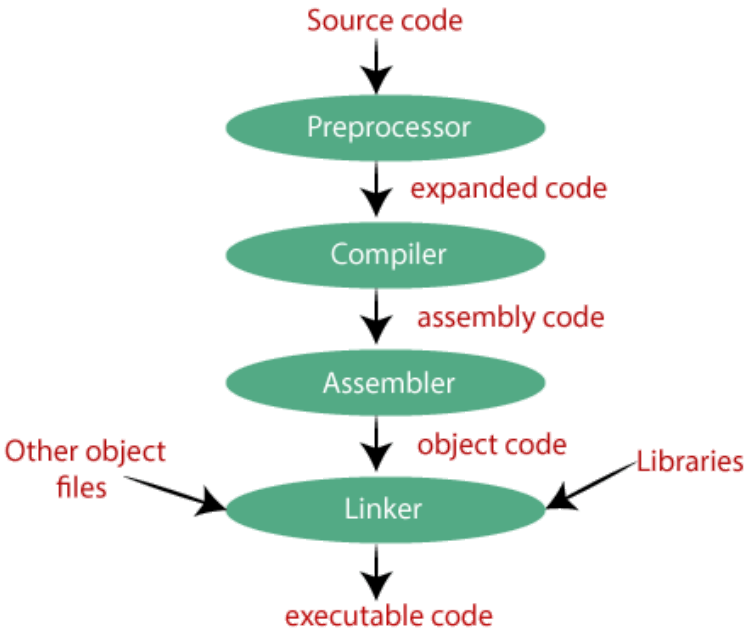
OPERATING SYSTEMS

ASSIGNMENT 0.1

UNDERSTANDING PROGRAM COMPILATION PROCESS

SUBMITTED BY
DOLLY SIDAR (2019304)

THE STEPS



Source Code

hello.c

```
home > dolly > Desktop > C hello.c > ...
1  # include <stdio.h> // stdio header file contain function printf
2
3
4  //main function
5  int main(){
6      int a=2;    //1st local variable intialised value 2
7      int b=3;    //2nd local variable inialised value 3
8      printf("a = %d \n" ,a); //print the value of a
9      printf("b = %d", b);    //print the value of b
10     return 0;
11 }
12
```

Pre - Processing

This is the first phase through which source code is passed. It obeys the commands that begin with # (which is known as preprocessor directives) by:

➢ rid of all the comments in the source file.

- ➢ Expanding macros - it replaces all of the macros by their values. (like symbolic constant defined using #define is replaced by their value)
- ➢ Expanding included files, i.e., header files that contain C function declaration and macro definitions.

The output of the above step will be stored in a file with a " **.i** " extension, in my case it is **hello.i**, and in order to stop the compilation right after this step. we can use the flag "**-E**" with the GCC command on the source file and press enter ."**-o"** is used for the output file name.

```
gcc -E hello.c -o hello.i
```

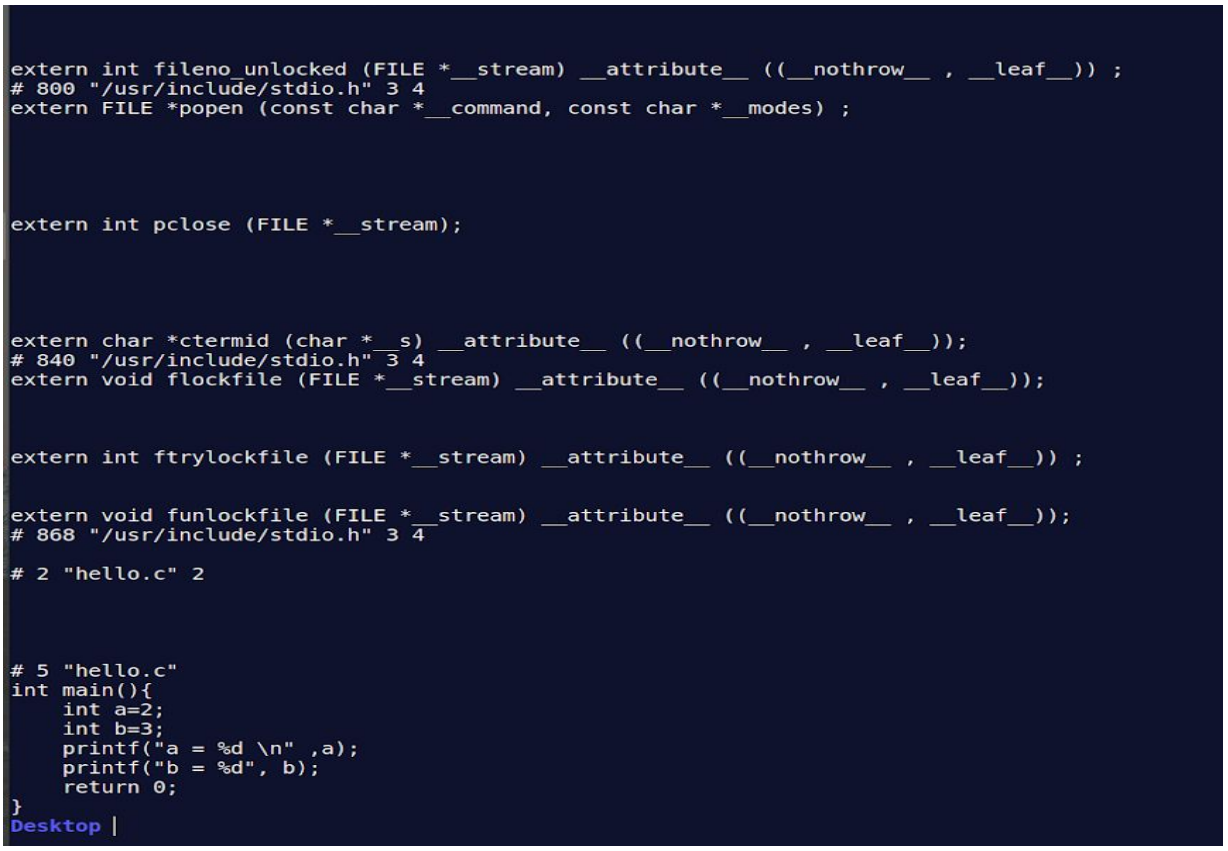
We will be using Makefile to run this command. Makefile is a set of commands similar to the terminal with variables names and targets to create the object files.
Here my target name for the above step compilation is a preprocessor then followed by command with tab space. (if you forgot to put the tab this program will not run)

```
Desktop make preprocessor
gcc -E hello.c -o hello.i
cat hello.i
# 1 "hello.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "hello.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
# 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 424 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
# 427 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 428 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/long-double.h" 1 3 4
# 429 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 425 "/usr/include/features.h" 2 3 4
# 448 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 1 3 4
# 10 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/stubs-64.h" 1 3 4
# 11 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 2 3 4
# 449 "/usr/include/features.h" 2 3 4
# 34 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 2 3 4
# 28 "/usr/include/stdio.h" 2 3 4




# 1 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 1 3 4
# 216 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 3 4

# 216 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 3 4
typedef long unsigned int size_t;
# 34 "/usr/include/stdio.h" 2 3 4
```

To execute the Makefile, you have to use the **make** command followed by the target name, but this not necessary if you do not specify the target name, it will automatically run all the targets in Makefile with their commands or run just the default target.

As you can see, its also showing commands under preprocessor target followed by their compilation. I am using **cat hello.i** to show what inside the hello.i.

The hello.i file contains tons of info, but I am showing the end part of the code.



```
extern int fileno_unlocked (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;
# 800 "/usr/include/stdio.h" 3 4
extern FILE *popen (const char *__command, const char *__modes) ;




extern int pclose (FILE *__stream);




extern char *ctermid (char *__s) __attribute__ ((__nothrow__ , __leaf__));
# 840 "/usr/include/stdio.h" 3 4
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));


extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;


extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
# 868 "/usr/include/stdio.h" 3 4

# 2 "hello.c" 2




# 5 "hello.c"
int main(){
    int a=2;
    int b=3;
    printf("a = %d \n" ,a);
    printf("b = %d", b);
    return 0;
}
Desktop |
```

Analysis of the output file:

- ➢ Comments are stripped off.
- ➢ #include <stdio.h> is not there; instead, we can see lots of code that means files have been expanded and included in our source file, or we can say system search for the stdio.h file and copy the header file into the source code file.

Compiling:

It is the second step. It takes the output of the preprocessor, i.e., hello.i file and generate assembly language, an intermediate human-readable language, specific to the target processor. This step will produce a **“.s”** file, and we can stop after this step with the **“-S”** flag on the gcc command, and press enter.

```
gcc -S hello.c -o hello.s
```

Compilation by makefile

```
Desktop make compiler
gcc -S hello.c -o hello.s
cat hello.s
        .file   "hello.c"
        .text
        .section        .rodata
.LC0:
        .string "a = %d \n"
.LC1:
        .string "b = %d"
        .text
        .globl  main
        .type   main, @function
main:
.LFB5:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $2, -8(%rbp)
        movl    $3, -4(%rbp)
        movl    -8(%rbp), %eax
        movl    %eax, %esi
        leaq    .LC0(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        movl    -4(%rbp), %eax
        movl    %eax, %esi
        leaq    .LC1(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

Similar to above, but here using **make compiler**.

hello.s file contains assembly instruction, which can be understood by the assembler.

Assembly

During this stage, the assembler takes the assembly instruction, i.e., hello.s file and transforms it into object code, which is code in machine language, i.e., binary form. This will produce a file ending with **“.o”.**we can stop the compilation process after this step by using the flag **“-c”** with the gcc command, and pressing enter.

```
gcc -c hello.c -o hello.o
```

For compilation using **make assembler**, **hello.o** the file contains machine-level instruction (it’s not human-readable).

Linking

This is the final phase; the object code generated within the assembly stage consists of machine instructions that the processor understands, but some parts of the program are out of order or missing. The linker does some extra work. It adds some additional code to our program, which is required when the program starts and ends. To produce a final executable program, the existing pieces need to be rearranged, and therefore the missing ones should be filled in. Linker arranges the piece of object code so that function in some parts can successfully call the methods in other ones. The linker exactly knows where to look for the function definitions, like in the static libraries or dynamic libraries. In Dynamic libraries, the code is not copied. It is done by just placing the name of the library in the binary file.
In contrast, Static libraries result from the linker making a copy of all used library functions to the executable file. gcc uses by default dynamic libraries. In my source code, this is when the linker will find the definition of the **"printf"** function and dynamically linked it.
This stage results in the final executable program, i.e., when you type the **"gcc hello.c"** command without any flag, the compiler will automatically create an executable program with file name a.out, that we can run with the command **"./a.out,"** or we can also name the file something else by adding the **"-o"** option to the gcc command, placed after the name of the file we are compiling and then run with the command **./<filename>.**

```
Desktop gcc hello.c
Desktop ./a.out
a = 2
b = 3%
Desktop gcc hello.c -o hello
Desktop ./hello
a = 2
b = 3%
Desktop
```

```
Desktop make linker
gcc hello.c -o hello
./hello
a = 2
b = 3%
Desktop
```

Using **make linker** command through compilation by Makefile.
So we can see, it makes the file executable.