

# 课程大纲

第0部分 课程简介 - 从Hello World谈起

第1部分 Linux编译 过程总体概览

1.1 预编译

1.2 编译

1.3 汇编

1.4 链接

第2部分 Linux 目标文件

2.1 初识Linux目标文件

2.2 Linux目标文件分段解析(ELF格式)

第3部分 Linux 静态链接

3.1 Linux静态链接的过程

第4部分 Linux可执行文件的装载

4.1 可执行文件的装载过程

4.2 从内核角度解读可执行文件的装载

第5部分 Linux动态链接

5.1 为什么要进行动态链接

5.2 动态链接的例子

5.3 动态链接在Linux下的实现机制

第6部分 库和运行库

6.1 Linux程序入口函数和程序初始化

6.2 运行库简介

本课程演示过程中所使用的Linux环境：CentOS6.6 - GCC版本 4.8.2

读者可以自行选择自己熟悉的Linux操作系统发行版进行实验环境的搭建和学习。

# 第0部分

-从Hello World谈起

# 首先问问你自己

- 你真的理解你闭着眼睛都能敲出来的Hello World程序么？
  - 为什么需要头文件？
  - 编译器在把C语言程序转换成可执行机器码的过程中究竟做了什么？
  - 最后编译出来的可执行文件里面究竟有什么？
  - 程序是怎么运行起来的，操作系统是怎么装载它的，调用main函数之前又发生了什么？
  - 程序在运行时，它在内存中的布局又是什么样子的？
  - ...

源代码

编译和运行

```
#include <stdio.h>

int main()
{
    printf("Hello World\n");
    return 0;
}
```

```
gcc -o helloworld HelloWorld.c
./helloworld
Hello World
```

如果你对以上细节了然于胸，那么很遗憾，本课程不是为你准备的。

如果你对以上问题不太了解，那么恭喜你，本课程就是为你准备的。

很多时候我们使用IDE环境比如Visual Studio进行软件开发，这种情况下一般都是编译和链接合并成一步完成，通常我们称其为构建Build。但是无论是集成编译环境还是命令行环境，当遇到莫名其妙的编译或链接错误时还是要求我们对底层的机制有所了解。

事实上只要我们能够深刻理解x86平台下程序运行的背后机理，当你某一天要在MIPS或者PowerPC指令集的嵌入式平台上做开发，很快就能找到相通之处。

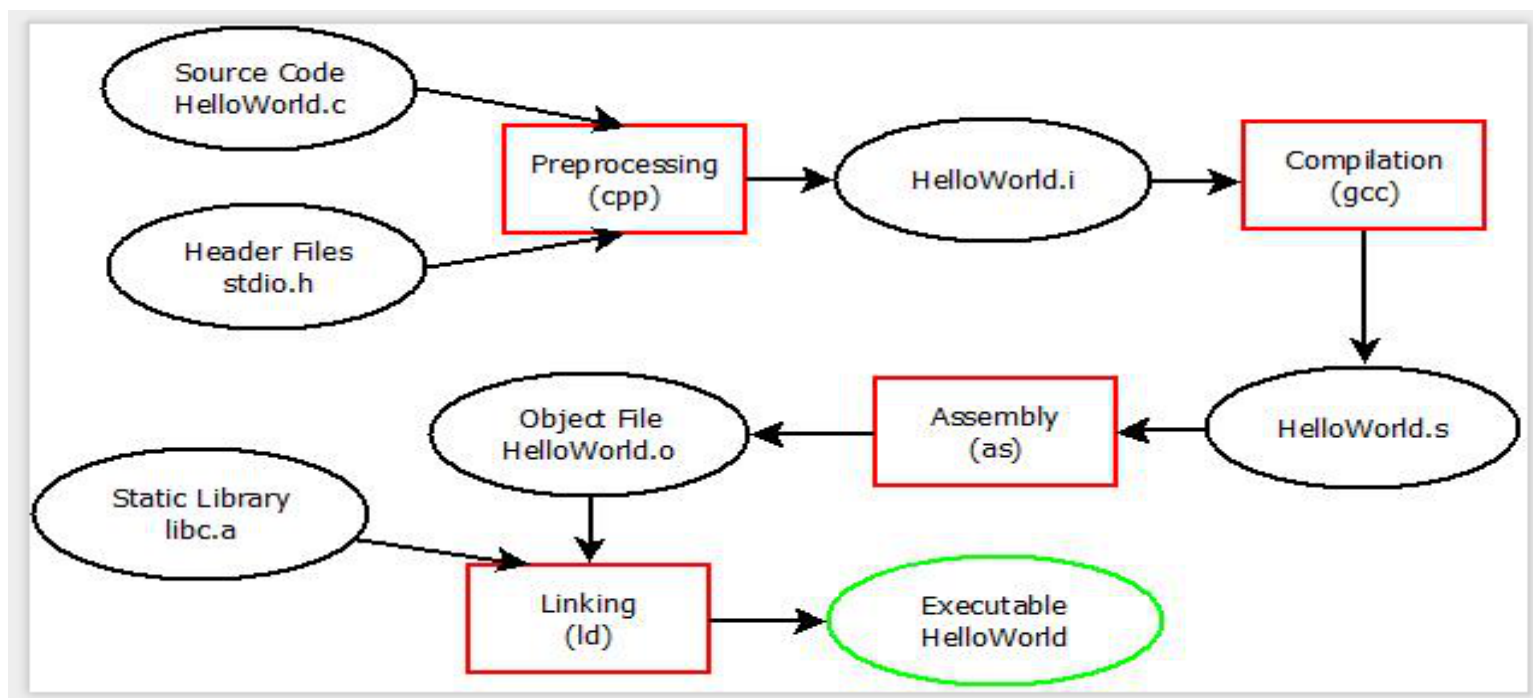
# 第1部分

-Linux编译过程总体概览

# GCC编译过程总体剖析

\*首先让我们一起来看一下GCC的手册\*

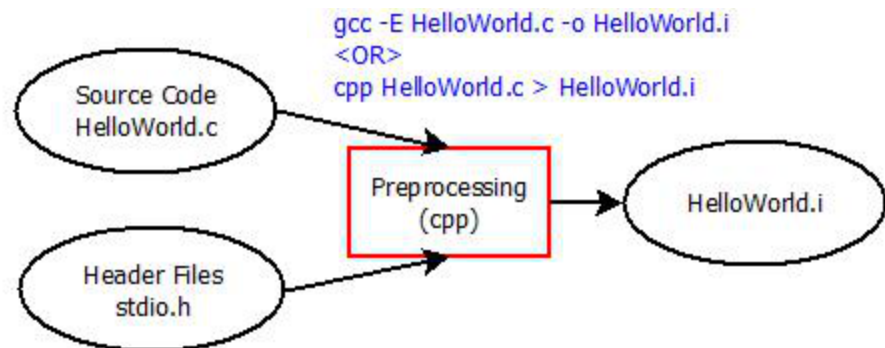
- 事实上GCC整个编译过程可以分为这几个步骤： 预处理，编译，汇编和链接



# GCC编译过程之-预编译

- 预编译过程主要是处理源代码文件中那些以#开头的预编译指令，具体来说

- 处理#define, 将所有的宏定义#define展开
- 处理#if, #else, #endif等等条件编译指令
- 处理#include, 原地插入文件  
(有时候你可以看到很有意思的头文件包含)
- 删除注释



- 可以使用以下命令来调用预编译器

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

或者

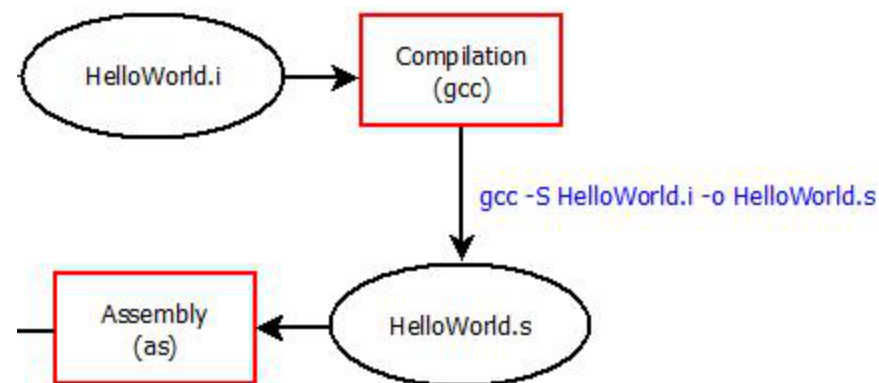
`cpp HelloWorld.c > HelloWorld.i`

- 如果你觉得头文件的包含或者宏定义有问题时可以通过查看预编译结果确认。

# GCC编译过程之-编译

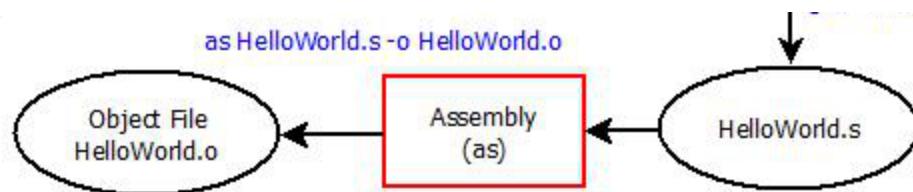
- 编译过程是最核心也是最复杂的过程， 复杂性从GCC的编译选项中就可以略知一二， 它主要是对预编译完的文件进行词法， 语法， 语义分析并进行优化， 最终产生对应的汇编代码文件。

- 可以使用以下命令来调用编译器  
`gcc -S HelloWorld.i -o HelloWorld.s`



# GCC编译过程之-汇编

- 汇编器把汇编代码转换成目标文件，比较单纯。



- 可以使用以下命令来调用汇编器  
as HelloWorld.s -o HelloWorld.o



# GCC编译过程之-链接

- 链接器最终将多个目标文件链接起来形成可执行文件

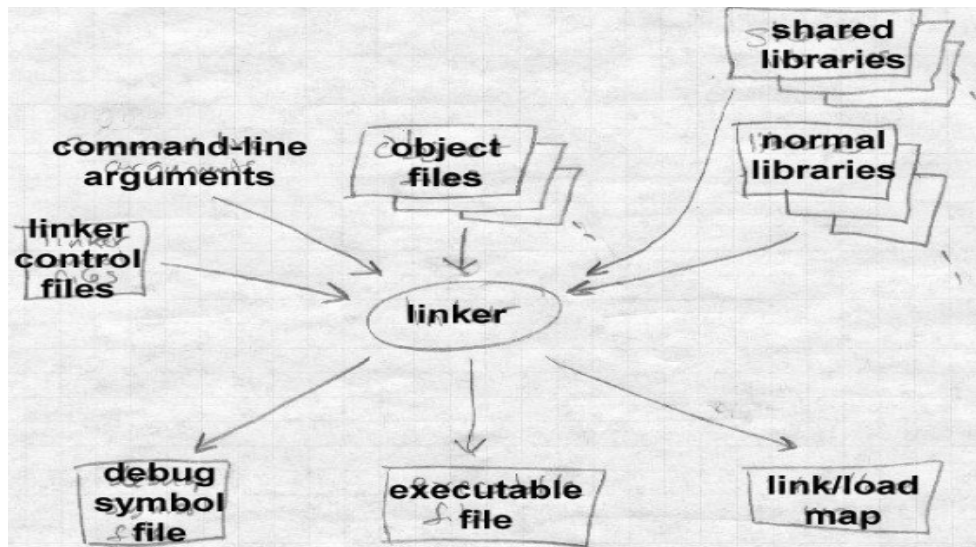
- 为什么需要链接?

简单来说因为每个源代码模块都独立地进行编译，链接就是要处理各个模块之间的相互引用，使他们之间能够互相衔接。

比如说我们的HelloWorld.c中的main函数并不知道printf这个函数的地址，链接器在链接的时候会根据引用到的符号printf，自动去相应的模块查找printf的地址，然后将HelloWorld.c模块中引用到printf的指令进行重新修正，让它的目标地址成为真正的printf函数的地址，这个就是链接的基本过程。

链接分为静态链接和动态链接，后续的章节会进行专门的探讨。

不过让我们首先来看看静态链接的过程



# 第2部分

-Linux目标文件

# Linux可执行文件(目标文件)的格式

- Linux下的**可执行文件**的格式称为ELF格式(Executable Linkage Format)  
`helloworld: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs)`
- Linux下的**目标文件**只是没有经过链接的过程，它本身的格式就是按照可执行文件的格式存储的，只是结构稍有不同。  
`HelloWorld.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped`
- 另外动态链接库.so和核心转储文件core dump也都是以ELF格式存储的  
`ld-2.12.so: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, not stripped`  
`core.14089: ELF 64-bit LSB core file x86-64, version 1 (SYSV), SVR4-style, from './corecump'`

建议大家首先使用**man elf**命令看下它的手册页

- ELF文件的内容是按照所谓的‘段’的形式(section)进行存储的,看看有哪些段

接下来我们通过具体的例子来看看ELF文件里面究竟是什么样子的

# Linux目标文件究竟是怎么样的(1)

首先介绍一个查看目标文件的工具**objdump**, 它可以辅助我们查看目标文件的内容

可以用命令**objdump -h xxxx.o**查看各个段的分布


```
#include <stdio.h>

int g_init_var1 = 1;
int g_uninit_var2;

void foo(int i)
{
    printf("%d", i);
}

int main(void)
{
    static int var3 = 2;
    static int var4;

    int x = 3;
    foo(x);
    return 0;
}
```



Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	00000041	0000000000000000	0000000000000000	00000040	2**2
	CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE					
1	.data	00000008	0000000000000000	0000000000000000	00000084	2**2
	CONTENTS, ALLOC, LOAD, DATA					
2	.bss	00000004	0000000000000000	0000000000000000	0000008c	2**2
	ALLOC					
3	.rodata	00000003	0000000000000000	0000000000000000	0000008c	2**0
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
4	.comment	00000012	0000000000000000	0000000000000000	0000008f	2**0
	CONTENTS, READONLY					
5	.note.GNU-stack	00000000	0000000000000000	0000000000000000	000000a1	2**0
	CONTENTS, READONLY					
6	.eh_frame	00000058	0000000000000000	0000000000000000	000000a8	2**3
	CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA					

可以用命令**objdump -s -d xxxx.o**挖掘各个段的内容

接下来我们来看看几个重要的段: **.text**段, **.data**段, **.rodata**段, **.bss**段, **.comment**段

# Linux目标文件究竟是怎么样的(2)

## 代码段.text

可以看到整个代码段的大小正好是0x41个字节

Contents of section .text:

```
0000 554889e5 4883ec10 897dfc8b 45fc89c6  .UH..H....}..E...
0010 bf000000 00b80000 0000e800 000000c9  .....
0020 c3554889 e54883ec 10c745fc 03000000  .UH..H....E.....
0030 8b45fc89 c7e80000 0000b800 000000c9  .E.....
0040 c3
```

0000000000000000 <foo>:

```
0: 55          push    %rbp
1: 48 89 e5    mov     %rsp,%rbp
4: 48 83 ec 10 sub     $0x10,%rsp
8: 89 7d fc    mov     %edi,-0x4(%rbp)
b: 8b 45 fc    mov     -0x4(%rbp),%eax
e: 89 c6      mov     %eax,%esi
10: bf 00 00 00 mov     $0x0,%edi
15: b8 00 00 00 mov     $0x0,%eax
1a: e8 00 00 00 callq   1f <foo+0x1f>
1f: c9         leaveq  %eax
20: c3         retq
```

0000000000000021 <main>:

```
21: 55          push    %rbp
22: 48 89 e5    mov     %rsp,%rbp
25: 48 83 ec 10 sub     $0x10,%rsp
29: c7 45 fc 03 00 00 00 movl    $0x3,-0x4(%rbp)
30: 8b 45 fc    mov     -0x4(%rbp),%eax
33: 89 c7      mov     %eax,%edi
35: e8 00 00 00 callq   3a <main+0x19>
3a: b8 00 00 00 mov     $0x0,%eax
3f: c9         leaveq  %eax
40: c3         retq
```

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	00000041	0000000000000000	0000000000000000	00000040	2**2
			CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE			
1	.data	00000008	0000000000000000	0000000000000000	00000084	2**2
			CONTENTS, ALLOC, LOAD, DATA			
2	.bss	00000004	0000000000000000	0000000000000000	0000008c	2**2
			ALLOC			
3	.rodata	00000003	0000000000000000	0000000000000000	0000008c	2**0
			CONTENTS, ALLOC, LOAD, READONLY, DATA			
4	.comment	00000012	0000000000000000	0000000000000000	0000008f	2**0
			CONTENTS, READONLY			
5	.note.GNU-stack	00000000	0000000000000000	0000000000000000	000000a1	2**0
			CONTENTS, READONLY			
6	.eh_frame	00000058	0000000000000000	0000000000000000	000000a8	2**3
			CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA			

# Linux目标文件究竟是怎么样的(3)

## 数据段.data和只读数据段.rodata

.data段保存初始化的全局变量和局部静态变量，  
在我们的例子中就是g\_init\_var1和var3，一共8个字节

.rodata段保存只读数据，一般是只读变量和字符串  
常量，在我们的例子中就是%d  
`#include <stdio.h>`

```
int g_init_var1 = 1;
int g_uninit_var2;

void foo(int i)
{
    printf("%d", i);
}

int main(void)
{
    static int var3 = 2;
    static int var4;

    int x = 3;
    foo(x);
    return 0;
}
```

Sections:						
Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	00000041	0000000000000000	0000000000000000	00000040	2**2
	CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE					
1	.data	00000008	0000000000000000	0000000000000000	00000084	2**2
	CONTENTS, ALLOC, LOAD, DATA					
2	.bss	00000004	0000000000000000	0000000000000000	0000008c	2**2
	ALLOC					
3	.rodata	00000003	0000000000000000	0000000000000000	0000008c	2**0
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
4	.comment	00000012	0000000000000000	0000000000000000	0000008f	2**0
	CONTENTS, READONLY					
5	.note.GNU-stack	00000000	0000000000000000	0000000000000000	000000a1	2**0
	CONTENTS, READONLY					
6	.eh_frame	00000058	0000000000000000	0000000000000000	000000a8	2**3
	CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA					

```
Contents of section .data:
0000 01000000 02000000      .....
Contents of section .rodata:
0000 256400          %d.
```

可以看到.data的前面的四个字节是从低到高分别是01,00,00,00，这正是我们的g\_init\_var1的值(按小端序)。

# Linux目标文件究竟是怎么样的(4)

## BSS段

.bss段通常保存未初始化的全局变量和局部静态变量

照理说我们的例子里面应该有两个变量g\_uninit\_va2

和var4都属于这个段，大小应该是8，但是我们明明

看到大小是4个字节？

这里全局变量g\_uninit\_va2并没有放到bss段里面，只是

一个Common符号(后面在讲到符号表的时候我们会

看到)。实际上这和不同的编译器实现有关。

BSS = "Block Started by Symbol"

- Dennis Ritchie says

From <http://www.faqs.org/faqs/unix-faq/faq/part1/section-3.html>

```
#include <stdio.h>
```

```
int g_init_var1 = 1;
```

```
int g_uninit_var2;
```

```
void foo(int i)
```

```
{
    printf("%d", i);
}
```

```
int main(void)
```

```
{
    static int var3 = 2;
    static int var4;
```

```
    int x = 3;
    foo(x);
    return 0;
}
```

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	00000041	0000000000000000	0000000000000000	00000040	2**2
			CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE			
1	.data	00000008	0000000000000000	0000000000000000	00000084	2**2
			CONTENTS, ALLOC, LOAD, DATA			
2	.bss	00000004	0000000000000000	0000000000000000	0000008c	2**2
			ALLOC			
3	.rodata	00000003	0000000000000000	0000000000000000	0000008c	2**0
			CONTENTS, ALLOC, LOAD, READONLY, DATA			
4	.comment	00000012	0000000000000000	0000000000000000	0000008f	2**0
			CONTENTS, READONLY			
5	.note.GNU-stack	00000000	0000000000000000	0000000000000000	000000a1	2**0
			CONTENTS, READONLY			
6	.eh_frame	00000058	0000000000000000	0000000000000000	000000a8	2**3
			CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA			

## Comment段

.comment段比较明显它就是用来存放编译器的版本信息

Contents of section .comment:

```
0000 00474343 3a202847 4e552920 342e382e .GCC: (GNU) 4.8.
0010 3200
```

# Linux目标文件究竟是怎么样的

## -ELF头

接下来我们进一步探究一下ELF文件的结构

介绍一个查看目标文件的工具`readelf`它可以辅助我们查看ELF文件的内容

可以用命令`readelf -h xxxx`查看ELF文件头

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                   ELF64
  Data:                     2's complement, little endian
  Version:                  1 (current)
  OS/ABI:                   UNIX - System V
  ABI Version:              0
  Type:                     REL (Relocatable file)
  Machine:                  Advanced Micro Devices X86-64
  Version:                  0x1
  Entry point address:      0x0
  Start of program headers: 0 (bytes into file)
  Start of section headers: 360 (bytes into file)
  Flags:                    0x0
  Size of this header:      64 (bytes)
  Size of program headers:  0 (bytes)
  Number of program headers: 0
  Size of section headers:  64 (bytes)
  Number of section headers: 13
  Section header string table index: 10
```

其实左侧显示的内容可以直接对应到相应的/  
`usr/include/elf.h`头文件中的结构体`Elf64_Ehdr`

```
typedef struct
{
    unsigned char e_ident[EI_NIDENT]; /* Magic number and other info */
    Elf64_Half e_type; /* Object file type */
    Elf64_Half e_machine; /* Architecture */
    Elf64_Word e_version; /* Object file version */
    Elf64_Addr e_entry; /* Entry point virtual address */
    Elf64_Off e_phoff; /* Program header table file offset */
    Elf64_Off e_shoff; /* Section header table file offset */
    Elf64_Word e_flags; /* Processor-specific flags */
    Elf64_Half e_ehsize; /* ELF header size in bytes */
    Elf64_Half e_phsize; /* Program header table entry size */
    Elf64_Half e_phnum; /* Program header table entry count */
    Elf64_Half e_shsize; /* Section header table entry size */
    Elf64_Half e_shnum; /* Section header table entry count */
    Elf64_Half e_shstrndx; /* Section header string table index */
} Elf64_Ehdr;
```



# Linux目标文件究竟是怎么样的

## -ELF段表

可以用命令`readelf -S xxxx`查看ELF段表

注意：之前我们使用`objdump -h`命令的时候其实只是显示了关键的段而省略了一些辅助性的段

Section Headers:						
[Nr]	Name	Type	Address	Offset		
	Size	EntSize	Flags Link Info Align			
[ 0]		NULL	0000000000000000	00000000		
	0000000000000000	0000000000000000	0 0 0			
[ 1]	.text	PROGBITS	0000000000000000	00000040		
	0000000000000041	0000000000000000	AX 0 0 4			
[ 2]	.rela.text	RELA	0000000000000000	00000678		
	0000000000000048	0000000000000018	11 1 8			
[ 3]	.data	PROGBITS	0000000000000000	00000084		
	0000000000000008	0000000000000000	WA 0 0 4			
[ 4]	.bss	NOBITS	0000000000000000	0000008c		
	0000000000000004	0000000000000000	WA 0 0 4			
[ 5]	.rodata	PROGBITS	0000000000000000	0000008c		
	0000000000000003	0000000000000000	A 0 0 1			
[ 6]	.comment	PROGBITS	0000000000000000	0000008f		
	0000000000000012	0000000000000001	MS 0 0 1			
[ 7]	.note.GNU-stack	PROGBITS	0000000000000000	000000a1		
	0000000000000000	0000000000000000	0 0 1			
[ 8]	.eh_frame	PROGBITS	0000000000000000	000000a8		
	0000000000000058	0000000000000000	A 0 0 8			
[ 9]	.rela.eh_frame	RELA	0000000000000000	000006c0		
	0000000000000030	0000000000000018	11 8 8			
[10]	.shstrtab	STRTAB	0000000000000000	00000100		
	0000000000000061	0000000000000000	0 0 1			
[11]	.symtab	SYMTAB	0000000000000000	000004a8		
	0000000000000180	0000000000000018	12 11 8			
[12]	.strtab	STRTAB	0000000000000000	00000628		
	000000000000004c	0000000000000000	0 0 1			

其实左侧显示的内容可以直接对应到相应的/  
*usr/include/elf.h*头文件中的结构体`Elf64_Shdr`  
段表其实就是一个 `Elf64_Shdr`结构的数组

```
typedef struct
{
    Elf64_Word  sh_name;           /* Section name (string tbl index) */
    Elf64_Word  sh_type;           /* Section type */
    Elf64_Xword sh_flags;          /* Section flags */
    Elf64_Addr  sh_addr;           /* Section virtual addr at execution */
    Elf64_Off   sh_offset;         /* Section file offset */
    Elf64_Xword sh_size;           /* Section size in bytes */
    Elf64_Word  sh_link;           /* Link to another section */
    Elf64_Word  sh_info;           /* Additional section information */
    Elf64_Xword sh_addralign;      /* Section alignment */
    Elf64_Xword sh_entsize;        /* Entry size if section holds table */
} Elf64_Shdr;
```

# Linux目标文件究竟是怎么样的

## -字符串表

可以用命令`readelf -p section_name xxxx`查看字符串表

注意：ELF文件里面都用字符串表的偏移来索引字符串

```
String dump of section '.shstrtab':  
[    1] .symtab  
[    9] .strtab  
[   11] .shstrtab  
[   1b] .rela.text  
[   26] .data  
[   2c] .bss  
[   31] .rodata  
[   39] .comment  
[   42] .note.GNU-stack  
[   52] .rela.eh_frame
```

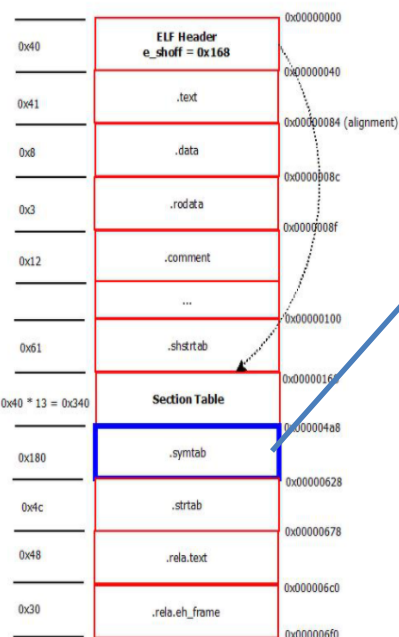
```
String dump of section '.strtab':  
[    1] c_code_obj.c  
[    e] var4.2184  
[   18] var3.2183  
[   22] g_init_var1  
[   2e] g_uninit_var2  
[   3c] foo  
[   40] printf  
[   47] main
```

# Linux目标文件究竟是怎么样的

## -符号表

可以用命令`readelf -s xxxx`查看符号表

ELF File Main Structure



Symbol table '.symtab' contains 16 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	c_code_obj.c
2:	0000000000000000	0	SECTION	LOCAL	DEFAULT	1	
3:	0000000000000000	0	SECTION	LOCAL	DEFAULT	3	
4:	0000000000000000	0	SECTION	LOCAL	DEFAULT	4	
5:	0000000000000000	0	SECTION	LOCAL	DEFAULT	5	
6:	0000000000000000	4	OBJECT	LOCAL	DEFAULT	4	var4.2184
7:	0000000000000004	4	OBJECT	LOCAL	DEFAULT	3	var3.2183
8:	0000000000000000	0	SECTION	LOCAL	DEFAULT	7	
9:	0000000000000000	0	SECTION	LOCAL	DEFAULT	8	
10:	0000000000000000	0	SECTION	LOCAL	DEFAULT	6	
11:	0000000000000000	4	OBJECT	GLOBAL	DEFAULT	3	g_init_var1
12:	0000000000000004	4	OBJECT	GLOBAL	DEFAULT	COM	g_uninit_var2
13:	0000000000000000	33	FUNC	GLOBAL	DEFAULT	1	foo
14:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	printf
15:	0000000000000021	32	FUNC	GLOBAL	DEFAULT	1	main

Ndx表示符号所在的段，如果符号定义在本目标文件中，那么指示该符号所在段在段表中的下标。UND表示该符号未定义。

其实上面显示的内容可以直接对应到相应的/usr/include/elf.h头文件中的

```
typedef struct
{
    Elf64_Word    st_name;           /* Symbol name (string tbl index) */
    unsigned char st_info;           /* Symbol type and binding */
    unsigned char st_other;          /* Symbol visibility */
    Elf64_Section st_shndx;          /* Section index */
    Elf64_Addr    st_value;          /* Symbol value */
    Elf64_Xword   st_size;           /* Symbol size */
} Elf64_Sym;
```

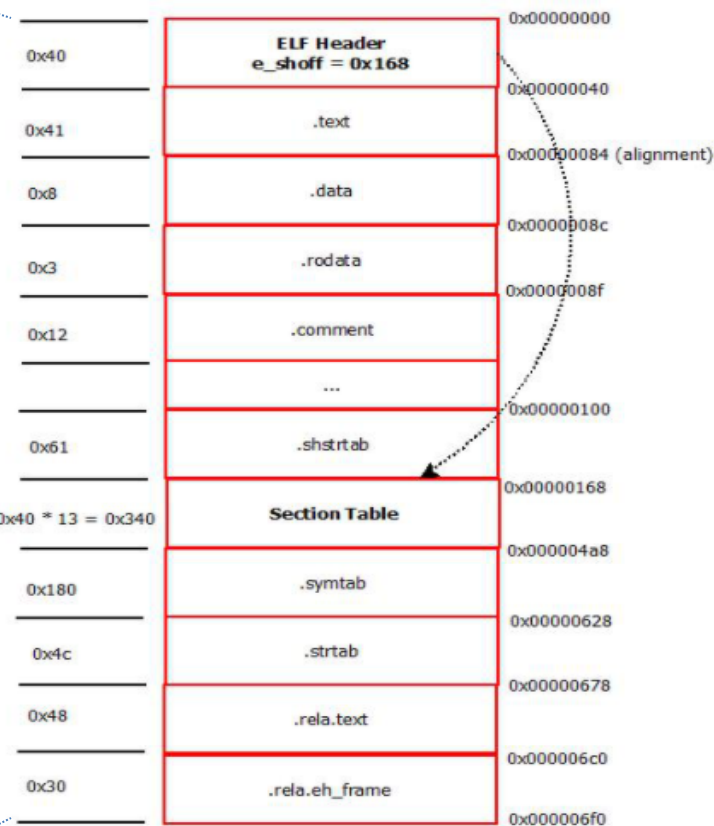
# Linux目标文件究竟是怎么样的

## -推导出ELF文件总体结构图

```
ELF Header:
Magic:  7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Class:                                ELF64
Data:      2's complement, little endian
Version:    1 (current)
OS/ABI:     UNIX - System V
ABI Version: 0
Type:       REL (Relocatable file)
Machine:    Advanced Micro Devices X86-64
Version:    0x1
Entry point address: 0x0
Start of program headers: 0 (bytes into file)
Start of section headers: 360 (bytes into file)
Flags:      0x0
Size of this header: 64 (bytes)
Size of program headers: 0 (bytes)
Number of program headers: 0
Size of section headers: 64 (bytes)
Number of section headers: 13
Section header string table index: 10
```

```
Section Headers:
[Nr] Name      Type           Address             Offset
     Size      EntSize          Flags Link Info Align
[ 0]           NULL                0000000000000000   00000000
     0000000000000000 0000000000000000      0 0 0
[ 1] .text       PROGBITS        0000000000000000   00000040
     0000000000000041 0000000000000000  AX  0 0 4
[ 2] .rela.text  RELA            0000000000000000   00000678
     0000000000000048 0000000000000018      11 1 8
[ 3] .data       PROGBITS        0000000000000000   00000084
     0000000000000008 0000000000000000  WA  0 0 4
[ 4] .bss        NOBITS          0000000000000000   0000008c
     0000000000000004 0000000000000000  WA  0 0 4
[ 5] .rodata     PROGBITS        0000000000000000   0000008c
     0000000000000003 0000000000000000  A  0 0 1
[ 6] .comment    PROGBITS        0000000000000000   0000008f
     0000000000000012 0000000000000001  MS  0 0 1
[ 7] .note.GNU-stack PROGBITS        0000000000000000   000000a1
     0000000000000000 0000000000000000      0 0 1
[ 8] .eh_frame    PROGBITS        0000000000000000   000000a8
     0000000000000058 0000000000000000  A  0 0 8
[ 9] .rela.eh_frame RELA            0000000000000000   000006c0
     0000000000000030 0000000000000018      11 8 8
[10] .shstrtab    STRTAB          0000000000000000   00000100
     0000000000000061 0000000000000000      0 0 1
[11] .symtab      SYMTAB          0000000000000000   000004a8
     00000000000000180 0000000000000018      12 11 8
[12] .strtab      STRTAB          0000000000000000   00000628
     000000000000004c 0000000000000000      0 0 1
```

ELF File Main Structure



sizeof(Elf64\_Shdr) = 0x40

this is exactly ELF file size

实际上可以ELF Header为线索推导出上边的图，唯一需要注意的是有些段会有字节对齐的要求

# 第3部分

-Linux静态链接

# Linux静态链接

## -空间和地址分配

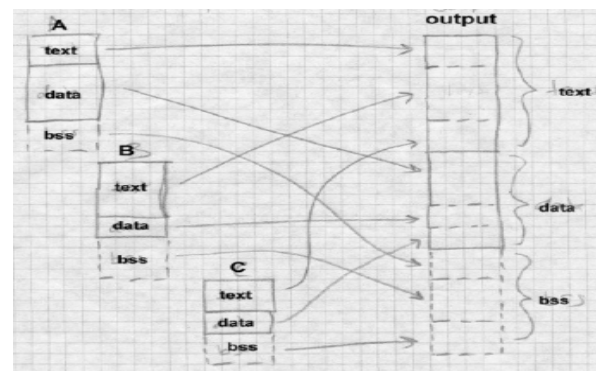
- 通过前面的学习相信大家对ELF目标文件有了一定的了解，那么接下来的问题是如果我们有两个目标文件，他们是怎么形成可执行文件的呢？这就需要我们的链接器ld的帮助了,先来看看静态链接的过程。

```
main.o: file format elf64-x86-64

Sections:
Idx Name          Size      VMA           LMA           File off  Algn
 0 .text          0000002c  0000000000000000  0000000000000000  00000040  2**2
CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
 1 .data          00000000  0000000000000000  0000000000000000  0000006c  2**2
CONTENTS, ALLOC, LOAD, DATA
 2 .bss           00000000  0000000000000000  0000000000000000  0000006c  2**2
ALLOC
 3 .comment        00000012  0000000000000000  0000000000000000  0000006c  2**0
CONTENTS, READONLY
 4 .note.GNU-stack 00000000  0000000000000000  0000000000000000  0000007e  2**0
CONTENTS, READONLY
 5 .eh_frame       00000038  0000000000000000  0000000000000000  00000080  2**3
CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
```

```
swap.o: file format elf64-x86-64

Sections:
Idx Name          Size      VMA           LMA           File off  Algn
 0 .text          0000002c  0000000000000000  0000000000000000  00000040  2**2
CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
 1 .data          00000004  0000000000000000  0000000000000000  0000006c  2**2
CONTENTS, ALLOC, LOAD, DATA
 2 .bss           00000000  0000000000000000  0000000000000000  00000070  2**2
ALLOC
 3 .comment        00000012  0000000000000000  0000000000000000  00000070  2**0
CONTENTS, READONLY
 4 .note.GNU-stack 00000000  0000000000000000  0000000000000000  00000082  2**0
CONTENTS, READONLY
 5 .eh_frame       00000038  0000000000000000  0000000000000000  00000088  2**3
CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
```



ld main.o swap.o -e main -o stlink

```
stlink: file format elf64-x86-64

Sections:
Idx Name          Size      VMA           LMA           File off  Algn
 0 .text          00000058  00000000004000e8  00000000004000e8  000000e8  2**2
CONTENTS, ALLOC, LOAD, READONLY, CODE
 1 .eh_frame       00000058  0000000000400140  0000000000400140  00000140  2**3
CONTENTS, ALLOC, LOAD, READONLY, DATA
 2 .data          00000004  0000000000600198  0000000000600198  00000198  2**2
CONTENTS, ALLOC, LOAD, DATA
 3 .comment        00000011  0000000000000000  0000000000000000  0000019c  2**0
CONTENTS, READONLY
```

# Linux静态链接

## -链接时重定位

- 重定位前

```
main.o:      file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
  0:  55                      push   %rbp
  1:  48 89 e5                mov     %rsp,%rbp
  4:  48 83 ec 10             sub     $0x10,%rsp
  8:  c7 45 fc 64 00 00 00    movl    $0x64,-0x4(%rbp)
  f:  48 8d 45 fc             lea     -0x4(%rbp),%rax
 13:  48 89 c6                mov     %rax,%rsi
 16:  b5 00 00 00 00          mov     $0x0,%edi
 1b:  b8 00 00 00 00          mov     $0x0,%eax
 20:  e8 00 00 00 00          callq   25 <main+0x25>
 25:  b8 00 00 00 00          mov     $0x0,%eax
 2a:  c9                      leaveq  %rax
 2b:  c3                      retq
```

### 重定位后

```
stlink:      file format elf64-x86-64

Disassembly of section .text:

00000000004000e8 <main>:
 4000e8:  55                      push   %rbp
 4000e9:  48 89 e5                mov     %rsp,%rbp
 4000ec:  48 83 ec 10             sub     $0x10,%rsp
 4000f0:  c7 45 fc 64 00 00 00    movl    $0x64,-0x4(%rbp),%rax
 4000f7:  48 8d 45 fc             lea     -0x4(%rbp),%rax
 4000fb:  48 89 c6                mov     %rax,%rsi
 4000fe:  bf 98 01 60 00          mov     $0x600198,%edi
 400103:  b8 00 00 00 00          mov     $0x0,%eax
 400108:  e8 07 00 00 00          callq   400114 <swap>
 40010d:  b8 00 00 00 00          mov     $0x0,%eax
 400112:  c9                      leaveq  %rax
 400113:  c3                      retq

0000000000400114 <swap>:
 400114:  55                      push   %rbp
```

绝对寻址

相对寻址

链接器怎么知道那些符号要重定位呢？

重定位表。

objdump -r main.o

```
main.o:      file format elf64-x86-64

RELOCATION RECORDS FOR [.text]:
OFFSET          TYPE           VALUE
0000000000000017 R_X86_64_32      x
0000000000000021 R_X86_64_PC32    swap-0x0000000000000004
```

## 重新解读编译过程(实例)

```
[zhoulei@LocalCentOS verbose_compile]$ gcc -o hello --verbose hello.c
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/local/libexec/gcc/x86_64-unknown-linux-gnu/4.8.2/lto-wrapper
Target: x86_64-unknown-linux-gnu
Configured with: ../gcc-4.8.2/configure --enable-checking=release --enable-languages=c,c++ --disable-multilib
Thread model: posix
gcc version 4.8.2 (GCC)
COLLECT_GCC_OPTIONS='-O' '-hello' '-v' '-mtune=generic' '-march=x86-64'
/usr/local/libexec/gcc/x86_64-unknown-linux-gnu/4.8.2/cc1 quiet -v hello.c -quiet -dumpbase hello.c -mtune=generic -march=x86-64 -auxbase hello -version -o /tmp/ccEMH0mW.s
GNU C (GCC) version 4.8.2 (x86_64-unknown-linux-gnu)
        compiled by GNU C version 4.8.2, GMP version 4.3.2, MPFR version 2.4.2, MPC version 0.8.1
GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
ignoring nonexistent directory "/usr/local/lib/gcc/x86_64-unknown-linux-gnu/4.8.2/../.././lib64:/usr/local/lib/gcc/x86_64-unknown-linux-gnu/4.8.2/include"
#include "... " search starts here:
#include <...> search starts here:
/usr/local/lib/gcc/x86_64-unknown-linux-gnu/4.8.2/include
/usr/local/include
/usr/local/lib/gcc/x86_64-unknown-linux-gnu/4.8.2/include-fixed
/usr/include
End of search list.
GNU C (GCC) version 4.8.2 (x86_64-unknown-linux-gnu)
        compiled by GNU C version 4.8.2, GMP version 4.3.2, MPFR version 2.4.2, MPC version 0.8.1
GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
Compiler executable checksum: 73df20b972d2fc231a974768fc25869a
COLLECT_GCC_OPTIONS='-O' '-hello' '-v' '-mtune=generic' '-march=x86-64'
as -v --64 -o /tmp/cc8hVDKu.o /tmp/ccEMH0mW.s
GNU assembler version 2.20.51.0.2 (x86_64-redhat-linux) using BFD version 2.20.51.0.2-5.42.el6_20100205
COMPILER_PATH=/usr/local/libexec/gcc/x86_64-unknown-linux-gnu/4.8.2:/usr/local/libexec/gcc/x86_64-unknown-linux-gnu/4.8.2:/usr/local/libexec/gcc/x86_64-unknown-linux-gnu/4.8.2:/usr/local/lib/gcc/x86_64-unknown-linux-gnu/4.8.2:/usr/local/lib/gcc/x86_64-unknown-linux-gnu/4.8.2/../.././lib64:/usr/lib/..
LIBRARY_PATH=/usr/local/lib/gcc/x86_64-unknown-linux-gnu/4.8.2:/usr/local/lib/gcc/x86_64-unknown-linux-gnu/4.8.2/../.././lib64:/usr/lib/..
COLLECT_GCC_OPTIONS='-O' '-hello' '-v' '-mtune=generic' '-march=x86-64'
/usr/local/libexec/gcc/x86_64-unknown-linux-gnu/4.8.2/collect2 --eh-frame-hdr -m elf_x86_64 -dynamic-linker /lib64/ld-linux-x86-64.so.2 -o hello /usr/lib/..
lib64/crti.o /usr/lib/.. lib64/crti.o /usr/local/lib/gcc/x86_64-unknown-linux-gnu/4.8.2/crtbegin.o -L/usr/local/lib/gcc/x86_64-unknown-linux-gnu/4.8.2 -L/usr
r/local/lib/gcc/x86_64-unknown-linux-gnu/4.8.2/../.././lib64 -L/lib/.. lib64 -L/usr/lib/.. lib64 -L/usr/local/lib/gcc/x86_64-unknown-linux-gnu/4.8.2/../.
.././tmp/cc8hVDKu.o -lgcc--as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed /usr/local/lib/gcc/x86_64-unknown-linux-gnu/4.8.2/
crtend.o /usr/lib/.. lib64/crtn.o
```



# 第4部分

## -可执行文件的装载

# 进程和装载的基本介绍

程序(可执行文件)和进程的区别 (静态 vs 动态)

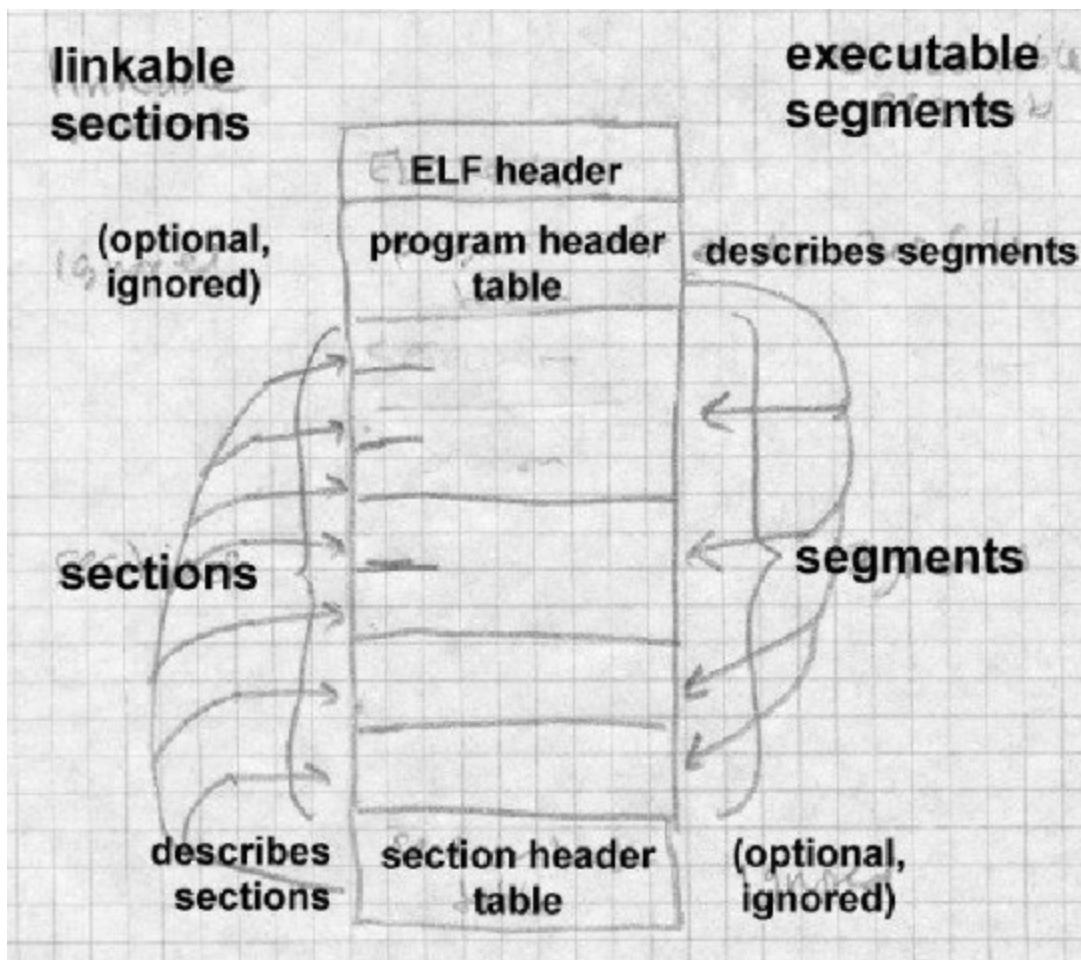
**现代操作系统如何装载可执行文件**

- 给进程分配独立的虚拟地址空间
- 建立虚拟地址空间和可执行文件的映射关系
- 把CPU指令寄存器设置成可执行文件的入口地址，启动执行

可执行文件在装载的过程中实际上如我们所说的那样是映射的虚拟地址空间，所以可执行文件通常被叫做映像文件(或者Image文件).

# ELF可执行文件介绍

## ELF文件的两种视角



ELF格式具有不寻常的双重特性，编译器、汇编器和链接器将这个文件看作是被区段（section）头部表描述的一系列逻辑区段的集合，而系统加载器将文件看成是由程序头部表描述的一系列段（segment）的集合。一个段（segment）通常会由多个区段（section）组成。例如，一个“可加载只读”段可以由可执行代码区段、只读数据区段和动态链接器需要的符号组成。

区段（section）是从链接器的视角来看ELF文件，而段（segment）是从执行的视角来看ELF文件，也就是它会被映射到内存中。

我们用命令`readelf -a xxx`再来看一下可执行文件的样子(段的映射关系)

# ELF可执行文件在Linux下的装载过程

首先我们来探究一下ELF可执行文件的结构

可以用命令`readelf -l xxxx`查看ELF可执行文件的程序头

(可执行文件中程序头保存 segment 的信息)

```
Elf file type is EXEC (Executable file)
Entry point 0x4003c0
There are 8 program headers, starting at offset 64

Program Headers:
Type           Offset             VirtAddr           PhysAddr
               FileSiz             MemSiz             Flags   Align
PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
               0x00000000000001c0 0x00000000000001c0 R E     8
INTERP         0x0000000000000200 0x0000000000400200 0x0000000000400200
               0x000000000000001c 0x000000000000001c R       1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
               0x00000000000006c4 0x00000000000006c4 R E     200000
LOAD           0x00000000000006c8 0x00000000000006c8 0x00000000000006c8
               0x00000000000001f8 0x0000000000000208 RW     200000
DYNAMIC         0x00000000000006f0 0x00000000000006f0 0x00000000000006f0
               0x0000000000000190 0x0000000000000190 RW      8
NOTE           0x000000000000021c 0x000000000040021c 0x000000000040021c
               0x0000000000000020 0x0000000000000020 R       4
GNU_EH_FRAME   0x0000000000000624 0x0000000000400624 0x0000000000400624
               0x0000000000000024 0x0000000000000024 R       4
GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
               0x0000000000000000 0x0000000000000000 RW      8
```

其实左侧显示的内容可以直接对应到相应的/  
*usr/include/elf.h*头文件中的结构体Elf64\_Phdr

```
typedef struct
{
    Elf64_Word    p_type;           /* Segment type */
    Elf64_Word    p_flags;         /* Segment flags */
    Elf64_Off     p_offset;        /* Segment file offset */
    Elf64_Addr     p_vaddr;        /* Segment virtual address */
    Elf64_Addr     p_paddr;        /* Segment physical address */
    Elf64_Xword    p_filesz;       /* Segment size in file */
    Elf64_Xword    p_memsz;       /* Segment size in memory */
    Elf64_Xword    p_align;       /* Segment alignment */
} Elf64_Phdr;
```

# ELF可执行文件在Linux下的装载过程

## ELF可执行文件和进程虚拟地址空间的映射关系

### 一个实例（以静态链接为例）

`/proc/<PID>/maps`

查看进程的虚拟地址空间是如何使用的。

该文件有6列，分别为：

地址：虚拟内存区域的起始和终止地址

权限：虚拟内存的权限，r=读,w=写,x=执行,s=共享,p=私有

偏移量：虚拟内存区域在被映射文件中的偏移量

设备：映像文件的主设备号和次设备号；

节点：映像文件的节点号；

路径：映像文件的路径

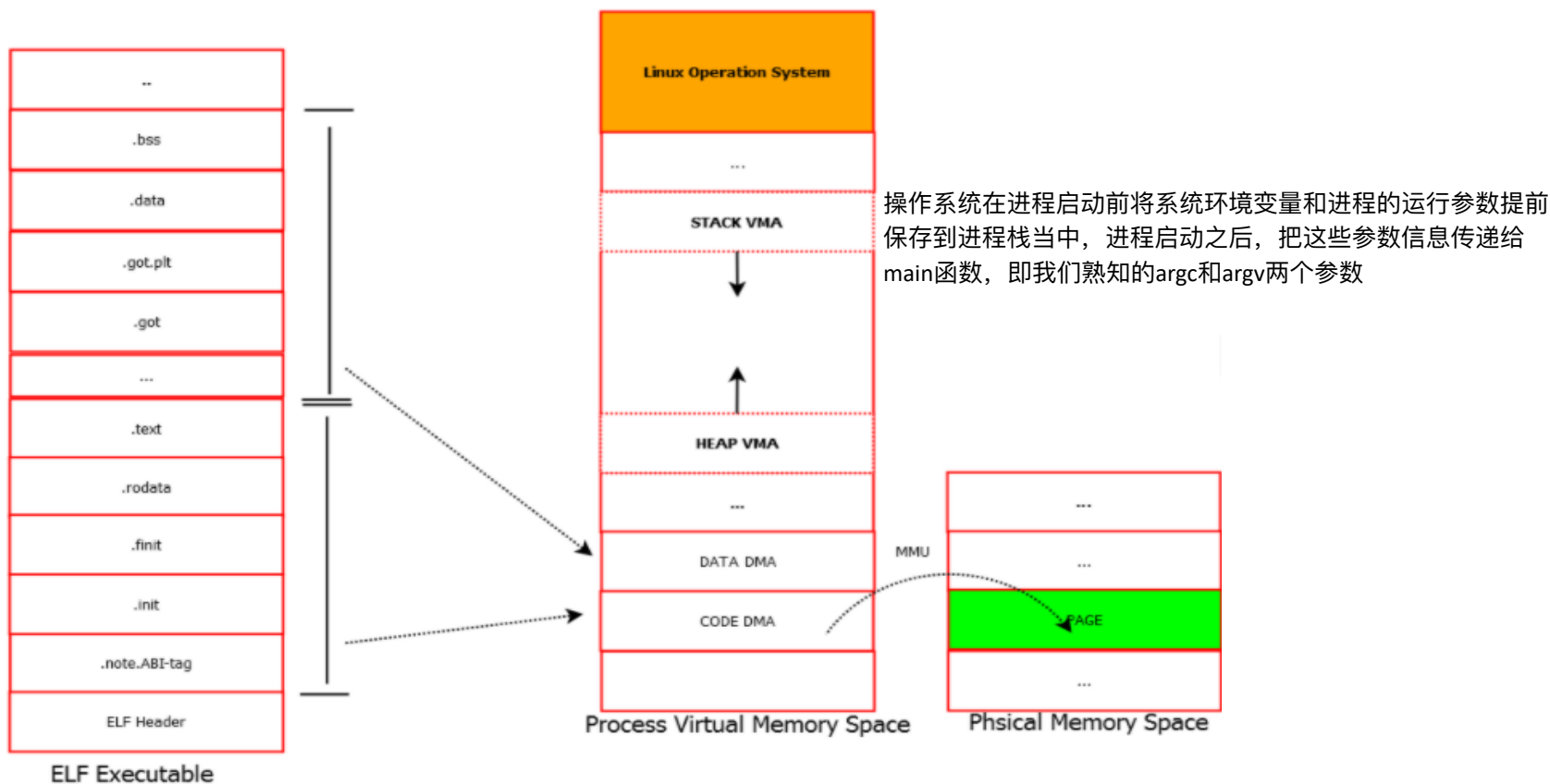
```
[zhoulei@LocalCentOS PartIV]$ cat /proc/3602/maps
00400000-004a6000 r-xp 00000000 fd:00 687306 /home/zhoulei/Desktop/Demo_Linker_and_Loader/PartIV/sta
006a6000-006a8000 rw-p 000a6000 fd:00 687306 /home/zhoulei/Desktop/Demo_Linker_and_Loader/PartIV/sta
006a8000-006aa000 rw-p 00000000 00:00 0
020eb000-0210e000 rw-p 00000000 00:00 0 [heap]
7fffa9d6c000-7fffa9d81000 rw-p 00000000 00:00 0 [stack]
7fffa9dda000-7fffa9ddb000 r-xp 00000000 00:00 0 [vdso]
fffffffff600000-fffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

vdso的全称是虚拟动态共享库（virtual dynamic shared library），而vsyscall的全称是虚拟系统调用（virtual system call），

关于这部分内容有兴趣的读者可以看看<https://0xax.gitbooks.io/linux-insides/content/SysCall/syscall-3.html>

# ELF可执行文件在Linux下的装载过程

ELF可执行文件,进程虚拟地址空间和物理地址空间的映射关系



# ELF可执行文件在Linux下的装载过程

接下来我们进一步探究一下Linux是怎么识别和装载ELF文件的，我们需要深入Linux内核去寻找答案 (内核实际处理过程涉及更多的过程，我们这里主要关注和ELF文件处理相关的代码)

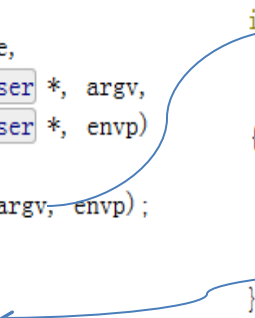
当我们在bash下输入命令执行某一个ELF文件的时候，首先bash进程调用fork()系统调用创建一个新的进程，然后新的进程调用execve()系统调用执行指定的ELF文件，内核开始真正的装载工作

## [linux/fs/exec.c](#)

```
SYSCALL_DEFINE3(execve,
    const char __user *, filename,
    const char __user *const __user *, argv,
    const char __user *const __user *, envp)
{
    return do_execve(getname(filename), argv, envp);
}

static int do_execveat_common(...)
{
    //some error handling
    prepare_binprm(bprm) //read the first 128 (BINPRM_BUF_SIZE) bytes
    exec_binprm(bprm)
}

int do_execve(struct filename *filename,
    const char __user *const __user *__argv,
    const char __user *const __user *__envp)
{
    struct user_arg_ptr argv = { .ptr.native = __argv };
    struct user_arg_ptr envp = { .ptr.native = __envp };
    return do_execveat_common(AT_FDCWD, filename, argv, envp, 0);
}
```



## [/fs/binfmt\\_elf.c](#)

### Load\_elf\_binary的代码走读

检查ELF文件头部信息(一致性检查)

加载程序头表(可以看到一个可执行程序必须至少有一个段(segment)，而所有段的大小之和不能超过64K(65536u))

寻找和处理解释器段(动态链接部分会介绍)

装入目标程序的段(elf\_map)

填写目标程序的入口地址

填写目标程序的参数，环境变量等信息(create\_elf\_tables)

start\_thread会将eip和esp改成新的地址，就使得CPU在返回用户空间时就进入新的程序入口

...

## [linux/fs/binfmt\\_elf.c](#)

```
search_binary_handler(struct linux_binprm *bprm)
{

```

//内核会通过list\_for\_each\_entry遍历所有注册的对象

**load\_elf\_binary**

```
}
```

# 第5部分

## -动态链接



# 什么是动态链接以及为什么要动态链接

- 事实上我们之前介绍的程序大部分都是动态链接的，链接程序在链接时一般是优先链接动态库的，除非我们显式地使用-static参数指定链接静态库，像这样：

```
[zhoulei@LocalCentOS PartIII]$ gcc -o loop -static loop.c
/usr/bin/ld: cannot find -lc
collect2: error: ld returned 1 exit status
```

发生上述错误说明找不到glibc静态库, 此时你可以使用yum install glibc-static安装静态库，然后重新编译。

```
[zhoulei@LocalCentOS PartIII]$ ls -l dyn_loop sta_loop
-rwxrwxr-x. 1 zhoulei zhoulei 6945 Aug 9 20:30 dyn_loop
-rwxrwxr-x. 1 zhoulei zhoulei 894811 Aug 9 20:30 sta_loop
```

- 可以看到静态链接和动态链接的可执行文件的大小差距还是很显著的。

因为静态库被链接后库就直接嵌入可执行文件中了，这样就带来了两个弊端：

- 1.首先就是系统空间被浪费了。这是显而易见的，想象一下，如果多个程序链接了同一个库，则每一个生成的可执行文件就都会有一个库的副本，必然会浪费系统空间。
- 2.再者，一旦发现了库中有bug或者是需要升级，必须把链接该库的程序找出来，然后全部需要重新编译。

动态库的出现正是为了弥补静态库的弊端。因为动态库是在程序运行时被链接的，所以磁盘上只要保留一份副本，因此节约了磁盘空间。如果发现了bug或要升级也很简单，只要用新的库把原来的替换掉就行了。

事实上，Linux环境下的动态链接对象都是以.so为扩展名的共享对象(Shared Object)。

# 动态链接的例子

- 让我们来看一个具体的动态链接的例子，假设我们有两个程序main\_1.c和main\_2.c需要调用同一个函数，而这个函数在动态链接库hello.so中，首先来生成我们自己的这个动态链接库hello.so，像这样：
- 参数-shared 表示产生共享对象，-fPIC表示产生位置无关代码

现在我们可以编译时链接我们的这个动态链接库了，像这样：

```
[zhoulei@LocalCentOS dynamic_example]$ gcc -fPIC -shared -o hello.so hello.c
[zhoulei@LocalCentOS dynamic_example]$ ls
hello.c hello.h hello.so main_1.c main_2.c
```

- 可以看到整个进程的空间中多了我们的hello.so, 它用到了动态链接库libc-2.12.so

```
[zhoulei@LocalCentOS dynamic_example]$ gcc -o main_1 main_1.c ./hello.so
[zhoulei@LocalCentOS dynamic_example]$ ls
hello.c hello.h hello.so main_1 main_1.c main_2.c
[zhoulei@LocalCentOS dynamic_example]$ ./main_1
call hello.so x=1
```

```
[zhoulei@LocalCentOS dynamic_example]$ cat /proc/4323/maps
00400000-00401000 r-xp 00000000 fd:00 687330 /home/zhoulei/Desktop/Demo_Linker_and_Loader/PartIII/dynamic_example/main_1
00600000-00601000 rw-p 00000000 fd:00 687330 /home/zhoulei/Desktop/Demo_Linker_and_Loader/PartIII/dynamic_example/main_1
7f2e74734000-7f2e748be000 r-xp 00000000 fd:00 653570 /lib64/libc-2.12.so
7f2e748be000-7f2e74abe000 ---p 0018a000 fd:00 653570 /lib64/libc-2.12.so
7f2e74abe000-7f2e74ac2000 r--p 0018a000 fd:00 653570 /lib64/libc-2.12.so
7f2e74ac2000-7f2e74ac4000 rw-p 0018e000 fd:00 653570 /lib64/libc-2.12.so
7f2e74ac4000-7f2e74ac8000 rw-p 00000000 00:00 0
7f2e74ac8000-7f2e74ac9000 r-xp 00000000 fd:00 653577 /home/zhoulei/Desktop/Demo_Linker_and_Loader/PartIII/dynamic_example/hello.so
7f2e74ac9000-7f2e74acc000 ---p 00001000 fd:00 653577 /home/zhoulei/Desktop/Demo_Linker_and_Loader/PartIII/dynamic_example/hello.so
7f2e74acc000-7f2e74cc9000 rw-p 00000000 fd:00 653577 /home/zhoulei/Desktop/Demo_Linker_and_Loader/PartIII/dynamic_example/hello.so
7f2e74cc9000-7f2e74ce9000 r-xp 00000000 fd:00 687325 /lib64/ld-2.12.so
7f2e74ce9000-7f2e74ed1000 rw-p 00000000 00:00 0
7f2e74ed1000-7f2e74ee9000 rw-p 00000000 00:00 0
7f2e74ee9000-7f2e74eea000 r--p 00020000 fd:00 687325 /lib64/ld-2.12.so
7f2e74eea000-7f2e74eeb000 rw-p 00021000 fd:00 687325 /lib64/ld-2.12.so
7f2e74eeb000-7f2e74eec000 rw-p 00000000 00:00 0
7fff35ecd000-7fff35ee2000 rw-p 00000000 00:00 0 [stack]
7fff35fff000-7fff36000000 r-xp 00000000 00:00 0 [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

- 此外值得我们关注的一个对象就是ld-2.12.so, 它就是Linux的动态链接器

# 动态链接的实现机制

- 首先我们用 `readelf -l` 命令来看看so文件的属性

```
[zhoulei@LocalCentOS dynamic_example]$ readelf -l hello.so

Elf file type is DYN (Shared object file)
Entry point 0x4f0
There are 5 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz          MemSiz          Flags  Align
LOAD             0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x000000000000070c 0x000000000000070c R E    200000
LOAD             0x0000000000000710 0x00000000000200710 0x00000000000200710
                 0x0000000000000200 0x0000000000000210 RW    200000
DYNAMIC          0x0000000000000738 0x00000000000200738 0x00000000000200738
                 0x0000000000000180 0x0000000000000180 RW     8
GNU_EH_FRAME     0x00000000000006bc 0x00000000000006bc 0x00000000000006bc
                 0x0000000000000014 0x0000000000000014 R      4
GNU_STACK        0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000 RW     8
```

```
[zhoulei@LocalCentOS dynamic_example]$ objdump -s main_1

main_1:      file format elf64-x86-64

Contents of section .interp:
 400200 2f6c6962 36342f6c 642d6c69 6e75782d /lib64/ld-linux-
 400210 7838362d 36342e73 6f2e3200 x86-64.so.2.
```

可以看到编译完成之后地址是从0x00000000开始的，即编译完成之后最终的装载地址是不确定的。

之前在静态链接的过程中我们提到过重定位的过程，那个时候其实属于链接时的重定位，现在我们需要装载时的重定位 - 使用了诸如1)PIC位置无关代码, 2)延迟绑定等技术(主要是性能优化)来实现。

- 引入动态链接之后，实际上在操作系统开始运行我们的应用程序之前，首先会把控制权交给动态链接器，它完成了动态链接的工作之后再把控制权交给应用程序。
- 可以看到动态链接器的路径在.interp这个段中体现，并且通常它是个软链接。

```
[zhoulei@LocalCentOS dynamic_example]$ ls -l /lib64/ld-linux-x86-64.so.2
lrwxrwxrwx. 1 root root 10 Aug  9 20:28 /lib64/ld-linux-x86-64.so.2 -> ld-2.12.so
```

# 动态链接的实现机制(续)

- 我们来看一下和动态链接相关的.dynamic段和它的结构

```
[zhoulei@LocalCentOS dynamic_example]$ readelf -d hello.so

Dynamic section at offset 0x738 contains 20 entries:
   Tag               Type              Name/Value
0x0000000000000001 (NEEDED)           Shared library: [libc.so.6]
0x000000000000000c (INIT)             0x4a8
0x000000000000000d (FINI)             0x698
0x0000000000000004 (HASH)             0x158
0x0000000000000005 (STRTAB)           0x2f8
0x0000000000000006 (SYMTAB)           0x1a8
0x000000000000000a (STRSZ)            178 (bytes)
0x000000000000000b (SYMENT)           24 (bytes)
0x0000000000000003 (PLTGOT)           0x2008e0
0x0000000000000002 (PLTRELSZ)         48 (bytes)
0x0000000000000014 (PLTREL)           RELA
0x0000000000000017 (JMPREL)           0x478
0x0000000000000007 (RELA)             0x3e8
0x0000000000000008 (RELASZ)           144 (bytes)
0x0000000000000009 (RELAENT)          24 (bytes)
0x000000006ffffffe (VERNEED)          0x3c8
0x000000006fffffff (VERNEEDNUM)       1
0x000000006ffffff0 (VERSYM)           0x3aa
0x000000006ffffff9 (RELACOUNT)        1
0x0000000000000000 (NULL)             0x0
```

其实左侧显示的内容可以直接对应到相应的/  
*usr/include/elf.h*头文件中的结构体Elf64\_Dyn

```
typedef struct
{
    Elf64_Sxword d_tag;                /* Dynamic entry type */
    union
    {
        Elf64_Xword d_val;             /* Integer value */
        Elf64_Addr d_ptr;              /* Address value */
    } d_un;
} Elf64_Dyn;
```

# 动态链接的实现机制(续)

- 动态符号表(和静态符号表几乎一样)

```
[zhoulei@LocalCentOS dynamic_example]$ readelf -sD hello.so

Symbol table for image:

```

Num	Buc:	Value	Size	Type	Bind	Vis	Ndx	Name
13	0:	000000000000004a8	0	FUNC	GLOBAL	DEFAULT	8	_init
12	0:	0000000000200910	0	NOTYPE	GLOBAL	DEFAULT	ABS	__edata
4	0:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	_Jv_RegisterClasses
3	0:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	__gmon_start__
11	1:	0000000000200920	0	NOTYPE	GLOBAL	DEFAULT	ABS	__end
8	1:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	_ITM_registerTMCloneTable
7	1:	0000000000000638	35	FUNC	GLOBAL	DEFAULT	10	print x
6	1:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	_ITM_deregisterTMCloneTab
5	1:	0000000000000698	0	FUNC	GLOBAL	DEFAULT	11	_fini
10	2:	0000000000200910	0	NOTYPE	GLOBAL	DEFAULT	ABS	__bss_start
9	2:	0000000000000000	0	FUNC	WEAK	DEFAULT	UND	__cxa_finalize
2	2:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	printf

我们已经知道在静态链接中未知符号的引用在最终链接的时候被修正，在动态链接中，需要在运行时修正，也就是重定位。

# 动态链接的实现机制(续)

- 重定位表

```
[zhoulei@LocalCentOS dynamic_example]$ readelf -r hello.so

Relocation section '.rela.dyn' at offset 0x3e8 contains 6 entries:
   Offset                Info                Type                Sym. Value          Sym. Name + Addend
000000200908  00000000000008 R_X86_64_RELATIVE  0000000000000000    0000000000200908
0000002008b8  00030000000006 R_X86_64_GLOB_DAT  0000000000000000    __gmon_start__ + 0
0000002008c0  00040000000006 R_X86_64_GLOB_DAT  0000000000000000    Jv_RegisterClasses + 0
0000002008c8  00060000000006 R_X86_64_GLOB_DAT  0000000000000000    ITM_deregisterTMClone + 0
0000002008d0  00080000000006 R_X86_64_GLOB_DAT  0000000000000000    ITM_registerTMCloneTa + 0
0000002008d8  00090000000006 R_X86_64_GLOB_DAT  0000000000000000    __cxa_finalize + 0

Relocation section '.rela.plt' at offset 0x478 contains 2 entries:
   Offset                Info                Type                Sym. Value          Sym. Name + Addend
0000002008f8  00020000000007 R_X86_64_JUMP_SLO  0000000000000000    printf + 0
000000200900  00090000000007 R_X86_64_JUMP_SLO  0000000000000000    __cxa_finalize + 0
```

我们已经知道在静态链接中未知符号的引用在最终链接的时候被修正，在动态链接中，需要在运行时修正，也就是重定位。

比如我们这里的printf, 它的类型是R\_X86\_64\_JUMP\_SLO, 重定位偏移offset=2008f8, 可以看到实际上它位于 .got.plt这个段  
当链接器需要进行重定位时，先查找printf 的地址，它位于libc.so, 链接器把这个地址填到.got.plt中偏移为2008f8的位置  
中，从而实现了地址的重定位。

[19]	.got	PROGBITS	00000000002008b8	000008b8
	0000000000000028	0000000000000008	WA	0 0 8
[20]	.got.plt	PROGBITS	00000000002008e0	000008e0
	0000000000000028	0000000000000008	WA	0 0 8
[21]	.data	PROGBITS	0000000000200908	00000908
	0000000000000008	0000000000000000	WA	0 0 8

# 动态链接的实现机制(续)

- 对于动态链接的可执行文件，内核会分析它的动态链接器地址，把动态链接器映射到进程的地址空间，把控制权交给动态链接器。
- 动态链接器本身也是so文件，但是它比较特殊，它是可执行的。

```
[zhoulei@LocalCentOS lib64]$ ./ld-2.12.so
Usage: ld.so [OPTION]... EXECUTABLE-FILE [ARGS-FOR-PROGRAM...]
You have invoked 'ld.so', the helper program for shared library executables.
This program usually lives in the file '/lib/ld.so', and special directives
in executable files using ELF shared libraries tell the system's program
loader to load the helper program from this file. This helper program loads
the shared libraries needed by the program executable, prepares the program
to run, and runs it. You may invoke this helper program directly from the
command line to load and run an ELF executable file; this is like executing
that file itself, but always uses this helper program from the file you
specified, instead of the helper program file specified in the executable
file you run. This is mostly of use for maintainers to test new versions
of this helper program; chances are you did not intend to run this program.

--list                list all dependencies and how they are resolved
--verify              verify that given object really is a dynamically linked
                      object we can handle
--library-path PATH   use given PATH instead of content of the environment
                      variable LD_LIBRARY_PATH
--inhibit-rpath LIST  ignore RUNPATH and RPATH information in object names
                      in LIST
--audit LIST           use objects named in LIST as auditors
```

- Linux的动态链接器本身是静态链接的，本身不依赖任何其他的共享对象也不能使用全局和静态变量。

```
[zhoulei@LocalCentOS lib64]$ ldd ld-2.12.so
statically linked
```

- Linux的动态链接器是Glibc的一部分，入口地址是sysdeps/x86\_64/dl-machine.h中的\_start，然后调用elf/rtld.c的\_dl\_start函数，最终调用dl\_main(动态链接器的主函数)

```
/* Initial entry point code for the dynamic linker.
   The C function '_dl_start' is the real entry point;
   its return value is the user program's entry point. */
#define RTLD_START asm ("\\n\\
.text\\n\\
    .align 16\\n\\
.globl _start\\n\\
.globl _dl_start_user\\n\\
_start:\\n\\
    movq %rsp, %rdi\\n\\
    call _dl_start\\n\\
```

# 动态链接(后续)

- 谈一谈共享库的使用，使用我们自己写的共享库

我们把hello.so拷贝到/usr/lib

```
[zhoulei@LocalCentOS dynamic_example]$ ls
hello.c hello.h hello.so.bak main_1 main_1.c main_2.c
[zhoulei@LocalCentOS dynamic_example]$ gcc -o main main_1.c
/tmp/ccbUmqw4.o: In function `main':
main_1.c:(.text+0xa): undefined reference to `print_x'
collect2: error: ld returned 1 exit status
```

- 注意：最好要符合命名规范，起名为libhello.so

```
[root@LocalCentOS dynamic_example]# ./main
./main: error while loading shared libraries: libhello.so: cannot open shared object file: No such file or directory
[root@LocalCentOS dynamic_example]# ldd main
linux-vdso.so.1 => (0x00007fffd45b7000)
libhello.so => not found
libc.so.6 => /lib64/libc.so.6 (0x0000003fcd600000)
/lib64/ld-linux-x86-64.so.2 (0x0000003fcd200000)
```

可以执行ldconfig,这个程序的作用是为共享库更新符号链接

```
[root@LocalCentOS dynamic_example]# ldconfig
[root@LocalCentOS dynamic_example]# ldd main
linux-vdso.so.1 => (0x00007fff8a9ff000)
libhello.so => /usr/lib/libhello.so (0x00007f5c1bbe7000)
libc.so.6 => /lib64/libc.so.6 (0x0000003fcd600000)
/lib64/ld-linux-x86-64.so.2 (0x0000003fcd200000)
```



# 第6部分

## -入口函数和运行库

# 程序从main函数开始执行？

我们有充分的理由证明main函数并不是操作系统装载程序后首先运行的代码

```
#include <stdio.h>

__attribute__((constructor)) void before_main()
{ printf("%s\n", __FUNCTION__); }

int main() {
    printf("%s\n", __FUNCTION__);
}
```

```
[root@LocalCentOS PartV]# gcc -o entry entry.c
[root@LocalCentOS PartV]# ./entry
before_main
main
```

```
#include <stdio.h>

void post(void)
{
    printf("goodbye!\n");
}

int main()
{
    atexit(&post);
    printf("exiting from main\n");
}
```

```
[root@LocalCentOS PartV]# gcc -o atexit atexit.c
[root@LocalCentOS PartV]# ./atexit
exiting from main
goodbye!
```

事实上操作系统装载程序之后首先运行的代码并不是我们编写的主函数的第一行，而是某些**运行库**的代码，它们负责初始化main函数正常执行所需要的环境，负责调用main函数，并且在main返回之后，记录main函数的返回值，调用atexit注册的函数，最后结束进程。

以Linux的运行库glibc为例，所谓的入口函数，其实就是指ld 默认的连接脚本所指定的程序入口\_start (默认情况下)

# Linux运行库glibc介绍

<http://www.gnu.org/software/libc/>

glibc = GNU C library

Linux环境下的C语言运行库glibc包括

-启动和退出相关的函数

-C标准库函数的实现 (标准输入输出, 字符处理, 数学函数等等)

...

事实上运行库是和平台相关的, 和操作系统联系的非常紧密, 我们可以把运行库理解成我们的C语言(包括c++)程序和操作系统之间的抽象层, 使得大部分时候我们写的程序不用直接和操作系统的API和系统调用直接打交道, 运行库把不同的操作系统API抽象成相同的库函数, 方便应用程序的使用和移植。

# glibc入口函数导读(1)

从可执行文件的ELF头中我们可以看到程序的入口

我们的实例程序来看, 0x400460就是这个程序\_start的地址

```
[zhoulel@LocalCentOS PartV]$ readelf -h main
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                           ELF64
  Data:                               2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                               EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:                0x400460
```

```
0000000000400460 <_start>:
400460: 31 ed                xor    %ebp,%ebp
400462: 49 89 d1             mov    %rdx,%r9
400465: 5e                  pop    %rsi
400466: 48 89 e2             mov    %rsp,%rdx
400469: 48 83 e4 f0          and    $0xfffffffffffffff0,%rsp
40046d: 50                  push   %rax
40046e: 54                  push   %rsp
40046f: 49 c7 c0 00 06 40 00 mov    $0x400600,%r8
400476: 48 c7 c1 10 06 40 00 mov    $0x400610,%rcx
40047d: 48 c7 c7 c0 05 40 00 mov    $0x4005c0,%rdi
400484: e8 af ff ff ff      callq 400438 <_libc_start_main@plt>
400489: f4                  hlt
40048a: 90                  nop
40048b: 90                  nop
```

接下来我们从源代码角度看一下这个入口函数做了一些什么事情

[glibc/csu/libc-start.c](#)

[glibc/sysdeps/x86\\_64/start.S](#)

xor是异或运算, ebp指向栈底, 所以与自身做异或运算会把ebp设置为零, 表明当前是程序的最外层函数, rdx中其实存放的是rtld\_fini的函数指针, 并将其存入r9, pop将argc存入rsi中, 然后将栈指针指向argv, 再通过mov将该地址存入rdx中, 栈指针寄存器与掩码进行and运算来重置自己, 接下来, 我们通过查看反编译的代码, 会发现0x400600是\_\_libc\_csu\_fini的地址, 被存入r8中, 0x400610是\_\_libc\_csu\_init的地址, 被存入rcx中, 0x4005c0是main的地址, 被存入rdi中。最后代码调用\_\_libc\_start\_main, 这个函数才是实际执行代码的函数, 而之前的那些对寄存器的设置其实就是对\_\_libc\_start\_main的函数参数的设置。

```
int LIBC_START_MAIN (int (*main) (int, char **, char ** MAIN_AUXVEC_DECL), int
argc, char **argv, __typeof (main) init, void (*fini) (void), void (*rtld_fini) (void), void
*stack_end)
```

几个重要的调用

1. \_\_cxa\_atexit ((void (\*)(void \*)) rtld\_fini, NULL, NULL); //动态装载机收尾工作
2. \_\_cxa\_atexit ((void (\*)(void \*)) fini, NULL, NULL); //main结束后收尾工作
3. (\*init) (argc, argv, \_\_environ MAIN\_AUXVEC\_PARAM); //main调用前的工作
4. result = main (argc, argv, \_\_environ MAIN\_AUXVEC\_PARAM); //main函数调用
5. exit (result);

由\_\_libc\_start\_main函数的实现可以看出, 程序在执行完main函数后都会执行exit函数(具体实现在stdlib/exit.c中)。所以, 在main函数中返回一个整型数值与在main末尾用该值调用exit函数是等价的。exit会执行通过atexit注册过的函数, 然后调用\_exit(由它来调用操作系统提供的exit系统调用)来结束进程。

# glibc入口函数导读(2)

```
/usr/local/libexec/gcc/x86_64-unknown-linux-gnu/4.8.2/collect2 -m elf_x86_64 -s
tatic -o main /usr/lib/../../lib64/crt1.o /usr/lib/../../lib64/crti.o /usr/local/lib/g
cc/x86_64-unknown-linux-gnu/4.8.2/crtbeginT.o -L/usr/local/lib/gcc/x86_64-unknow
n-linux-gnu/4.8.2 -L/usr/local/lib/gcc/x86_64-unknown-linux-gnu/4.8.2/../../..
./lib64 -L/lib/../../lib64 -L/usr/lib/../../lib64 -L/usr/local/lib/gcc/x86_64-unknown-
linux-gnu/4.8.2/../../.. /tmp/ccALCpgi.o --start-group -lgcc -lgcc_eh -lc --end-
group /usr/local/lib/gcc/x86_64-unknown-linux-gnu/4.8.2/crtend.o /usr/lib/../../lib
64/crtn.o
```

Glibc有几个重要的辅助程序运行的库 /usr/lib64/crt1.o, /usr/lib64/crti.o, /usr/lib64/crtn.o

其中crt1包含了基本的启动退出代码, ctri和crtn包含了关于.init段及.finit段相关处理的代码(实际上是\_init()和\_fini()的开始和结尾部分)

0000000004003f0 <\_init>:

4003f0:	48 83 ec 08	<b>sub</b> \$0x8,%rsp
4003f4:	e8 93 00 00 00	<b>callq</b> 40048c <call_gmon_start>
4003f9:	e8 92 01 00 00	<b>callq</b> 400590 <frame_dummy>
4003fe:	e8 9d 02 00 00	<b>callq</b> 4006a0 <__do_global_ctors_aux>
400403:	48 83 c4 08	<b>add</b> \$0x8,%rsp
400407:	c3	<b>retq</b>

0000000004006d8 <\_fini>:

4006d8:	48 83 ec 08	<b>sub</b> \$0x8,%rsp
4006dc:	e8 3f fe ff ff	<b>callq</b> 400520 <__do_global_dtors_aux>
4006e1:	48 83 c4 08	<b>add</b> \$0x8,%rsp
4006e5:	c3	<b>retq</b>

Glibc是运行库, 它对语言的实现并不太了解, 真正实现C++语言特性的是gcc编译器, 所以gcc提供了两个目标文件crtbeginT.o和crtend.o来实现C++的全局构造和析构 – 实际上以上两个高亮出来的函数就是gcc提供的, 有兴趣的读者可以自己翻阅gcc源代码进一步深入学习。

# 参考文献资料

- **Linker and Loader - by John R. Levine**
- **Computer Systems A Programmer's Perspective - by Randal E. Bryant & David O'Hallaron**
- **Intel Architecture Software Developer's Manual - by Intel Company**