/home/usertest0616110# ls
0616110_directory 0616116_dir Desktop hola
root@ubuntu:/home/usertest0616110# rm -r 0616116_dir/
root@ubuntu:/home/usertest0616110#
root@ubuntu:/home/usertest0616110# ls
0616110_directory Desktop hola
root@ubuntu:/home/usertest0616110# rm -r 0616110_directory/
root@ubuntu:/home/usertest0616110# ls
Desktop hola
root@ubuntu:/home/usertest0616110# _

```

**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.



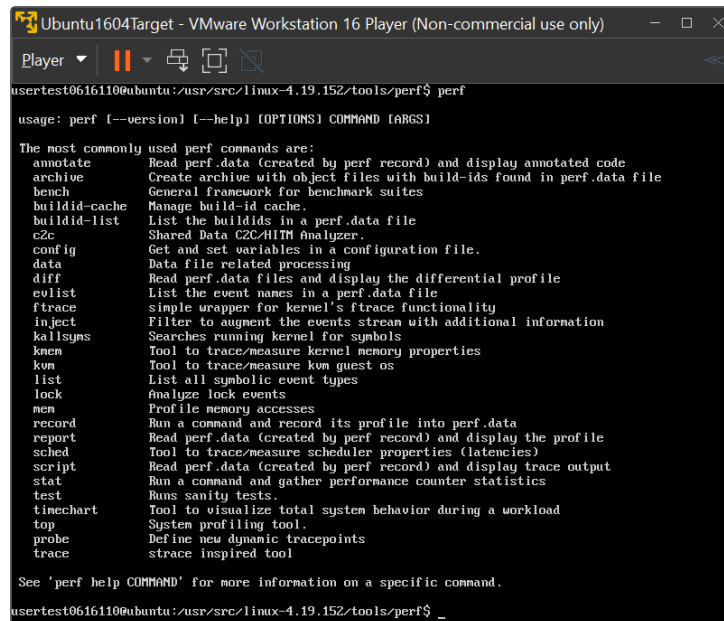
```
Ubuntu1604Host - VMware Workstation 16 Player (Non-commercial use only)
Player
[New Thread 6831]
[New Thread 6721]
[New Thread 6841]
[New Thread 6981]
[New Thread 7091]
[New Thread 7151]
[New Thread 8351]
[New Thread 8641]
[New Thread 8651]
[New Thread 8681]
[New Thread 8801]
[New Thread 8811]

Thread 70 received signal SIGTRAP, Trace/breakpoint trap.
[Switching to Thread 8821]
kgdb_breakpoint () at kernel/debug/debug_core.c:1086
1086      in kernel/debug/debug_core.c
(gdb) continue
Continuing.

ls
^[
Thread 70 received signal SIGTRAP, Trace/breakpoint trap.
kgdb_breakpoint () at kernel/debug/debug_core.c:1086
1086      in kernel/debug/debug_core.c
(gdb)
Continuing.
continue
x
^C^CThe target is not responding to interrupt requests.
Stop debugging it? (y or n) y
Disconnected from target.
(gdb) break do_rmdir
Breakpoint 1 at 0xffffffff812875c0: file fs/namei.c, line 3901.
(gdb)
```

**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

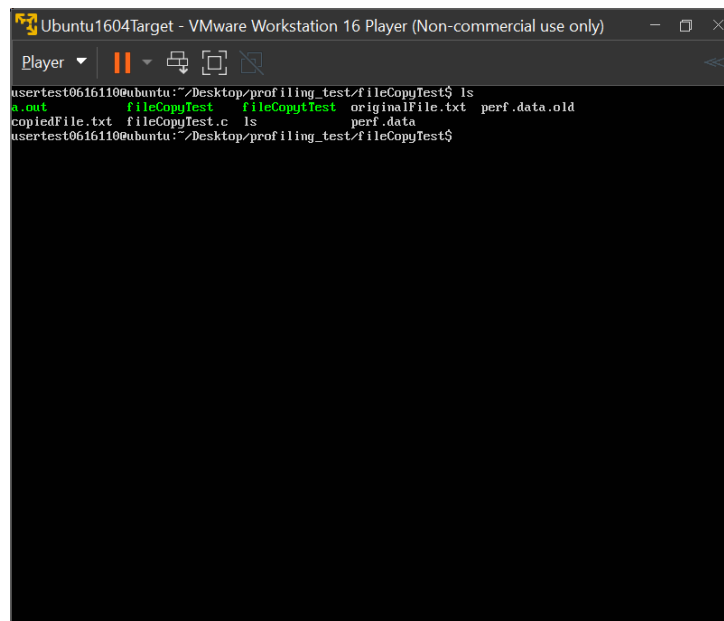


```
Ubuntu1604Target - VMware Workstation 16 Player (Non-commercial use only)
Player
usertest0616110@ubuntu:/usr/src/linux-4.19.152/tools/perf$ perf

usage: perf [--version] [--help] [OPTIONS] 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 build-id cache.
  buildid-list  List the buildids in a perf.data file
  c2c          Shared Data C2C/ALTM Analyzer.
  config       Get and set variables in a configuration file.
  data         Data file related processing
  diff         Read perf.data files and display the differential profile
  evlist       List the event names in a perf.data file
  ftrace       simple wrapper for kernel's ftrace functionality
  inject       Filter to augment the events stream with additional information
  kallsyms     Searches running kernel for symbols
  kmem         Tool to trace/measure kernel memory properties
  kvm          Tool to trace/measure kvm guest os
  list         List all symbolic event types
  lock         Analyze lock events
  mem          Profile memory accesses
  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.
  probe        Define new dynamic tracepoints
  trace        strace inspired tool

See 'perf help COMMAND' for more information on a specific command.
usertest0616110@ubuntu:/usr/src/linux-4.19.152/tools/perf$ _
```

**Figure 16:** Perf command help

```
Ubuntu1604Target - VMware Workstation 16 Player (Non-commercial use only)
Player
usertest0616110@ubuntu:~/Desktop/profiling_test/fileCopyTest$ ls
a.out      fileCopyTest  fileCopyTest  originalFile.txt  perf.data.old
copiedFile.txt  fileCopyTest.c  ls            perf.data
usertest0616110@ubuntu:~/Desktop/profiling_test/fileCopyTest$
```

**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.

```

2.192 ( 0.004 ms): fileCopyTest/50145 arch_prctl(option: 4098, arg2: 140170879305472
) = 0
2.253 ( 0.013 ms): fileCopyTest/50145 mprotect(start: 0x7f7c1355b000, len: 16384, prot: READ
) = 0
2.273 ( 0.009 ms): fileCopyTest/50145 mprotect(start: 0x600000, len: 4096, prot: READ
) = 0
2.290 ( 0.008 ms): fileCopyTest/50145 mprotect(start: 0x7f7c1378a000, len: 4096, prot: READ
) = 0
2.303 ( 0.013 ms): fileCopyTest/50145 munmap(addr: 0x7f7c13782000, len: 32270
) = 0
2.360 ( 0.178 ms): fileCopyTest/50145 clone(clone_flags: CHILD_CLEARID|CHILD_SETTID|0x11, chil
d_tidptr: 0x7f7c137809d0) = 50146 (fileCopyTest)
2.568 ( 0.010 ms): fileCopyTest/50145 open(filename: 0x3c172809
) = 3
2.581 ( 0.038 ms): fileCopyTest/50145 open(filename: 0x3c17289a, flags: RDWR|CREAT|TRUNC, mode:
IRUGO|ISGID|ISUTX|IXUSR|IXGRP|IXOTH|IXOTH) = 4
2.626 ( 0.005 ms): fileCopyTest/50145 fstat(fd: 3, statbuf: 0x7ff43c1715a0
) = 0
2.635 ( 0.004 ms): fileCopyTest/50145 lseek(fd: 4, offset: 0601, whence: SET
) = 0601
2.643 ( 0.015 ms): fileCopyTest/50145 write(fd: 4, buf: 0x400b0b, count: 1
) = 1
2.664 ( 0.008 ms): fileCopyTest/50145 mmap(len: 0602, prot: READ, flags: SHARED, fd: 3
) = 0x7f7c13787000
2.677 ( 0.006 ms): fileCopyTest/50145 mmap(len: 0602, prot: READ|WRITE, flags: SHARED, fd: 4
) = 0x7f7c13784000
2.709 ( 0.006 ms): fileCopyTest/50145 fstat(fd: 1, statbuf: 0x7ff43c170de0
) = 0
2.756 ( 0.005 ms): fileCopyTest/50145 brk(
) = 0x09e000
2.764 ( 0.006 ms): fileCopyTest/50145 brk(brk: 0x8c0000
) = 0x8c0000
2.812 ( 0.057 ms): fileCopyTest/50145 write(fd: 1, buf: 0x09e010, count: 19
) = 19
2.878 (
): fileCopyTest/50145 exit_group(
)
usertest0616110@ubuntu:~/Desktop/profiling_test/fileCopyTest$ _

```

**Figure 18:** Perf program output

```

> 1.377 ( 0.019 ms): emptyTest/49598 access(filename: 0x7a10bd24
) = -1 EIO No such file or directory
> 1.431 ( 0.016 ms): emptyTest/49598 open(filename: 0x7a313d60, flags: CLOEXEC
) = 3
> 1.478 ( 0.088 ms): emptyTest/49598 read(fd: 3</lib/x86_64-linux-gnu/libc-2.23.so>, buf: 0x7ff
c59b058a0, count: 832) = 832
> 1.600 ( 0.007 ms): emptyTest/49598 fstat(fd: 3</lib/x86_64-linux-gnu/libc-2.23.so>, statbuf:
0x7ffcc59b05740) = 0
> 1.632 ( 0.015 ms): emptyTest/49598 mmap(len: 4096, prot: READ|WRITE, flags: PRIVATE|ANONYMOUS
) = 0x7f697a308000
> 1.675 ( 0.024 ms): emptyTest/49598 mmap(len: 3971488, prot: EXEC|READ, flags: PRIVATE|DENYWRIT
E, fd: 3) = 0x7f6979d22000
> 1.735 ( 0.100 ms): emptyTest/49598 mprotect(start: 0x7f6979ee2000, len: 2097152
) = 0
> 1.866 ( 0.021 ms): emptyTest/49598 mmap(addr: 0x7f697a0e2000, len: 24576, prot: READ|WRITE, f
lags: PRIVATE|DENYWRITE|FIXED, fd: 3, off: 1835008) = 0x7f697a0e2000
> 1.916 ( 0.016 ms): emptyTest/49598 mmap(addr: 0x7f697a0e8000, len: 14752, prot: READ|WRITE, f
lags: PRIVATE|ANONYMOUS|FIXED) = 0x7f697a0e8000
> 2.005 ( 0.008 ms): emptyTest/49598 close(fd: 3</lib/x86_64-linux-gnu/libc-2.23.so>)
) = 0
> 2.062 ( 0.023 ms): emptyTest/49598 mmap(len: 4096, prot: READ|WRITE, flags: PRIVATE|ANONYMOUS
) = 0x7f697a307000
> 2.110 ( 0.013 ms): emptyTest/49598 mmap(len: 4096, prot: READ|WRITE, flags: PRIVATE|ANONYMOUS
) = 0x7f697a306000
> 2.155 ( 0.005 ms): emptyTest/49598 arch_prctl(option: 4098, arg2: 140090998290176
) = 0
> 2.400 ( 0.029 ms): emptyTest/49598 mprotect(start: 0x7f697a0e2000, len: 16384, prot: READ
) = 0
> 2.464 ( 0.022 ms): emptyTest/49598 mprotect(start: 0x600000, len: 4096, prot: READ
) = 0
> 2.521 ( 0.016 ms): emptyTest/49598 mprotect(start: 0x7f697a311000, len: 4096, prot: READ
) = 0
> 2.565 ( 0.086 ms): emptyTest/49598 munmap(addr: 0x7f697a309000, len: 32270
) = 0
> 2.721 (
): emptyTest/49598 exit_group(
)
68,1 Bot

```

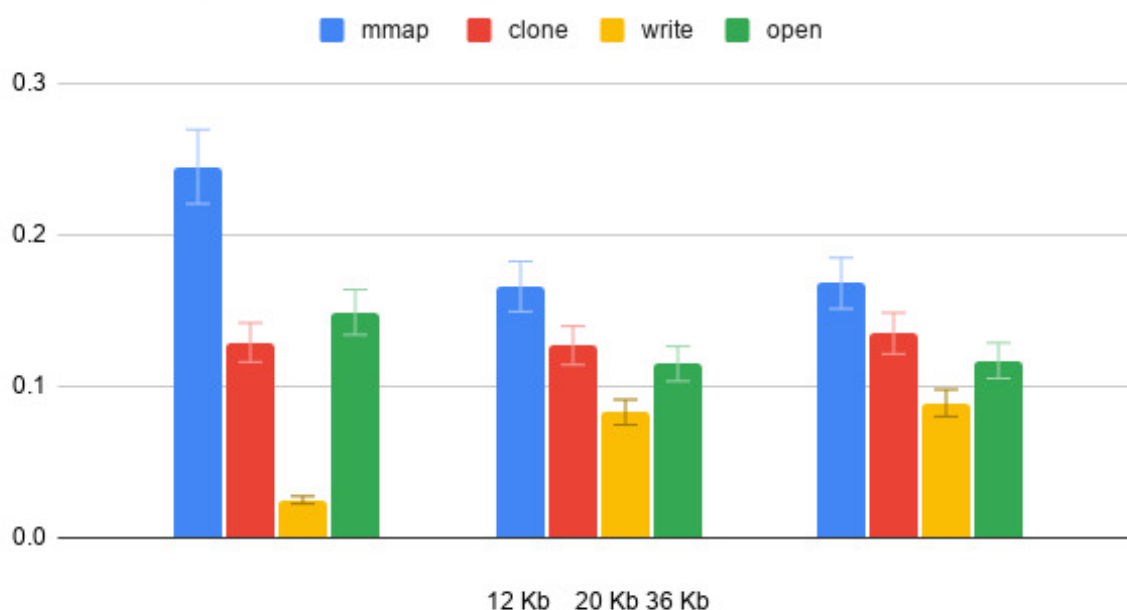
**Figure 19:** Diff program output

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 milliseconds, and we can profile the amount of time it takes to execute.

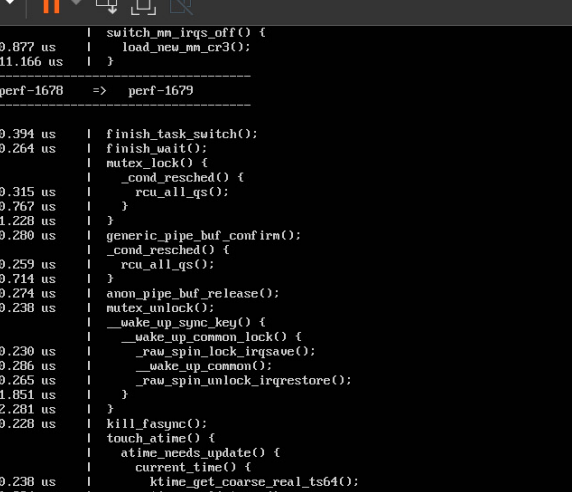
Once we execute the `perf` program on the empty C file and the C program that copies the programs, we see the difference between the two files using the `diff` program in Linux. This program will point the common and different lines in two files. When we use it on the two `perf` outputs, we can see the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

### 0616110 System Call Comparison



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

The final analysis of the execution time for the system calls used reveal the relative invariance of the time taken by the system calls regardless of file size. The file sizes were 12 Kb, 20 Kb, and 36 Kb, respectively. The text used was just the contents of `OriginalFile.txt` copied multiple times. The only syscall that showed a spike in time required was the `mmap` for a file size of 12 Kb.



Ubuntu1604Target - VMware Workstation 16 Player (Non-commercial use only)

Player ▾ || ▾ ↵ ↶ ↷

```

0) | switch_mm_irqs_off() {
0) 0.877 us | | load_new_mm_cr3();
0) + 11.166 us | | }

0) perf-1678 => perf-1679

0) 0.394 us | | finish_task_switch();
0) 0.264 us | | finish_wait();
0) | | mutex_lock() {
0) | | _cond_resched() {
0) 0.315 us | | | rcu_all_qs();
0) 0.767 us | | }
0) 1.228 us | | }
0) 0.280 us | | generic_pipe_buf_confirm();
0) | | _cond_resched() {
0) 0.259 us | | | rcu_all_qs();
0) 0.714 us | | }
0) 0.274 us | | anon_pipe_buf_release();
0) 0.238 us | | mutex_unlock();
0) | | _wake_up_sync_key() {
0) | | | _wake_up_common_lock() {
0) 0.230 us | | | | _raw_spin_lock_irqsave();
0) 0.206 us | | | | _wake_up_common();
0) 0.265 us | | | | _raw_spin_unlock_irqrestore();
0) 1.851 us | | | }
0) 2.281 us | | }
0) 0.228 us | | kill_fasync();
0) | | touch_atime() {
0) | | | atime_needs_update() {
0) | | | | current_time() {
0) 0.230 us | | | | | ktime_get_coarse_real_ts64();
0) 0.234 us | | | | | timespec64_trunc();
0) 1.146 us | | | | }
0) 1.640 us | | | }
0) 0.257 us | | | _sb_start_write();

```

**Figure 22:** sudo perf ftrace

The `sudo perf ftrace` command shows the function call stack. We can see all the functions, and which functions are called inside any particular function. This is incredibly useful if we are experiencing unexpected resource utilization or unexpected behavior.