CHAPTER 3
INTRODUCTION TO LINUX AND SYSTEM CALLS

Learning Objectives

1. Students are able to use frequently used commands in the Linux terminal.
2. Students can use the system calls provided by Linux operating systems.

Introduction

Linux is a family of open-source operating systems based on the Linux kernel. The Linux kernel itself was released for the first time in 1991 by Linus Torvalds and is maintained to this day. To date, there are many Linux distributions that shipped the kernel with their own set of applications. Some examples of well-known Linux distributions are Ubuntu, Fedora, and Debian.

In Linux operating systems, there are two modes of operation:

- Kernel mode is the privileged mode that provides system call interfaces to manage processes and hardware.
- User Mode is the regular mode in which users can run their typical applications, such as a web browser, word processor, or terminal. When a user application requires elevated access, the necessary system calls are called and there will be a switch from user mode to kernel mode.

As presented in Figure 1, system calls serve as interfaces for user or system programs or applications to access and manage computer and device resources. If a user application requires access to a device, it will call the necessary system calls and switch from user mode to kernel mode. Using system calls is similar to calling a function. System calls also accepting arguments, process them, and return values. The key difference between system calls and regular function calls is that system calls silently interact with the kernel in kernel mode to complete the required tasks.

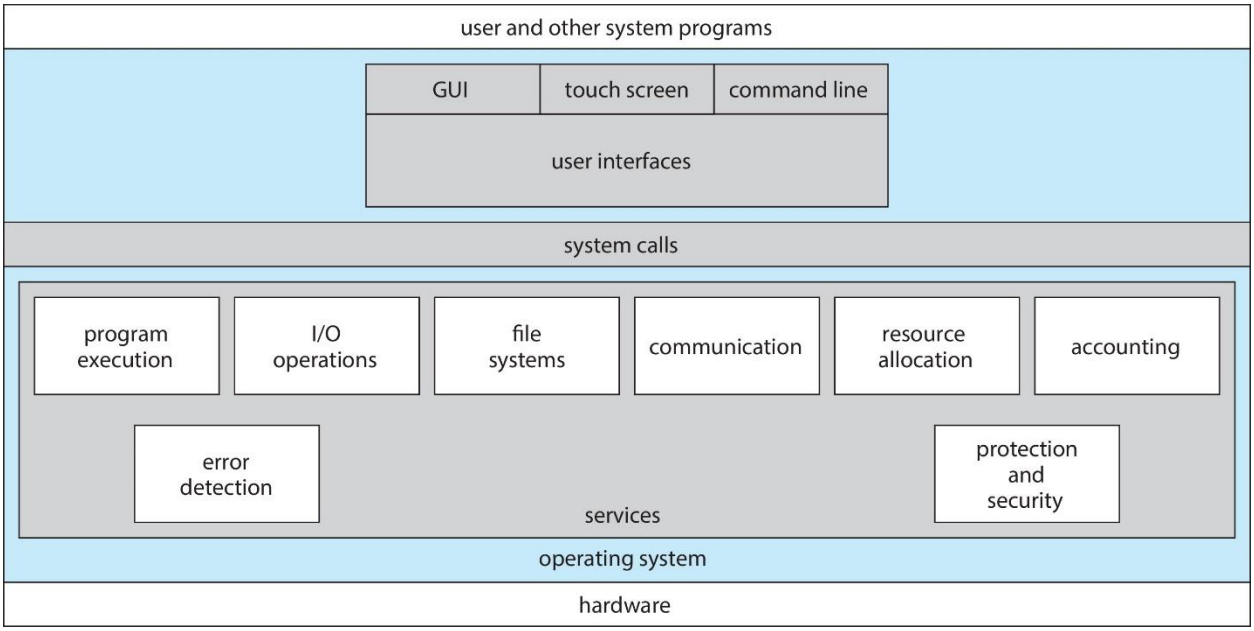

Figure 1 System calls in operating systems (taken from [1])

In general, there are several types of system calls in operating systems [1]:

- Process control system calls, such as creating and terminating process, end or aborting, etc.
- File management system calls, such as create, open, close, delete file.
- Device management system calls, such as request and release device.
- Information maintenance system calls, such as get time or date, get, and set system data.
- Communications system calls, such as creating and deleting connections, sending, and receiving messages.
- Protection system calls, such as control access to resources, get and set permissions.

In this session of lab work, we use Ubuntu as a distribution of the Linux operating system to demonstrate the use of system calls. First, we try and discuss some Linux terminal commands. Then, we explore how these commands use system calls. Finally, we use some libraries to develop simple programs that utilize system calls.

3.1 Selected Linux Terminal Commands

The Linux terminal, also known as the shell, is a program that receives commands from a user, processes them, and returns output. Although modern Linux operating systems provide graphical user interfaces (GUI), the terminal is still an important part of the operating system. Because we are using Ubuntu, we can use the shortcut "Ctrl+Alt+T" to open the terminal window. Alternatively, you can search for "terminal" in the Ubuntu menu. Figure 2 shows an example of a

terminal in an Ubuntu derivative. The remainder of this subsection describes some selected terminal commands.

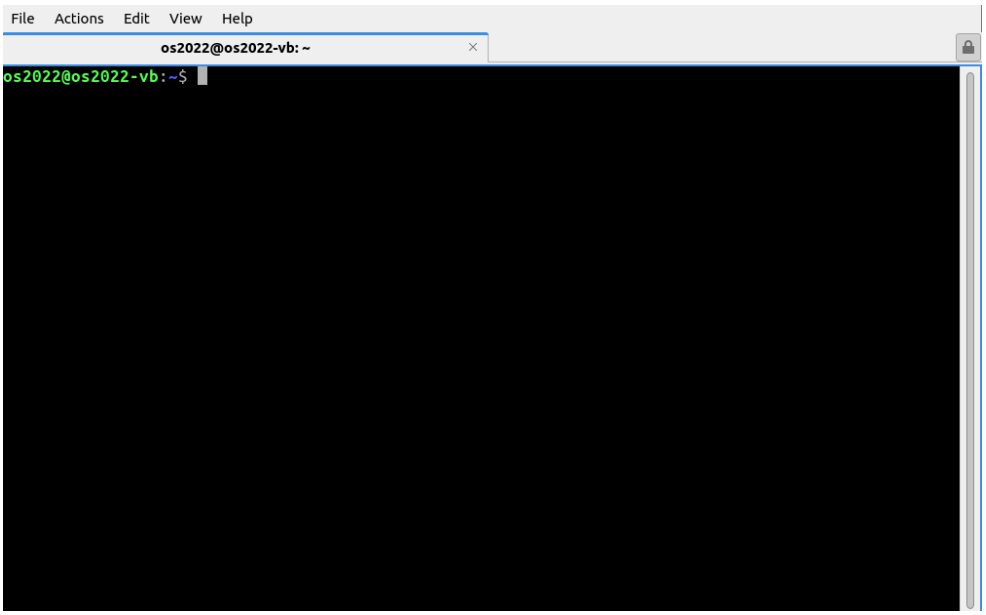

Figure 2 Terminal window

1. **pwd** is a command to display the current directory in which we are in. The directory structure of Linux operating systems starts with the root directory, denoted by the forward slash symbol (/), followed by the directory path to each file or folder. For example, if we start the terminal for the first time, our default directory is our home directory (/home/username). Thus, if we run pwd, the terminal will display this path.

```
os2022@os2022-vb:~$ pwd
/home/os2022
```

   This result above means that our current directory is in the os2022 directory located in the home directory.

2. **ls** is a command to list all files and folders located in our current directory. For example, when we run ls on our home directory, the terminal shows all directories located within.

```
os2022@os2022-vb:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
```

   We can also list all files and folders in a particular directory path by specifying the path after the ls command. For example, listing all files and directories within the root could be done as follows:

```
os2022@os2022-vb:~$ ls /
```

```
bin   dev  home  lib32  libx32      media  opt   root  sbin  srv       sys
usr
boot  etc  lib   lib64  lost+found  mnt    proc  run   snap  swapfile  tmp
var
```

We can also see the type and access level of each file and directory using the option -l.

```
os2022@os2022-vb:~$ ls -l
total 32
drwxrwxr-x 2 os2022 os2022 4096 Sep  4 11:39 Desktop
drwxr-xr-x 2 os2022 os2022 4096 Sep  4 11:25 Documents
drwxr-xr-x 2 os2022 os2022 4096 Sep  4 11:25 Downloads
drwxr-xr-x 2 os2022 os2022 4096 Sep  4 11:25 Music
drwxr-xr-x 2 os2022 os2022 4096 Sep  4 11:25 Pictures
drwxr-xr-x 2 os2022 os2022 4096 Sep  4 11:25 Public
drwxr-xr-x 2 os2022 os2022 4096 Sep  4 11:25 Templates
drwxr-xr-x 2 os2022 os2022 4096 Sep  4 11:25 Videos
```

The type and access level of files and directories in Linux are explained as follows:

(1) the first character denotes whether the corresponding item is a directory (denoted by d) or a file (denoted by -). In this case, all items presented above are directories.

(2) the next three characters represent the access level for the owner of the corresponding item. In our case above, the owner of the items presented (os2022) has read, write, and execute permission.

(3) the next three characters represent the access level for the group members of the corresponding item. In our case above, the members of the os2022 group can read, write, and execute the Desktop directory, but can only read and execute on the other directories.

(4) the last three characters represent the access level for the other users. In our case above, other users have read and execute access only to all directories.

3. **chmod** is a command to change the access level of files and directories. Each group of three characters above is represented as a single number from the binary value of the characters. For example, if read, write, and execute are permitted for a file, then this will translate into a value of 7 (or 111 in binary). Another example, read and execute access will translate into a value of 5 (or 101 in binary, 0 for write access). To change the access level of a directory we can combine chmod with three digit numbers for the three group above.

```
os2022@os2022-vb:~$ ls -l
total 32
drwxrwxr-x 2 os2022 os2022 4096 Sep 10 17:43 Desktop
drwxr-xr-x 2 os2022 os2022 4096 Sep 10 10:52 Documents
drwxr-xr-x 2 os2022 os2022 4096 Sep  5 10:31 Downloads
```

```
drwxr-xr-x 2 os2022 os2022 4096 Sep  4 11:25 Music
drwxr-xr-x 2 os2022 os2022 4096 Sep  4 11:25 Pictures
drwxr-xr-x 2 os2022 os2022 4096 Sep  4 11:25 Public
drwxr-xr-x 2 os2022 os2022 4096 Sep  4 11:25 Templates
drwxr-xr-x 2 os2022 os2022 4096 Sep  4 11:25 Videos
os2022@os2022-vb:~$ chmod 777 Documents/
os2022@os2022-vb:~$ ls -l
total 32
drwxrwxr-x 2 os2022 os2022 4096 Sep 10 17:43 Desktop
drwxrwxrwx 2 os2022 os2022 4096 Sep 10 10:52 Documents
drwxr-xr-x 2 os2022 os2022 4096 Sep  5 10:31 Downloads
drwxr-xr-x 2 os2022 os2022 4096 Sep  4 11:25 Music
drwxr-xr-x 2 os2022 os2022 4096 Sep  4 11:25 Pictures
drwxr-xr-x 2 os2022 os2022 4096 Sep  4 11:25 Public
drwxr-xr-x 2 os2022 os2022 4096 Sep  4 11:25 Templates
drwxr-xr-x 2 os2022 os2022 4096 Sep  4 11:25 Videos
```

In the command above, first we list that the permission level for Documents folder is read and execute for group and other users. After we execute the command "chmod 777", the permission level changed to read, write, and execute for the group and other users.

4. **cd** is a command to change to a directory that we specified. For example, if we are in the home directory and want to change into the Downloads directory, we can use the following command.

```
os2022@os2022-vb:~$ cd Downloads/
os2022@os2022-vb:~/Downloads$ pwd
/home/os2022/Downloads
```

The command "cd Downloads/" asks the terminal to change to the Downloads directory in the home folder. In the beginning of second line, we can see that the path before the command sign is appended into "~/Downloads". The command pwd is used to check that our current directory is now in the Downloads directory.

If the directory or path that we specify has a space on it, then we need to append the space using the backslash (\) operator to let the terminal knows that the space is part of the path that we want.

```
os2022@os2022-vb:~/Downloads$ cd Path\ with\ space/
os2022@os2022-vb:~/Downloads/Path with space$
```

5. **mkdir and rmdir** are commands to make and remove a directory, respectively. For example, if we want to create a directory called "Assignments", we can use "mkdir Assignments". Then, if to remove that directory, we can use "rmdir Assignments". Another thing to remember, rmdir

is only capable of removing an empty directory. The rule about the space in the directory name mentioned in the cd command also applies in these commands.

```
os2022@os2022-vb:~$ mkdir Assignments
os2022@os2022-vb:~$ ls
Assignments  Desktop  Documents  Downloads  Music  Pictures  Public
Templates  Videos
os2022@os2022-vb:~$ rmdir Assignments/
os2022@os2022-vb:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
os2022@os2022-vb:~$
```

6. **rm**, as mentioned before, is a command to delete files and directories. To delete a directory and all files inside the directory, we use "rm -r"

```
os2022@os2022-vb:~/Documents$ ls
hello.py  Notes
os2022@os2022-vb:~/Documents$ rm hello.py
os2022@os2022-vb:~/Documents$ ls
Notes
os2022@os2022-vb:~/Documents$ rm Notes/
rm: cannot remove 'Notes/': Is a directory
os2022@os2022-vb:~/Documents$ rm -r Notes
os2022@os2022-vb:~/Documents$ ls
os2022@os2022-vb:~/Documents$
```

7. **touch** is a command to create a file. It could be used to create any file, such as a simple text file.

```
os2022@os2022-vb:~/Documents$ ls
os2022@os2022-vb:~/Documents$ touch hello.py
os2022@os2022-vb:~/Documents$ ls
hello.py
os2022@os2022-vb:~/Documents$
```

8. **man** is a command to open a manual file of a command. This manual file informs about the command, its accepted parameters, and how to use the command. For example, running "man ls" will present the manual for the ls command (see Figure 3). Alternatively, we can also use the parameter '**--help**' after a command name.

```
os2022@os2022-vb:~/Documents$ ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.

Mandatory arguments to long options are mandatory for short options too.
  -a, --all                  do not ignore entries starting with .
  -A, --almost-all           do not list implied . and ..
```

```
      --author               with -l, print the author of each file
  -b, --escape               print C-style escapes for nongraphic
characters
      --block-size=SIZE      with -l, scale sizes by SIZE when printing
them;
                               e.g., '--block-size=M'; see SIZE format
below
  -B, --ignore-backups       do not list implied entries ending with ~
  -c                         with -lt: sort by, and show, ctime (time of
last
                               modification of file status information);
                               with -l: show ctime and sort by name;
                               otherwise: sort by ctime, newest first
```

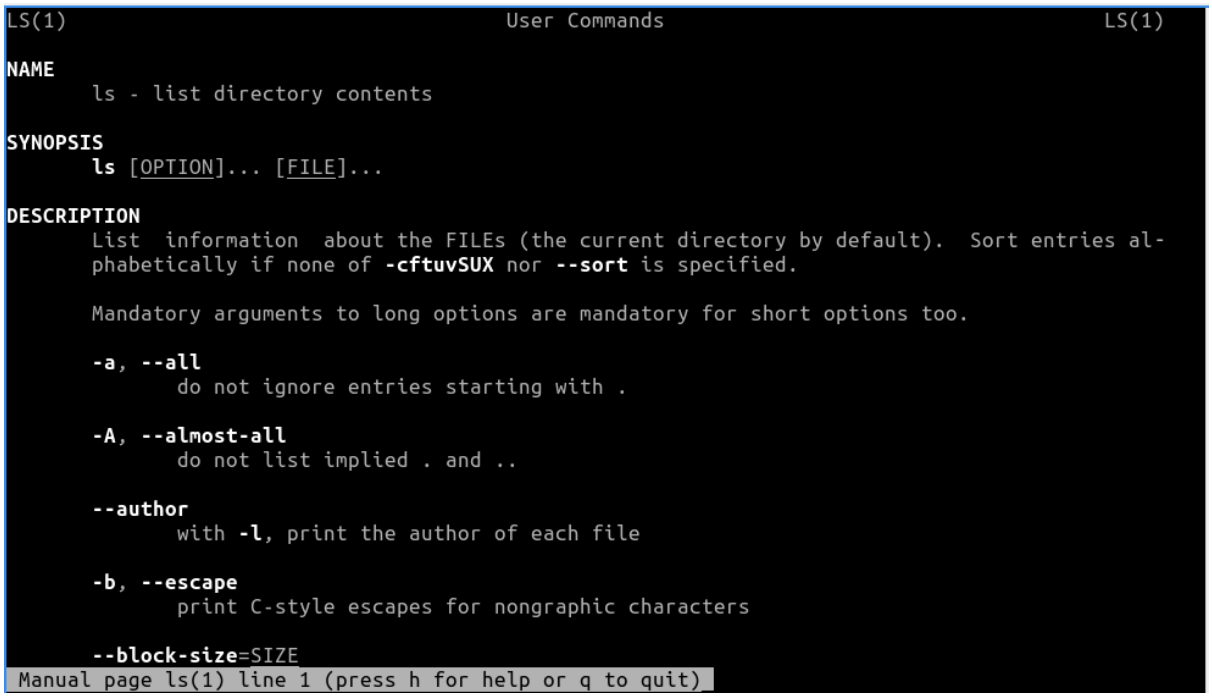

Figure 3 Manual page for the ls command

9. **cp** is a command to copy a file or directory via the terminal. This command requires two arguments: the location of the file or directory that we want to copy, and the location where we want to copy the file or the directory.

```
os2022@os2022-vb:~$ cp Documents/hello.py .
os2022@os2022-vb:~$ ls
Desktop  Documents  Downloads  hello.py  Music  Pictures  Public
Templates  Videos
```

In the above command, we copy the file “hello.py” into our current directory, denoted by the period symbol (.).

10. **mv** is a command to move a file or directory via the terminal. Similarly to the cp command, this command also requires the same arguments.

```
os2022@os2022-vb:~$ mv Documents/hello.py Downloads
os2022@os2022-vb:~$ ls Documents/
os2022@os2022-vb:~$ ls Downloads/
hello.py
```

11. **locate** is a command to locate a file in a Linux system. This command is useful if we do not remember the location or the name of a file. We can use the argument '-i' to activate the ignore case of the search. For instance, running “locate -i hello” will return all files that have a name containing the word hello, regardless of the case. To locate a file using two words, we can separate these words using the asterisk symbol (*). For example, to locate a file containing the words “hello” and “world”, we can use the command “locate -i hello*world”. If the terminal cannot find the locate command, we first need to install it using “sudo apt install plocate” (more on this command later).

```
os2022@os2022-vb:~$ locate hello.py
/home/os2022/hello.py
/home/os2022/Downloads/hello.py
/snap/gnome-3-38-2004/112/usr/lib/x86_64-linux-gnu/peas-
demo/plugins/pythonhello/pythonhello.py
```

12. **nano** is one of the terminal-based text editors installed in Ubuntu. We can use nano to create a new file or to edit an existing file. To create a new file, run nano followed by the name of the file that we want to create that does not exist yet. Similarly, we can run nano followed by the name of the file that we want to edit. For example, if we want to edit the file “hello.py”, we can run “nano hello.py”.

Figure 4 nano user interface

13. **cat** is a command to display the contents of a file. This command is a useful tool to have a glimpse of a text file.

```
os2022@os2022-vb:~$ cat hello.py
print("hello world")
```

14. **echo** is a command to show text that we defined after the command. We can also use this command to append text to a file. If the file does not exist yet, this command will create the file.

```
os2022@os2022-vb:~$ echo hello hi
hello hi
os2022@os2022-vb:~$ echo 'print("new line added")' >> hello.py
os2022@os2022-vb:~$ cat hello.py
print("hello world")
print("new line added")
```

15. **sudo** is a command to execute other commands in super-user or root privilege mode. This command stands for “SuperUser Do”. This command is particularly useful for executing commands that require administrative access, such as editing configuration files on operating systems or installing applications. For example, we want to edit the “alsa-base.conf” file that requires root privileges.

```
os2022@os2022-vb:~$ ls -l /etc/modprobe.d/alsa-base.conf
-rw-r--r-- 1 root root 2507 Feb 22  2021 /etc/modprobe.d/alsa-base.conf
os2022@os2022-vb:~$ sudo nano /etc/modprobe.d/alsa-base.conf
[sudo] password for os2022:
```

The ls -l command shows that the alsa-base.conf file requires root privileges to be edited. To edit it using nano, we add sudo before calling nano. The terminal then asks for our password to continue editing the file.

16. **apt** is a command to manage application packages in the Ubuntu and Debian family. Because this command changes the system, it requires root privileges to execute this command. To install an application with all the required packages, we can use the "sudo apt install" command.

```
os2022@os2022-vb:~$ sudo apt install jed
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  jed-common libonig5 libslang2-modules slsh
Suggested packages:
  gpm
The following NEW packages will be installed:
  jed jed-common libonig5 libslang2-modules slsh
0 upgraded, 5 newly installed, 0 to remove and 0 not upgraded.
Need to get 940 kB of archives.
After this operation, 4,338 kB of additional disk space will be used.
Do you want to continue? [Y/n]
```

In the example above, we use apt to install jed, another text editor. During execution, the "apt install" command will fetch the necessary packages from the Ubuntu package repository and install the required packages on our system.

Sometimes, we need to update the packages that we already installed in our system with the latest version available in the repository. To do this, we can use the command "sudo apt update". This command will check the repository to find if there is any newer version of the packages that we have in the system from the repository. To find a package in the repository, we can use "apt search". In this case, root privileges are not required.

To remove an application that we installed, we can use the command "sudo apt remove". This command will list the packages that will be removed from the system.

```
os2022@os2022-vb:~$ sudo apt remove jed
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following packages were automatically installed and are no longer
required:
  jed-common libonig5 libslang2-modules slsh
Use 'sudo apt autoremove' to remove them.
```

```
The following packages will be REMOVED:
  jed
0 upgraded, 0 newly installed, 1 to remove and 29 not upgraded.
After this operation, 455 kB disk space will be freed.
Do you want to continue? [Y/n]
(Reading database ... 251830 files and directories currently installed.)
Removing jed (1:0.99.20~pre.158+dfsg-1) ...
update-alternatives: using /bin/nano to provide /usr/bin/editor (editor)
in auto mode
Processing triggers for hicolor-icon-theme (0.17-2) ...
Processing triggers for man-db (2.10.2-1) ...
Processing triggers for mailcap (3.70+nmu1ubuntu1) ...
Processing triggers for desktop-file-utils (0.26-1ubuntu3) ...
```

Finally, if we want to remove the remaining packages that are no longer used due to the previous uninstallation, we can use the command “sudo apt autoremove”.

```
os2022@os2022-vb:~$ sudo apt autoremove
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following packages will be REMOVED:
  jed-common libonig5 libslang2-modules slsh
0 upgraded, 0 newly installed, 4 to remove and 29 not upgraded.
After this operation, 3,883 kB disk space will be freed.
Do you want to continue? [Y/n] Y
(Reading database ... 251820 files and directories currently installed.)
Removing jed-common (1:0.99.20~pre.158+dfsg-1) ...
Running /usr/share/jed/compile/jed-common...done
Removing slsh (2.3.2-5build4) ...
Removing libslang2-modules:amd64 (2.3.2-5build4) ...
Removing libonig5:amd64 (6.9.7.1-2build1) ...
Processing triggers for man-db (2.10.2-1) ...
Processing triggers for install-info (6.8-4build1) ...
Processing triggers for libc-bin (2.35-0ubuntu3.1) ...
```

Besides those commands mentioned above, there are still many other terminal commands available. Some of the tips that we can use for the Linux terminal are as follows:

- We can use a **clear** command to clear the terminal outputs.
- A shortcut Tab can be used to automatically finish the command or parameter that you specify. For example, in the home directory, typing “cd Doc” followed by a Tab press will automatically complete the command in “cd Documents”.
- Ctrl+C can be used to safely stop a running command. If it does not stop after pressing these buttons, we can use Ctrl+Z to forcely stop the command.

Activity 3.1

Try the following commands, capture your screen for their results, and explain what these commands are doing:

1. uname
2. df
3. hostname
4. hostname -I

3.2 System Calls

In this part of the lab work, we examine the use of system calls in some commands that we have discussed before. To this end, we use a tool called **strace**. The strace command captures the system calls used by a process. Before we use these tools, we first need to install them (if they are not installed yet).

```
os2022@os2022-vb:~$ sudo apt install strace
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
strace is already the newest version (5.16-0ubuntu3).
strace set to manually installed.
0 upgraded, 0 newly installed, 0 to remove and 29 not upgraded.
```

Next, we will try to see the system calls used in the ls command. To see this, we can use the strace command before the ls command, as follows:

```
os2022@os2022-vb:~$ cd
os2022@os2022-vb:~$ strace ls Documents/
execve("/usr/bin/ls", ["ls", "Documents/"], 0x7ffc2cca0058 /* 59 vars */) = 0
brk(NULL)                               = 0x56522f11a000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffd2ef70ad0) = -1 EINVAL (Invalid
argument)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f5b1a74c000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or
directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=65063, ...}, AT_EMPTY_PATH)
= 0
mmap(NULL, 65063, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f5b1a73c000
close(3)                                = 0
...
```

The output shows a sequence of system calls made during the execution of the ls command. To store this output into a file for analysis, we can save this using the -o parameter.

```
os2022@os2022-vb:~$ strace -o trace.log ls Documents/
os2022@os2022-vb:~$ cat trace.log
execve("/usr/bin/ls", ["ls", "Documents/"], 0x7ffe091addf8 /* 59 vars */) = 0
brk(NULL)                               = 0x558480e55000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffd5ab07840) = -1 EINVAL (Invalid
argument)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f937a54b000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or
directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=65063, ...}, AT_EMPTY_PATH)
= 0
mmap(NULL, 65063, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f937a53b000
...
```

The command above saves the strace result in a file named trace.log. We can then see this file using the cat command or using the text editor provided by Ubuntu (gedit). To open using gedit, we can use the command “gedit trace.log”. The gedit window will open the file (see Figure 5).

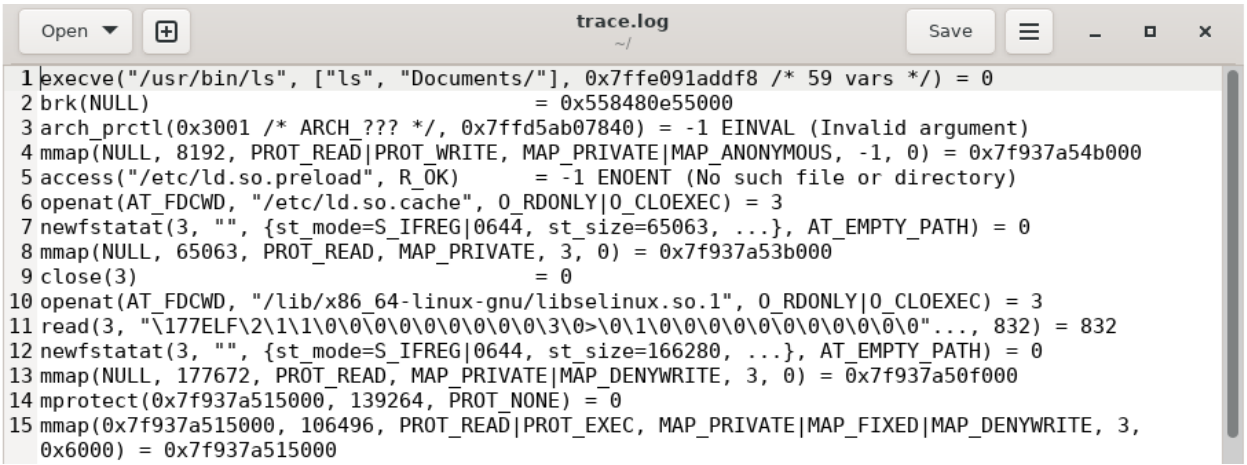

Figure 5 The trace.log file opened using gedit

Now, let us take a look at the trace.log file. The start of each line, such as **execve**, **brk,** and **arch_prctl**, shows the system calls that are made when we execute the ls command. We can also see that each of the system calls is followed by a parentheses that represent the arguments provided to the system calls. At the end of the line, we can see an equal sign (=) and a number after it. This represents a value returned by the corresponding system call.

To understand what a specific system call does, we can use the man command. However, we need to install the development manual pages first before we can open the corresponding manual for system calls.

```
os2022@os2022-vb:~$ sudo apt install manpages-dev
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following NEW packages will be installed:
  manpages-dev
0 upgraded, 1 newly installed, 0 to remove and 29 not upgraded.
Need to get 2,309 kB of archives.
After this operation, 4,037 kB of additional disk space will be used.
Get:1 http://archive.ubuntu.com/ubuntu jammy/main amd64 manpages-dev all
5.10-1ubuntu1 [2,309 kB]
Fetched 2,309 kB in 11s (205 kB/s)
Selecting previously unselected package manpages-dev.
(Reading database ... 255012 files and directories currently installed.)
Preparing to unpack .../manpages-dev_5.10-1ubuntu1_all.deb ...
Unpacking manpages-dev (5.10-1ubuntu1) ...
Setting up manpages-dev (5.10-1ubuntu1) ...
Processing triggers for man-db (2.10.2-1) ...
```

Next, let us take a look at the first line of the trace.log file:

```
execve("/usr/bin/ls", ["ls", "Documents/"], 0x7ffe091addf8 /* 59 vars */) = 0
```

To know what execve does, we can open the manual using “man 2 execve”. The 2 argument represents the section for the system calls in the manual page.

```
NAME
       execve - execute program

SYNOPSIS
       #include <unistd.h>

       int execve(const char *pathname, char *const argv[],
                  char *const envp[]);

DESCRIPTION
       execve()  executes  the  program referred to by pathname.  This
       causes the program that is currently being run by  the  calling
       process  to be replaced with a new program, with newly initial-
       ized stack, heap, and (initialized and uninitialized) data seg-
       ments.
```

From this manual, we can see that the execve executes the “/usr/bin/ls” program with an additional arguments “Documents/” as the path that we want to list.

Next, let us focus on the system calls involving the directory name that we specified: Documents. We can filter the trace.log file to find this document name.

```
os2022@os2022-vb:~$ grep Documents trace.log
execve("/usr/bin/ls", ["ls", "Documents/"], 0x7ffe091addf8 /* 59 vars */) = 0
statx(AT_FDCWD, "Documents/", AT_STATX_SYNC_AS_STAT, STATX_MODE,
{stx_mask=STATX_BASIC_STATS|STATX_MNT_ID, stx_attributes=0,
stx_mode=S_IFDIR|0755, stx_size=4096, ...}) = 0
openat(AT_FDCWD, "Documents/", O_RDONLY|O_NONBLOCK|O_CLOEXEC|O_DIRECTORY) = 3
```

The result of the grep command shows three system calls that specifically use “Documents” as their argument. We have discussed the first system calls. Let us have a look at the second system call in the result. The second line shows the use of the **statx** system call with “Documents/” as one of its arguments. Based on the statx manual page, this system call gets the status of a file. The third line shows the use of **openat** to open the specified directory “Documents/” returning 3 as the value of the file descriptor. The value of the file descriptor is 3 because the previous numbers are used as default descriptors: 0 represents standard input, 1 represents standard output, and 2 represents standard error (more on this later).

Activity 3.2

1. Save the trace of a command `echo hello` to a file titled “echo.log”.
2. Filter the “echo.log” file to find the text “hello” being mentioned. Screen capture the result.
3. Explain what the system call that related to the text in question number 2 is doing based on the manual page of the system call.

3.3 Using Linux System Calls

In this last part of the lab work, we are going to develop some simple programs that utilize system calls. For this purpose, we will use the C programming language. First, we need to make sure that the compiler for the C programming language, or gcc, is installed in our operating system. To install the compiler, we can use the build-essential package in Ubuntu.

```
os2022@os2022-vb:~/OS_arif$ sudo apt install build-essential
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  dpkg-dev fakeroot g++ g++-11 gcc libalgorithm-diff-perl
  libalgorithm-diff-xs-perl libalgorithm-merge-perl libdpkg-perl
  libfakeroot libfile-fcntllock-perl libstdc++-11-dev
  lto-disabled-list make
Suggested packages:
```

```
  debian-keyring g++-multilib g++-11-multilib gcc-11-doc gcc-multilib
  autoconf automake libtool flex bison gdb gcc-doc bzr
  libstdc++-11-doc make-doc
The following NEW packages will be installed:
  build-essential dpkg-dev fakeroot g++ g++-11 gcc
  libalgorithm-diff-perl libalgorithm-diff-xs-perl
  libalgorithm-merge-perl libdpkg-perl libfakeroot
  libfile-fcntllock-perl libstdc++-11-dev lto-disabled-list make
0 upgraded, 15 newly installed, 0 to remove and 29 not upgraded.
Need to get 12.9 MB/15.0 MB of archives.
After this operation, 54.9 MB of additional disk space will be used.
Do you want to continue? [Y/n]
```

Next, we will try some examples of system calls provided in the glibc library. This library provides, among others, wrapper functions for system calls in Linux operating systems.

3.3.1 Read and Write

System calls that can be used to read and write data are functions read() and write() respectively. These functions read and write data to a file descriptor. The read() function is used to read data from a file descriptor. Please note that hardware are also referred to as files in Linux. We can see the syntax for the read function using the command "man 2 read". The syntax is as follows.

```
ssize_t read(int fd, const void *buf, size_t count);
```

The read function accepts three parameters: (1) fd denotes the file descriptor that we want to read from; (2) *buf denotes the placeholder of the data that we read; and (3) count denotes the total bytes that we want to read. This function returns the number of bytes read, if successful, and returns -1 if it fails.

The write() function is used to write data to a file descriptor. We can see the syntax for the write function using the command "man 2 write". The syntax for the write system call is as follows.

```
ssize_t write(int fd, const void *buf, size_t count);
```

The write function accepts three parameters: (1) fd denotes the file descriptor that we want to write; (2) *buf denotes the data that we want to write; and (3) count denotes the total bytes that are to be written. This function returns the number of bytes written if successful, and returns -1 if it fails.

Let us try to write a simple program using these system calls. First, create a file titled "readwrite.c" and write the following code.

```
#include <unistd.h>

int main()
```

```
{
    char databuffer[12];
    read(0, databuffer, 12);
    write(1, databuffer, 12);
}
```

To use the read and write system call functions, we need to import the unistd.h library that provides the functions. Then, we declare a variable named “databuffer” to hold the data.

Afterwards, we set the parameters for the read function. We specify the first parameter with a value of 0 as the file descriptor for the standard input, i.e., keyboard. Then, we set the second parameter with databuffer variable to hold the input. For the third parameter of the read function, we set a value of 15 as the size of the bytes that we want to get from the keyboard.

In the last line with the write function, we specify a value of 1 as the file descriptor for standard output, i.e., computer screen. Then, because we want to display the input from the keyboard, we specify the data buffer as the second parameter. Finally, we specify a value of 15 as the size of the bytes that we want to display.

After we done, let us compile this file and run it.

```
os2022@os2022-vb:~$ gcc -o readwrite.out readwrite.c
os2022@os2022-vb:~$ ./readwrite.out
hello world
hello world
```

The first command compiles the file “readwrite.c” and produces an executable named “readwrite.out” as a result. Then, we run the file “readwrite.out”. When we run the file, it will wait for our input. If we type “hello world” as shown above and press enter, the program will return the text that we wrote.

3.3.2 Open

The open system call is used to open, either an existing or non-existing file. This system call returns a file descriptor that we can use with the previous read and write system calls. The syntax of the open function system call is as follows.

```
 int open(const char *pathname, int flags, [mode_t mode]);
```

The first parameter of this function denotes the file name that we want to open. The second parameter denotes the mode of opening. Some of the modes that we can use are O_RDONLY and O_WRONLY for read and write only modes, respectively. To read and write a file, we can use the O_RDWR mode. The third parameter is not required if we want to open an existing file. If we

want to open a file that already exists, we need to specify the third parameter with the permissions of the file that we create (read/write/execute). This function returns the file descriptor of the file if successful or -1 if it fails.

Let us write a program to demonstrate the use of the open() function. First, create a file titled "myname.txt" and write down our name inside. This is the file that we want to open. Then, create another file titled "open.c" and write the following program.

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>

int main()
{
    int fd;
    char databuffer[20];

    fd = open("myname.txt", O_RDONLY);
    printf("The file descriptor: %d\n",fd);
    read(fd,databuffer,20);
    write(1,databuffer,20);
}
```

The first three lines import the necessary libraries. The unistd.h library provides the open() function. The stdio.h library provides the printf() function to write to the screen. The fcntl.h library provides the file open mode constants for the open system call.

The next block of codes are the declaration of variables. The first variable fd is used to store the file descriptor. The second variable databuffer is used to store the data. The first line of the next block calls the open function with two parameters: (1) "myname.txt" for the file name that we want to open; and (2) O_RDONLY as the mode of reading the file, i.e., read only. We then store the file descriptor resulting from this function in the fd variable.

The next line prints the value of the file descriptor on the screen using the function printf(). Then, we read 20 bytes of data from the file "myname.txt" using the file descriptor returned from the open function and save it in the databuffer variable. The statement in the last line writes 20 bytes of data from the databuffer variable to the screen (file descriptor = 1). Finally, let us compile and run this program.

```
os2022@os2022-vb:~$ gcc -o open.out open.c
os2022@os2022-vb:~$ ./open.out
The file descriptor: 3
myname
```

The program shows the value of the file descriptor variable, i.e., 3, resulting from the open function. Then, the write function displays the content of the file to the screen.

Resources

- Install Ubuntu in VirtualBox: https://linuxhint.com/install-ubuntu22-04-virtual-box/
- Install VirtualBox Guest Addition for Ubuntu: https://www.tecmint.com/install-virtualbox-guest-additions-in-ubuntu/

References

[1] A. Silberschatz, P. B. Galvin, and G. Gagne, Operating Systems Concepts, Tenth Edition, John Wiley & Sons, 2018.