Name: Abanob Yousry Fahmy

ID: 23101861

Exercise 1: Using fork() in C

Write a C program that demonstrates process creation using the fork() system call.

Code Example:

```
#include <stdio.h>
#include <unistd.h> int
main() {

pid_t pid = fork(); if
(pid == 0) {

printf("This is the child process. PID: %d\n", getpid());

} else if (pid > 0) {

printf("This is the parent process. PID: %d\n", getpid());

} else {

printf("Fork failed!\n");

}

return 0;

}
```

Task: Compile the program

```
gcc process_creation.c -o process_creation

./process_creatio
```

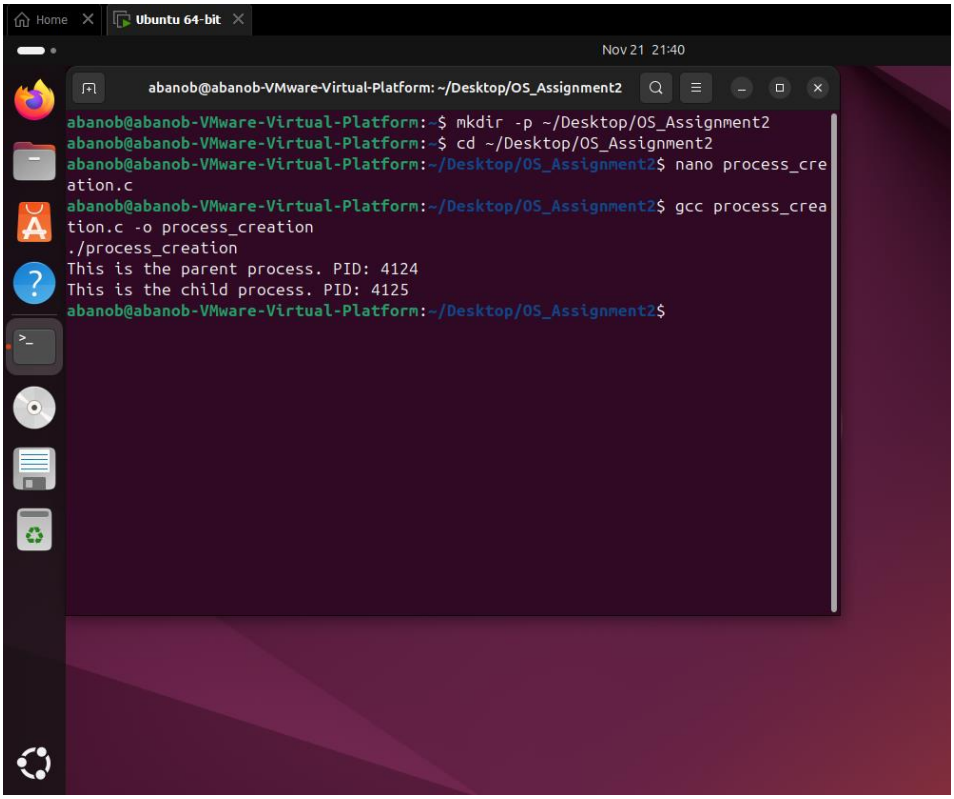
Explanation:

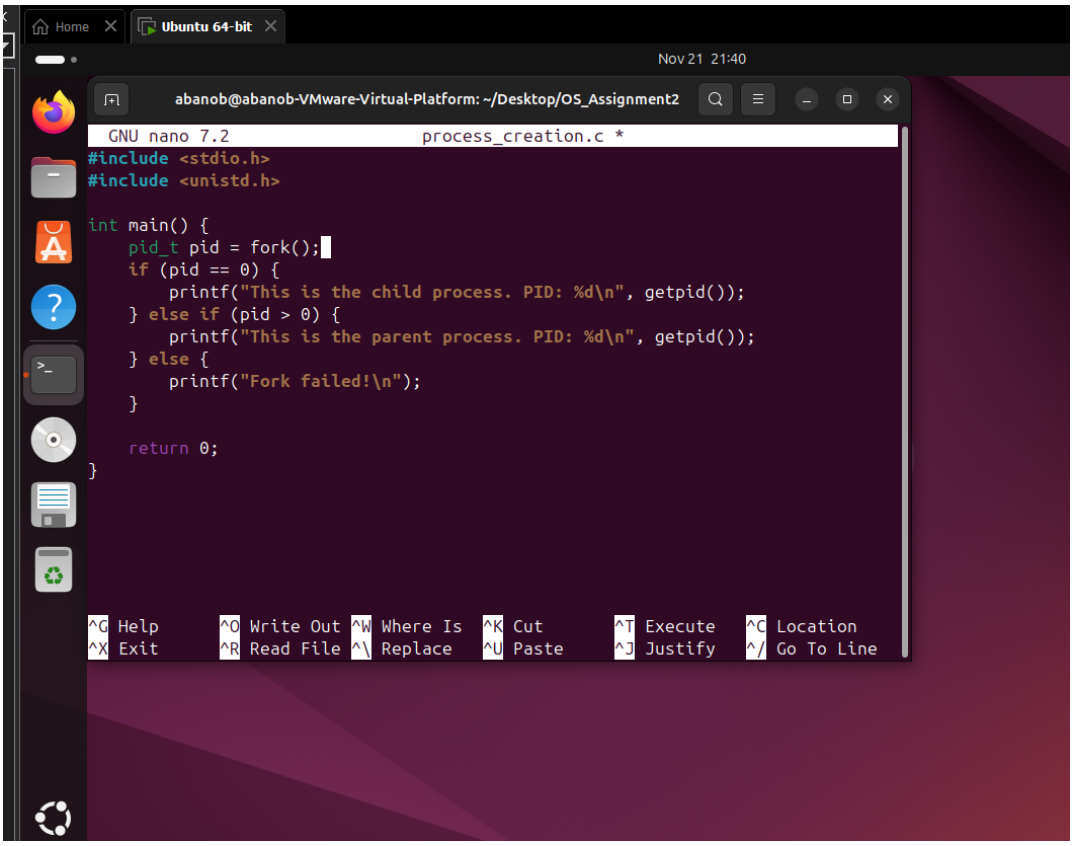
This program is all about creating a new process using fork().

When fork() gives 0, it means we're inside the child process, and that part of the code runs there.

If fork() returns a positive number, that's the child's PID, and it tells the parent process to run its code.

If fork() returns -1, it means something went wrong and the process couldn't be created.





Exercise 2: Starting Processes in the Background

Start a process in the background using the & operator: sleep 300 &

List the running background processes using:

Jobs

Explanation:

This program just sleeps for 300 seconds to simulate a process that takes time.

We run it in the background using & so we can still use the terminal for other commands.

That's basically it for Q2. It's simple: the process runs while you can do other stuff in the terminal.

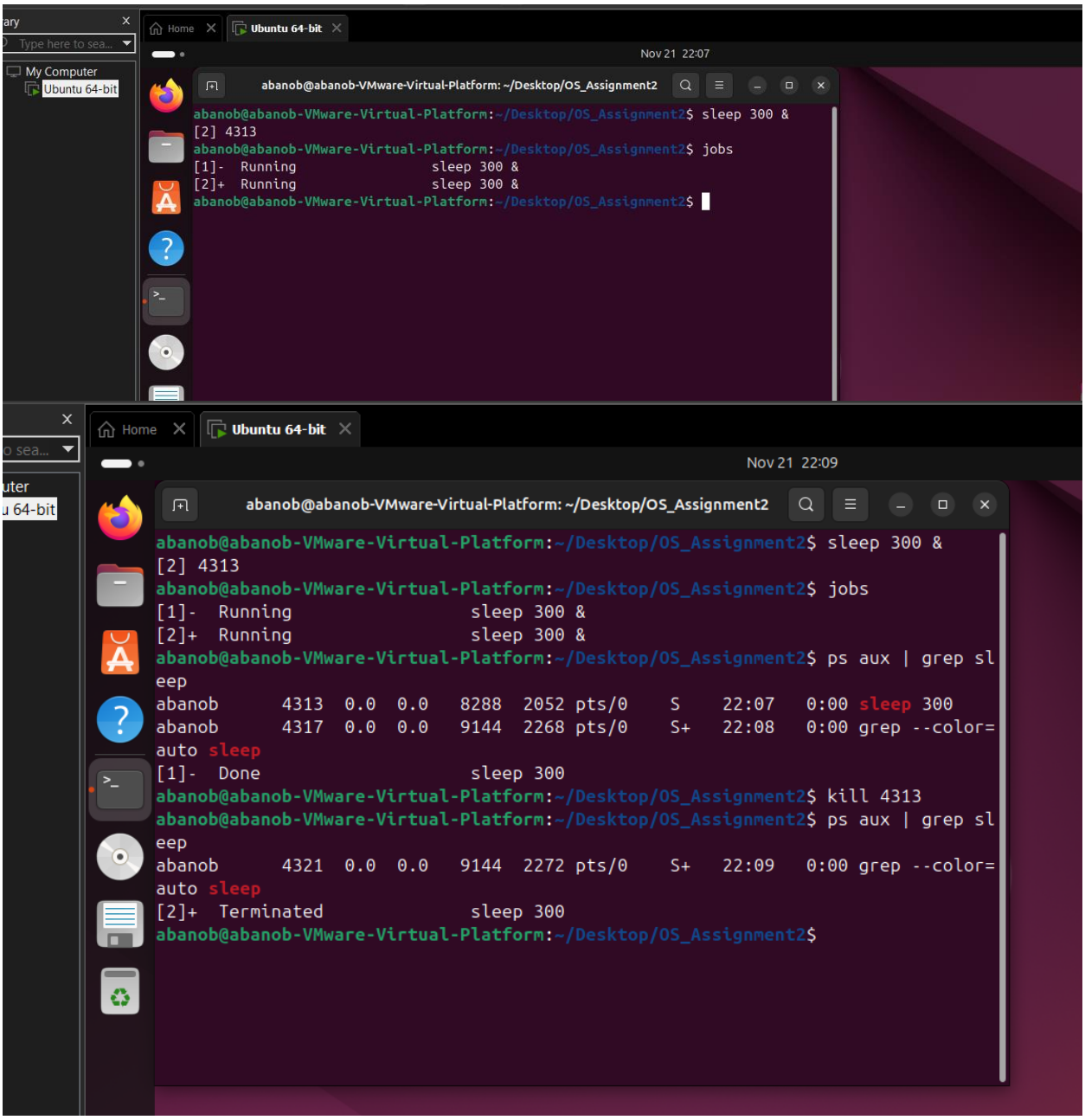
Exercise 3: Stopping Processes

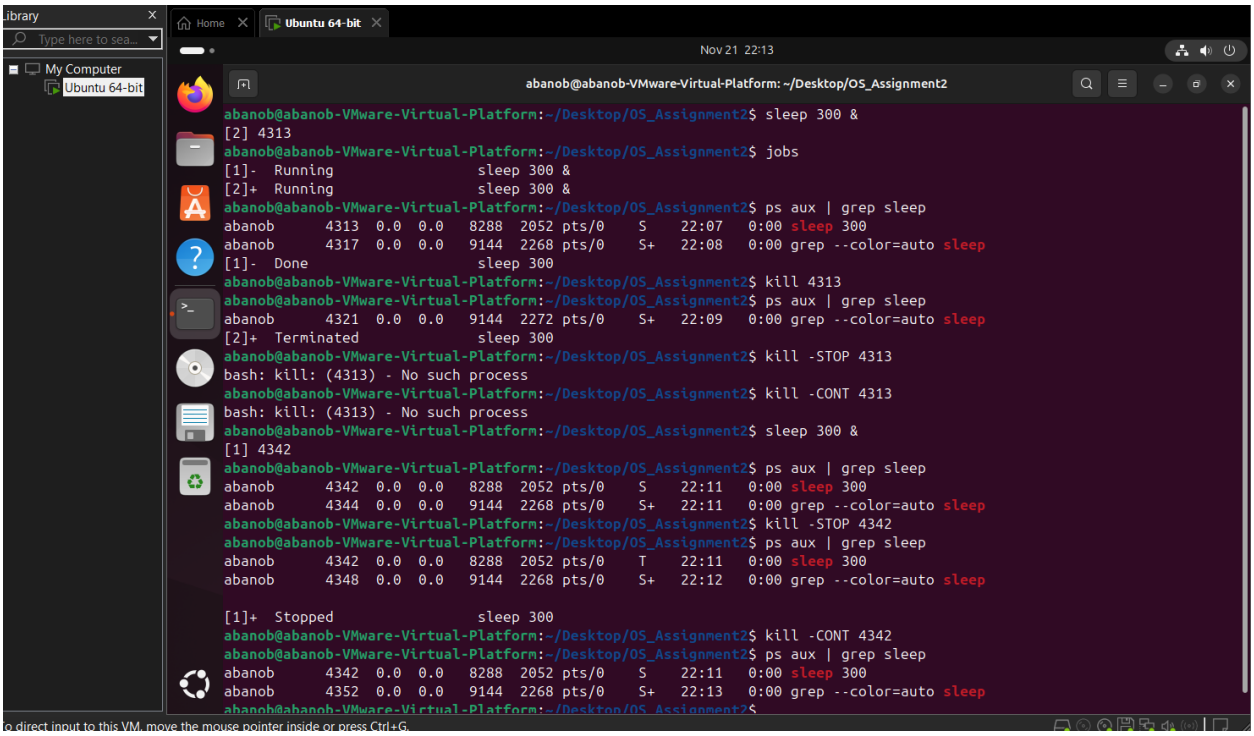
1- Use ps to find the process ID (PID) of the sleep process you started:
ps aux | grep sleep

2- Stop the process using kill:
kill <PID>
3- Verify the process is no longer running:
ps aux | grep sleep

Explanation:

Ideally, we use `kill -STOP` to **freeze** the process without deleting it. Later, we use `kill -CONT` to **resume** it from the same point. Finally, standard `kill` is used to **terminate** it completely.


Home
Ubuntu 64-bit
Nov 21 22:07
abanob@abanob-VMware-Virtual-Platform: ~/Desktop/OS_Assignment2
abanob@abanob-VMware-Virtual-Platform:~/Desktop/OS_Assignment2$ sleep 300 &
[2] 4313
abanob@abanob-VMware-Virtual-Platform:~/Desktop/OS_Assignment2$ jobs
[1]- Running sleep 300 &
[2]+ Running sleep 300 &
abanob@abanob-VMware-Virtual-Platform:~/Desktop/OS_Assignment2$
Home
Ubuntu 64-bit
Nov 21 22:09
abanob@abanob-VMware-Virtual-Platform: ~/Desktop/OS_Assignment2
abanob@abanob-VMware-Virtual-Platform:~/Desktop/OS_Assignment2$ sleep 300 &
[2] 4313
abanob@abanob-VMware-Virtual-Platform:~/Desktop/OS_Assignment2$ jobs
[1]- Running sleep 300 &
[2]+ Running sleep 300 &
abanob@abanob-VMware-Virtual-Platform:~/Desktop/OS_Assignment2$ ps aux | grep sl
eep
abanob 4313 0.0 0.0 8288 2052 pts/0 S 22:07 0:00 sleep 300
abanob 4317 0.0 0.0 9144 2268 pts/0 S+ 22:08 0:00 grep --color=
auto sleep
[1]- Done sleep 300
abanob@abanob-VMware-Virtual-Platform:~/Desktop/OS_Assignment2$ kill 4313
abanob@abanob-VMware-Virtual-Platform:~/Desktop/OS_Assignment2$ ps aux | grep sl
eep
abanob 4321 0.0 0.0 9144 2272 pts/0 S+ 22:09 0:00 grep --color=
auto sleep
[2]+ Terminated sleep 300
abanob@abanob-VMware-Virtual-Platform:~/Desktop/OS_Assignment2$



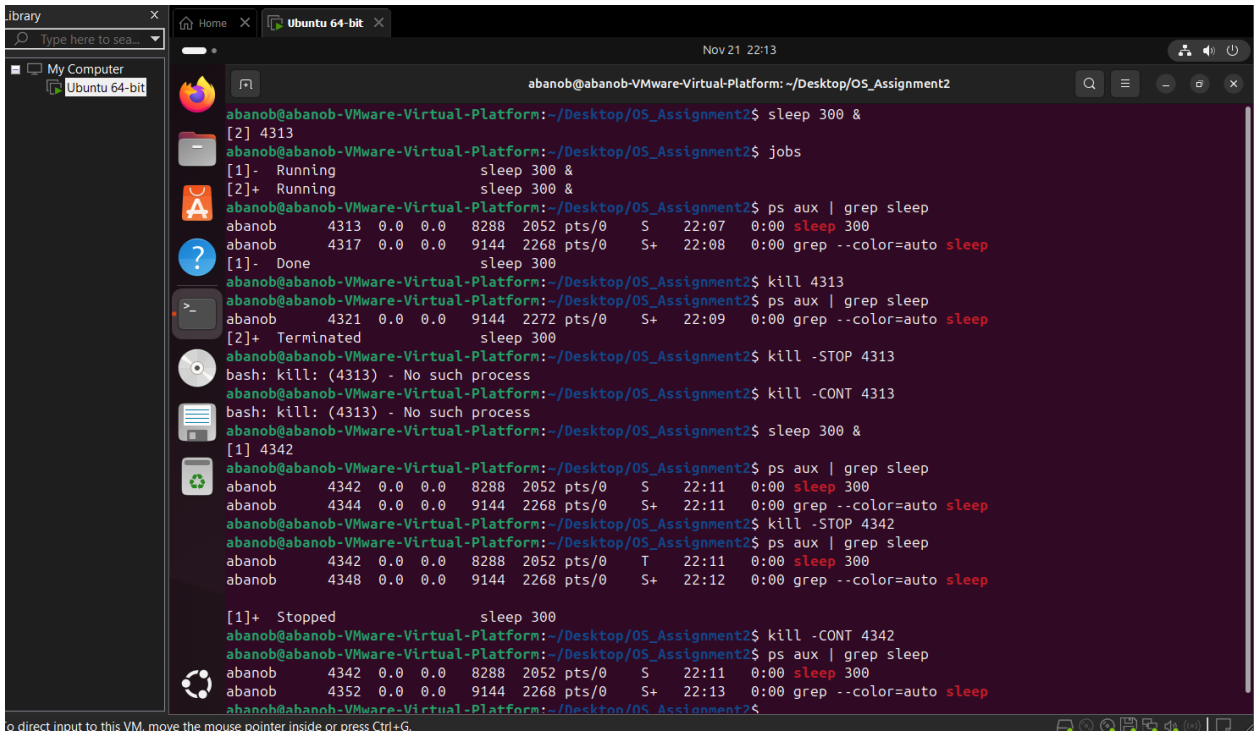
Exercise 4: Pausing and Resuming a Process

- Pause a process
  kill -STOP <PID>

- Resume the process
  kill -CONT <PID>

Explanation:

We use `-STOP` to **freeze** the process temporarily, and `-CONT` to let it **continue** working again.



Exercise 5: Role of the Linker

Write two separate C files, then compile and link them together. File 1: " file1.c ":

```
#include <stdio.h>
void hello() {

printf("Hello from file1!\n");

}
```

File 2: " file2.c ": void hello();

```
int main()
{ hello();
return 0;
}
```

Compile both files and link them:

```
gcc file1.c file2.c -o output_program
./output_program
```

Task: Modify one of the files, recompile, and observe the impact of linking.

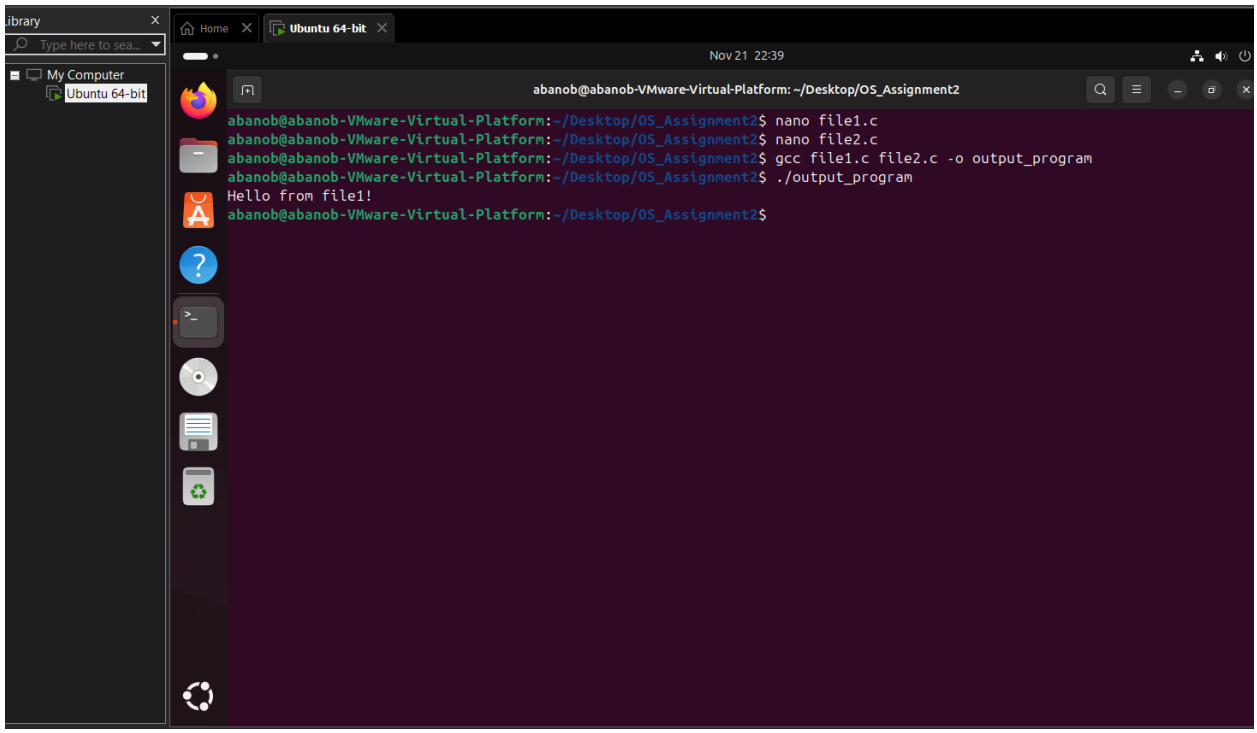
Explanation:

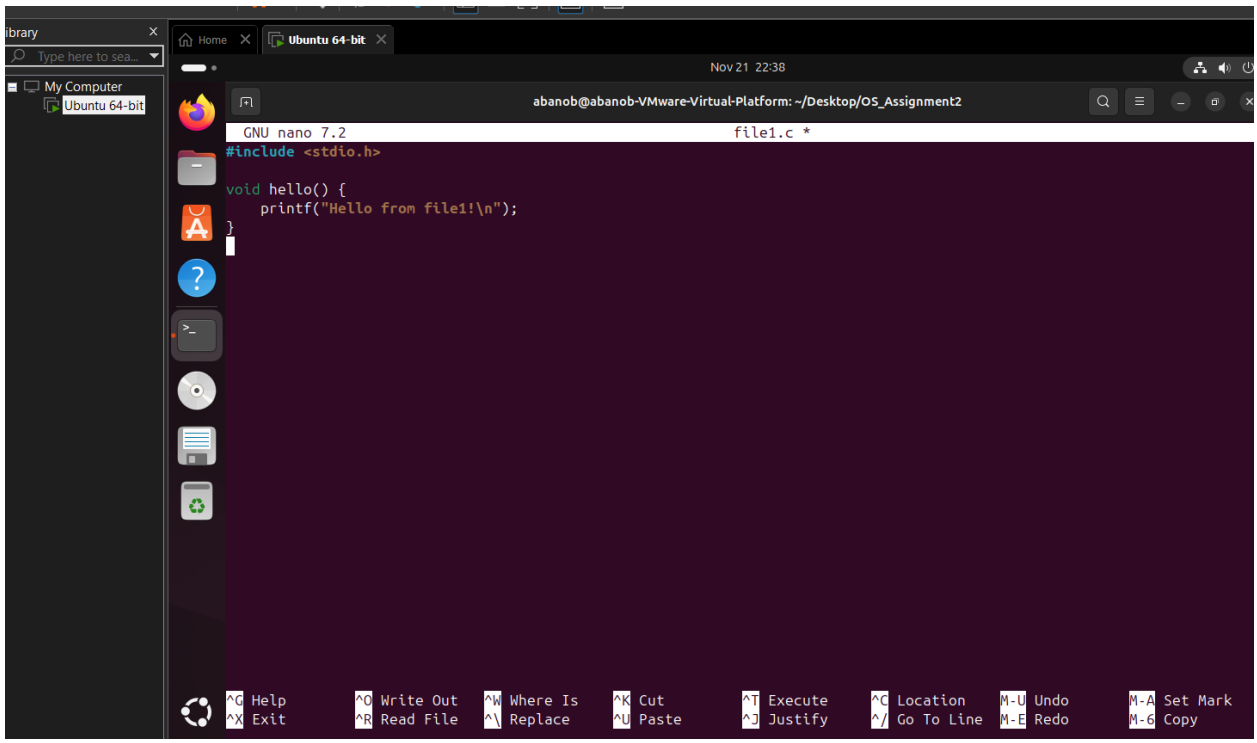
Here we have two files. file1.c has the function hello() and file2.c calls it.

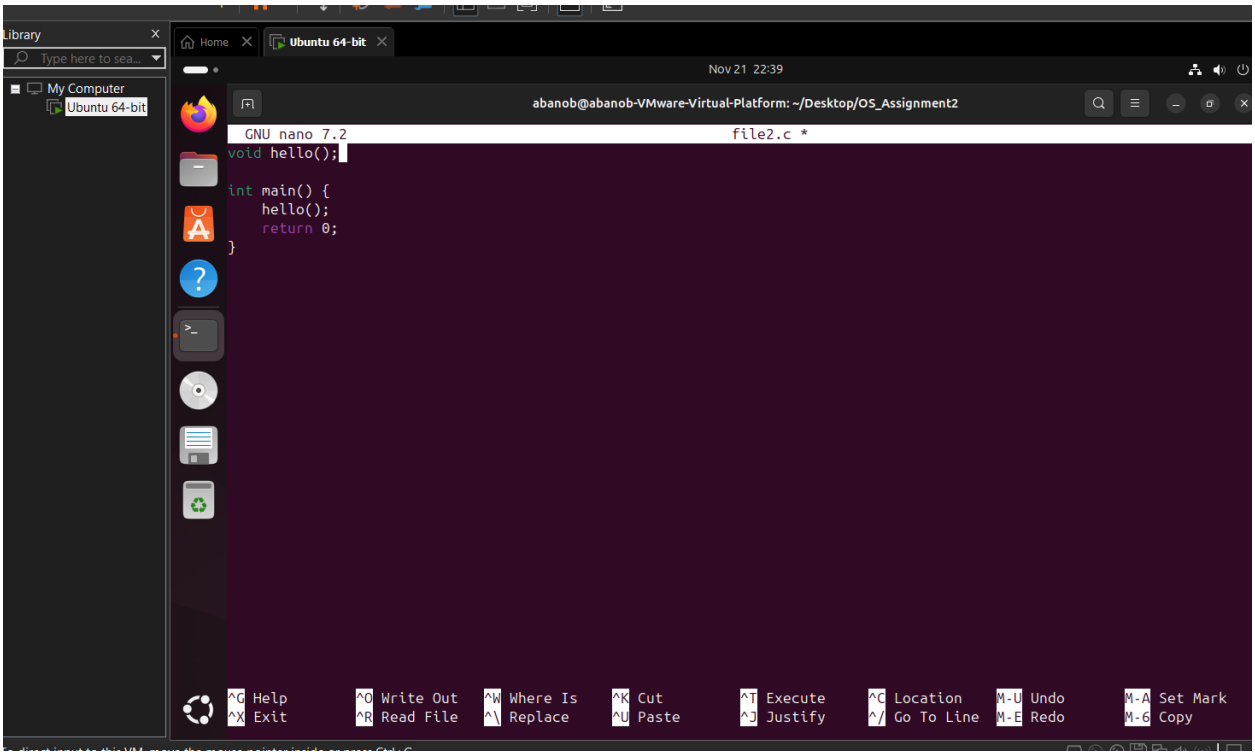
The linker's job is to connect the function call in file2.c to the actual function in file1.c.

After compiling and linking both files, the program runs and prints "Hello from file1!".

If we change something in file1.c, we need to recompile both files so the linker can update the executable with the new changes.


Library
Type here to sea...
My Computer
Ubuntu 64-bit
Home
Ubuntu 64-bit
Nov 21 22:39
abanob@abanob-VMware-Virtual-Platform: ~/Desktop/OS_Assignment2
abanob@abanob-VMware-Virtual-Platform:~/Desktop/OS_Assignment2$ nano file1.c
abanob@abanob-VMware-Virtual-Platform:~/Desktop/OS_Assignment2$ nano file2.c
abanob@abanob-VMware-Virtual-Platform:~/Desktop/OS_Assignment2$ gcc file1.c file2.c -o output_program
abanob@abanob-VMware-Virtual-Platform:~/Desktop/OS_Assignment2$ ./output_program
Hello from file1!
abanob@abanob-VMware-Virtual-Platform:~/Desktop/OS_Assignment2$


Library
Type here to sea...
My Computer
Ubuntu 64-bit
Home
Ubuntu 64-bit
Nov 21 22:38
abanob@abanob-VMware-Virtual-Platform: ~/Desktop/OS_Assignment2
GNU nano 7.2 file1.c *
#include <stdio.h>
void hello() {
printf("Hello from file1!\n");
}
^G Help ^O Write Out ^W Where Is ^K Cut ^T Execute ^C Location M-U Undo M-A Set Mark
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^/ Go To Line M-E Redo M-6 Copy



Exercise 6: Role of the Loader

Write a simple program and inspect the libraries it uses with ldd Simple Program:

#include <stdio.h> int
main() {

printf("This is a simple program.\n"); return 0;

}

Compile the program:

gcc simple_program.c -o simple_program Use
ldd to list the dynamic libraries:

ldd simple_program

Task: Identify which shared libraries are dynamically loaded when the program runs.

Explanation:

This program just prints a simple message.

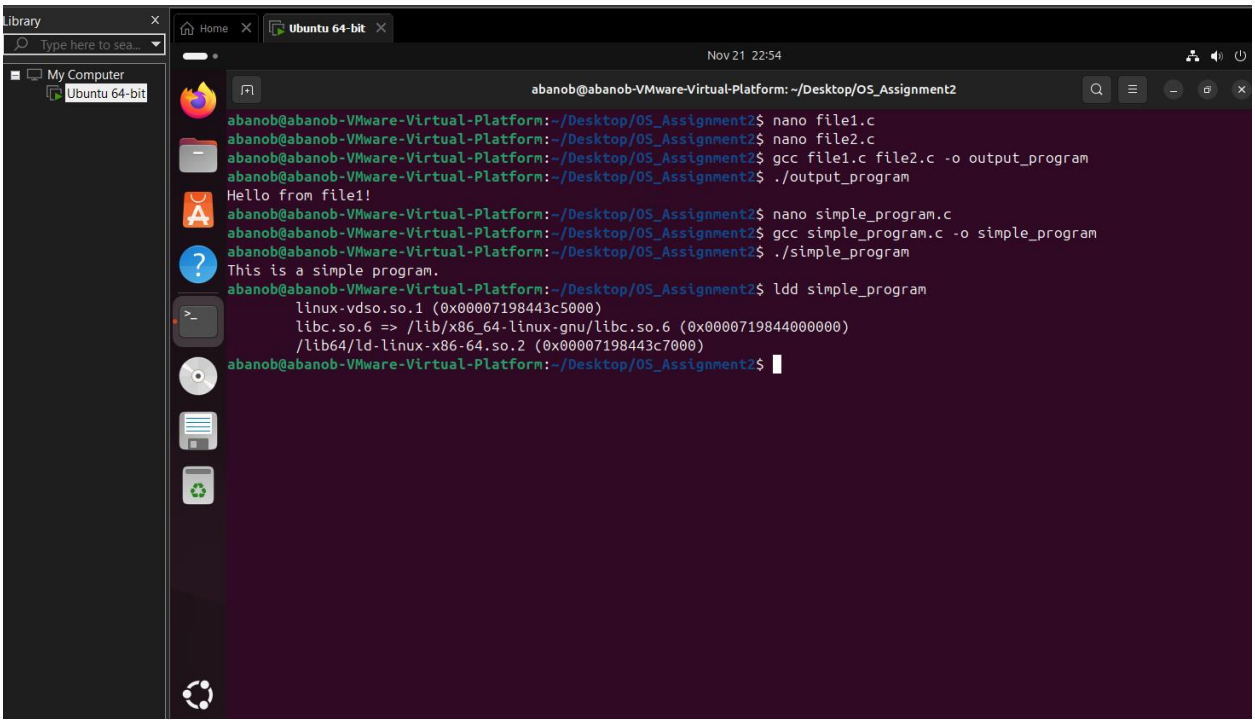
The loader is responsible for preparing the program to run.

When we run ldd on the executable, it shows which dynamic libraries are loaded into memory at runtime.

For example, libc.so.6 is the standard C library that provides

functions like printf.

The loader automatically links these libraries when the program starts.



Q2/ what is the job of the Linker.

The linker's job is to take all the compiled object files (.o) and combine them into a single executable program.

It connects function calls to their actual definitions. For example, if main() in one file calls hello() in another file, the linker makes sure the call in main() points to the correct function in memory.

Basically, the linker "links" all the pieces of your program together so it can run as one program.

Q3/ what is the job of the Loader.

The loader's job is to take the executable file created by the linker and load it into memory so the CPU can run it.

It sets up the program's memory space, loads all the necessary dynamic libraries, and then starts the program.

Basically, the loader makes sure the program is ready to run on your computer by putting everything in the right place in memory.