

Processor Arch

13 元培数科 常欣海

Here we go! →

2024/11/20



Knowledge Review

This part is cited from Arthals

链接

Linking

链接：将多个目标文件组合成一个可执行文件。

目标文件：编译器将源代码文件编译后的产物，但还未链接成最终的可执行文件。

可执行文件：链接后的最终产物，这个文件可被加载（复制）到内存并执行。

链接在 **编译时、加载时、运行时** 均可执行（后文会说都是啥）。

链接的优势：

- **代码模块化：**方便查找问题与维护，可以建立常见函数的库
- **时间和空间效率：**只需要改一处代码可同时影响多个代码，只会编译你引用的库中用到的代码

从源代码到可执行文件

From source code to executable file

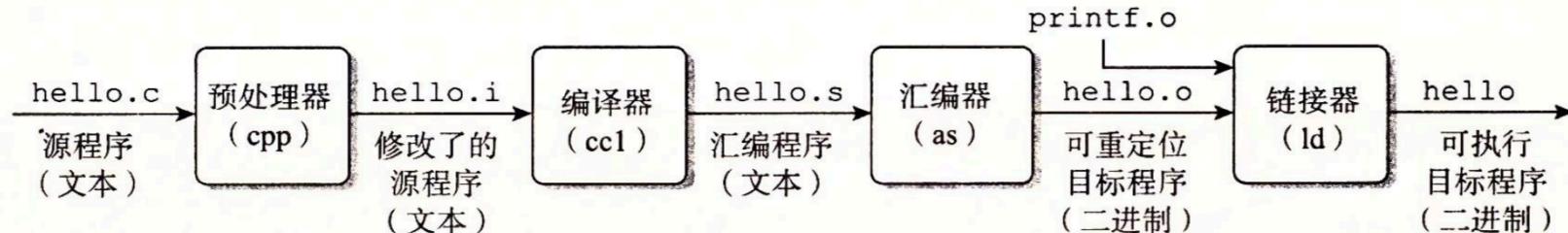


图 1-3 编译系统

$$\text{.c} \xrightarrow{\text{cpp}} \text{.i} \xrightarrow{\text{cc1}} \text{.s} \xrightarrow{\text{as}} \text{.o} \xrightarrow{\text{ld}} \text{prog}$$

- gcc : 编译器驱动程序
- cpp : C 预处理 (c preprocessor) , 将 xx.c 翻译成一个 ASCII 码的 中间文件 xx.i (intermediate file)
- cc1 : C 编译器 (c compiler) , 将 xx.i 翻译成一个 ASCII 汇编语言文件 xx.s (assembly language file)
- as : 汇编器 (assembler) , 将 xx.s 翻译成一个 可重定位目标文件 xx.o (relocatable object file)
- ld : 链接器 (linker) , 将 xx.o 和其他的 xx.o, 以及必要的系统目标文件组合起来, 创建一个 可执行目标文件 prog

* 书 1.2 章, 知道大家都没看, 可以回去看。

静态链接

Static Linking

main.c

```
int sum(int *a, int n); // 这里声明了函数，但未定义
int array[2] = {1, 2};
```

```
int main()
{
    int val = sum(array, 2);
    return val;
}
```

sum.c

```
int sum(int *a, int n) // 这里定义了函数
{
    int i, s = 0;
    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

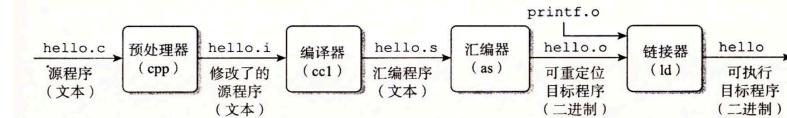


图 1-3 编译系统

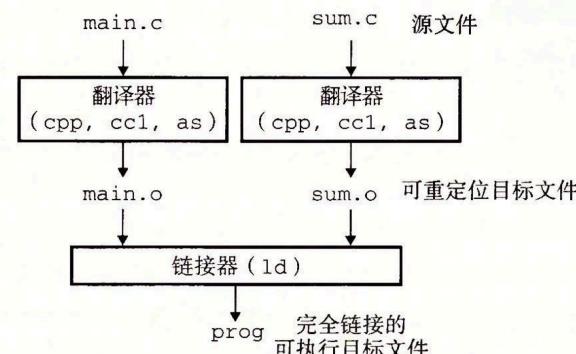


图 7-2 静态链接。链接器将可重定位目标文件组合起来，形成一个可执行目标文件 prog

静态链接

Static Linking

在静态链接中，链接器需要实现如下功能：

- **符号解析**：将 **符号引用** 与输入的可重定位目标文件中的 **符号表** 中的一个确定的符号（全局变量、函数等）关联起来。
- **重定位**：**合并重定位输入模块**，将符号定义与一个地址关联起来，并为找到的每个符号指向对应的该地址。

目标文件

Object File

1. 可重定位目标文件

- 这是一个包含计算机能直接理解的代码和数据的文件，**但它还不能单独运行。**
- 它就像是一个半成品，需要和其他半成品合并在一起才能变成一个完整的产品。

2. 可执行目标文件

- 这是一个已经准备好 **可以直接在计算机上运行的文件。**
- 它就像是一个已经组装好的玩具，拿到手就可以玩了，不需要再做其他操作。

3. 共享目标文件

- **特殊类型的可重定位目标文件**，可以在程序运行的时候 **动态加载** 进来使用。
- 它就像是一个插件，可以在需要的时候装上去，不需要的时候可以取下来。

可重定位目标文件

Relocatable Object File

ELF 可重定位目标文件

- ELF 头 (ELF Header) [低地址]

- 节 (Sections)

- .text (机器代码)
- .rodata (只读数据)
- .data (已初始化数据)
- .bss (未初始化数据)
- .symtab (符号表)
- .rel.text (重定位信息)
- .rel.data (重定位信息)
- .debug (调试信息)
- .strtab (字符串表)

- 节头部表 (Section Header Table) [高地址]

- 节头部条目 1
- 节头部条目 2
- ...
- 节头部条目 N

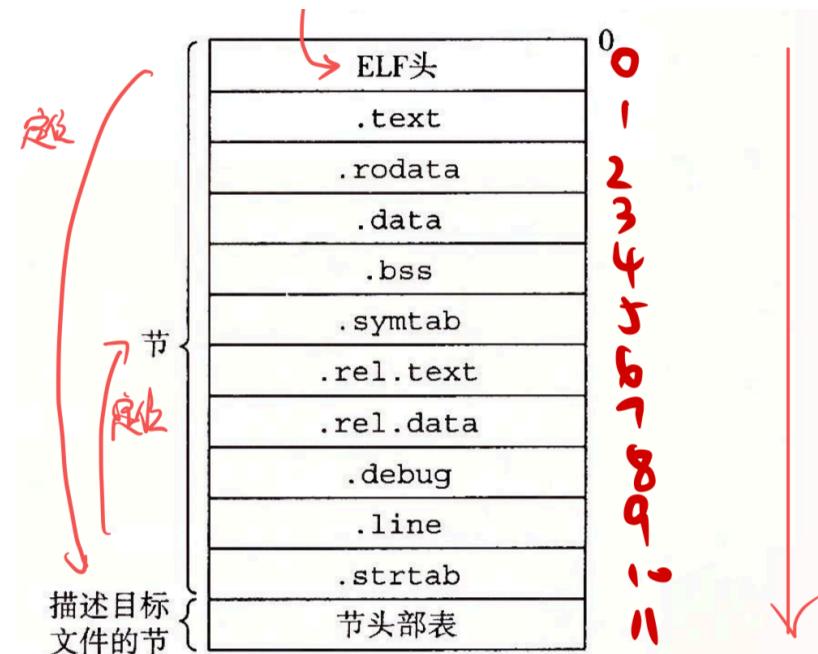


图 7-3 典型的 ELF 可重定位目标文件

可重定位目标文件

Relocatable Object File

ELF 头

存储关于这个二进制文件的一般信息：

文件的类型、字节序、ELF 头长度、节头部表偏移量、节头部表条目大小和数量等信息。

运行时，会根据这些信息：

1. 先定位到 ELF 头
2. 然后根据 ELF 头中的信息找到节头部表
3. 再根据节头部表找到相应的节

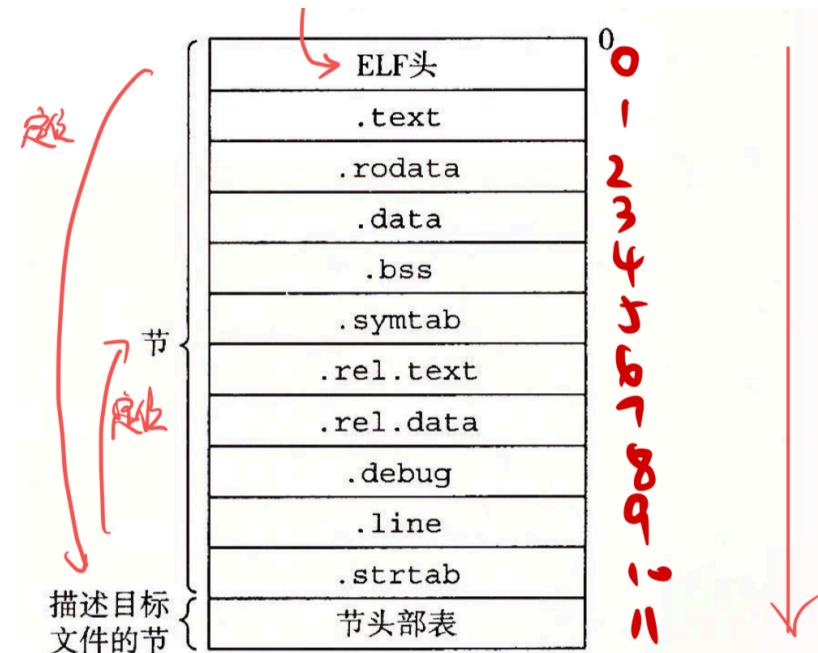


图 7-3 典型的 ELF 可重定位目标文件

可重定位目标文件

Relocatable Object File

节

.text : 已编译程序的机器代码

.rodata : 只读 (read-only) 数据, 如 printf 中的格式串、开关语句的跳转表、字符串常量等

.data : 已初始化的全局与静态 C 变量

.bss : 未初始化的全局和静态 C 变量, 以及所有被初始化为 0 的全局和静态 C 变量

(Block Storage Start / Better Save Space)

* 将初始化为 0 的变量放在 .bss 段而不是 .data 段, 可以使可执行文件更小, 因为 .bss 段只需要记录变量的大小和位置, 而不需要存储实际的 0 值 (直到运行时才真的去内存中分配空间)。

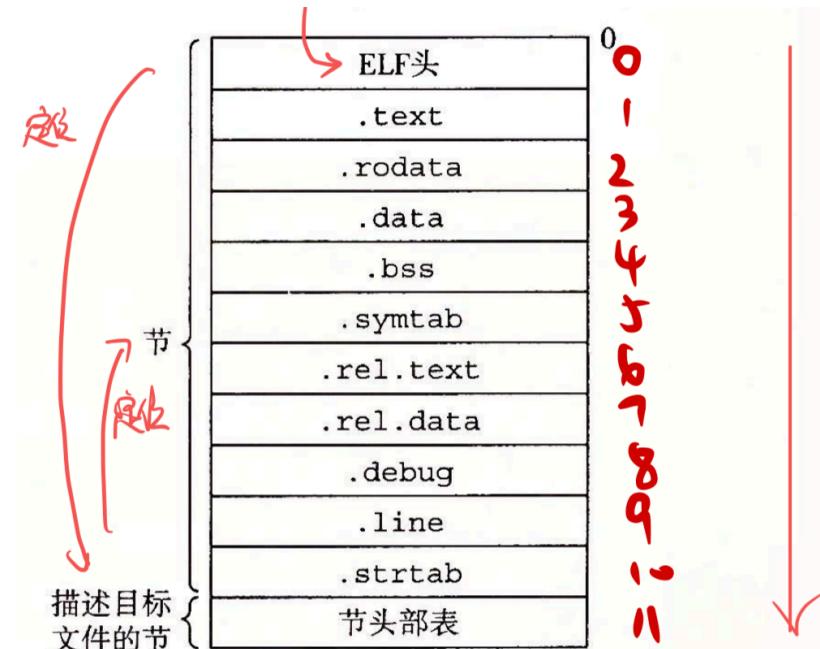


图 7-3 典型的 ELF 可重定位目标文件

可重定位目标文件

Relocatable Object File

节

.symsymtab : 符号表 (symbol table) , 存放定义和引用的函数与全局变量信息

.rel.text : 重定位信息 (relocate text)

.rel.data : 重定位信息 (relocate data) , (未初始化或为 0 的变量不需要重定位)

.strtab : 字符串表 (string table) , 包括:

- .symsymtab 和 .debug 节中的符号表
- 节头部中的节名字

字符串表就是以 null (\0) 结尾的字符串的序列。

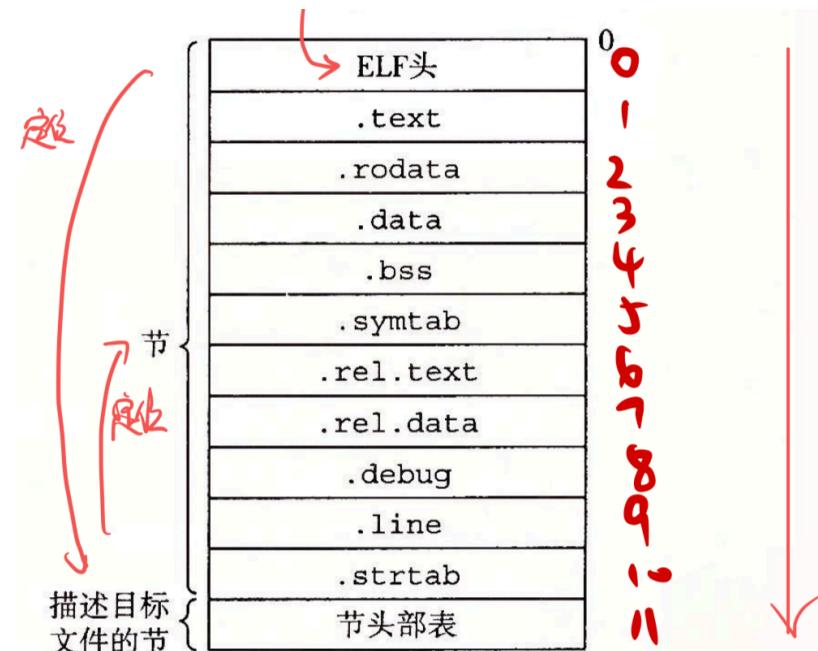


图 7-3 典型的 ELF 可重定位目标文件

可重定位目标文件

Relocatable Object File

节头部表

存储不同部分的起始位置。

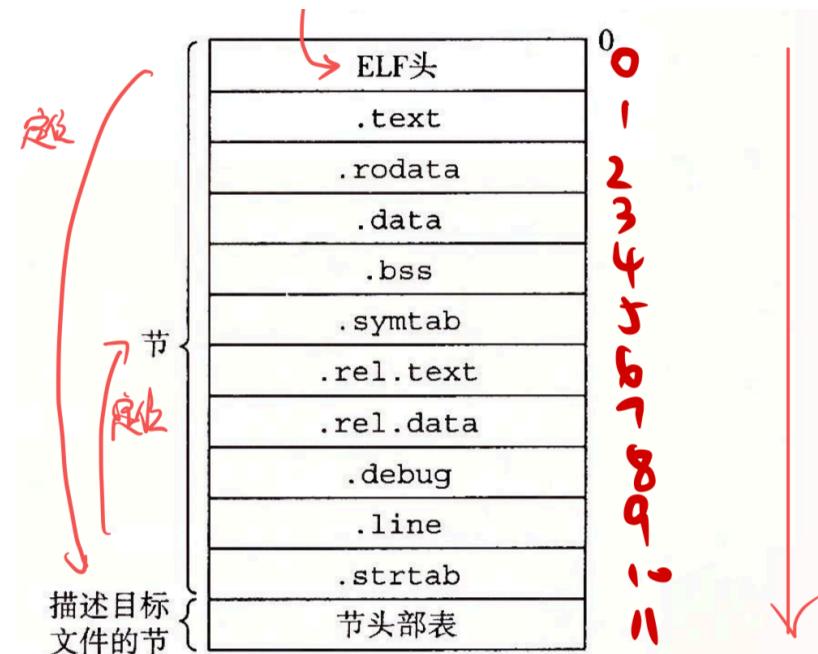


图 7-3 典型的 ELF 可重定位目标文件

符号和符号表

Symbols and Symbol Table

每个可重定位目标模块 m 都有一个符号表，包含符号的定义和引用详情，它们包括：

- **全局符号**: m 定义并能被其他模块 n 引用的，对应非静态的 C 函数和全局变量。
- **局部符号**: m 定义且 只 在模块 m 内自己使用的，对应带 `static` 属性的 C 函数和全局变量，这些符号在模块 m 内任何位置都可见。
- **外部符号**: m 引用但 m 中 未定义 的，对应其他模块 n 定义的函数或全局变量。

符号表 不关心本地非静态程序变量，如：

```
int main() {  
    int a = 1;  
    return 0;  
}
```

这里的 `a` 是局部变量，不会出现在符号表中，而是出现在栈中。



Static 属性

`static` Attribute

`static` 定义的函数和全局变量 **只能在其声明模块中使用（类似 C++ 的 `private`）**，对其他模块不可见。

编译器会在 `.data` 或 `.bss` 中为其分配空间，创造一个具有唯一名字的符号。

在右侧代码中：

- 带有 `static` 关键字的变量会出现在符号表中，且在 `.data` 或 `.bss` 中分配空间
- 未带有 `static` 关键字的变量不会出现在符号表中，而是出现在栈中
- 由于出现了重名的 `static` 变量，编译器生成不同名称的局部链接器符号，如 `x.1` 表示函数 `f` 中的静态变量，而 `x.2` 表示函数 `g` 中的静态变量

```
int f() {
    static int x = 0;
    int y = 1;
    return x + y;
}

int g() {
    static int x = 1;
    int y = 2;
    return x + y;
}
```

符号表结构

Symbol Table Structure

- `name` : 指向符号的 null (`\0`) 结束的字符串名字
- `value`
 - 对于可重定位的模块来说, 表示相对于定义所在节 (`section`) 起始位置的偏移
 - 对于可执行目标文件来说, 该值是一个绝对运行时地址
- `size` : 目标的大小 (字节为单位)
- `type` : 符号类型 (函数或数据)
- `binding` : 符号是否是本地的还是全局的 (是否可以被其他模块引用, 即有无 `static` 关键字) 
- `section` : 符号所在的节

```
typedef struct {
    int name;           // .strtab 中的偏移
    char type:4;        // 函数或数据
    binding:4;          // 局部或全局
    char reserved;      // 是否使用
    short section;      // 节头部表索引
    long value;         // 节内偏移或绝对地址
    long size;          // 对象大小 (字节)
} Elf64_Symbol;
```

符号表条目分配

Allocation of Symbol Table Entries

每个符号将被分配到目标文件的某个节 (section) , 如 .text .data .bss 等。

有三个特殊的伪节 (pseudosection) , 它们在节头部表中是没有条目的:

- ABS : 代表不该被重定位的符号 (absolute)
- UNDEF : 代表未定义的符号 (undefined)
- COMMON : 未被分配位置的未初始化的数据目标

COMMON vs .bss :

- COMMON : 未初始化的全局变量, **从而它们可能定义在其他模块中**
- .bss : 未初始化的静态变量, 以及初始化为 0 的全局或静态变量, **必然定义在当前模块中**

```
int a; // 未初始化的全局变量, 属于 COMMON
static int b; // 未被初始化的静态变量, 属于 .bss
```

符号表条目分配

Allocation of Symbol Table Entries

对于 `m.o` 和 `swap.o` 模块，对每个符号在 `swap.o` 中定义或引用的符号，指出是否在 **模块 swap.o 的 .symtab 节** 中有一个符号条目。如果是，请指出该符号的模块（`swap.o` 或者 `m.o`）、符号类型（局部、全局或者外部）以及它在模块中被分配到的节（`.text`、`.data`、`.bss` 或 `COMMON`）。

```
// m.c
void swap();
int buf[2] = {1, 2};

int main()
{
    swap();
    return 0;
}
```

```
// swap.c
extern int buf [];

int *bufp0 = &buf[0];
int *bufp1;

void swap(){
    int temp;

    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```

符号	.symtab 条目？	符号类型	在哪个模块中定义	节
buf	✓	外部	m.o	UND
bufp0	✓	全局	swap.o	.data
bufp1	✓	全局	swap.o	COMMON
swap	✓	全局	swap.o	.text
temp	✗	局部	swap.o	

符号解析

Symbol Resolution

符号解析：在多个模块间，将每个引用与一个确定的符号定义关联起来。

链接时，将编译器输出的全局符号分为：

- **强符号**：包含函数和已初始化的全局变量
- **弱符号**：未初始化的全局变量

选择规则：

1. 不允许有多个同名的强符号
2. 如果有一个强符号和多个弱符号同名，则选择强符号
3. 如果有多个弱符号同名，则从这些弱符号中任意选择一个（一般选择第一个）

多个全局符号同名的时候，即使不违背上述规则，也有可能会导致潜在问题：类型混淆、预期以外的作用等。

为了安全，尽量使用 `static` 属性定义全局变量。

符号解析的问题

Symbol Resolution Problems

Case 1

```
// foo1.c
int main()
{
    return 0;
}

// bar1.c
int main() // 重复定义强符号
{
    return 0;
}
```

Case 2

```
// foo2.c
int x = 15213;
int main()
{
    return 0;
}

// bar2.c
int x = 15213; // 重复定义强符号
int f()
{
    return 0;
}
```

Case 3

```
// foo3.c
#include <stdio.h>
void f(void);
int x = 15213;
int main()
{
    f(); // 会修改 x 的值
    printf("x = %d\n", x);
    return 0;
}

// bar3.c
int x;
void f()
{
    x = 15212;
}
```

符号解析的问题

Symbol Resolution Problems

Case 4

```
// foo4.c
#include <stdio.h>
void f(void);

int y = 15212; // 高地址
int x = 15213; // 低地址

int main()
{
    f();
    printf("x = 0x%x y = 0x%x\n",
           x, y);
    // 0x0 0x80000000
    return 0;
}
```

```
// bar4.c
double x;

void f()
{
    x = -0.0;
}
```

x 会按照 double (8 字节) 的方式计算，但实际上 x 只占用 4 个字节的空间（因为它是 int 类型），这会导致对内存的非法访问。

静态库

Static Libraries

静态库 (Static Library)：是一组目标文件的集合，通过将这些目标文件打包成一个单独的文件来实现代码的重用。静态库在编译时链接到应用程序中，生成一个包含所有代码的可执行文件。

Why 静态库？

1. 区分标准函数 / 程序函数：复杂
2. 所有标准函数放一起，存为单独的可重定位目标文件：文件体积太大，维护要重新打包
3. 每个函数一个模块：文件多、维护复杂，手动指定链接累死了
4. 相关的函数才放在一起： ✓ ✓ ✓

静态库的创建和使用

Creating and Using Static Libraries

创建静态库

1. 编译源文件生成目标文件 (.o文件, object file)。

```
gcc -c addvec.c multvec.c # -c 只编译, 不链接
```

2. 使用 ar (archive) 工具将目标文件打包成静态库:

```
ar rcs libvector.a addvec.o multvec.o
```

使用静态库

1. 编译使用静态库的程序文件。

```
gcc -c main2.c
```

2. 链接生成可执行文件, 指定静态库的位置。

```
gcc -static -o prog2c main2.o ./libvector.a
```

-static : 指定静态链接, 无需运行时链接

或者使用 -L 和 -l 参数 (library) :

```
gcc -static -o prog2c main2.o -L. -lvector
```

解析静态库引用

Resolving Static Library References

对于一行代码，链接器从左到右按需解析库中的符号：

```
gcc foo.c libx.a liby.a libz.a
```

集合定义：

- E : 最终所需要的成员目标文件 (Exported)，目标文件肯定丢进去，存档文件看有没有用上再决定。
- U : 未解析的符号 (Undefined)，没定义的符号就先加进去，遇到存档文件再看能不能匹配上。
- D : 已定义的符号 (Defined)，已经匹配上、找到定义的符号。

解析完成后，不在 E 中的成员目标文件会被丢掉（用不上）。

解析完成后， U 非空，链接器会输出一个错误并终止。

解析静态库引用

Resolving Static Library References

因为从左到右解析，所以在命令行链接的顺序变得至关重要。

```
gcc foo.c libx.a liby.a libz.a
```

和

```
gcc foo.c libz.a liby.a libx.a
```

结果不一样，前者会报错，后者不会。

- `.a` 文件中也可能有未定义的符号，也可能依赖于其他 `.a` 文件
- 在解析过程中，一个 `.a` 或 `.o` 文件可以出现多次。

怎么做题：画出依赖图，满足最终的次序能够遍历每条有向边。

重定位

Relocation

重定位节: 将所有相同类型的节合并为同一类型的新的聚合节，并把运行时内存地址赋给新的聚合节。这样，程序中的每一条指令和全局变量都有唯一的运行时内存地址了。

重定位符号: 链接器会修改代码节和数据节中对每个符号的引用，使得它们指向正确的运行时内存地址。这一步依赖于可重定位目标模块中的 **重定位条目**。

重定位条目

Relocation Entries

当 **汇编器** 生成一个目标模块时，它并不知道数据和代码最终将放在内存中的什么位置。

所以，它就会生成一个**重定位条目**，告诉 **链接器** 将目标文件合并成可执行文件时如何修改这个引用。

代码的重定位条目放在 `.rel.text` 中，数据的重定位条目放在 `.rel.data` 中。

ELF 重定位条目的格式

```
typedef struct {
    long offset;      /* 偏移量：引用重定位的偏移量 */
    long type:32;    /* 重定位类型 */
    long symbol:32;  /* 符号表索引 */
    long addend;     /* 常量部分：重定位表达式的常量部分 */
} Elf64_Rela;
```

重定位过程

Relocation Process

重定位类型

- R_X86_64_PC32 : 重定位一个使用 32 位 PC 相对地址的引用
- R_X86_64_32 : 重定位一个使用 32 位绝对地址的引用

相对地址：距离 PC 的**当前运行时值**的偏移量

重定位过程

Relocation Process

`r.addend`：对于绝对重定位来讲，一般就设置为 0 了。

此时，我们直接就有：

```
*refptr = ADDR(r.symbol)
```

```

foreach section s { /* 迭代节 */
    foreach relocation entry r { /* 迭代重定位条目 */
        refptr = s + r.offset; /* 需要重定位的引用的指针，要写哪里 */

        /* 重定位 PC-relative 引用 */
        if (r.type == R_X86_64_PC32) {
            refaddr = ADDR(s) + r.offset; /* 引用的运行时地址 */
            /* addend = -4 */
            *refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr);
        }

        /* 重定位绝对引用 */
        if (r.type == R_X86_64_32) {
            /* addend = 0 */
            *refptr = (unsigned) (ADDR(r.symbol) + r.addend);
        }
    }
}

```

相对重定位举例

PC-relative Relocation Example

```
r.offset = 0xf
r.symbol = sum
r.type = R_X86_64_PC32
r.addend = -4
```

- r.offset : call 指令第二个字节与 main 函数起始地址的偏移量 = 0xf
- r.retptr : 要修改的地址, $s + r.offset = \text{main} + 0xf = 0xf$
- refaddr : 引用的运行时地址 = $\text{ADDR}(s) + r.offset = 0x4004d0 + 0xf = 0x4004df$
- PC : 运行时的 PC, 下一条指令的地址 = 0x4004e3
- r.addend : 补偿 refaddr 和运行时 PC 之间的差值 = $0x4004df - 0x4004e3 = -4$
- ADDR(r.symbol) : 真实要得到的地址, 也即 sum 的运行时地址 = 0x4004e8

```
*refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr) =
(unsigned) (0x4004e8 + (-4) - 0x4004df) = 5
```

```
1 0000000000000000 <main>:
2   0: 48 83 ec 08          sub  $0x8,%rsp
3   4: be 02 00 00 00       mov  $0x2,%esi
4   9: bf 00 00 00 00       mov  $0x0,%edi
5   Oxf = retptr, r.offset  a: R_X86_64_32 array
6   e: e8 00 00 00 00       callq 13 <main+0x13>
7   r.addend
8   13: 48 83 c4 08       f: R_X86_64_PC32 sum=0x4
9   17: c3                 add  $0x8,%rsp
                           retq
```

链接前

```
1 0000000004004d0 <main>:
2   4004d0: 48 83 ec 08      sub  $0x8,%rsp
3   4004d1: be 02 00 00 00    mov  $0x2,%esi
4   4004d2: bf 18 10 60 00    mov  $0x601018,%edi
PC 4004d3: e8 05 00 00 00    callq 4004e8 <sum> %edi = &array
6   4004e3: 48 83 c4 08      add  $0x8,%rsp
7   4004e7: c3                retq
```

ADDR(r.symbol)

0x4004df = refaddr

```
8   0000000004004e8 <sum>:
9   4004e8: b8 00 00 00 00    mov  $0x0,%eax
10  4004e9: ba 00 00 00 00    mov  $0x0,%edx
11  4004f0: eb 09            jmp  4004fd <sum+0x15>
12  4004f1: 48 63 ca          movslq %edx,%rcx
13  4004f2: 03 04 8f          add  (%rdi,%rcx,4),%eax
14  4004f3: 83 c2 01          add  $0x1,%edx
15  4004f4: 39 f2            cmp  %esi,%edx
16  4004f5: 7c f3            jl   4004f4 <sum+0xc>
17  400501: f3 c3            repz retq
```

(b) Relocated .data section

```
1 000000000601018 <array>:
2   601018: 01 00 00 00 02 00 00 00
```

链接后

绝对重定位举例

Absolute Relocation Example

```
r.offset = 0xa
r.symbol = array
r.type = R_X86_64_32
r.addend = 0
```

- r.offset : call 指令第二个字节与 main 函数起始地址的偏移量 = 0xa
- r.retptr : 要修改的地址, s + r.offset = main + 0xa = 0xa
- r.addend : 对于绝对重定位, 设置为 0。
- ADDR(r.symbol) : 真实要得到的地址, 也即 array 的运行时地址 = 0x601018

```
*retptr = (unsigned) (ADDR(r.symbol) + r.addend) = (unsigned)
(0x601018 + 0) = 0x601018
```

```
1 0000000000000000 <main>:
2   0: 48 83 ec 08          sub   $0x8,%rsp
3   4: be 02 00 00 00       mov    $0x2,%esi
4   9: bf 00 00 00 00       mov    $0x0,%edi
5   13: 48 83 c4 08        mov    %rax,%rcx
6   e: e8 00 00 00 00       callq 13 <main+0x13>
7   f: R_X86_64_PC32 sum=0x4
8   13: 48 83 c4 08        add    $0x8,%rsp
9   17: c3                  retq
```

链接前

```
1 0000000004004d0 <main>:
2  4004d0: 48 83 ec 08          sub   $0x8,%rsp
3  4004d4: be 02 00 00 00       mov    $0x2,%esi
4  4004d9: bf 18 10 60 00       mov    $0x601018,%edi
5  4004de: e8 05 00 00 00       callq 4004e8 <sum>
6  4004e3: 48 83 c4 08          add    $0x8,%rsp
7  4004e7: c3                  retq

8 0000000004004e8 <sum>:
9  4004e8: b8 00 00 00 00       mov    $0x0,%eax
10 4004ed: ba 00 00 00 00      mov    $0x0,%edx
11 4004f2: eb 09                jmp   4004fd <sum+0x15>
12 4004f4: 48 63 ca            movslq %edx,%rcx
13 4004f7: 03 04 8f            add    (%rdi,%rcx,4),%eax
14 4004fa: 83 c2 01            add    $0x1,%edx
15 4004fd: 39 f2                cmp    %esi,%edx
16 4004ff: 7c f3                jl    4004f4 <sum+0xc>
17 400501: f3 c3                repq  retq

(b) Relocated .data section ADDR(r.symbol)
1 000000000601018 <array>:
2  601018: 01 00 00 00 02 00 00 00
```

链接后

可执行目标文件

Executable Object File

ELF 头描述文件的整体格式，包含程序的入口点（entry point），即程序运行时执行的第一条指令地址。

段：链接器根据目标文件中 **属性相同的多个节合并后的节的集合**，这个集合称为段。

- `.init` 段：比可重定位目标文件多出来的，包含初始代码（`_init` 函数），会在程序开始时调用。
- `.text` `.rodata` `.data` 段：与同名节对应，已被重定位到最终的运行时内存地址。
- `.rel.text` 和 `.rel.data` 节：不再存在，因为已经完全链接。

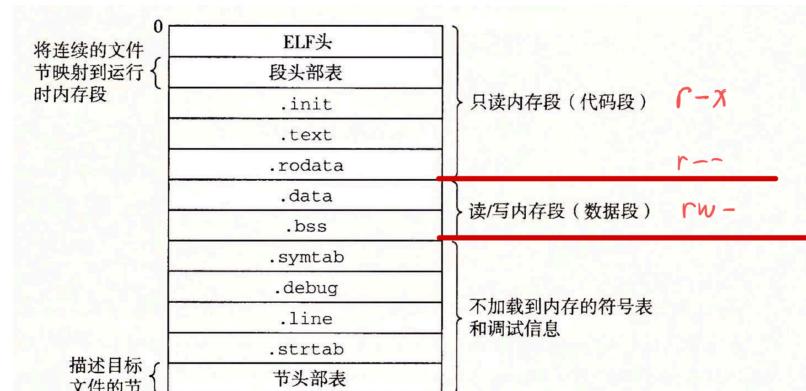


图 7-13 典型的 ELF 可执行目标文件

段代表的是可执行代码和数据等内存区域，而节则更加抽象，它代表的是文件中的一组相关数据。在 ELF 文件中，节是按照功能和目的来划分的，比如代码节、数据节、符号表节等等，而段则是按照内存区域来划分的。

加载可执行目标文件

Loading an Executable Object File

内存布局

- 代码段**: 从地址 $0x400000 = 2^{21}$ 开始, 后面是数据段
- 堆内存**: 由 `malloc` 分配, 运行时堆在数据段之后
- 用户栈**: 从最大合法用户地址 $2^{48} - 1$ 向下增长
- 内核区**: 从地址 2^{48} 开始, 用于内核代码和数据段

`_start` (入口点) \rightarrow `_libc_start_main` (定义在 `libc.so` 中) \rightarrow `main`

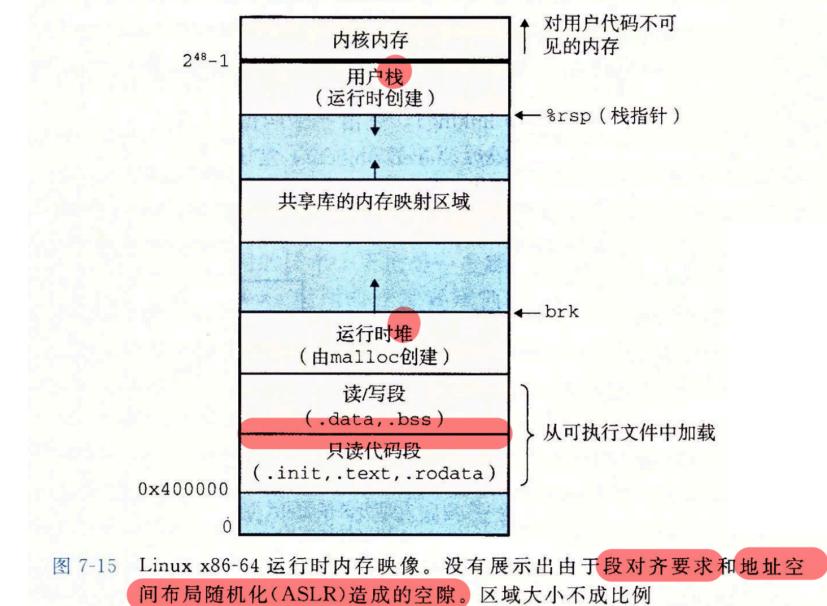


图 7-15 Linux x86-64 运行时内存映像。没有展示出由于段对齐要求和地址空间布局随机化(ASLR)造成的空隙。区域大小不成比例

动态链接共享库

Dynamic Linking Shared Libraries

共享库：在运行时被动态加载和链接的库文件，通常以 `.so` (Linux) 或 `.dll` (Windows) 为后缀。

- 节省资源：避免静态库复制很多次
- 简化更新：更新共享库只需替换库文件，无需重新编译所有依赖的程序
- 动态链接：运行时加载库文件，多个程序共享同一份库文件

C 标准库 `libc.so` 通常是动态链接的。

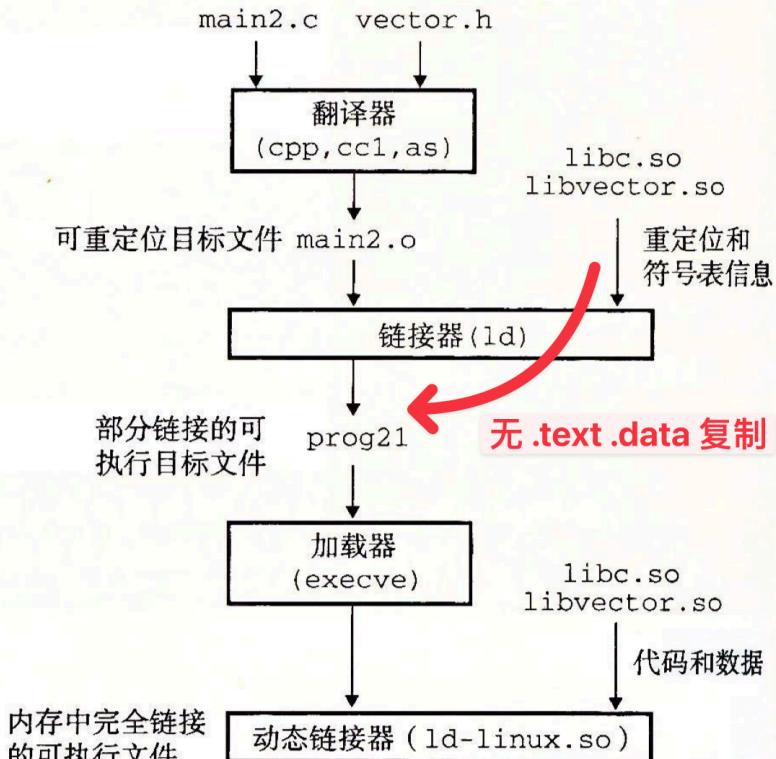


图 7-16 动态链接共享库

动态链接共享库

Dynamic Linking Shared Libraries

链接器、加载器、动态链接器

- 链接器**: 在编译阶段之后工作, 将多个目标文件和库文件链接成一个可执行文件或库文件。
- 加载器**: 在程序执行阶段工作, 将可执行文件加载到内存并准备好执行环境。
- 动态链接器**: 在加载器将程序加载到内存后、程序开始执行前工作, 负责在运行时加载动态库并解析符号(重定位)。

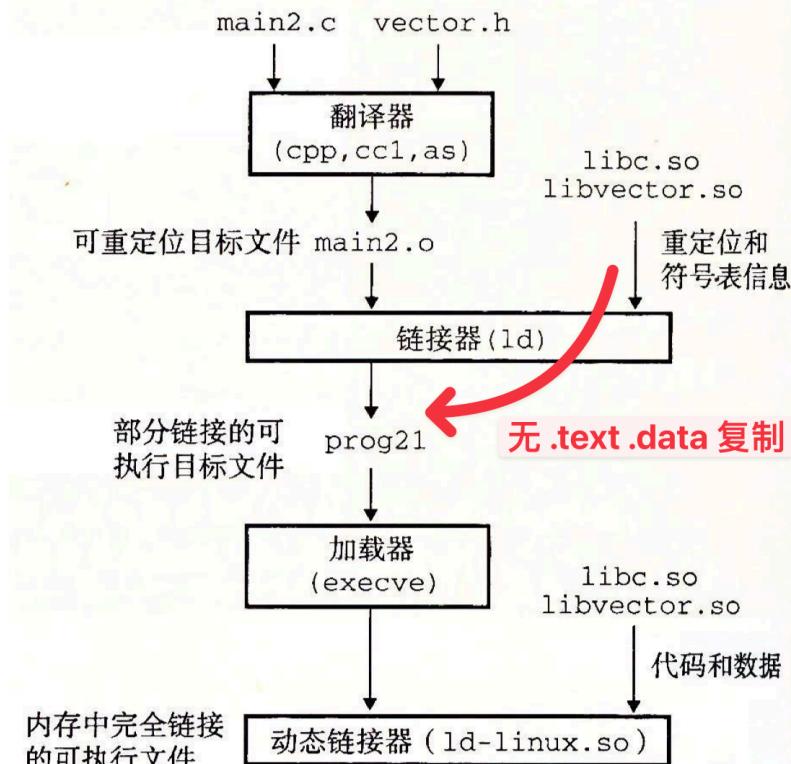


图 7-16 动态链接共享库

位置无关代码

Position Independent Code

位置固定：会引起内存的浪费，无法高效利用（Why?）

位置无关：共享库可以被加载到内存的任何位置，从而多个程序可以同时使用同一个共享库实例，节省内存。

需要使用 `-fpic` (flag position independent code) 选项编译共享库

过程链接表 (PLT) 和 全局偏移表 (GOT)

Procedure Link Table and Global Offset Table, PLT & GOT

假设有一个共享库函数 `foo`，以下是调用过程：

1. 在第一次调用 `foo` 时，程序跳转到 PLT 表的 `foo` 入口
2. PLT 表的 `foo` 入口跳转到 **动态链接器的解析函数**
3. 动态链接器解析 `foo` 的地址，并将地址写入 GOT 表中相应的条目
4. 动态链接器返回，程序继续执行 `foo` 函数
5. 下一次调用 `foo` 时，PLT 表直接从 GOT 表中读取 `foo` 的地址并跳转，**无需再次解析**

PLT 和 GOT 举例

PLT & GOT Example

```
# 数据段
# 全局偏移量表 (GOT)
GOT[0]: addr of .dynamic
GOT[1]: addr of reloc entries # 重定位条目的地址
GOT[2]: addr of dynamic linker # 动态链接器的地址
GOT[3]: 0x4005b6 # sys startup
GOT[4]: 0x4005c6 # addvec()
GOT[5]: 0x4005d6 # printf()
```

```
# 代码段
callq 0x4005c0 # call addvec()
```

```
# 过程链接表 (PLT)
# PLT[0]: call dynamic linker
4005a0: pushq *GOT[1]
4005a6: jmpq *GOT[2]
...
# PLT[2]: call addvec()
4005c0: jmpq *GOT[4]
4005c6: pushq $0x1 # addvec 的 ID
4005cb: jmpq 4005a0
```

```
# 数据段
# 全局偏移量表 (GOT)
GOT[0]: addr of .dynamic
GOT[1]: addr of reloc entries
GOT[2]: addr of dynamic linker
GOT[3]: 0x4005b6 # sys startup
GOT[4]: &addvec()
GOT[5]: 0x4005d6 # printf()
```

```
# 代码段
callq 0x4005c0 # call addvec()
```

```
# 过程链接表 (PLT)
# PLT[0]: call dynamic linker
4005a0: pushq *GOT[1]
4005a6: jmpq *GOT[2]
...
# PLT[2]: call addvec()
4005c0: jmpq *GOT[4]
4005c6: pushq $0x1
4005cb: jmpq 4005a0
```

库打桩机制

Library Interposition

打桩: 在运行时替换库函数的行为。

打桩机制有三种主要方法:

- 编译时打桩
- 链接时打桩
- 运行时打桩

编译时打桩

Compile-Time Interposition

概念：通过在编译源代码时插入打桩函数来实现。

实现方式：使用 `-D` 编译选项将标准函数替换为自定义函数。

编译命令：

```
gcc -I. -o intc int.c mymalloc.o
```

- `-I.`：告诉编译器在当前目录（`.`）中查找头文件（Include）
- `-o intc`：指定输出文件名为 `intc`
- `int.c`：源文件
- `mymalloc.o`：自定义动态库

这样做后，会在搜索 `malloc` 时，优先搜索 `mymalloc.o` 中的 `malloc`，然后再搜索通常的系统目录。

链接时打桩

Link-Time Interposition

概念：在链接阶段，用自定义函数替换标准库函数。

实现方式：使用 `--wrap` 选项告诉链接器用自定义函数替换标准函数。

链接命令：

```
gcc -c mymalloc.c # -c 只编译，不链接，得到 mymalloc.o
gcc -c int.c # 得到 int.o
# -Wl: 将后面的选项传递给链接器 ld。
gcc -Wl,--wrap,malloc \
    -Wl,--wrap,free \
    -o intl int.o mymalloc.o # 链接时打桩
```

```
// mymalloc.c
void *__real_malloc(size_t size);
void *__wrap_malloc(size_t size) {
    void *ptr = __real_malloc(size); // 调用原始malloc
    printf("malloc(%zu) = %p\n", size, ptr);
    return ptr;
}
...

// int.c
#include <stdio.h>
#include <malloc.h>
int main(){
    int *p = malloc(32);
    free(p);
    return 0;
}
```

运行时打桩

Runtime Interposition

概念：在程序运行时，动态替换库函数。

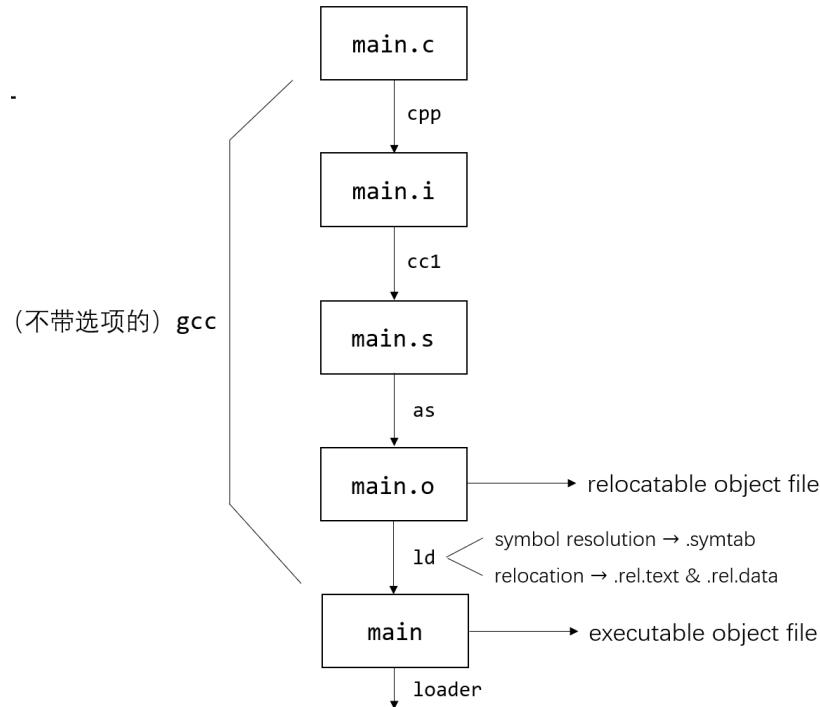
实现方式：使用 `LD_PRELOAD` 环境变量加载自定义动态库。

```
gcc -shared -fpic -o mymalloc.so mymalloc.c -ldl  
LD_PRELOAD="./mymalloc.so" ./myprogram
```

- `-shared`：生成共享库
- `-fpic`：生成位置无关代码，**这是共享库所必需的，因为共享库可以加载到内存中的任何位置**
- `-ldl`：链接动态链接器库，`libdl` 是动态链接器库
- `LD_PRELOAD`：环境变量，指定在运行程序时加载的自定义动态库

Emphasis

编译流程



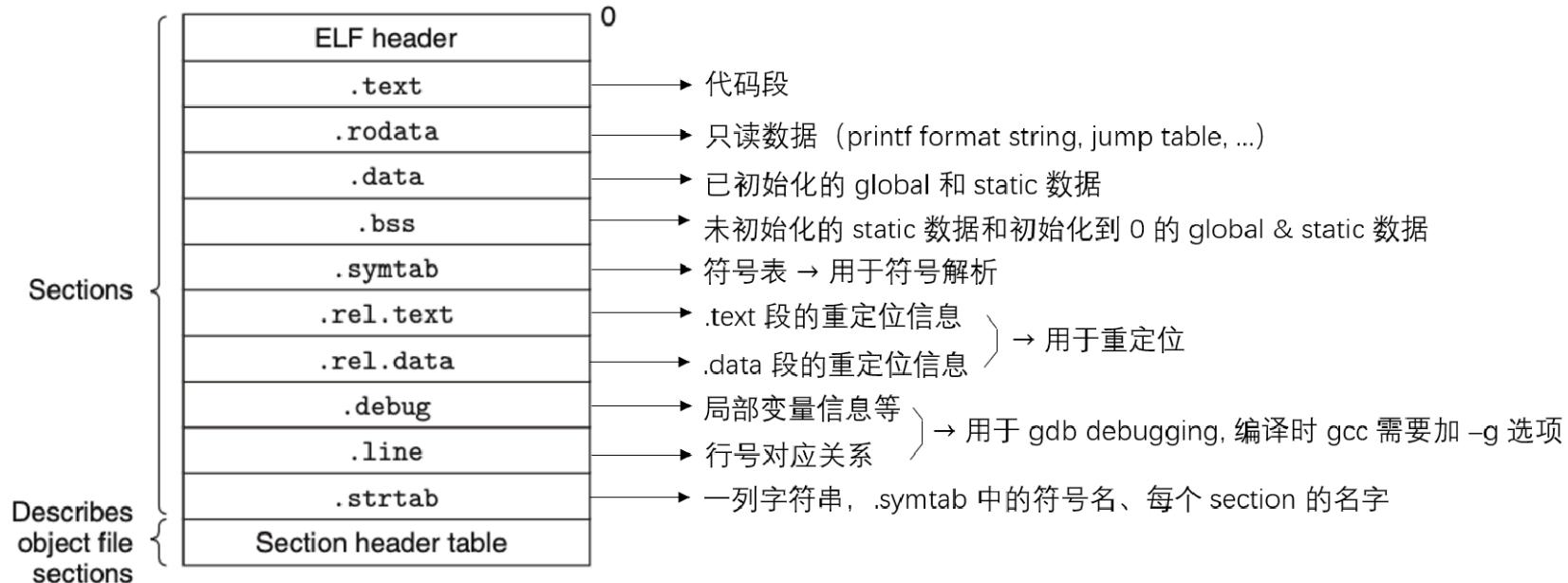
预处理器 cpp → 编译器 cc1 → 汇编器 as → 链接 ld → 加载器 loader

文件符号

- **源文件 (.c)** : 包含人类可读的源代码。
- **预处理文件 (.i)** : 包含展开的宏和头文件内容（通过预处理生成）。
- **汇编文件 (.s)** : C 文件编译后的汇编代码（通过编译生成）。
- **目标文件 (.o)** : 汇编文件或 C 文件编译后的机器代码，包含符号表和重定位信息。
- **静态库文件 (.a)** : 包含多个 .o 文件的归档文件，用于静态链接。
- **共享库文件 (.so)** : 动态链接库文件，程序运行时加载。
- **可执行文件**: 最终的可执行程序，包含所有已解析的符号和重定位的代码。

.c -> .o -> .a / .so -> **可执行文件** 是一个常见的编译-链接-执行的完整流程

目标文件-ELF



符号和符号表

- `.symtab` : 一个符号表，它存放在程序中**定义和引用的函数和全局变量的信息**

- ▶ 符号表中的变量：所有**有声明的非局部变量**
- ▶ 区分符号与变量：**函数内部的静态变量是变量而不是符号**，因为其进入符号表时会增加后缀（如 `x.0`）
- ▶ `readelf` 导出的 ELF 信息：
 - ▶ **Value**: 该符号相对节头的位置
 - ▶ **Size**: 该符号大小
 - ▶ **Ndx**: 该符号所在节的编号，或是 ABS, COM, UND
- ▶ 三种符号与对应的 Bind 信息：
 - ▶ 全局符号：非静态函数 + 非静态全局变量 → `global`
 - ▶ 外部符号：其他模块定义的全局符号 → `global`
 - ▶ 局部符号：静态函数 + 静态变量（包括静态全局和静态局部变量）→ `local`

```

1  typedef struct {
2      int     name;        /* String table offset */
3      char    type:4,      /* Function or data (4 bits) */
4          binding:4; /* Local or global (4 bits) */
5      char    reserved;   /* Unused */
6      short   section;    /* Section header index */
7      long    value;       /* Section offset or absolute address */
8      long    size;        /* Object size in bytes */
9  } Elf64_Symbol;

```

Figure 7.4 ELF symbol table entry. The type and binding fields are 4 bits each.

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
8:	0000000000000000	24	FUNC	GLOBAL	DEFAULT	1	main
9:	0000000000000000	8	OBJECT	GLOBAL	DEFAULT	3	array
10:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	sum

伪节

- 伪节只存在于可重定位目标文件中，不存在于可执行目标文件中
- ABS : 不该被重定位的符号
- UNDEF : 未定义的符号（只有引用）
- COMMON : 未初始化的全局变量（适配符号解析规则）
 - 初始化为 0 的全局变量与所有静态变量存放于 .bss 中
 - 在可执行目标文件中 COMMON 保存的数据会进入 .bss

COMMON Uninitialized global variables

.bss Uninitialized static variables, and global or static variables that are initialized to zero

示例

```

int global_uninit;
int global_init_0 = 0;
int global_init_1 = 1;

static int global_static_uninit;
static int global_static_init_0 = 0;
static int global_static_init_1 = 1;

int global_func_def(int x)
{
    return x + 1;
}

static int global_static_func_def(int x)
{
    return x + 1;
}

int global_func_decl_with_ref(int);
int global_func_decl_without_ref(int);

int main()
{
    int local_uninit;
    int local_init_0 = 0;
    int local_init_1 = 1;

    static int local_static_uninit;
    static int local_static_init_0 = 0;
    static int local_static_init_1 = 1;

    global_func_decl_with_ref(3);
    return 0;
}

```

```

ubuntu@yaen:~/ICS/class-in/10-linking$ readelf -s example.o

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 example.c
 2: 0000000000000000 0 SECTION LOCAL DEFAULT .text
 3: 0000000000000008 4 OBJECT LOCAL DEFAULT 4 global_static_uninit
 4: 000000000000000c 4 OBJECT LOCAL DEFAULT 4 global_static_init_0
 5: 0000000000000004 4 OBJECT LOCAL DEFAULT 3 global_static_init_1
 6: 0000000000000013 19 FUNC LOCAL DEFAULT 1 global_static_fu[...]
 7: 0000000000000008 4 OBJECT LOCAL DEFAULT 3 local_static_init_1.2
 8: 0000000000000010 4 OBJECT LOCAL DEFAULT 4 local_static_init_0.1
 9: 0000000000000014 4 OBJECT LOCAL DEFAULT 4 local_static_uninit.0
10: 0000000000000000 4 OBJECT GLOBAL DEFAULT 4 global_uninit
11: 0000000000000004 4 OBJECT GLOBAL DEFAULT 4 global_init_0
12: 0000000000000000 4 OBJECT GLOBAL DEFAULT 3 global_init_1
13: 0000000000000000 19 FUNC GLOBAL DEFAULT 1 global_func_def
14: 0000000000000026 43 FUNC GLOBAL DEFAULT 1 main
15: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND global_func_decl[...]
ubuntu@yaen:~/ICS/class-in/10-linking$ █

```

可重定位目标文件

文件类型

- **文件后缀**: 通常以 `.o` (Unix/Linux) 、 `.obj` (Windows) 为后缀。
- **生成方式**: 通过编译源文件生成。例如:

```
gcc -c example.c -o example.o
```

- **示例**: 假设有文件 `example.c` , 将其编译为 `example.o` , 得到的就是一个可重定位目标文件。

可重定位目标文件包含以下内容:

- **代码段**: 如 `.text` 段, 包含编译生成的机器指令。
- **数据段**: 如 `.data` (已初始化的全局变量) 和 `.bss` (未初始化的全局变量) 段。
- **符号表**: 列出文件中的符号 (函数、变量等) 定义和引用。
- **重定位信息**: 标识出需要在链接阶段调整的地址。

可执行目标文件

文件类型

- **文件后缀**: 在 Unix/Linux 系统上没有特定后缀, 一般命名为 `a.out` 或直接以程序名称命名; 在 Windows 系统上通常以 `.exe` 作为后缀。
- **生成方式**: 通过链接器将多个 `.o` 文件组合并重定位生成。例如:

```
gcc main.o example.o -o example_program
```

- **示例**: 运行 `gcc main.o example.o -o example_program` 后生成的 `example_program` 就是一个可执行目标文件。

可执行目标文件包含以下内容:

- **代码段**: 包含程序的指令, 设置为可执行。
- **数据段**: 包含程序的数据, 已完成重定位。
- **程序入口**: 指明程序开始执行的入口地址。
- **动态链接信息** (若为动态链接) : 包含对共享库的依赖信息。

共享目标文件

文件类型

- **文件后缀**: 在 Unix/Linux 系统上以 `.so` 为后缀, 在 Windows 系统上以 `.dll` 为后缀。
- **生成方式**: 通过编译器或链接器指定生成共享库。例如:

```
gcc -fPIC -shared example.c -o libexample.so
```

- **示例**: 运行 `gcc -fPIC -shared example.c -o libexample.so` 后生成的 `libexample.so` 就是一个共享目标文件。

共享目标文件包含以下内容:

- **代码段**: 包含共享库的代码, 可以被多个程序加载并共享。
- **数据段**: 包含共享库的全局数据。
- **动态段**: 记录共享库的依赖和符号表, 用于动态链接时符号解析。
- **全局偏移表 (GOT) 和过程链接表 (PLT)** : 用于延迟绑定的符号解析机制。

特性	可重定位目标文件 (.o)	可执行目标文件 (无特定后缀或 .exe)	共享目标文件 (.so / .dll)
文件生成阶段	编译阶段	链接阶段	动态链接库生成
文件是否可执行	否，需进一步链接	是，可直接加载并运行	否，需动态链接到进程中
地址类型	相对地址 (需重定位)	绝对地址	可相对也可延迟解析
主要内容	代码段、数据段、符号表、重定位信息	代码段、数据段、入口地址	代码段、数据段、动态段、符号表
应用场景	链接输入文件，生成可执行文件或库	可执行文件，程序入口由系统加载	共享库，在多个程序中共享使用
链接方式	静态链接	静态或动态链接	动态链接
重定位需求	需要在链接时重定位	已重定位完成	加载时动态重定位
共享性	独立，不共享	独立，不共享	支持多个进程共享

流程梳理

■ 静态链接

- 符号解析
 - 符号定义
 - 解析规则：强、弱、未定义
 - 解析方法：E、U、D

■ 重定位

- 重定位条目（Relocation Entry）
- 重定位类型（PC-relative、absolute）
- 重定位算法（Relocation Algorithm）

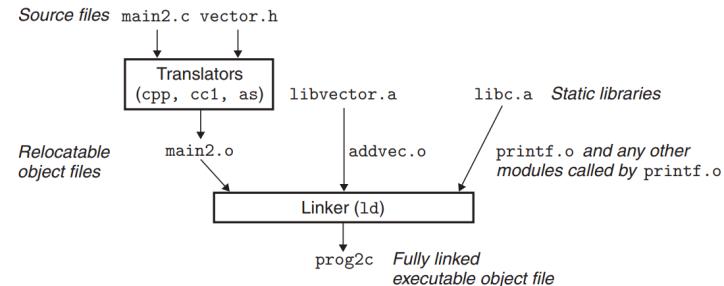


Figure 7.8 Linking with static libraries.

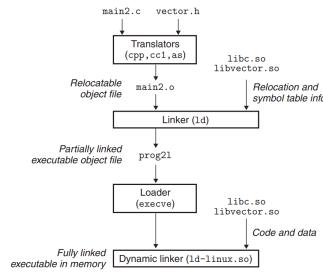
- 注意：`.o` 与 `.a` 的不同
 - `.o` 在链接算法中会直接重定位加入汇编而 `.a` 只会加入其被引用的符号

流程梳理

■ 动态链接

- dynamic linker:
 - (1) 将动态库加载到虚拟内存空间中;
 - (2) 对可执行文件中的 reference 进行重定位
(修改 .text 和 .data)
- PIC: (编译) 将源文件编译为位置无关代码, 加载到不同进程的不同地址时依然能够正确执行
- GOT: (链接) 为每个全局变量和外部函数保留一个条目, 并在运行时将实际地址填入
- PLT: (加载) 对于动态库中的函数调用, PLT 会保存函数的入口地址

Figure 7.16
Dynamic linking with shared libraries.



1. 加载可执行文件和共享库, 并启动动态链接器。
2. 解析依赖关系, 加载所有依赖的共享库。
3. 准备 GOT 和 PLT 表, 确保符号可以被动态解析。
4. 符号解析和重定位, 在初次加载时完成大部分符号的解析。
5. 延迟绑定, 当第一次调用共享库的函数时, 动态链接器解析符号并绑定。
6. 完成初始化并跳转到入口点, 程序开始正常执行。

位置无关代码 (PIC)

- ▶ 上述过程的问题：仍然浪费内存空间
 - ▶ .so 也由 relocatable obj file 得到 gcc a.o -shared -o a.so
 - ▶ 也有 .text 和 .data，也有可能引用其他 .so 里的函数 / 变量
 - ▶ 如果 .so 也会按上述过程被 dynamic linker 重定位，那么每个进程里的 .so 依然是一份拷贝而非共享
- ▶ 解决方案：PIC（位置无关代码，Position-Independent Code）
 - ▶ 保证 .so 的 .text 不需要重定位
 - ▶ 物理内存中只需要一份 liba.so (举例) 的 .text，所有引用 liba.so 的程序可以以只读的方式将自己虚拟内存中的一段映射到这段代码，实现共享
- ▶ 共享库 .so 必须是 PIC
 - ▶ gcc -c -fPIC a.c -o a.o
 - ▶ gcc a.o -shared -o liba.so
- ▶ **-fPIC 生成的共享库在加载时作为一个整体被加载进内存中，内部相对位置不变**

GOT

- ▶ GOT: 全局偏移量表 (Global Offset Table)
 - ▶ 每个条目为 8 字节地址
 - ▶ 存储用户定义的符号
- ▶ 调用数据时, 通过将数据地址存储在 GOT 条目表中的形式达到无需修改代码段的目的
- ▶ 动态链接器在加载过程中只对 GOT 进行重定位, 不需要修改 .text

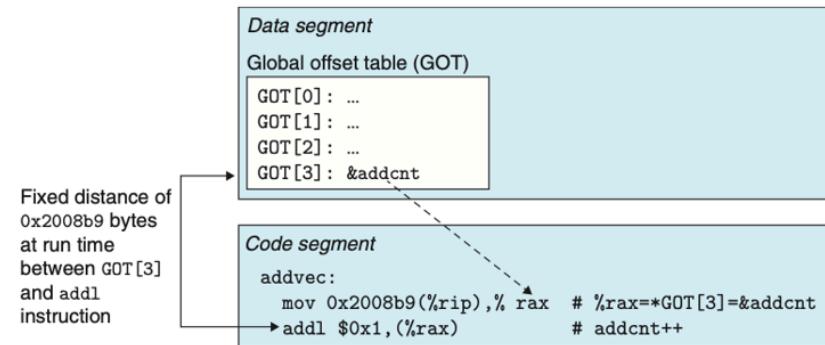
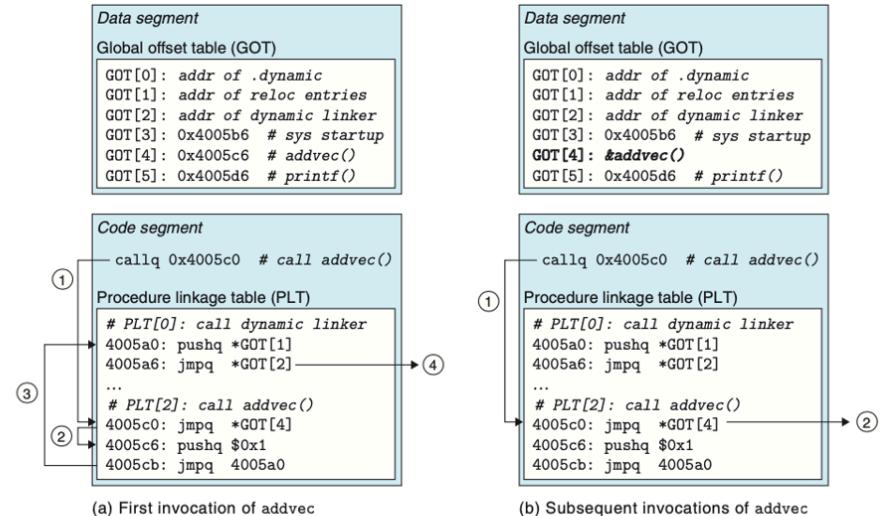


Figure 7.18 Using the GOT to reference a global variable. The addvec routine in libvector.so references addcnt indirectly through the GOT for libvector.so.

PLT

- ▶ PLT: 过程链接表 (Procedure-Linkage Table)
 - ▶ 每个条目为 16 字节机器码
 - ▶ PLT[0] 调用动态链接器
 - ▶ PLT[1] 调用系统启动函数 `__libc_start_main()`
 - ▶ PLT[2] 开始, 调用用户定义的函数
 - ▶ 有 PLT 时, GOT 用法有一定变化
 - ▶ GOT[0] 和 GOT[1] 存储动态链接器需要的信息
 - ▶ GOT[2] 存储动态链接器的入口
 - ▶ GOT[3] 开始, 存储函数调用对应的 PLT 条目位置, 其中 GOT[3] 对应系统启动函数 `__libc_start_main()`
- ▶ 第一次调用时, 函数入口为 PLT 条目的起始位置, 根据汇编跳转至 GOT 表初始所指向的该句汇编的下一句汇编
- ▶ PLT 将该函数 ID 传入动态链接器并调用其修改 GOT 条目, 使其指向函数入口的真正地址
- ▶ 以后调用时, 经过第一个 GOT 跳转时直接进入函数
- ▶ PLT + GOT 实现 lazy binding, 运行时再确定函数的地址
- ▶ 现在需要加 `-z lazy` 选项才能观测到这个现象



Homework Review

HW6

学号	HW-分数	HW-问题
2300012932	10	
2300012935	9	提交格式问题
2300012950	10	
2300012951	10	
2300013083	10	
2300013115	10	
2300013158	10	
2300013201	10	
2300013222	10	
2300013230	10	
2300013272	10	
2300017788	10	
2300094610	10	

Exercises

E1

9. 以下关于 Linux 系统上处理可执行文件的工具的说法错误的是：
- A 使用 objdump 反汇编 .text 节的机器码。
 - B 使用 readelf 读取文件的节头部表 (section header) 和程序头部表 (program header)。
 - C 使用 ls 查询文件是文本文件还是二进制文件。
 - D 使用 gdb 加载可执行文件、设置断点，然后单步调试运行。

E1

9. 以下关于 Linux 系统上处理可执行文件的工具的说法错误的是：
- A 使用 objdump 反汇编 .text 节的机器码。
 - B 使用 readelf 读取文件的节头部表 (section header) 和程序头部表 (program header)。
 - C 使用 ls 查询文件是文本文件还是二进制文件。
 - D 使用 gdb 加载可执行文件、设置断点，然后单步调试运行。

C 错误 ↶

解析： ↶

- A 正确。objdump -dj .text [file] ↶
- B 正确。节头部表：readelf -S [file] 程序头部表：readelf -l [file] ↶
- C 错误。ls 只是取得文件的元数据，这与文件内容无关，而 linux 文件元数据中也不包含对 binary 或者 text 编码属性的描述。其他诸如 grep 和 file 的工具使用 heuristics 确定文件的类型。 ↶
- D 正确。gdb 可以支持单步调试。 ↶

E2

2. 对如下两个 C 程序，用 gcc 生成对应的.o 模块，再链接在一起得到 a.out。则下列说法正确的是：←

```
// main.c←
#include <stdio.h>←
static int a;←
int main() {←
    int *func();←
    printf("%d\n", func() - &a);←
    return 0;←
}
```

```
// util.c←
int a = 0;←
int *func() {←
    return &a;←
}
```

- A. 在 main.o 中，符号 a 位于 COM 伪节←
- B. 在 util.o 中，符号 a 位于 COM 伪节←
- C. 无论怎样链接和运行 a.out，a.out 输出的结果都一样，但必不为 0←
- D. 以上说法都不正确←

E2

D 正确 ↵

A 错。它位于 .bss 节。 ↵

B 错。它位于 .bss 节。 ↵

C 错。注意 ld 时文件顺序的交换会改变 a.out 中两个符号的相对偏移。 ↵
于是 D 正确。 ↵

E3

8. 文件 f1.c 和 f2.c 的 C 源代码如下图所示：

```
//f1.c
#include<stdio.h>
extern float x;
extern void f();
int main()
{
    f();
    printf("%d\n", (int)x);
    return 0;
}
```

```
//f2.c
int x = 0;
void f()
{
    x++;
}
```

已知这两个文件在同一个目录下，在该目录下用“gcc -Og -o f f1.c f2.c”编译，然后用“./f”运行，这个过程中会出现的情况是：

- A. 编译错误
- B. 输出0
- C. 输出1
- D. 链接错误

E3

8. B。请注意编译器看不到两边 `x` 类型的不同，因此不会报错。链接器会将 `x` 初始化为 0，在 `f1.c` 中编译器生成的代码视其为浮点而在另一边视其为整数，所以自增结果是一个很小的浮点数，转为整数是 0。

E4

6. 在链接时，对于下列哪些符号需要进行重定位？

- (1) 不同 C 语言源文件中定义的函数
 - (2) 同一 C 语言源文件中定义的全局变量
 - (3) 同一函数中定义时不带 static 的变量
 - (4) 同一函数中定义时带有 static 的变量
- A. (1) (3) B. (2) (4) C. (1) (2) (4) D. (1) (2) (3) (4)

E4

6. 在链接时，对于下列哪些符号需要进行重定位？

- (1) 不同 C 语言源文件中定义的函数
- (2) 同一 C 语言源文件中定义的全局变量
- (3) 同一函数中定义时不带 static 的变量
- (4) 同一函数中定义时带有 static 的变量

A. (1) (3) B. (2) (4) C. (1) (2) (4) D. (1) (2) (3) (4)

6. C。实际上对符号谈重定位是不正确的，应当对引用谈重定位，例如全局变量如果初始化为值且不使用，则不需要重定位。需要留意，如果是同一 C 语言源文件中定义的函数则一般不需要重定位。

E5

4. 下列关于链接技术的描述，错误的是（ ）
- A. 在 Linux 系统中，对程序中全局符号的不恰当定义，会在链接时刻进行报告。
 - B. 在使用 Linux 的默认链接器时，如果有多个弱符号同名，那么会从这些弱符号中任意选择一个占用空间最大的符号。
 - C. 编译时打桩 (interpositioning) 需要能够访问程序的源代码，链接时打桩需要能够访问程序的可重定位对象文件，运行时打桩只需要能够访问可执行目标文件。
 - D. 链接器的两个主要任务是符号解析和重定位。符号解析将目标文件中的全局符号都绑定到唯一的定义，重定位确定每个符号的最终内存地址，并修改对那些目标的引用。

E5

4. 下列关于链接技术的描述，错误的是（ ）
- A. 在 Linux 系统中，对程序中全局符号的不恰当定义，会在链接时刻进行报告。
 - B. 在使用 Linux 的默认链接器时，如果有多个弱符号同名，那么会从这些弱符号中任意选择一个占用空间最大的符号。
 - C. 编译时打桩 (interpositioning) 需要能够访问程序的源代码，链接时打桩需要能够访问程序的可重定位对象文件，运行时打桩只需要能够访问可执行目标文件。
 - D. 链接器的两个主要任务是符号解析和重定位。符号解析将目标文件中的全局符号都绑定到唯一的定义，重定位确定每个符号的最终内存地址，并修改对那些目标的引用。

答案：选 A。参考机械工业出版社第三版中文教材，四个选项分别对应 P464 第 5 自然段、P471 页文末、P494 页文末、P496 文末的文字。其中选项 B 略有调整，增加了“占用空间最大的”，是 Linux binutils 中链接器实现的原则，仍是正确选项。
A 错误是因为全局符号的不恰当定义并不总会报警（甚至不报告 warning）。

E6

7. C 源文件 f1.c 和 f2.c 的代码分别如下所示，编译链接生成可执行文件后执行，输出结果为（ ）
- A. 100 B. 200
C. 201 D. 链接错误

```
// f1.c
#include <stdio.h>
static int var = 100;
int main(void)
{
    extern int var ;
    extern void f() ;
    f() ;
    printf("%d\n", var) ;
    return 0;
}
```

```
//f2.c
int var = 200;

void f()
{
    var++;
}
```

E6

7. C 源文件 f1.c 和 f2.c 的代码分别如下所示，编译链接生成可执行文件后执行，输出结果为（ ）
- A. 100 B. 200
C. 201 D. 链接错误

```
// f1.c
#include <stdio.h>
static int var = 100;
int main(void)
{
    extern int var ;
    extern void f() ;
    f() ;
    printf("%d\n", var) ;
    return 0;
}
```

```
//f2.c
int var = 200;

void f()
{
    var++;
}
```

答案: A(f1.c 中的 `extern int var`, 会搜索 f1.c 之前定义过的全局变量, 因为之前已经已经有了 var 的定义, 所以 `printf` 会打印该 var 的值 100)

E7

8. C 源文件 m1.c 和 m2.c 的代码分别如下所示，编译链接生成可执行文件后执行，结果最可能为（ ）

```
$ gcc -o a.out m2.c m1.c ; ./a.out
```

0x1083020

- A. 0x1083018, 0x108301c B. 0x1083028, 0x1083024
C. 0x1083024, 0x1083028 D. 0x108301c, 0x1083018

// m1.c	//m2.c
#include <stdio.h>	int a4 = 10 ;
int a1 ;	int main()
int a2 = 2 ;	{
extern int a4 ;	extern void hello() ;
void hello()	hello() ;
{	return 0 ;
printf("%p ", &a1);	}

printf("%p ", &a2);	
printf("%p\n", &a4);	
}	

E7

答案: D (a2 和 a4 在.data 中, a1 在.bss 中, 按照布局, .data 地址比.bss 要小, 又由于 m2.c 的编译先于 m1.c, 故选 D 不选 A)

E9

3. 以下程序可以使用 `gcc dl.c -ldl` 编译并正常运行。如果缺少了 `-ldl` 标志，链接时会报错 `undefined reference to `dlopen'`。基于你对于动态链接的理解，请分析出以下说法中不正确的一项。←

```
// dl.c←
#include <dlfcn.h>←
←
const char *path = "/lib/libc.so";←
int (*printf) (const char *x);←
←
int main() {←
    // 加载共享库←
    void *handle = dlopen(path, RTLD_NOW);←
    // 解析符号 "printf" 并返回地址←
    printf = dlsym(handle, "printf");←
    // 调用←
    printf("2022 is coming!\n");←
    // 关闭共享库←
    dlclose(handle);←
}←
```

- A. 该机器上 `libdl.so` 模块中包含符号名为 `dlopen` 的动态链接符号表条目←
- B. 在 `a.out` 文件中包含 `printf` 的 PLT 条目和相应的 GOT 条目←
- C. 在 `a.out` 文件中包含 `dlopen` 的 PLT 条目和相应的 GOT 条目←
- D. 如果使用 `gcc -ldl dl.c` 编译程序，则会在链接时发生同样错误←

E9

B 错误。 ↵

解析： ↵

省略 `-ldl` 标志报错，表明 `ld` 默认不会包含 `libdl.so` (与之对比，`libc.so` 默认包含)，并且 `dlopen` 的定义来自于该共享库。 ↵

A 正确。`.dynsym` 含有一个符号表条目。格式形如 ↵

`000000000001390 g DF .text 0000000000000085 GLIBC_2.2.5 dlopen` ↵

“动态链接符号表”的描述是准确的，也不影响理解。 ↵

B 错误，`printf` 只是一个未初始化的全局变量。它不是内置的 `printf` 函数。 ↵

C 正确。默认程序动态绑定 `dlopen` 到共享库，它需要自己的 PLT 表和 GOT 表。 ↵

D 正确。`gcc` 按照命令行顺序解析。不管是动态库还是静态库只解析当前已经被引用的符号 (这一点容易推断，否则没有必要建立专门的库文件格式了。因此没有补充在题目中交代动态链接符号解析的规则。)，所以 `-ldl` 放在第一个位置没有任何效果。最后会在链接阶段产生 `dlopen`、`dlsym` 或者 `dlclose` 未能解析的错误。 ↵

额外说明，`libc.so` 和 `libdl.so` 在实际系统上可能会带上版本号，路径名一般也更复杂。这里为了出题，做了合适的简化。 ↵

E10

7. 下列关于链接的描述，正确的是：

- A. 打桩 (interpositioning)机制需要能够访问程序的源代码。
- B. 链接器生成的文件不再有ABS、UNDEF 和COMMON 伪节。
- C. 对于Glibc和GCC生成的程序，其入口点位于 `_libc_start_main` 函数的第一条指令处。
- D. 用户使用GCC进行共享库的编译必须使用-fpic选项指示生成位置无关代码。

E10

7. 下列关于链接的描述，正确的是：

- A. 打桩 (interpositioning)机制需要能够访问程序的源代码。
 - B. 链接器生成的文件不再有ABS、UNDEF 和COMMON 伪节。
 - C. 对于GlibC和GCC生成的程序，其入口点位于 `_libc_start_main` 函数的第一条指令处。
 - D. 用户使用GCC进行共享库的编译必须使用-fpic选项指示生成位置无关代码。
7. D。只有编译时打桩需要访问源代码，A 错误。B，可执行目标文件中仍然会有 ABS 节（文件名）。程序的入口是 `_start`，后者跳到题干所说的位置，C 错误。D 是正确的（书 P489 原话），但是 -fPIC（大写）选项更好。

Notices

THANKS

Made by WalkerCH

changxinhai@stu.pku.edu.cn

Reference: [Weicheng Lin]'s presentation.

Reference: [Arthals]'s templates and content.

