OS Refresher Module
Assignment 0.1

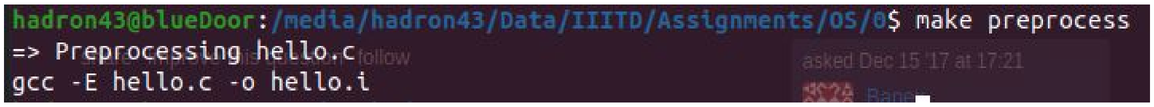
Name : Harsh Kumar **Roll No:** 2019043

The compilation of a C program to the final executable (binary) file is not direct, and in fact involves many intermediate steps. To perform any specific phase in the complete compilation process, I have made a makefile to perform one step at a time.

Let us observe each of these steps one by one:

1. **Preprocessing :** This step mainly does three major tasks:
    a. include all the header files in the code
    b. substitute the values of the macros defined
    c. get rid of all the comments in the code file

   This process leaves us with a '.i' file, which acts as an indication to the gcc compiler, that this file is preprocessed, and need not be preprocessed again.

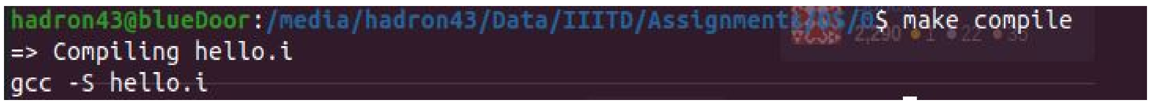
```
hadron43@blueDoor:/media/hadron43/Data/IIITD/Assignments/OS/0$ make preprocess
=> Preprocessing hello.c
gcc -E hello.c -o hello.i
```

   To stop the compilation at this stage, we have to pass the flag -E to the gcc compiler. In this particular example, we name the file 'hello.i' .

   The output file, "hello.i" contains preprocessed code, where the code from 'stdio.h' is included in the file, and replaced with the '#include' command. We haven't written any comments, or used any macros, or else they would have been resolved too.

2. **Compiling :** In this step, the compiler transforms the pre processed file to assembly language code. In this particular example, we are compiling on an x86_64 processor based system. Thus, we get a file named 'hello.s' which contains assembly instructions.

   Assembly code is architecture specific pneumonic based code. The extension used for assembly code is '.s'.

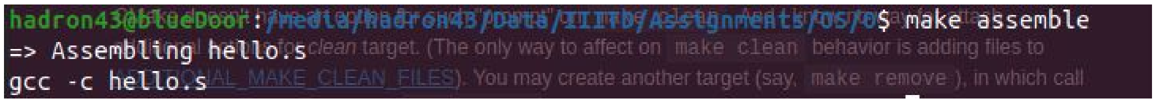
   

```
hadron43@blueDoor:/media/hadron43/Data/IIITD/Assignments/OS/0$ make compile
=> Compiling hello.i
gcc -S hello.i
```

To stop the compilation at this stage, we have to pass the flag -S. This tells the GCC compiler to terminate after the compilation step. By default, the output file is created with the same name as the source file, with the extension ‘.s’.

The output file, “hello.s” contains assembly instructions, which is essentially the **mnemonics** for machine level instructions for x86_64 architecture based CPU. It needs to be sent to assembler for further processing.

3. **Assembling :** This steps involves the assembler decoding the pneumonics based code from the compiled code, and finally transforming it to object code. The object code is still not executable. Though it is the lowest level of code, and consists of basic binary code, it is not yet executable, because it is not yet linked.



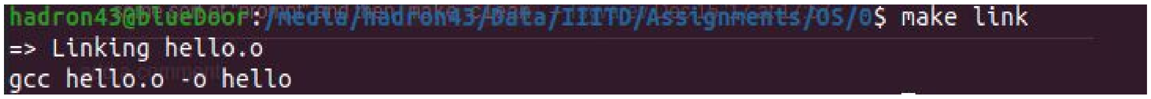
```
hadron43@blueDoor:/media/hadron43/Data/IIITD/Assignments/OS/0$ make assemble
=> Assembling hello.s
gcc -c hello.s
```

To stop compilation at this step, we have to pass the flag -c to the GCC compiler. The output is saved in a file with the same name as the source file, but the extension is changed to ‘.o’.

The output file “hello.o” contains the binary code, **without** resolving the references made to library function calls, and calls to functions from other files. This file needs to be linked with library files, and other required files for the project before it could be read by the machine.

4. **Linking :** In this step, mainly two things happen:
    a. Linking the function calls with their actual definition from the static as well as dynamic libraries
    b. Linking all source files together, to finally produce one single executable file

This step involves preparing the final executable file from the object code. Now the instructions are ready to be executed on the CPU. Also, any function calls are now linked to their respective code, and no references exist. The linker knows where to look for the definition in the static or the dynamic libraries.



```
hadron43@blueDoor:/media/hadron43/Data/IIITD/Assignments/OS/0$ make link
=> Linking hello.o
gcc hello.o -o hello
```

To link the files, we don’t have to pass any flag. The GCC compiler takes any file, at any of the previous defined stages, and prepares a binary executable file for it. The output file is by default named as ‘a.out’. He, we are specifying the file name as hello, without any extension.

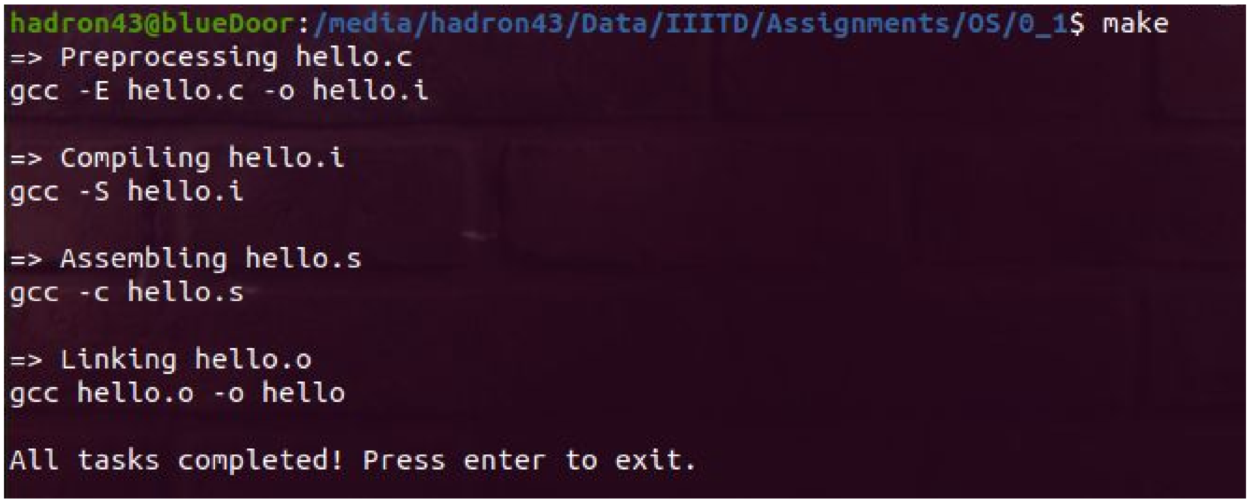
The output file “hello” is complete binary code for the program, ready to be executed on any x86_64 based machine. Though when we try to open the file in a text editor, we see hexadecimal form of the binary code, the code actually consists of just zeroes and ones.

To sum it all, we use the following flags for GCC for stopping the compilation procedure at various steps:

a. -E flag : stop after preprocessing
b. -S flag : stop after compiling
c. -c flag : stop after assembling
d. no flag : complete compilation, and prepare binary executable file

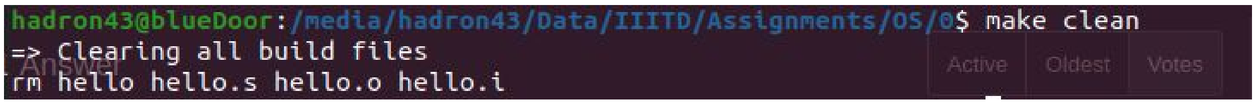
In the default mode, where we don’t specify any target while calling make, we perform each step, and wait for the user to press enter. Then we proceed with the next step, and so on.

Here, --no-print-directory flag is passed to the GCC compiler, to make it silent while entering and leaving any directory, to avoid unnecessary clutter in the output window.



```
hadron43@blueDoor:/media/hadron43/Data/IIITD/Assignments/OS/0_1$ make
=> Preprocessing hello.c
gcc -E hello.c -o hello.i

=> Compiling hello.i
gcc -S hello.i

=> Assembling hello.s
gcc -c hello.s

=> Linking hello.o
gcc hello.o -o hello

All tasks completed! Press enter to exit.
```

We have also added a clean target in makefile, to just clean all the build files that we generate during the program.



```
hadron43@blueDoor:/media/hadron43/Data/IIITD/Assignments/OS/0$ make clean
=> Clearing all build files
rm hello hello.s hello.o hello.i
```