THE GEEK STUFF
Linux | DB | Open Source | Web

≡ Menu

- Home
- Free eBook
- Start Here
- Contact
- About

Journey of a C Program to Linux Executable in 4 Stages

by Himanshu Arora on October 5, 2011

You write a C program, use gcc to compile it, and you get an executable. It is pretty simple. Right?

Have you ever wondered what happens during the compilation process and how the C program gets converted to an executable?

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
2. Compilation
3. Assembly
4. Linking

In Part-I of this article series, we will discuss the steps that the gcc compiler goes through when a C program source code is compiled into an executable.

Before going any further, lets take a quick look on how to compile and run a 'C' code using gcc, using a simple hello world example.

```
$ vi print.c
#include <stdio.h>
#define STRING "Hello World"
int main(void)
{
/* Using a macro to print 'Hello World'*/
printf(STRING);
return 0;
}
```

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
```

Note: When you are working on a big project that contains several C program, use make utility to manage your C program compilation as we discussed earlier.

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:

1. Macro substitution
2. Comments are stripped off

3. Expansion of the included files

To understand preprocessing better, you can compile the above 'print.c' program using flag -E, which will print the preprocessed output to stdout.

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

Even better, you can use 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
```

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.

```
$ 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.

1. 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.
2. 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.
3. 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().

You can use gdb to debug your c programs. 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 I am 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 dig deeper into the ELF format in some other future article).

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 There is lot more to the linking stage, which we will cover in our next article in this series.

If you enjoyed this article, you might also like..

1. 50 Linux Sysadmin Tutorials
2. 50 Most Frequently Used Linux Commands (With Examples)
3. Top 25 Best Linux Performance Monitoring and Debugging Tools
4. Mommy, I found it! – 15 Practical Linux Find Command Examples
5. Linux 101 Hacks 2nd Edition eBook Free

- Awk Introduction – 7 Awk Print Examples
- Advanced Sed Substitution Examples
- 8 Essential Vim Editor Navigation Fundamentals
- 25 Most Frequently Used Linux IPTables Rules Examples
- Turbocharge PuTTY with 12 Powerful Add-Ons