**EX NO:1 FUNDAMENTALS**

**Date:**

Fundamental Study for

OS161 - Instructional Operating System

This tutorial is aimed at those who have just started using Linux. Generally when users from the Windows background enter the Linux scene,they are totally stumped by the software installation method. They were used to the luxury of double clicking on a single file and getting their software installed. But now they have to type cryptic commands to do the same.

Though the installation instructions in **the Next Experiment** tell you what to do, we may not have any idea what those steps actually do. This article shall explain the basics of software installation. After reading this article we might feel more at home.

We Will Start With The Floor Level Fundamentals.

A surgery on “C Program Compilation”

If you have ever wondered what happens during the compilation process and how the C program gets converted to an executable – then here it is;

There are four main stages through which a source code passes in order to finally become an executable.

The four stages for a C program to become an executable are the following:

1. Pre-processing

**You shall also write without macro – but not this time**

#include <stdio.h>

Void main()

{

printf(“hello world”);

getch();

}

2. Compilation

3. Assembly

4. Linking

Let’s take a quick look on

**how to compile and run a ‘C’ code using gcc, using a simple hello world example.**

//open any text editor in your system & have the following program in it.

// Using a macro to print 'Hello World' //

**#include <stdio.h>**

**#define STRING "Hello World"**

**int main(void)**

**{**

**printf(STRING);**

**return 0;**

**}**

// Save the program as **print.c**

Now, lets run gcc compiler over this source code to create the executable.

**$ gcc -Wall print.c -o print**

In the above command:

* gcc – Invokes the GNU C compiler
* -Wall – gcc flag that enables all warnings. -W stands for warning, and we are passing “all” to -W.
* print.c – Input C program
* -o print – Instruct C compiler to create the C executable as print. If you don’t specify -o, by default C compiler will create the executable with name a.out

Finally, execute print which will execute the C program and display hello world.

**$ ./print**

**Hello World**

*When we work on a big project that contains several C program, we will use make utility to manage your C program compilation, which we will discuss in the sections below.*

Now that we have a basic idea about how gcc is used to convert a source code into binary, we’ll review the 4 stages a C program has to go through to become an executable.

**1. PRE-PROCESSING**

This is the very first stage through which a source code passes. In this stage the following tasks are done:

**Macro substitution**

**Comments are stripped off**

**Expansion of the included files**

**To understand better compilation, you can compile the above ‘print.c’ program using flag -save-temps,** as shown below.

‘-save-temps’ flag instructs compiler to store the temporary intermediate files used by the gcc compiler in the current directory.

**$ gcc -Wall -save-temps print.c -o print**

So when we compile the program print.c with -save-temps flag we get the following intermediate files in the current directory (along with the print executable)

$ ls

print.i

print.s

print.o

**You can also compile the above ‘print.c’ program using flag -E, which will print ONLY the preprocessed output to stdout.**

**$ gcc -Wall -E print.c**

The preprocessed output is stored in the temporary file that has the extension .i (i.e ‘print.i’ in this example)

Now, lets open print.i file and view the content. (using any text editor like GEDIT or VI)

**$ vi print.i**

**......**

**......**

**# 846 "/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\_\_));**

**# 886 "/usr/include/stdio.h" 3 4**

**extern void flockfile (FILE \*\_\_stream) \_\_attribute\_\_ ((\_\_nothrow\_\_));**

**extern int ftrylockfile (FILE \*\_\_stream) \_\_attribute\_\_ ((\_\_nothrow\_\_)) ;**

**extern void funlockfile (FILE \*\_\_stream) \_\_attribute\_\_ ((\_\_nothrow\_\_));**

**# 916 "/usr/include/stdio.h" 3 4**

**# 2 "print.c" 2**

**int main(void)**

**{**

**printf("Hello World");**

**return 0;**

**}**

In the above output, you can see that the source file is now filled with lots and lots of information, but still at the end of it we can see the lines of code written by us. Lets analyze on these lines of code first.

**The first observation is that the argument to printf() now contains directly the string “Hello World” rather than the macro.**

In fact the macro definition and usage has completely disappeared. This proves the first task that all the macros are expanded in the preprocessing stage.

**The second observation is that the comment that we wrote in our original code is not there.**

This proves that all the comments are stripped off.

**The third observation is that beside the line ‘#include’ is missing and instead of that we see whole lot of code in its place.**

So its safe to conclude that stdio.h has been expanded and literally included in our source file. Hence we understand how the compiler is able to see the declaration of printf() function.

When I searched print.i file, I found, The function printf is declared as:

extern int printf (\_\_const char \*\_\_restrict \_\_format, ...);

The keyword ‘extern’ tells that the function printf() is not defined here. It is external to this file. We will later see how gcc gets to the definition of printf().

Now that we have a decent understanding on what happens during the preprocessing stage. let us move on to the next stage.

**2. COMPILING**

After the compiler is done with the pre-processor stage,

The next step is to take print.i as input, compile it and produce an intermediate compiled output. The output file for this stage is ‘print.s’. The output present in print.s is assembly level instructions.

Open the print.s file in an editor and view the content.

$ vi print.s

.file "print.c"

.section .rodata

.LC0:

.string "Hello World"

.text

.globl main

.type main, @function

main:

.LFB0:

.cfi\_startproc

pushq %rbp

.cfi\_def\_cfa\_offset 16

movq %rsp, %rbp

.cfi\_offset 6, -16

.cfi\_def\_cfa\_register 6

movl $.LC0, %eax

movq %rax, %rdi

movl $0, %eax

call printf

movl $0, %eax

leave

ret

.cfi\_endproc

.LFE0:

.size main, .-main

.ident "GCC: (Ubuntu 4.4.3-4ubuntu5) 4.4.3"

.section .note.GNU-stack,"",@progbits

Though we are not much into assembly level programming but a quick look concludes that this assembly level output is in some form of instructions which the assembler can understand and convert it into machine level language.

**3. ASSEMBLY**

At this stage the print.s file is taken as an input and an intermediate file **print.o** is produced. This file is also **known as the object file**.

This file is produced by the assembler that understands and converts a ‘.s’ file with assembly instructions into a ‘.o’ object file which contains machine level instructions. **At this stage only the existing code is converted into machine language, the function calls like printf() are not resolved.**

Since the output of this stage is a machine level file (print.o). So we cannot view the content of it. If you still try to open the print.o and view it, you’ll see something that is totally not readable.

$ vi print.o

^?ELF^B^A^A^@^@^@^@^@^@^@^@^@^A^@>^@^A^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@0^

^@UH<89>å¸^@^@^@^@H<89>Ç¸Hello World^@^@GCC: (Ubuntu 4.4.3-4ubuntu5) 4.4.3^@^

T^@^@^@^@^@^@^@^AzR^@^Ax^P^A^[^L^G^H<90>^A^@^@^\^@^@]^@^@^@^@A^N^PC<86>^B^M^F

^@^@^@^@^@^@^@^@.symtab^@.strtab^@.shstrtab^@.rela.text^@.data^@.bss^@.rodata

^@.comment^@.note.GNU-stack^@.rela.eh\_frame^@^@^@^@^@^@^@^@^@^@^@^

...

...

…

The only thing we can explain by looking at the print.o file is about the string ELF.

**ELF stands for executable and linkable format.**

This is a relatively new format for machine level object files and executable that are produced by gcc. **Prior to this, a format known as a.out was used**. ELF is said to be more sophisticated format than a.out (We might in some other future article).

\*\* If you still need to dig deeper into the ELF format – Kindly Contact Faculty In-Charge.

Note: If you compile your code without specifying the name of the output file, the output file produced has name ‘a.out’ but the format now have changed to ELF. It is just that the default executable file name remains the same.

**4. LINKING**

This is the final stage at which all the linking of function calls with their definitions are done. As discussed earlier, till this stage gcc doesn’t know about the definition of functions like printf(). Until the compiler knows exactly where all of these functions are implemented, it simply uses a place-holder for the function call. **It is at this stage, the definition of printf() is resolved and the actual address of the function printf() is plugged in.**

The linker comes into action at this stage and does this task.

The linker also does some extra work; it combines some extra code to our program that is required when the program starts and when the program ends. For example, there is code which is standard for setting up the running environment like passing command line arguments, passing environment variables to every program. Similarly some standard code that is required to return the return value of the program to the system.

The above tasks of the compiler can be verified by a small experiment. Since now we already know that the linker converts .o file (print.o) to an executable file (print).

So if we compare the file sizes of both the print.o and print file, we’ll see the difference.

$ size print.o

text data bss dec hex filename

97 0 0 97 61 print.o

$ size print

text data bss dec hex filename

1181 520 16 1717 6b5 print

Through the size command we get a rough idea about how the size of the output file increases from an object file to an executable file. This is all because of that extra standard code that linker combines with our program.

Now you know what happens to a C program before it becomes an executable. You know about Preprocessing, Compiling, Assembly, and Linking stages, which we will cover in our next article in this series.

\*\* There is lot more to the linking stage. If you wanna dig deeper – Kindly Google.

|  |  |  |  |
| --- | --- | --- | --- |
| **Sl No:** | **Assessment Process Description** | | **Mark(s)** |
| 1 | Precise answering | | 5 |
| 2 | Explanation | | 5 |
|  | Total(10) | | 10 |
| Remarks |  | | |
| Date of Completion: | | Signature | |

**Result:**