Explore GCC Linking Process Using LDD, Readelf, and Objdump

by Himanshu Arora on October 17, 2011

Linking is the final stage of the gcc compilation process.

In the linking process, object files are linked together and all the references to external symbols are resolved, final addresses are assigned to function calls, etc.

In this article we will mainly focus on the following aspects of gcc linking process:

1. Object files and how are they linked together
2. Code relocations

Before you read this article, make sure you understand all the 4 stages that a C program has to go through before becoming an executable (pre-processing, compilation, assembly and linking).

LINKING OBJECT FILES

Lets understand this first step through an example. First create the following main.c program.

```
$ vi main.c
#include <stdio.h>

extern void func(void);

int main(void)
{
    printf("\n Inside main()\n");
    func();
```

```
    return 0;
}
```

Next create the following func.c program. In the file main.c we have declared a function func() through keyword 'extern' and have defined this function in a separate file func.c

```
$ vi func.c
void func(void)
{
    printf("\n Inside func()\n");
}
```

Create the object file for func.c as shown below. This will create the file func.o in the current directory.

```
$ gcc -c func.c
```

Similarly create the object file for main.c as shown below. This will create the file main.o in the current directory.

```
$ gcc -c main.c
```

Now execute the following command to link these two object files to produce a final executable. This will create the file 'main' in the current directory.

```
$ gcc func.o main.o -o main
```

When you execute this 'main' program you'll see the following output.

```
$ ./main
Inside main()
Inside func()
```

From the above output, it is clear that we were able to link the two object files successfully into a final executable.

What did we acheive when we separated function func() from main.c and wrote it in func.c?

The answer is that here it may not have mattered much if we would have written the function func() in the same file too but think of very large programs where we might have thousands of lines of code. A change to one line of code could result in recompilation of the whole source code which is not accceptable in most cases. So, very large programs are sometimes divided into small peices which are finaly linked together to produce the executable.

The make utility which works on makefiles comes into the play in most of these situations because this utility knows which source files have been changed and which object files need to be recompiled. The object files whose corresponding source files have not been altered are linked as it is. This makes the compilation process very easy and manageable.

So, now we understand that when we link the two object files func.o and main.o, the gcc linker is able to resolve the function call to func() and when the final executable main is executed, we see the printf() inside the function func() being executed.

Where did the linker find the definition of the function printf()? Since Linker did not give any error that surely means that linker found the definition of printf(). printf() is a function which is declared in stdio.h and defined as a part of standard 'C' shared library (libc.so)

We did not link this shared object file to our program. So, how did this work? Use the ldd tool to find out, which prints the shared libraries required by each program or shared library specified on the command line.

Execute ldd on the 'main' executable, which will display the following output.

```
$ ldd main
linux-vdso.so.1 =>  (0x00007fff1c1ff000)
libc.so.6 => /lib/libc.so.6 (0x00007f32fa6ad000)
```

```
/lib64/ld-linux-x86-64.so.2 (0x00007f32faa4f000)
```

The above output indicates that the main executable depends on three libraries. The second line in the above output is 'libc.so.6' (standard 'C" library). This is how gcc linker is able to resolve the function call to printf().

The first library is required for making system calls while the third shared library is the one which loads all the other shared libraries required by the executable. This library will be present for every executable which depends on any other shared libraries for its execution.

During linking, the command that is internally used by gcc is very long but from users prespective, we just have to write.

```
$ gcc <object files> -o <output file name>
```

CODE RELOCATION

Relocations are entries within a binary that are left to be filled at link time or run time. A typical relocation entry says: Find the value of 'z' and put that value into the final executable at offset 'x'

Create the following reloc.c for this example.

```
$ vi reloc.c
extern void func(void);

void func1(void)
{
    func();
}
```

In the above reloc.c we declared a function func() whose definition is still not provided, but we are calling that function in func1().

Create an object file reloc.o from reloc.c as shown below.

```
$ gcc -c reloc.c -o reloc.o
```

Use readelf utility to see the relocations in this object file as shown below.

```
$ readelf --relocs reloc.o
Relocation section '.rela.text' at offset 0x510 contains 1 entries:
Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000005  000900000002 R_X86_64_PC32     0000000000000000 func - 4
...
```

The address of func() is not known at the time we make reloc.o so the compiler leaves a relocation of type R_X86_64_PC32. This relocation indirectly says that "fill the address of the function func() in the final executable at offset 000000000005".

The above relocation was corresponding to the .text section in the object file reloc.o (again one needs to understand the structure of ELF files to understand various sections) so lets disassemble the .text section using objdump utility:

```
$ objdump --disassemble reloc.o
reloc.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <func1>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   e8 00 00 00 00          callq  9 <func1+0x9>
   9:   c9                      leaveq
   a:   c3                      retq
```

In the above output, the offset '5' (entry with value '4' relative to starting address 0000000000000000) has 4 bytes waiting to be writen with the address of function func().

So, there is a relocation pending for the function func() which will get resolved when we link reloc.o with the object file or library that contains the defination of function func().

Lets try and see whether this relocation gets reolved or not. Here is another file main.c that provides defination of func() :

```
$ vi main.c
#include<stdio.h>

void func(void) // Provides the defination
{
    printf("\n Inside func()\n");
}

int main(void)
{
    printf("\n Inside main()\n");
    func1();
    return 0;
}
```

Create main.o object file from main.c as shown below.

```
$ gcc -c main.c -o main.o
```

Link reloc.o with main.o and try to produce an executable as shown below.

```
$ gcc reloc.o main.o -o reloc
```

Execute objdump again and see whether the relocation has been resolved or not:

```
$ objdump --disassemble reloc > output.txt
```

We redirected the output because an executable contains lots and lots of information and we do not want to get lost on stdout.
View the content of the output.txt file.

```
$ vi output.txt
...
0000000000400524 <func1>:
400524:       55                        push   %rbp
400525:       48 89 e5                  mov    %rsp,%rbp
400528:       e8 03 00 00 00            callq  400530 <func>
40052d:       c9                        leaveq
40052e:       c3                        retq
40052f:       90                        nop
...
```

In the 4th line, we can clearly see that the empty address bytes that we saw earlier are now filled with the address of function func().

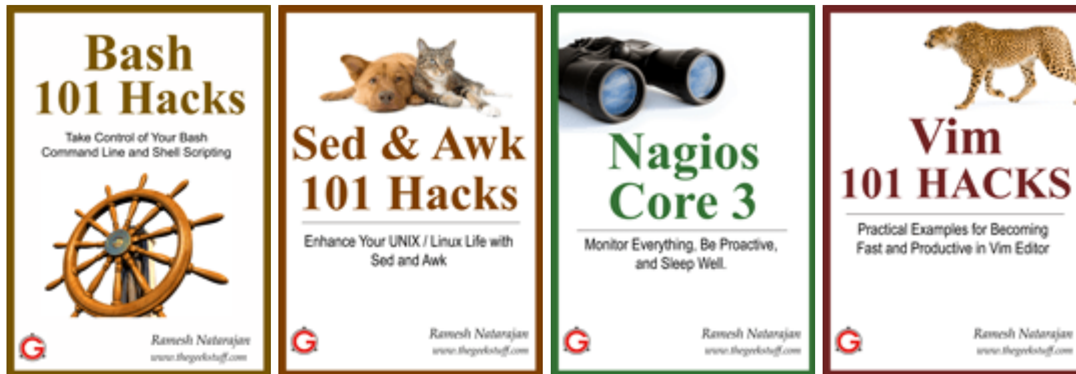
To conclude, gcc compiler linking is such a vast sea to dive in that it cannot be covered in one article. Still, this article made an attempt to peel off the first layer of linking process to give you an idea about what happens beneath the gcc command that promises to link different object files to produce an executable.

> Add your comment

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



{ 9 comments… add one }

- E. Menout October 17, 2011, 5:42 am

  Very Good. Thanks

  Link
- Jalal Hajigholamali October 17, 2011, 6:18 am

  Hi,
  Very nice and usable article

  Link
- behzad October 18, 2011, 11:32 am

  thanks again for another quality article,
  it will be great if you mention at the end a few sources or references that you would recommend for the people who want to know more, with a short comment on each.

  Link
- Himanshu October 18, 2011, 6:56 pm

  @behzad.
  Sure I'll take care of this from now on and will definitely add some references at the end of my articles.

  Link
- Viren October 19, 2011, 6:16 am

  In the last example's program you are calling func1() but the defined function name is func().
  Please correct it.

  Link
- Himanshu October 19, 2011, 11:05 pm

  @Viren
  I am calling func1() which is defined in reloc.c.

  Link
- Arun Saha November 11, 2011, 12:00 am

Consider including in func.c so that it does not throw implicit declaration warning.

Link

- vaibhav sharma February 24, 2013, 2:59 am

  very good material

  Link

- BSpider March 15, 2015, 8:22 am

  whats the difference between

  $ gcc -o

  and

  $ gcc -c reloc.c -o reloc.o

  "-o" operates differently in both lines
  could you explain it ?

  Thanks,

  Link

Leave a Comment

Name

Email

Website

Comment

Submit

Notify me of followup comments via e-mail

Next post: How to Backup Ubuntu Desktop Using sbackup Simple Backup GNOME Tool

Previous post: Dennis Ritchie – Father of C Programming Language

RSS | Email | Twitter | Facebook | Google+

Search

EBOOKS

- Free Linux 101 Hacks 2nd Edition eBook - Practical Examples to Build a Strong Foundation in Linux
- Bash 101 Hacks eBook - Take Control of Your Bash Command Line and Shell Scripting

- Sed and Awk 101 Hacks eBook - Enhance Your UNIX / Linux Life with Sed and Awk
- Vim 101 Hacks eBook - Practical Examples for Becoming Fast and Productive in Vim Editor
- Nagios Core 3 eBook - Monitor Everything, Be Proactive, and Sleep Well

The Geek Stuff
17 244 Me gusta

Me gusta esta página | Compartir

Sé el primero de tus amigos en indicar que te gusta.

POPULAR POSTS

- 15 Essential Accessories for Your Nikon or Canon DSLR Camera
- 12 Amazing and Essential Linux Books To Enrich Your Brain and Library
- 50 UNIX / Linux Sysadmin Tutorials
- 50 Most Frequently Used UNIX / Linux Commands (With Examples)
- How To Be Productive and Get Things Done Using GTD
- 30 Things To Do When you are Bored and have a Computer
- Linux Directory Structure (File System Structure) Explained with Examples
- Linux Crontab: 15 Awesome Cron Job Examples
- Get a Grip on the Grep! – 15 Practical Grep Command Examples
- Unix LS Command: 15 Practical Examples
- 15 Examples To Master Linux Command Line History
- Top 10 Open Source Bug Tracking System
- Vi and Vim Macro Tutorial: How To Record and Play
- Mommy, I found it! -- 15 Practical Linux Find Command Examples
- 15 Awesome Gmail Tips and Tricks
- 15 Awesome Google Search Tips and Tricks
- RAID 0, RAID 1, RAID 5, RAID 10 Explained with Diagrams
- Can You Top This? 15 Practical Linux Top Command Examples
- Top 5 Best System Monitoring Tools
- Top 5 Best Linux OS Distributions
- How To Monitor Remote Linux Host using Nagios 3.0
- Awk Introduction Tutorial – 7 Awk Print Examples
- How to Backup Linux? 15 rsync Command Examples
- The Ultimate Wget Download Guide With 15 Awesome Examples
- Top 5 Best Linux Text Editors
- Packet Analyzer: 15 TCPDUMP Command Examples
- The Ultimate Bash Array Tutorial with 15 Examples
- 3 Steps to Perform SSH Login Without Password Using ssh-keygen & ssh-copy-id
- Unix Sed Tutorial: Advanced Sed Substitution Examples
- UNIX / Linux: 10 Netstat Command Examples
- The Ultimate Guide for Creating Strong Passwords
- 6 Steps to Secure Your Home Wireless Network
- Turbocharge PuTTY with 12 Powerful Add-Ons

CATEGORIES

- Linux Tutorials
- Vim Editor
- Sed Scripting

- Awk Scripting
- Bash Shell Scripting
- Nagios Monitoring
- OpenSSH
- IPTables Firewall
- Apache Web Server
- MySQL Database
- Perl Programming
- Google Tutorials
- Ubuntu Tutorials
- PostgreSQL DB
- Hello World Examples
- C Programming
- C++ Programming
- DELL Server Tutorials
- Oracle Database
- VMware Tutorials

About The Geek Stuff

My name is **Ramesh Natarajan**. I will be posting instruction guides, how-to, troubleshooting tips and tricks on Linux, database, hardware, security and web. My focus is to write articles that will either teach you or help you resolve a problem. Read more about Ramesh Natarajan and the blog.

Contact Us

Email Me : Use this Contact Form to get in touch me with your comments, questions or suggestions about this site. You can also simply drop me a line to say hello!.

Follow us on Google+

Follow us on Twitter

Become a fan on Facebook

Support Us

Support this blog by purchasing one of my ebooks.

Bash 101 Hacks eBook

Sed and Awk 101 Hacks eBook

Vim 101 Hacks eBook

Nagios Core 3 eBook

Copyright © 2008–2018 Ramesh Natarajan. All rights reserved | Terms of Service