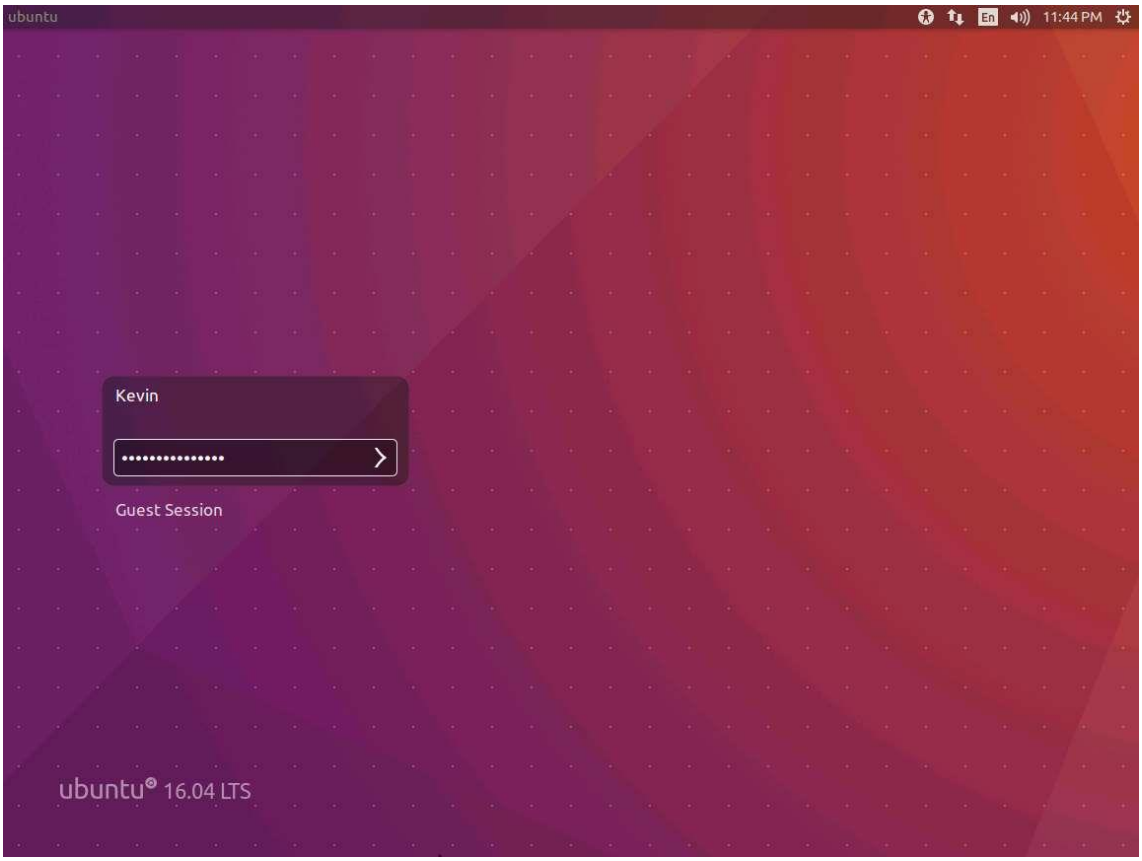
Exercise 2: Binary Analysis on a 64-bit Machine

Lab Objective:

In this lab, we will explore the creation and data for assembly language.

Lab Tasks

1. ☐ Login to the Software-Test-Linux machine using **studentpassword** as Password.



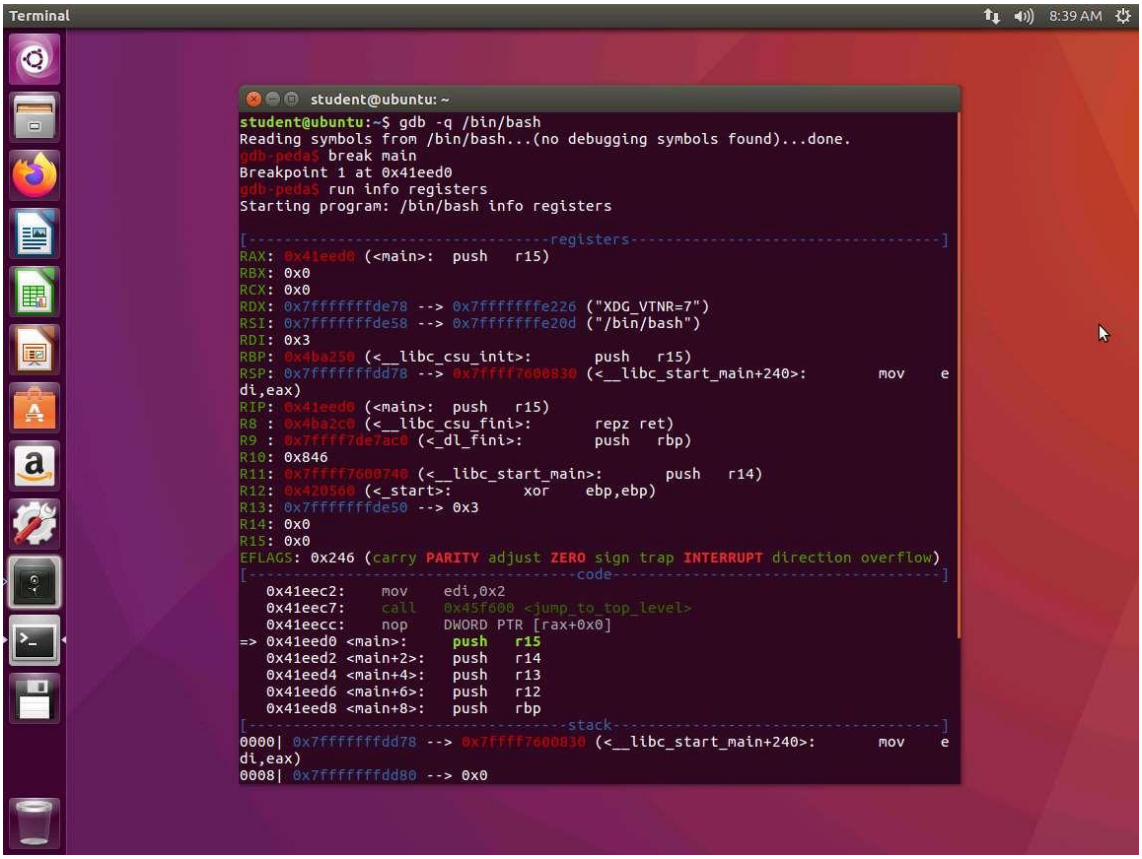
2. ☐ In the 64-bit machine, enter the following:

a. **gdb -q /bin/bash**

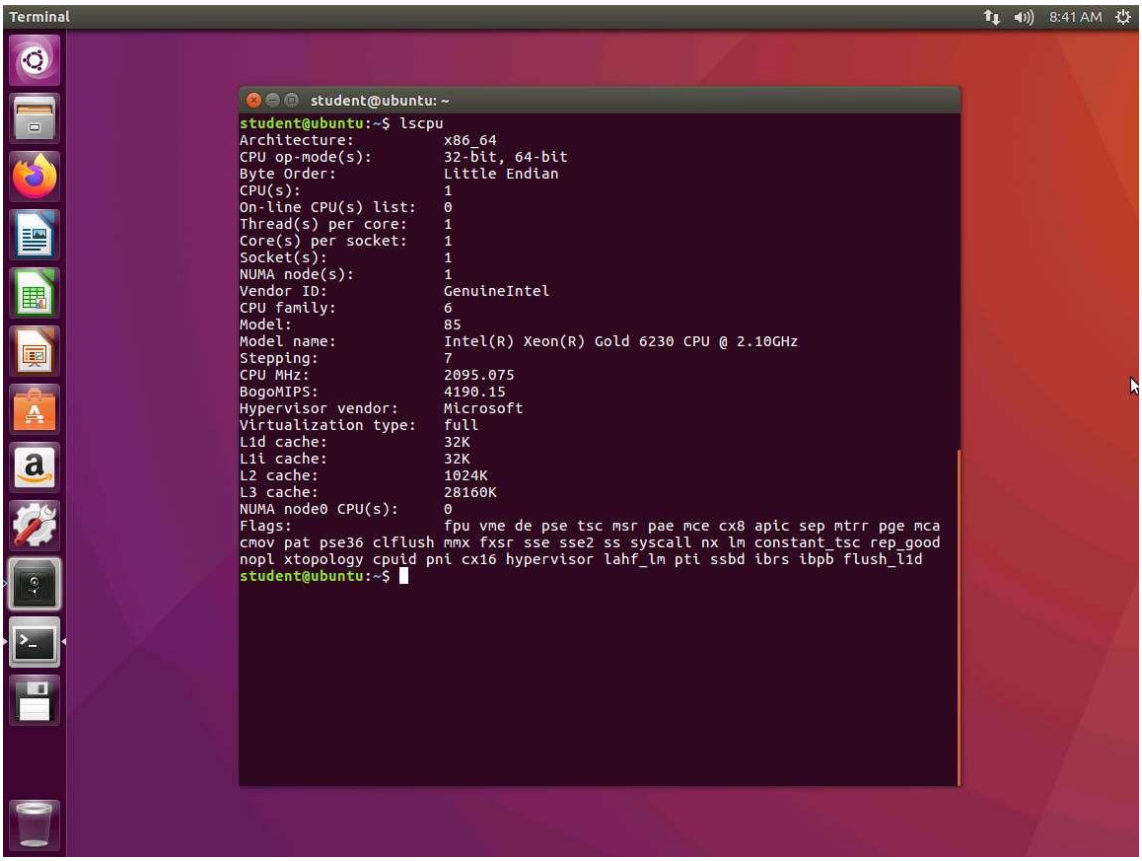
b. **break main**

c. **run info registers**

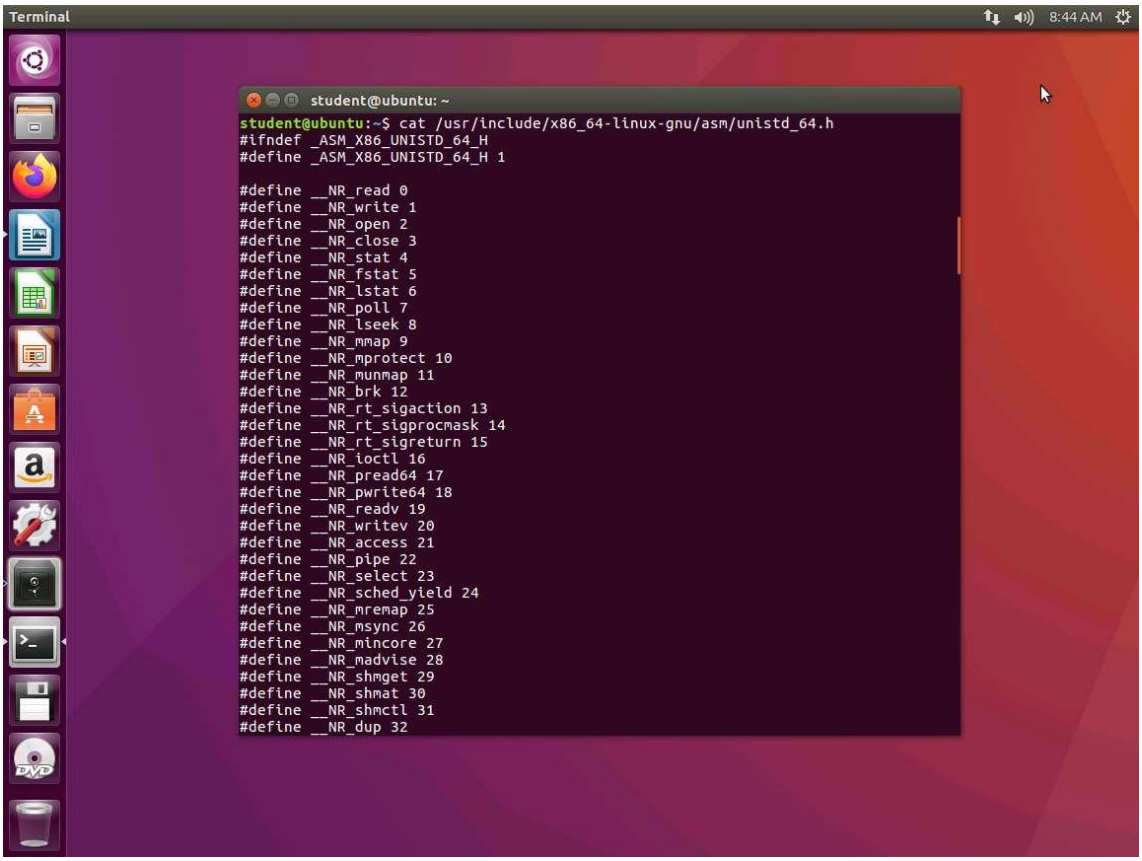
The output of this command is shown in the following screenshot.

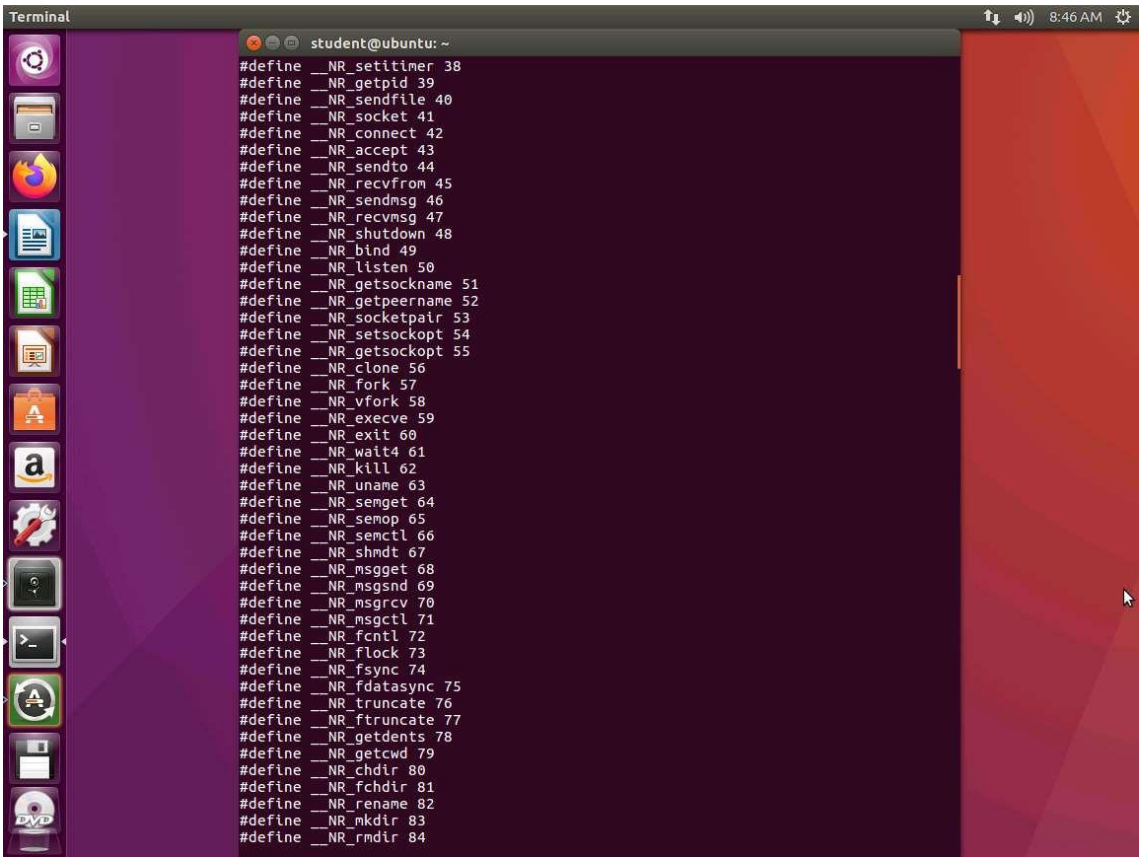


3. As the screenshot in Step 2 shows, we have additional registers along with the 64-bit version of the registers we have already discussed. Next, enter **lscpu**. The output of this command is shown in the following screenshot.



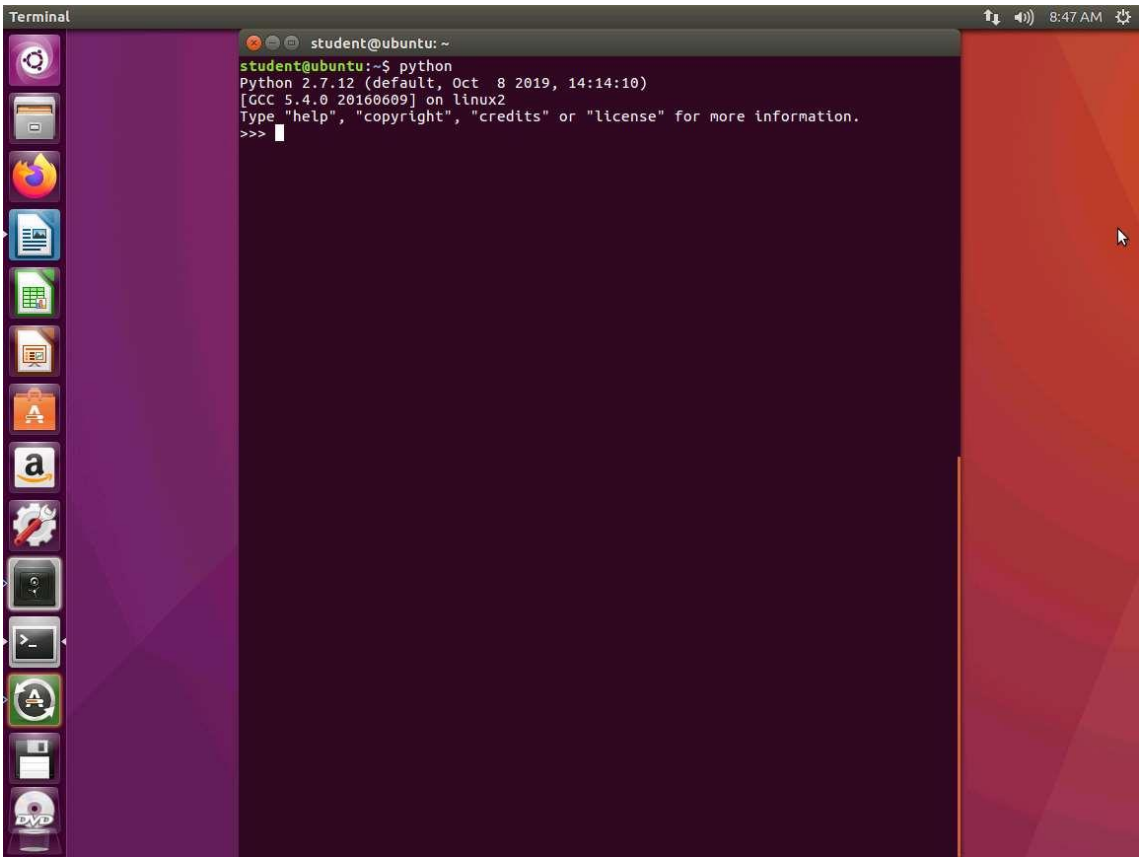
4. ☐ Next, enter **cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h**.


Terminal
8:44 AM
student@ubuntu: ~
student@ubuntu:~$ cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h
#ifndef _ASM_X86_UNISTD_64_H
#define _ASM_X86_UNISTD_64_H 1
#define __NR_read 0
#define __NR_write 1
#define __NR_open 2
#define __NR_close 3
#define __NR_stat 4
#define __NR_fstat 5
#define __NR_lstat 6
#define __NR_poll 7
#define __NR_lseek 8
#define __NR_mmap 9
#define __NR_mprotect 10
#define __NR_munmap 11
#define __NR_brk 12
#define __NR_rt_sigaction 13
#define __NR_rt_sigprocmask 14
#define __NR_rt_sigreturn 15
#define __NR_ioctl 16
#define __NR_pread64 17
#define __NR_pwrite64 18
#define __NR_readv 19
#define __NR_writev 20
#define __NR_access 21
#define __NR_pipe 22
#define __NR_select 23
#define __NR_sched_yield 24
#define __NR_mremap 25
#define __NR_msync 26
#define __NR_mincore 27
#define __NR_madvise 28
#define __NR_shmget 29
#define __NR_shmat 30
#define __NR_shmctl 31
#define __NR_dup 32



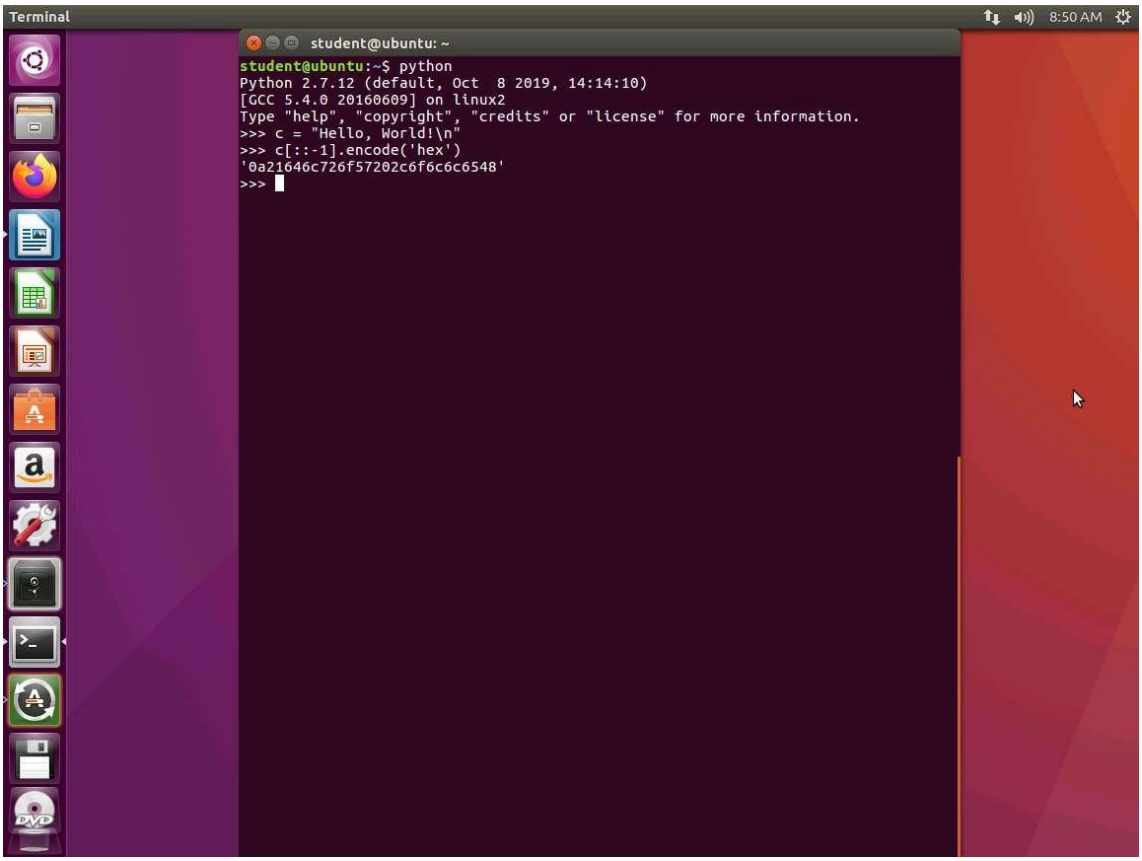
5. ☐ As you can see from the above screenshot, we have many more here with the write system call at 1. The exit is at 60.

6. ☐ Next, enter **python** to enter the python editor.



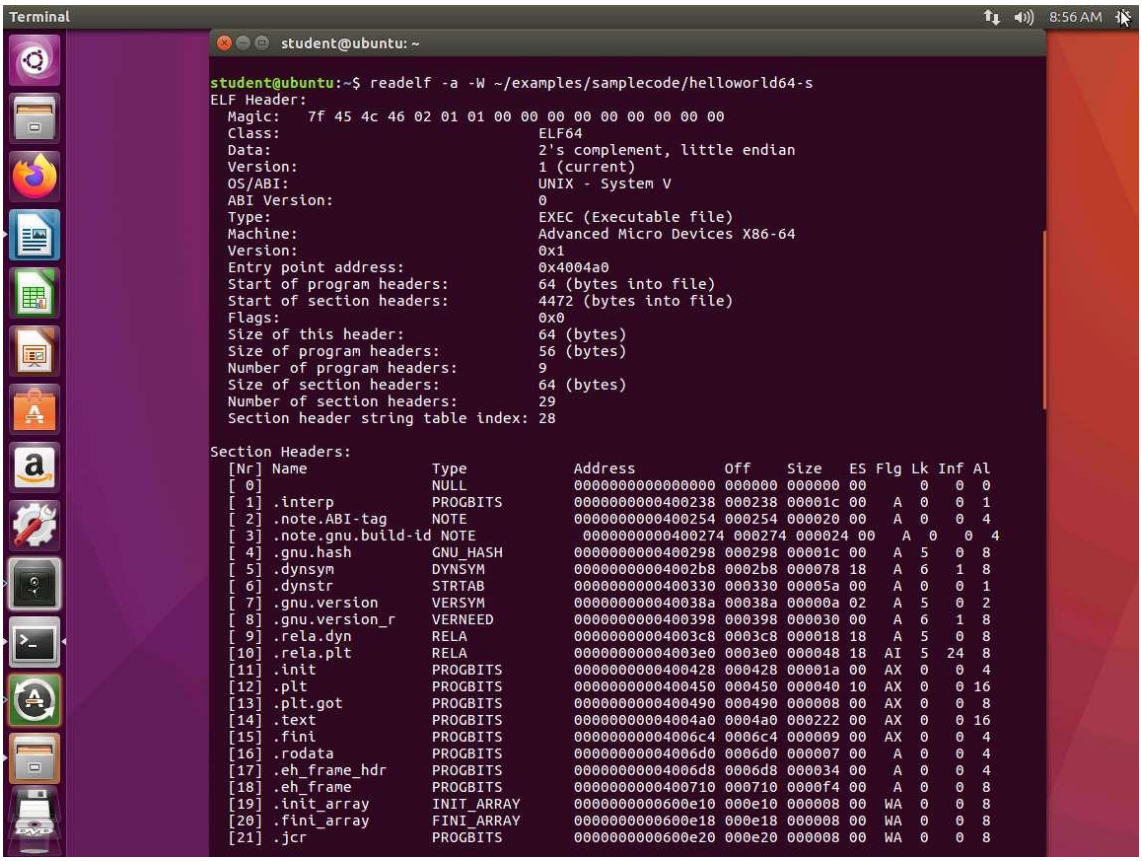
7. ☐ In the editor, enter **c = "Hello, World!¥n"**.

8. ☐ Next, enter **c[::-1].encode('hex')**. The output of the command is shown in the following screenshot.



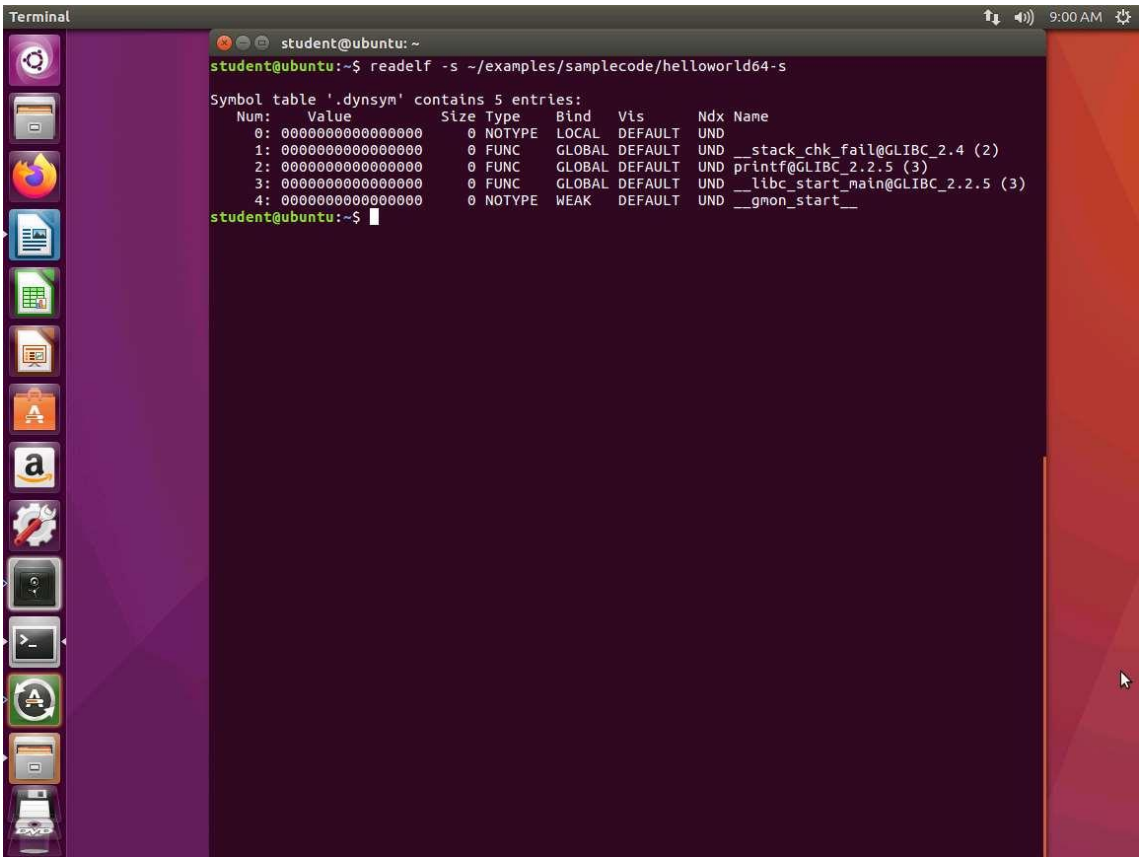
9. ☐ Once the Python prompt is available, we initialize a variable called **c** and store the string as **Hello, World!¥n**. Next, we use Python's power of string manipulation, and in one line, reverse the order of the characters in the string with the **c[::-1]** syntax. Then, we call the encode method to encode our characters in their hexadecimal representation. This is done with the **.encode('hex')** part of the syntax. This makes our string almost usable for assembly.

10. ☐ In the terminal window, enter **readelf -a -W ~/examples/samplecode/helloworld64-s**. The output of this command is shown in the following screenshot.



11. The **-a** argument displays the ELF header, program headers, section headers, symbols, relocations, dynamic section, version information, architecture-specific information, and a histogram of bucket list lengths. We can also see that we are dealing with an executable file as opposed to a relocatable object file and the address in memory where execution begins is **0x400a0**. There are nine program headers, each of which is 56 bytes in size, and thirty-one section headers, each of which is 64 bytes in size.

12. Next, enter **readelf -s ~/examples/samplecode/helloworld64-s**. The output of the symbol table is shown in the following screenshot.



13. ☐ What we can gather from this part of the output is that

    the **printf()** function is used somewhere in the program. Larger programs with more code and those that use additional functions from shared libraries will have many more functions linked dynamically similar to this. Continuing to review the output, the .symtab section shows us all symbol references in the program, including any variables or function names, and immediately.

14. ☐ The lab objectives have been achieved. Close all windows and clean up

    from the exercise as required.