链接

计算机系统基础

任课教师：
龚奕利
yiligong@whu.edu.cn

主要内容

- 链接
- 案例研究：库打桩/插桩（Library interpositioning）

C程序示例

```c
int sum(int *a, int n);

int array[2] = {1, 2};

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

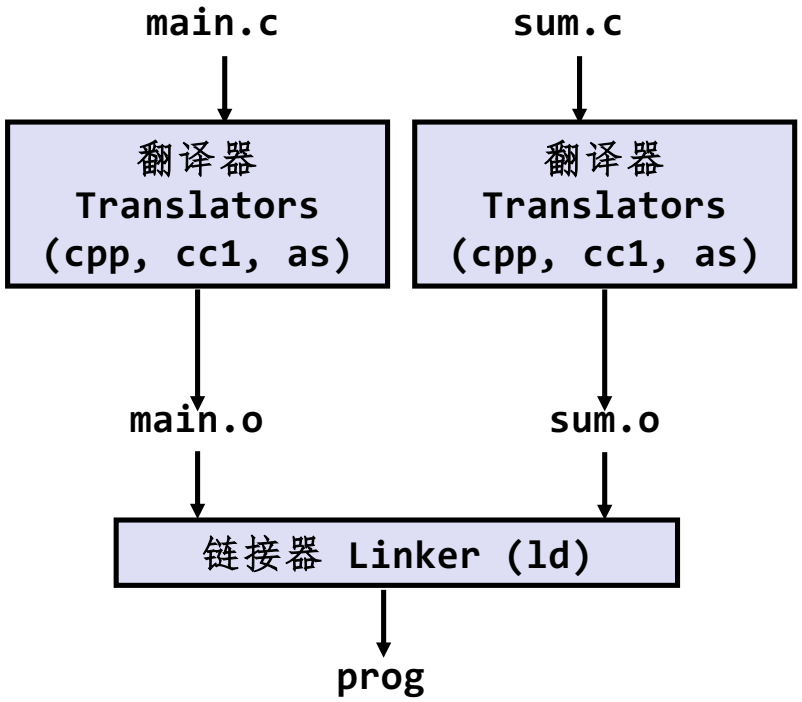
main.c

```c
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

静态链接（Static Linking）

- 用编译器翻译和链接程序：
  - linux> *gcc -Og -o prog main.c sum.c*
  - linux> *./prog*



静态链接

- 用编译器翻译和链接程序：
  - linux> *gcc -Og -o prog main.c sum.c*
  - linux> *./prog*

`-Og`

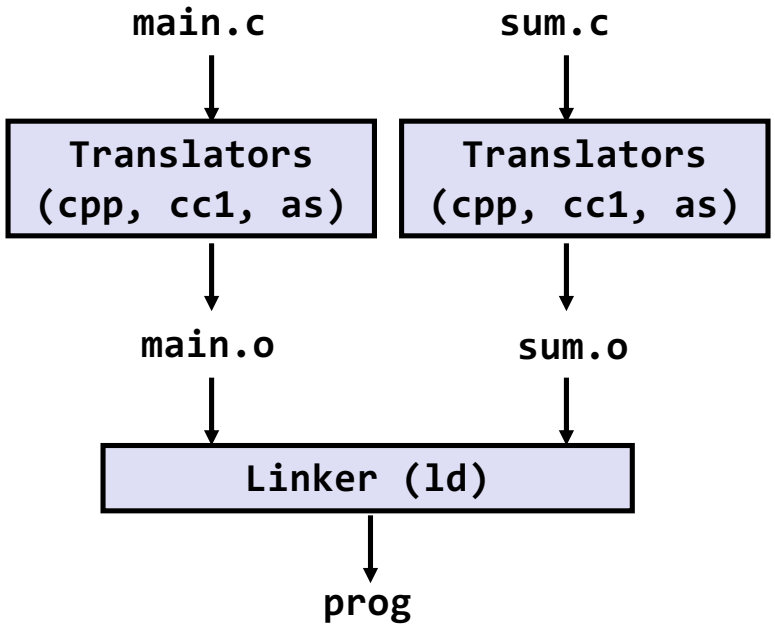
Optimize debugging experience. `-Og` should be the optimization level of choice for the standard edit-compile-debug cycle, offering a reasonable level of optimization while maintaining fast compilation and a good debugging experience. It is a better choice than `-O0` for producing debuggable code because some compiler passes that collect debug information are disabled at `-O0`.

Like `-O0`, `-Og` completely disables a number of optimization passes so that individual options controlling them have no effect. Otherwise `-Og` enables all `-O1` optimization flags except for those that may interfere with debugging:

```
-fbranch-count-reg  -fdelayed-branch
-fdse  -fif-conversion  -fif-conversion2
-finline-functions-called-once
-fmove-loop-invariants  -fmove-loop-stores  -fssa-phiopt
-ftree-bit-ccp  -ftree-dse  -ftree-pta  -ftree-sra
```

静态链接

- 用编译器翻译和链接程序：
  - linux> *gcc -Og -o prog main.c sum.c*
  - linux> *./prog*



源文件

预处理 **pre-processing (.i)**
编译 **compiling (.s)**
汇编 **assembling (.o)**

分别编译得到的
可重定位目标文件（**relocatable object files**）

完全链接的
可执行目标文件（**executable object file**）
（包含**main.c**和**sum.c**中定义的所有函数的代码和数据）

为什么要用链接器？

- **理由1：模块化**
  - 可以用一系列小的源文件来完成程序的功能，而不是一个超级大的文件
  - 可以将公共的函数做成库（后面会谈到）
    - 例如，数学库，标准C库

为什么要用链接器？（续）

- **理由2：效率**
  - 时间：可以分开编译
    - 修改一个源文件，编译，再重链接
    - 不需要重编译其他源文件
  - 空间：库
    - 公共的函数可以打包进一个文件里 ... ...
    - 而可执行文件和运行时内存镜像中只包含它们实际使用的函数的代码

链接器做什么?

- 第一步：符号解析 **symbol resolution**
  - 程序定义和引用符号 *symbols* （全局变量和函数）
    - `void swap() {…}  /* define symbol swap */`
    - `swap();  /* reference symbol swap */`
    - `int *xp = &x;  /* define symbol xp, reference x */`
  - 符号定义（由汇编器）存储在目标文件的符号表 *(symbol table)* 中
    - 符号表是一个 `structs` 的数组
    - 每个条目包括符号的名字、大小和位置
  - 符号解析步骤中，链接器将每个符号引用与刚好一个符号定义关联起来

链接器做什么？（续）

- 第二步：重定位 **relocation**
  - 将分开的代码段和数据段合并成统一的段
  - 将.o 文件中符号的相对位置重定位到它们在可执行文件中最后的、绝对内存位置
  - 更新这些符号的引用，使它们反映对应的新位置

让我们再仔细看看这两个步骤 ... ...

三类目标文件（模块）

- **可重定位目标文件 (.o 文件)**
  - 包含二进制代码和数据，其形式使得它可以与其他可重定位目标文件合并起来，形成一个可执行目标文件
    - 每个 .o 文件由一个源文件 (.c) 产生得到
- **可执行目标文件 (a.out 文件)**
  - 包含二进制代码和数据，其形式可使得它以被直接复制到内存并执行
- **共享目标文件 (.so 文件)**
  - 一种特殊类型的可重定位目标文件，可以在加载时或运行时被动态地加载进内存并链接
  - 在Windows上叫作动态链接库（Dynamic Link Libraries (DLLs)）

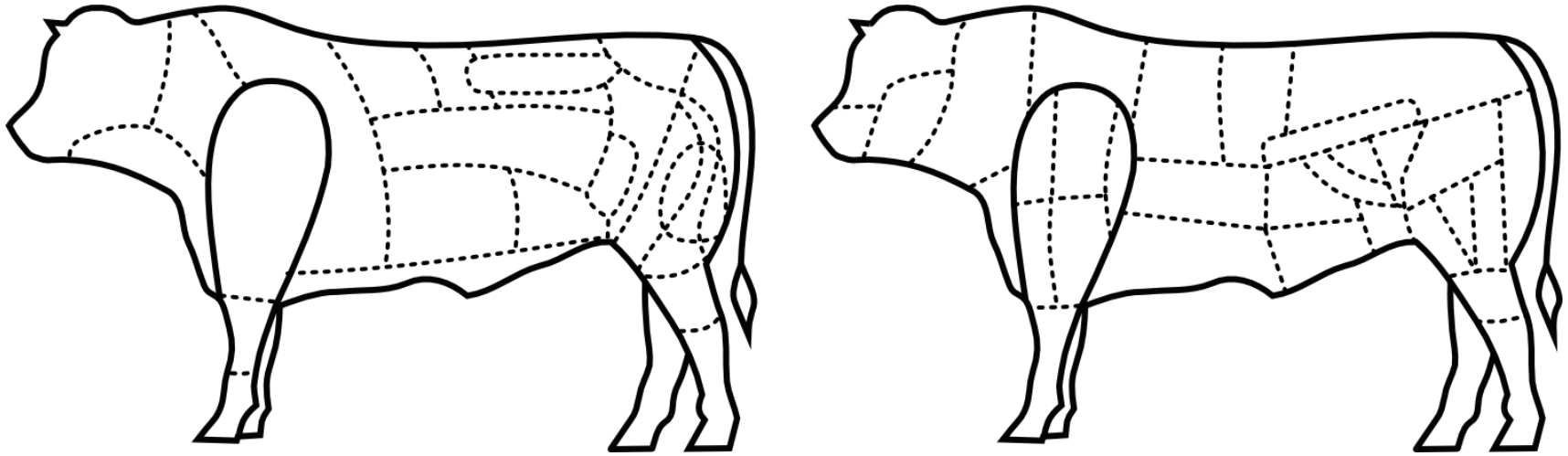
ELF 格式（Executable and Linkable Format）

- 目标文件的标准二进制格式
- 下面这些文件都采用统一的格式
  - 可重定位目标文件 (.o),
  - 可执行目标文件 (a.out)
  - 共享目标文件 (.so)
  - coredumps
- 通用名字：**ELF** 二进制文件

- Windows上使用PE（Portable Executable）格式
- Mac OS-X上使用Mach-O（Mach Object）格式

文件与文件分析器/解释器

- 文件本身没有特殊含义
- 文件的含义——它的类型、内容等——都是由分析器（parser）或者解释器（interpreter）给予的

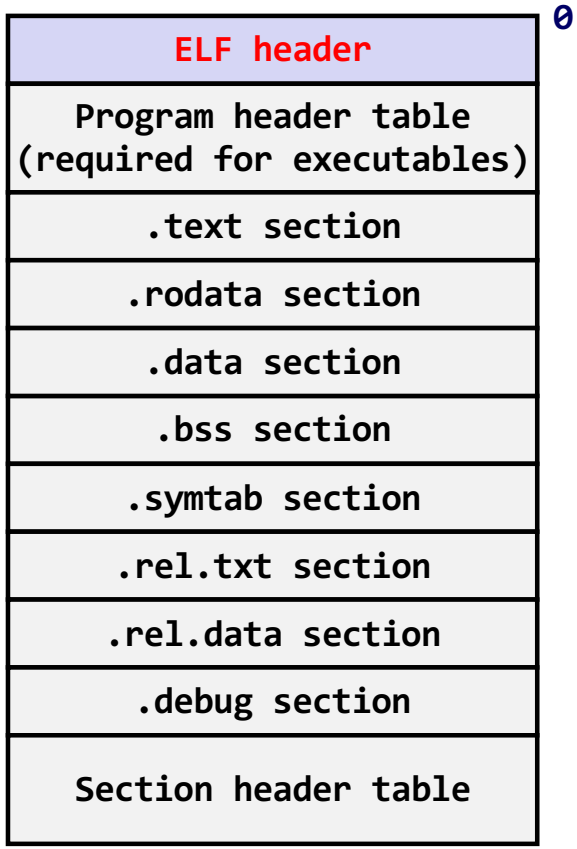


Brazilian and French
beef cuts

ELF 目标文件格式

- ELF 头部
  - 字长、字节序、文件类型 (.o, exec, .so)、机器类型等

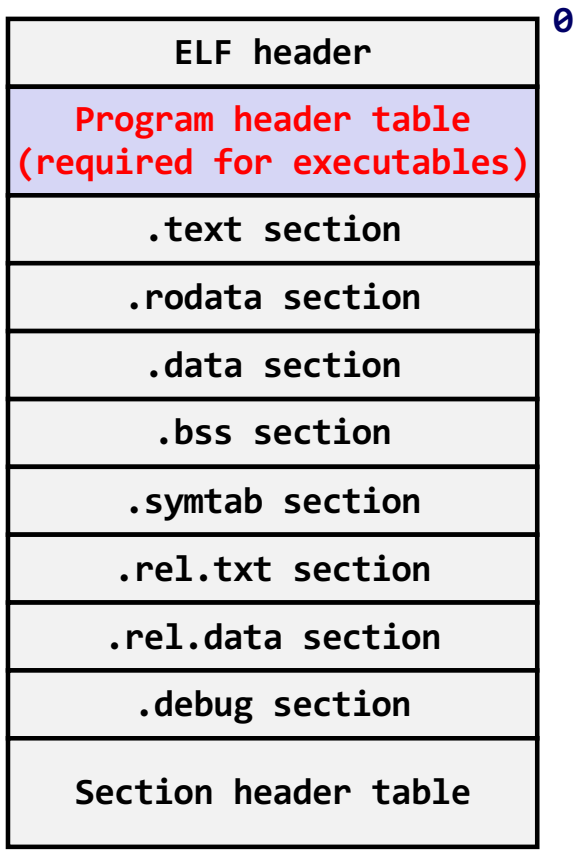
```
root@cg:~/Desktop/test# readelf -h test1
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x540
  Start of program headers:          64 (bytes into file)
  Start of section headers:          6448 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         29
  Section header string table index: 28
```



ELF 目标文件格式

- 程序头部表 **program header table**
  - 页大小、虚拟地址内存段 (节, sections)、段大小
  - 程序加载时，OS使用这些信息来对可执行程序的内容做内存映射

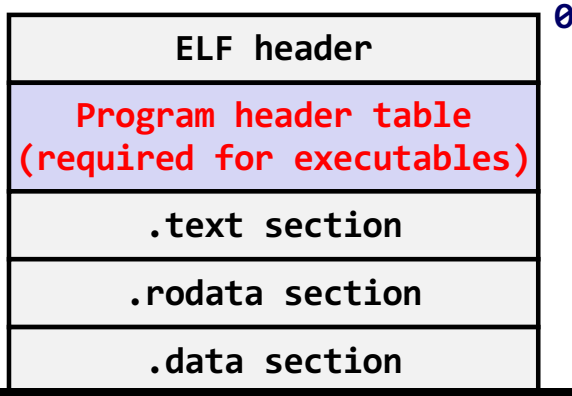
```
root@cg:~/Desktop# readelf -l gdb

Elf file type is DYN (Shared object file)
Entry point 0xa6d30
There are 10 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x0000000000000230 0x0000000000000230  R      0x8
  INTERP         0x0000000000000270 0x0000000000000270 0x0000000000000270
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000650554 0x0000000000650554  R E    0x200000
  LOAD           0x00000000006514a8 0x00000000008514a8 0x00000000008514a8
                 0x00000000000f22f0 0x0000000000118bb0  RW     0x200000
  DYNAMIC        0x000000000072db58 0x000000000092db58 0x000000000092db58
                 0x00000000000002d0 0x00000000000002d0  RW     0x8
  NOTE           0x000000000000028c 0x000000000000028c 0x000000000000028c
                 0x0000000000000044 0x0000000000000044  R      0x4
  TLS            0x00000000006514a8 0x00000000008514a8 0x00000000008514a8
                 0x0000000000000000 0x0000000000000010  R      0x8
  GNU_EH_FRAME   0x0000000000593204 0x0000000000593204 0x0000000000593204
                 0x000000000001a2a4 0x000000000001a2a4  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x00000000006514a8 0x00000000008514a8 0x00000000008514a8
                 0x000000000000ddb58 0x000000000000ddb58  R      0x1
```



ELF 目标文件格式

- 程序头部表 **program header table**
  - 页大小、虚拟地址内存段 (节, sections)、段大小
  - 程序加载时，OS使用这些信息来对可执行程序的内容做内存映射

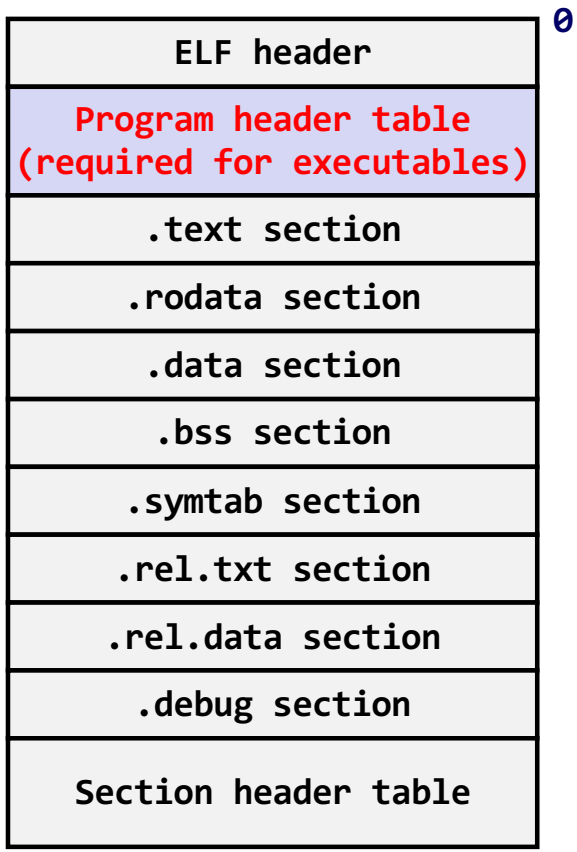
```
root@cg:~/Desktop# readelf -l gdb

Elf file type is DYN (Shared object file)
Entry point 0xa6d30
There are 10 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x0000000000000230 0x0000000000000230  R      0x8
  INTERP         0x0000000000000270 0x0000000000000270 0x0000000000000270
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000a58 0x0000000000000a58  R E    0x200000
  LOAD           0x0000000000000e10 0x0000000000200e10 0x0000000000200e10
                 0x0000000000000328 0x0000000000000330  RW     0x200000
  DYNAMIC        0x0000000000000e28 0x0000000000200e28 0x0000000000200e28
                 0x00000000000001d0 0x00000000000001d0  RW     0x8
  NOTE           0x0000000000000298 0x0000000000000298 0x0000000000000298
                 0x0000000000000044 0x0000000000000044  R      0x4
  TLS            0x0000000000000e10 0x0000000000200e10 0x0000000000200e10
                 0x0000000000000000 0x0000000000000018  R      0x8
  GNU_EH_FRAME   0x0000000000000a08 0x0000000000000a08 0x0000000000000a08
                 0x0000000000000034 0x0000000000000034  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000000e10 0x0000000000200e10 0x0000000000200e10
                 0x00000000000001f0 0x00000000000001f0  R      0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu
.version .gnu.version_r .rela.dyn .rela.plt .init .plt .plt.got .text .fini .roda
ta .stapsdt.base .eh_frame_hdr .eh_frame .gcc_except_table
   03     .init_array .fini_array .data.rel.ro .dynamic .got .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .tbss
   07     .eh_frame_hdr
   08
   09     .init_array .fini_array .data.rel.ro .dynamic .got
```



ELF 目标文件格式

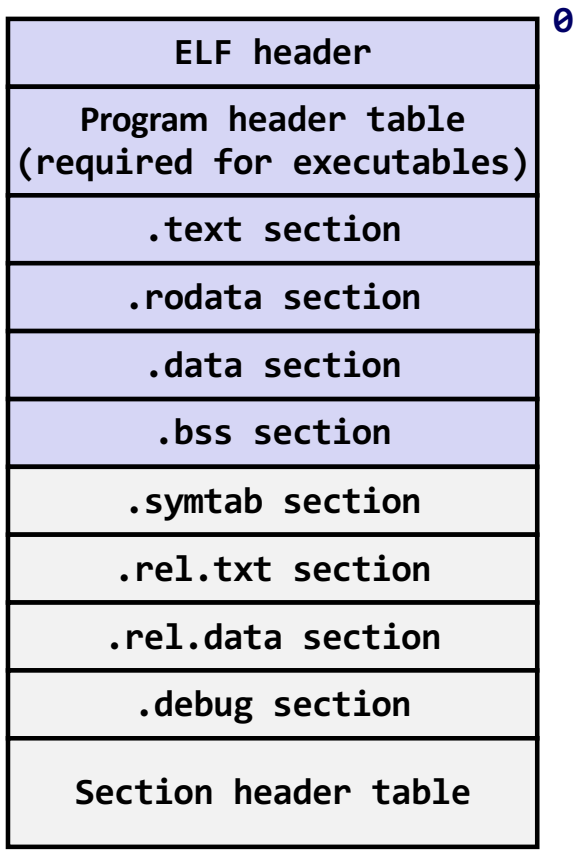
- 程序头部表 **program header table**
  - 页大小、虚拟地址内存段 (节，sections)、段大小
  - 程序加载时，OS使用这些信息来对可执行程序的内容做内存映射

```
root@cg:~/Desktop# readelf -l main.o

There are no program headers in this file.
```



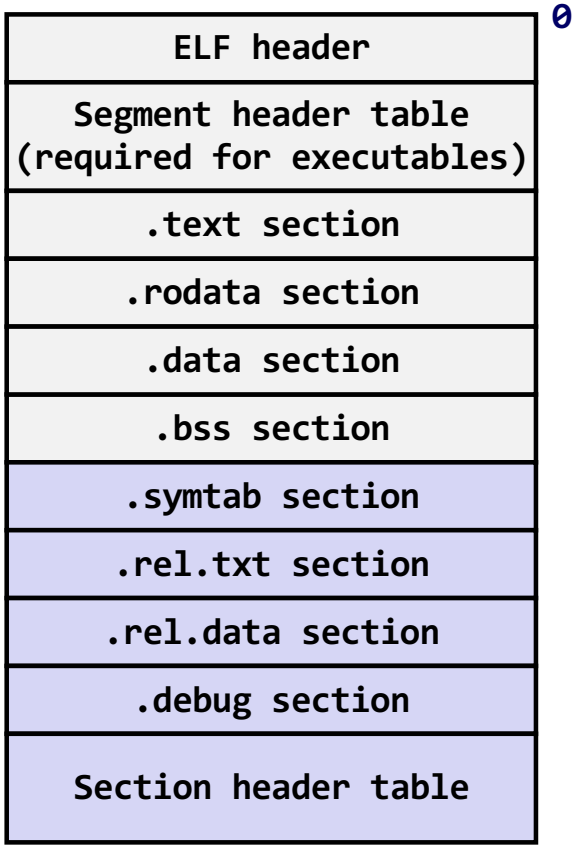
ELF 目标文件格式

- .text 节
  - 代码节，可读/可执行
- .rodata 节
  - 只读数据：跳转表 ... ...
- .data 节
  - 初始化了的全局变量，可读/可写
- .bss 节
  - 未初始化或初始化为0的全局和静态变量
  - 可读/可写
  - “Block Started by Symbol”
  - “Better Save Space”
  - 具有节头部，但是实际不占用空间



ELF 目标文件格式（续）

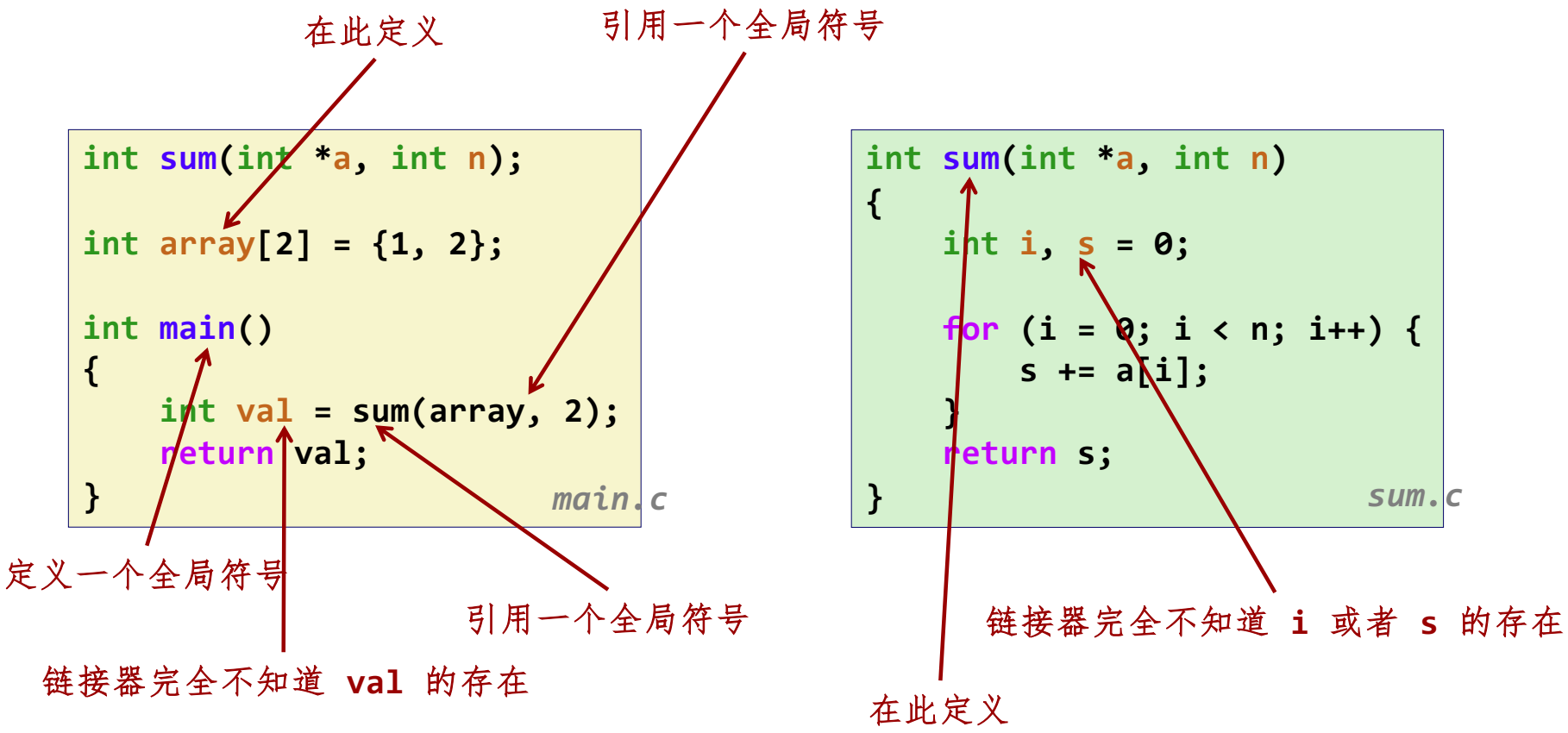
- **.symtab 节**
  - 符号表
  - 过程和静态变量的名字
  - 节名字和位置
- **.rel.text 节**
  - .text 节的重定位信息
  - 可执行文件中待修改指令的地址
  - 待修改的指令
- **.rel.data 节**
  - .data 节的重定位信息
  - 合并的可执行文件中待修改的指针数据的地址
- **.debug 节**
  - 有关符号调试的信息 **(gcc -g)**
- **节头部表 section header table**
  - 每个节的偏移量和大小



链接器符号

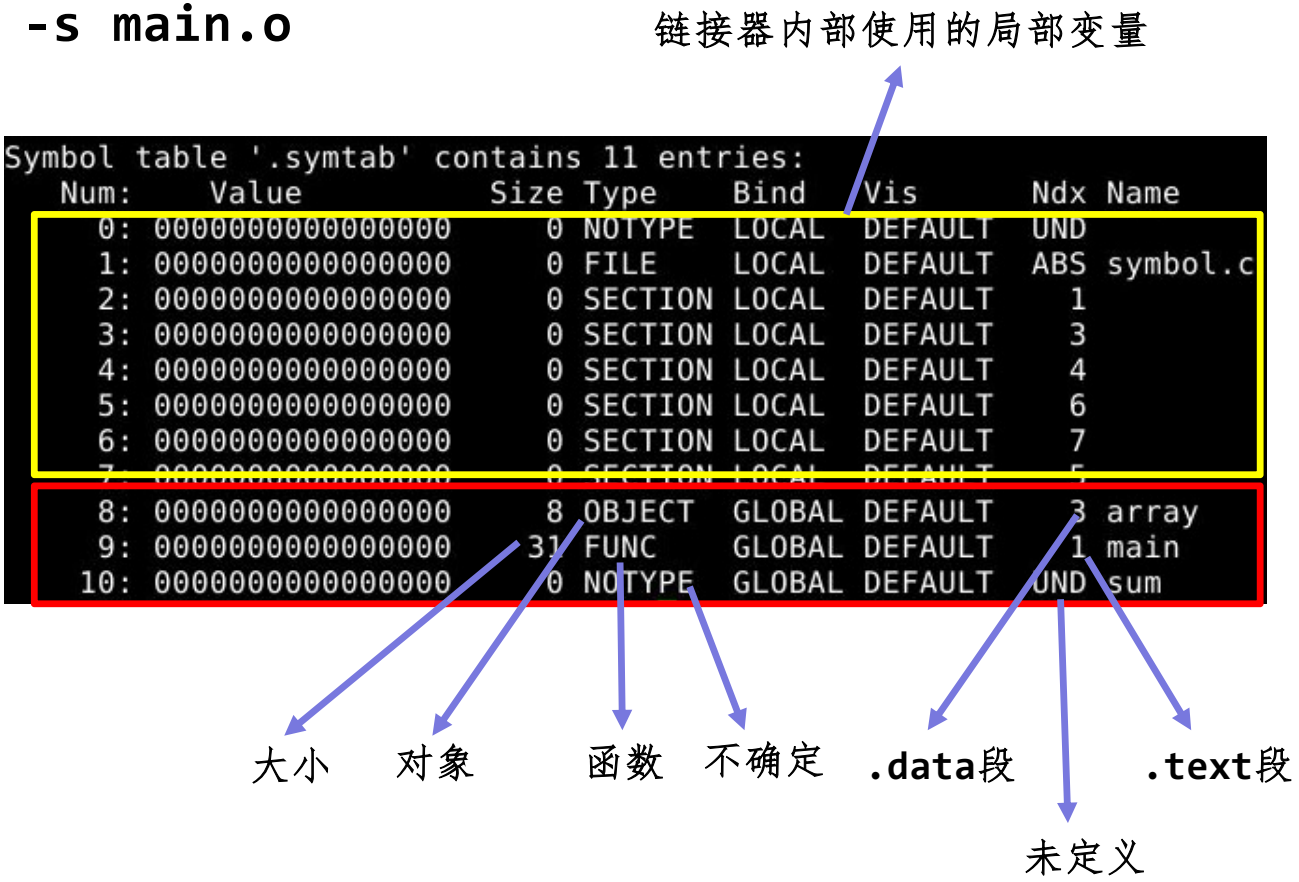
- 全局符号
  - 由模块 m 定义，能够被其他模块引用的符号
  - 例如：非静态函数和非静态全局变量
- 外部符号
  - 模块 m 引用的由其他模块定义的全局符号
- 局部符号
  - 由模块 m 独占地定义和引用的符号
  - 例如：以 static 属性定义的 C 函数和全局变量
  - 链接器局部符号不是程序的局部变量

步骤1：符号解析


在此定义
引用一个全局符号
int sum(int *a, int n);
int array[2] = {1, 2};
int main()
{
int val = sum(array, 2);
return val;
}
main.c
int sum(int *a, int n)
{
int i, s = 0;
for (i = 0; i < n; i++) {
s += a[i];
}
return s;
}
sum.c
定义一个全局符号
引用一个全局符号
链接器完全不知道 val 的存在
链接器完全不知道 i 或者 s 的存在
在此定义

查看符号表

- **readelf -s main.o**



查看sections

```
root@cg:~/Desktop# readelf -S main.o
There are 12 section headers, starting at offset 0x2d0:

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
       0000000000000021  0000000000000000  AX       0     0     1
  [ 2] .rela.text        RELA             0000000000000000  00000228
       0000000000000030  0000000000000018   I       9     1     8
  [ 3] .data             PROGBITS         0000000000000000  00000068
       0000000000000008  0000000000000000  WA       0     0     8
  [ 4] .bss              NOBITS           0000000000000000  00000070
       0000000000000000  0000000000000000  WA       0     0     1
  [ 5] .comment          PROGBITS         0000000000000000  00000070
       000000000000002a  0000000000000001  MS       0     0     1
  [ 6] .note.GNU-stack   PROGBITS         0000000000000000  0000009a
       0000000000000000  0000000000000000           0     0     1
  [ 7] .eh_frame         PROGBITS         0000000000000000  000000a0
       0000000000000038  0000000000000000   A       0     0     8
  [ 8] .rela.eh_frame    RELA             0000000000000000  00000258
       0000000000000018  0000000000000018   I       9     7     8
  [ 9] .symtab           SYMTAB           0000000000000000  000000d8
       0000000000000120  0000000000000018          10     8     8
  [10] .strtab           STRTAB           0000000000000000  000001f8
       000000000000002d  0000000000000000           0     0     1
  [11] .shstrtab         STRTAB           0000000000000000  00000270
       0000000000000059  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
```

练习题7.1

这个题目针对图7.5中的 m.o 和 swap.o 模块。对于每个在 swap.o 中定义或引用的符号，请指出它是否在模块 swap.o 中的 .symtab 节中有一个符号表条目。如果是，请指出定义该符号的模块（swap.o 或者 m.o）、符号类型（局部或全局）以及它在模块中被分配到的节（.text、.data、.bss或COMMON）。

(a) m.c — code/link/m.c

```
1   void swap();
2
3   int buf[2] = {1, 2};
4
5   int main()
6   {
7       swap();
8       return 0;
9   }
```

(b) swap.c — code/link/swap.c

```
1   extern int buf[];
2
3   int *bufp0 = &buf[0];
4   int *bufp1;
5
6   void swap()
7   {
8       int temp;
9
10      bufp1 = &buf[1];
11      temp = *bufp0;
12      *bufp0 = *bufp1;
13      *bufp1 = temp;
14  }
```

Figure 7.5 **Example program for Practice Problem 7.1.**

符号	.symtab条目？	符号类型	在哪个模块中定义	节
buf				
bufp0				
bufp1				
swap				
temp				

练习题7.1

这个题目针对图7.5中的 m.o 和 swap.o 模块。对于每个在 swap.o 中定义或引用的符号，请指出它是否在模块 swap.o 中的 .symtab 节中有一个符号表条目。如果是，请指出定义该符号的模块（swap.o 或者 m.o）、符号类型（局部或全局）以及它在模块中被分配到的节（.text、.data、.bss或COMMON）。

(a) m.c — *code/link/m.c*

```
1   void swap();
2
3   int buf[2] = {1, 2};
4
5   int main()
6   {
7       swap();
8       return 0;
9   }
```

code/link/m.c

(b) swap.c — *code/link/swap.c*

```
1   extern int buf[];
2
3   int *bufp0 = &buf[0];
4   int *bufp1;
5
6   void swap()
7   {
8       int temp;
9
10      bufp1 = &buf[1];
11      temp = *bufp0;
12      *bufp0 = *bufp1;
13      *bufp1 = temp;
14  }
```

code/link/swap.c

Figure 7.5 **Example program for Practice Problem 7.1.**

符号	.symtab条目？	符号类型	在哪个模块中定义	节
buf	是	GLOBAL	m.o	.data
bufp0	是	GLOBAL	swap.o	.data
bufp1	是	GLOBAL	swap.o	COMMON
swap	是	GLOBAL	swap.o	.text
temp	否	-	-	-

swap.o 的符号表

```
root@cg:~/Desktop/csapp# gcc -fno-pic -c swap.c
root@cg:~/Desktop/csapp# readelf -s swap.o

Symbol table '.symtab' contains 12 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS swap.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     8: 0000000000000000     8 OBJECT  GLOBAL DEFAULT    3 bufp0
     9: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND buf
    10: 0000000000000008     8 OBJECT  GLOBAL DEFAULT  COM bufp1
    11: 0000000000000000    60 FUNC    GLOBAL DEFAULT    1 swap
```

swap.o 的节表

```
root@cg:~/Desktop/csapp# readelf -S swap.o
There are 13 section headers, starting at offset 0x350:

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
       000000000000003c  0000000000000000  AX       0     0     1
  [ 2] .rela.text        RELA             0000000000000000  00000230
       0000000000000090  0000000000000018   I      10     1     8
  [ 3] .data             PROGBITS         0000000000000000  00000080
       0000000000000008  0000000000000000  WA       0     0     8
  [ 4] .rela.data        RELA             0000000000000000  000002c0
       0000000000000018  0000000000000018   I      10     3     8
  [ 5] .bss              NOBITS           0000000000000000  00000088
       0000000000000000  0000000000000000  WA       0     0     1
```

m.o 的符号表

```
root@cg:~/Desktop/csapp# gcc -fno-pic -c m.c
root@cg:~/Desktop/csapp# readelf -s m.o

Symbol table '.symtab' contains 11 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.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    6
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     8: 0000000000000000     8 OBJECT  GLOBAL DEFAULT    3 buf
     9: 0000000000000000    21 FUNC    GLOBAL DEFAULT    1 main
    10: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND swap
```

m.o 的节表

```
root@cg:~/Desktop/csapp# readelf -S m.o
There are 12 section headers, starting at offset 0x278:

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
       0000000000000015  0000000000000000  AX       0     0     1
  [ 2] .rela.text        RELA             0000000000000000  000001e8
       0000000000000018  0000000000000018   I       9     1     8
  [ 3] .data             PROGBITS         0000000000000000  00000058
       0000000000000008  0000000000000000  WA       0     0     8
  [ 4] .bss              NOBITS           0000000000000000  00000060
       0000000000000000  0000000000000000  WA       0     0     1
```

练习题7.1

swap.o 的符号表

```
root@cg:~/Desktop/csapp# gcc -fno-pic -c swap.c
root@cg:~/Desktop/csapp# readelf -s swap.o

Symbol table '.symtab' contains 12 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS swap.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     8: 0000000000000000     8 OBJECT  GLOBAL DEFAULT    3 bufp0
     9: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND buf
    10: 0000000000000008     8 OBJECT  GLOBAL DEFAULT  COM bufp1
    11: 0000000000000000    60 FUNC    GLOBAL DEFAULT    1 swap
```

(a) m.c — *code/link/m.c*

```
1   void swap();
2
3   int buf[2] = {1, 2};
4
5   int main()
6   {
7       swap();
8       return 0;
9   }
```

code/link/m.c

(b) swap.c — *code/link/swap.c*

```
1   extern int buf[];
2
3   int *bufp0 = &buf[0];
4   int *bufp1;
5
6   void swap()
7   {
8       int temp;
9
10      bufp1 = &buf[1];
```

gcc中：
COMMON: 未初始化的全局变量
.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
后面再具体解释

符号	.symtab条目？	符号类型	在哪个模块中定义	节
buf	是	GLOBAL	m.o	.data
bufp0	是	GLOBAL	swap.o	.data
bufp1	是	GLOBAL	swap.o	COMMON
swap	是	GLOBAL	swap.0	.text
temp	否	-	-	-

练习题7.1

swap.o 的符号表

```
root@cg:~/Desktop/csapp# gcc -fno-pic -c swap.c
root@cg:~/Desktop/csapp# readelf -s swap.o

Symbol table '.symtab' contains 12 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS swap.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     8: 0000000000000000     8 OBJECT  GLOBAL DEFAULT    3 bufp0
     9: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND buf
    10: 0000000000000008     8 OBJECT  GLOBAL DEFAULT  COM bufp1
    11: 0000000000000000    60 FUNC    GLOBAL DEFAULT    1 swap
```

(a) m.c — *code/link/m.c*

```
1   void swap();
2
3   int buf[2] = {1, 2};
4
5   int main()
6   {
7       swap();
8       return 0;
9   }
```

code/link/m.c

(b) swap.c — *code/link/swap.c*

```
1   extern int buf[];
2
3   int *bufp0 = &buf[0];
4   int *bufp1;
5
6   void swap()
7   {
8       int temp;
9
10      bufp1 = &buf[1];
11      temp = *bufp0;
12      *bufp0 = *bufp1;
13      *bufp1 = temp;
14  }
```

code/link/swap.c

buf 在 swap.o 中有表项吗？在哪一节中？

符号	.symtab条目？	符号类型	在哪个模块中定义	节
buf	是	GLOBAL	m.o	.data
bufp0	是	GLOBAL	swap.o	.data
bufp1	是	GLOBAL	swap.o	COMMON
swap	是	GLOBAL	swap.o	.text
temp	否	-	-	-

局部符号

- 局部非静态 C 变量 vs. 局部静态 C 变量
  - 局部非静态 C 变量：存储在栈上
  - 局部静态 C 变量：存储在 .bss 或 .data中

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

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

编译器为 x 的每个定义都在 .data 或 .bss 中分配空间

在符号表中为每个局部符号创建不同的、唯一的名字，例如 x.1 和 x.2

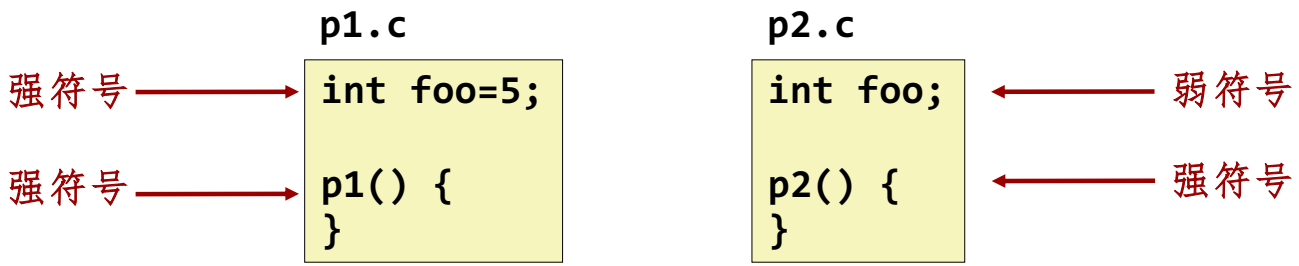
```
root@cg:~/Desktop# readelf -s static.o

Symbol table '.symtab' contains 12 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS static.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     4 OBJECT  LOCAL  DEFAULT    4 x.1794
     6: 0000000000000000     4 OBJECT  LOCAL  DEFAULT    3 x.1797
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     9: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
    10: 0000000000000000    12 FUNC    GLOBAL DEFAULT    1 f
    11: 000000000000000c    12 FUNC    GLOBAL DEFAULT    1 g
```

```c
int f()
{
    static int x = 0;
    return x;
}

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

```
root@cg:~/Desktop# readelf -S static.o
There are 12 section headers, starting at offset 0x2e0:

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
       0000000000000018  0000000000000000  AX       0     0     1
  [ 2] .rela.text        RELA            0000000000000000  00000220
       0000000000000030  0000000000000018   I       9     1     8
  [ 3] .data             PROGBITS        0000000000000000  00000058
       0000000000000004  0000000000000000  WA       0     0     4
  [ 4] .bss              NOBITS          0000000000000000  0000005c
       0000000000000004  0000000000000000  WA       0     0     4
```

链接器如何解析多个同名符号的定义

- 程序符号分为强（strong）和弱（weak）两种
  - 强符号：过程和初始化了的全局符号
  - 弱符号：未初始化的全局符号



链接器的符号解析规则

- **规则1：不允许有多个同名的强符号**
  - 每个强符号只能被定义一次
  - 否则：报链接错误

- **规则2：如果有一个强符号和多个弱符号，选择强符号**
  - 对弱符号的引用会被解析成强符号

- **规则3：如果有多个弱符号，任意选择一个**
  - 可以用 **gcc –fno-common** 来覆盖该规则（现在是默认选项了）

为什么要有 Common 和 .bss 两种类型？

gcc中：

- **COMMON**：未初始化的全局变量
- **.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
- 链接器允许多个模块定义同名的全局符号，编译器翻译时遇到一个弱全局符号（放到 **COMMON** 中），不能决定使用哪个定义，把决定权留给链接器

为什么未初始化的静态变量放到 **.bss** 中而不是 **COMMON** 中？

链接器谜题

`int x;` `p1() {}`	`p1() {}`	链接时错误：两个强符号 (p1)。
`int x;` `p1() {}`	`int x;` `p2() {}`	对 x 的引用会被指向同一个未被初始化的 int，这真的是你期望的吗？

`int x;`
`int y;`
`p1() {}`
	`double x;`
	`p2() {}`
		在 p2 中对 x 的写可能会覆盖 

`int x=7;`
`int y=5;`
`p1() {}`
	`double x;`
	`p2() {}`
		在 p2 中对 x 的写可能会覆盖 

`int x=7;` `p1() {}`	`int x;` `p2() {}`	对 x 的引用会指向同一个初始化了的变量。

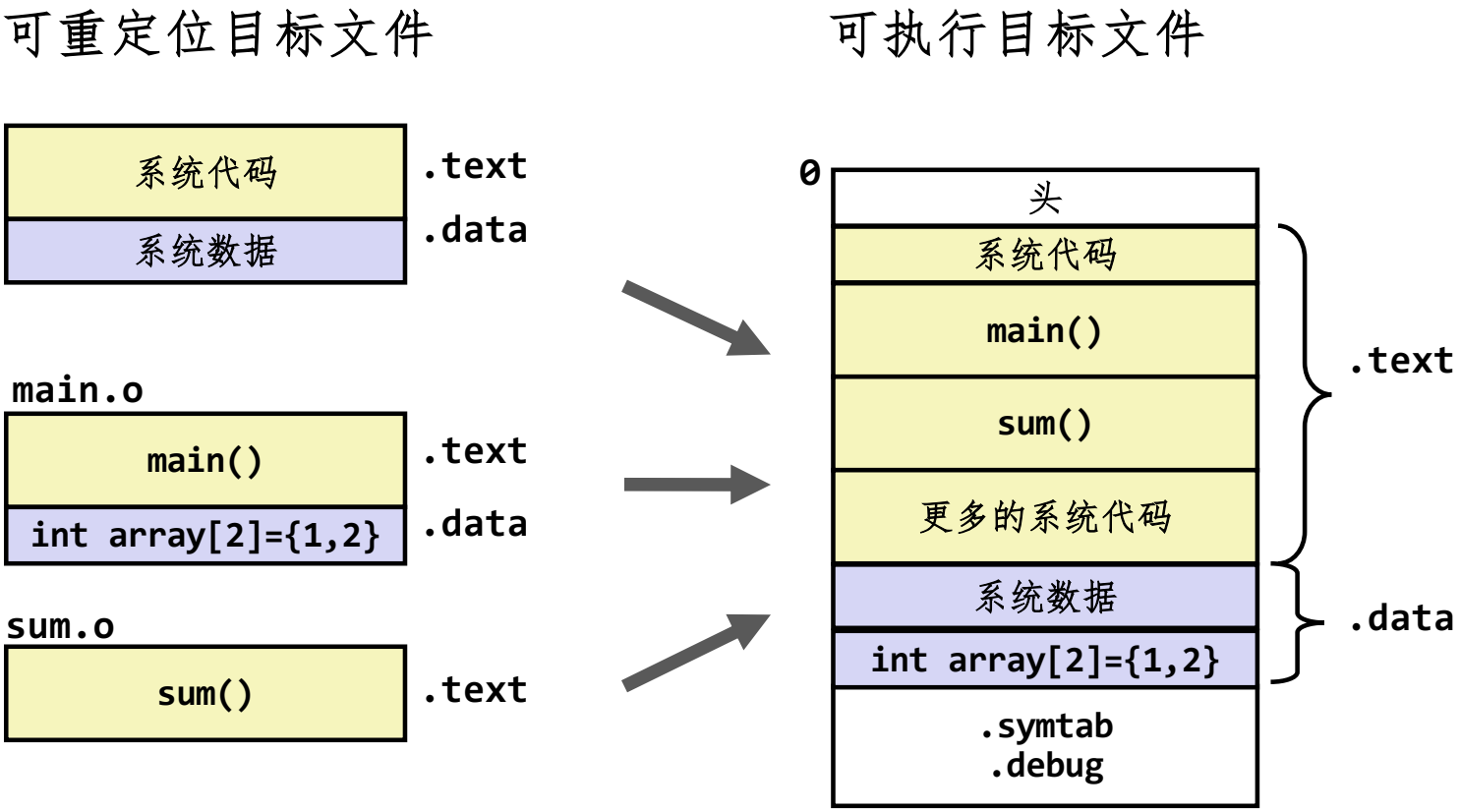
最可怕的场景：两个相同的弱符号结构体定义，被两个具有不同对齐规则的编译器编译.....

全局变量

- 如果可以，请尽量避免

- 否则
  - 如果可以，请使用 **static**
  - 如果你定义了一个全局变量，请初始化
  - 如果你引用了外部的全局变量，请使用 **extern**

步骤2：重定位


可重定位目标文件
可执行目标文件
系统代码
.text
系统数据
.data
main.o
main()
.text
int array[2]={1,2}
.data
sum.o
sum()
.text
0
头
系统代码
main()
sum()
更多的系统代码
.text
系统数据
int array[2]={1,2}
.data
.symtab
.debug

重定位条目

```
int array[2] = {1, 2};

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

```
0000000000000000 <main>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   be 02 00 00 00          mov    $0x2,%esi
   9:   bf 00 00 00 00          mov    $0x0,%edi        # %edi = &array
                        a: R_X86_64_32 array           # 重定位条目

   e:   e8 00 00 00 00          callq  13 <main+0x13> # sum()
                        f: R_X86_64_PC32 sum-0x4       # 重定位条目
  13:   48 83 c4 08             add    $0x8,%rsp
  17:   c3                      retq
```
main.o

32位绝对地址的引用

待重定位引用的位置（地址）

32位PC相对地址的引用

Source: `objdump -r -d main.o`

重定位条目

```
int array[2] = {1, 2

int main()
{
    int val = sum(ar
    return val;
}
```

code/link/elfstructs.c

```
1  typedef struct {
2      long offset;    /* Offset of the reference to relocate */
3      long type:32,   /* Relocation type */
4           symbol:32; /* Symbol table index */
5      long addend;    /* Constant part of relocation expression */
6  } Elf64_Rela;
```

code/link/elfstructs.c

图 7-9 ELF 重定位条目。每个条目表示一个必须被重定位的引用，并指明如何计算被修改的引用

```
0000000000000000 <main>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   be 02 00 00 00          mov    $0x2,%esi
   9:   bf 00 00 00 00          mov    $0x0,%edi       # %edi = &array
                        a: R_X86_64_32 array           # 重定位条目

   e:   e8 00 00 00 00          callq  13 <main+0x13> # sum()
                        f: R_X86_64_PC32 sum-0x4       # 重定位条目
  13:   48 83 c4 08             add    $0x8,%rsp
  17:   c3                      retq
```

32位绝对地址的引用

待重定位引用的位置（地址）

32位PC相对地址的引用

main.o

Source: objdump -r -d main.o

重定位后的 .text 节

```
00000000004004d0 <main>:
  4004d0:       48 83 ec 08        sub    $0x8,%rsp
  4004d4:       be 02 00 00 00     mov    $0x2,%esi
  4004d9:       bf 18 10 60 00     mov    $0x601018,%edi # %edi = &array
  4004de:       e8 05 00 00 00     callq  4004e8 <sum>    # sum()
  4004e3:       48 83 c4 08        add    $0x8,%rsp
  4004e7:       c3                 retq

00000000004004e8 <sum>:
  4004e8:       b8 00 00 00 00          mov    $0x0,%eax
  4004ed:       ba 00 00 00 00          mov    $0x0,%edx
  4004f2:       eb 09                   jmp    4004fd <sum+0x15>
  4004f4:       48 63 ca                movslq %edx,%rcx
  4004f7:       03 04 8f                add    (%rdi,%rcx,4),%eax
  4004fa:       83 c2 01                add    $0x1,%edx
  4004fd:       39 f2                   cmp    %esi,%edx
  4004ff:       7c f3                   jl     4004f4 <sum+0xc>
  400501:       f3 c3                   repz retq
```

Source: objdump -dx prog

```
1   foreach section s {
2       foreach relocation entry r {
3           refptr = s + r.offset;  /* ptr to reference to be relocated */
4
5           /* Relocate a PC-relative reference */
6           if (r.type == R_X86_64_PC32) {
7               refaddr = ADDR(s) + r.offset; /* ref's run-time address */
8               *refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr);
9           }
10
11          /* Relocate an absolute reference */
12          if (r.type == R_X86_64_32)
13              *refptr = (unsigned) (ADDR(r.symbol) + r.addend);
14      }
15  }
```

Figure 7.10 **Relocation algorithm.**

code/link/elfstructs.c

```
1   typedef struct {
2       long offset;      nce to relocate */
3       long type:32,
4            symbol:32;   /
5       long addend;      ocation expression */
6   } Elf64_Rela;
```

0xf
R_X86_64_PC32
sum
-4

code/link/elfstructs.c

Figure 7.9 **ELF relocation entry.** Each entry identifies a reference that must be relocated and specifies how to compute the modified reference.

sum(0x4004e8) - 4 - PC(0x4004df)
= 0x5

对sum()使用 PC 相对寻址:
0x4004e8 = 0x4004e3 + 0x5

位置无关代码

```
root@cg:~/Desktop/csapp/linking# objdump -d main.o

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:   be 02 00 00 00          mov    $0x2,%esi
   d:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 14 <main+0x14>
  14:   e8 00 00 00 00          callq  19 <main+0x19>
  19:   89 45 fc                mov    %eax,-0x4(%rbp)
  1c:   8b 45 fc                mov    -0x4(%rbp),%eax
  1f:   c9                      leaveq
  20:   c3                      retq
```

链接选项：**-fpic**

现在是默认选项

```
root@cg:~/Desktop/csapp/linking# objdump -r main.o

main.o:     file format elf64-x86-64

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000010 R_X86_64_PC32     array-0x0000000000000004
0000000000000015 R_X86_64_PLT32    sum-0x0000000000000004
```

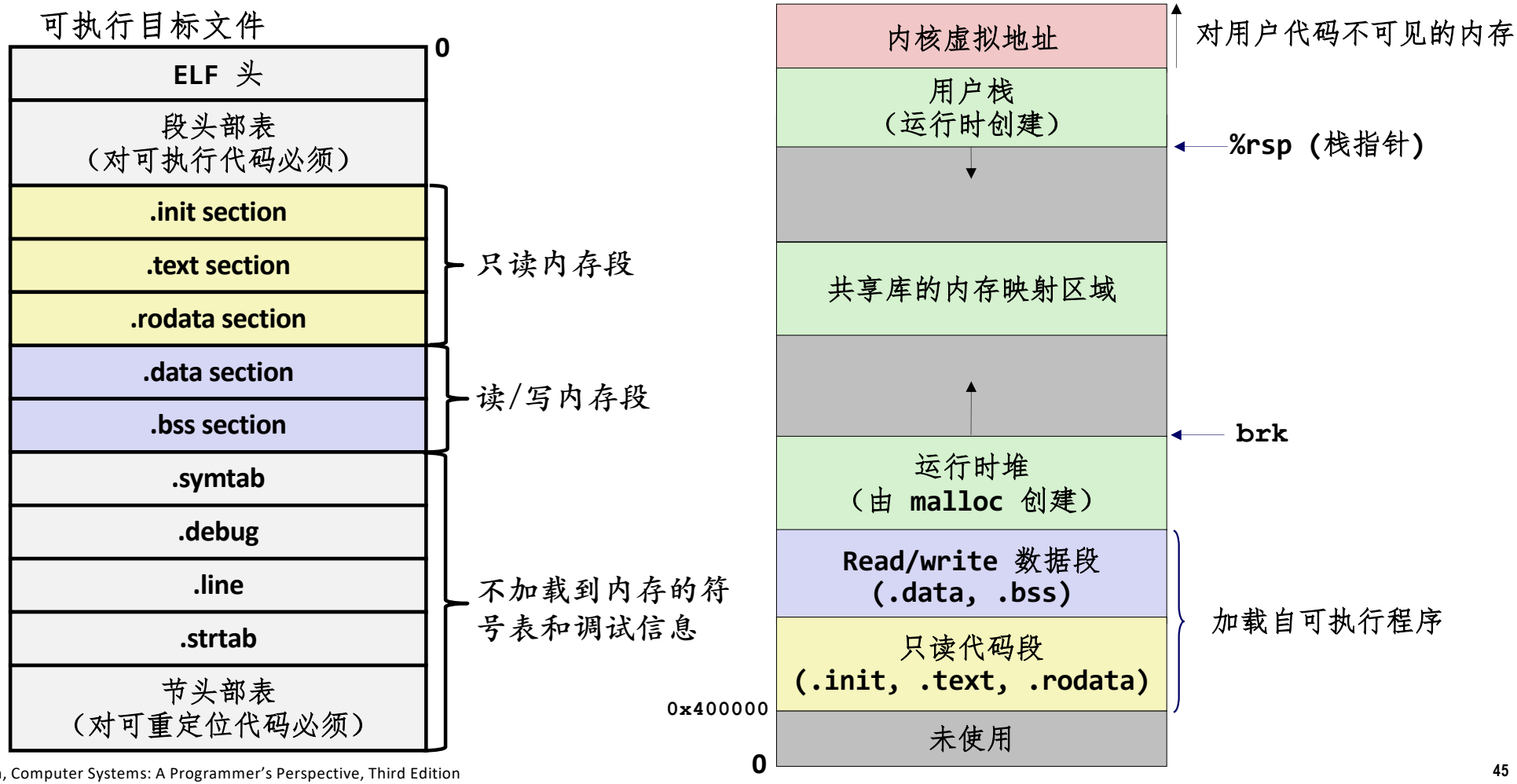
位置有关代码

```
root@cg:~/Desktop/csapp/linking# objdump -d main.abs.o

main.abs.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:   be 02 00 00 00          mov    $0x2,%esi
   d:   bf 00 00 00 00          mov    $0x0,%edi
  12:   e8 00 00 00 00          callq  17 <main+0x17>
  17:   89 45 fc                mov    %eax,-0x4(%rbp)
  1a:   8b 45 fc                mov    -0x4(%rbp),%eax
  1d:   c9                      leaveq
  1e:   c3                      retq
```

链接选项：**-fno-pic**

```
root@cg:~/Desktop/csapp/linking# gcc -c -fno-pic main.c -o main.abs.o
root@cg:~/Desktop/csapp/linking# objdump -r main.abs.o

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

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
000000000000000e R_X86_64_32       array
0000000000000013 R_X86_64_PC32     sum-0x0000000000000004
```

加载可执行目标文件


可执行目标文件
0
ELF 头
段头部表
（对可执行代码必须）
.init section
.text section
.rodata section
.data section
.bss section
.symtab
.debug
.line
.strtab
节头部表
（对可重定位代码必须）
只读内存段
读/写内存段
不加载到内存的符号表和调试信息
内核虚拟地址
对用户代码不可见的内存
用户栈
（运行时创建）
%rsp（栈指针）
共享库的内存映射区域
brk
运行时堆
（由 malloc 创建）
Read/write 数据段
(.data, .bss)
只读代码段
(.init, .text, .rodata)
加载自可执行程序
0x400000
未使用
0

打包常用函数

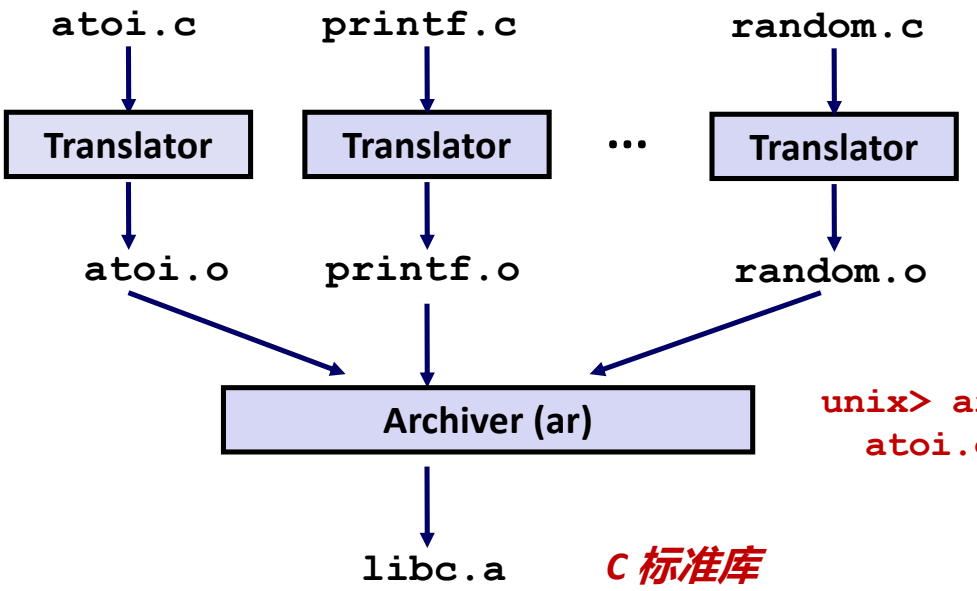
- 如何将程序员常用的函数打包到一起？
  - 数学，I/O，内存管理，字符串处理，等等

- 到目前为止，有两种方式：
  - 方式1：将所有的函数都放进一个单一的源文件中
    - 程序员把这个大的目标文件链接进他们的程序中
    - 空间和时间效率都很低
  - 方式2：将每个函数放在一个独立的原文件中
    - 程序员显示地将适当的二进制文件链接进他们的程序中
    - 更有效，但对程序员来说增加了负担

以前的解决方案：静态库

- 静态库（.a 存档文件）
  - 将一些相关联的可重定位目标文件连接成一个带索引的文件（称为档案 *archive*）
  - 改进链接器，使它可以在一个或多个档案文件中寻找符号，以解析未被解析的外部引用
  - 如果某个档案文件能够解析某个引用，就将它链接到可执行文件中

创建静态库




```
unix> ar rs libc.a \
  atoi.o printf.o … random.o
```

- **Archiver** 允许增量更新
- 重新编译发生了变化的函数，在存档文件中替换相应的 **.o** 文件

常见的库

libc.a （C 标准库)

- 4.6 MB 存档了 1496 个目标文件
- I/O，内存分配，信号处理，字符串处理，数据和时间，随机数，整数数学

libm.a （C 数学库）

- 2 MB 存档了 444 个目标文件
- 浮点数数学 (sin, cos, tan, log, exp, sqrt, …)

```
% ar -t libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
…
```

```
% ar -t libm.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
…
```

与静态库链接

```c
#include <stdio.h>
#include "vector.h"

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main()
{
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n", z[0], z[1]);
    return 0;
}
```

main2.c

libvector.a

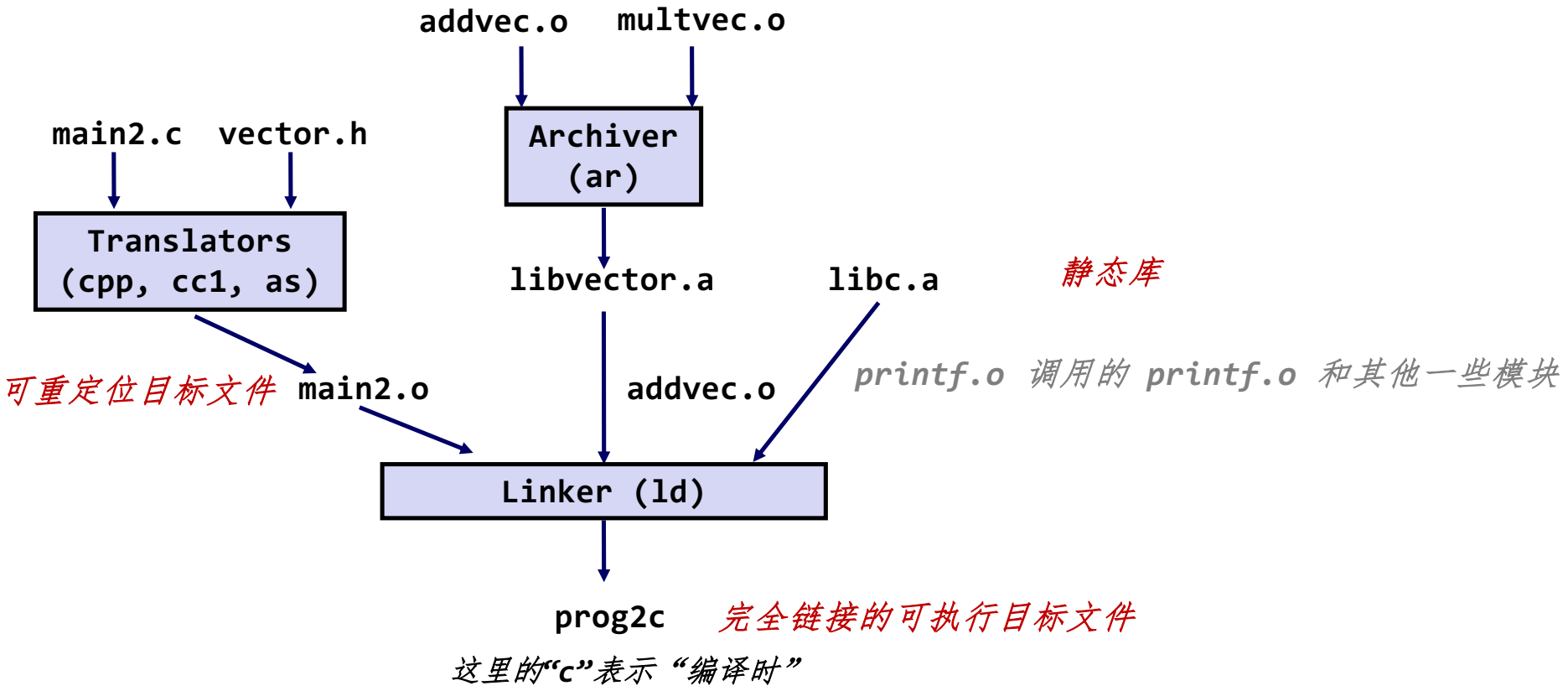
```c
void addvec(int *x, int *y,
            int *z, int n) {
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
```

addvec.c

```c
void multvec(int *x, int *y,
             int *z, int n)
{
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] * y[i];
}
```

multvec.c

与静态库链接


addvec.o
multvec.o
main2.c
vector.h
Archiver
(ar)
Translators
(cpp, cc1, as)
libvector.a
libc.a
静态库
可重定位目标文件
main2.o
addvec.o
printf.o 调用的 printf.o 和其他一些模块
Linker (ld)
prog2c
完全链接的可执行目标文件
这里的“c”表示“编译时”

使用静态库

- 链接器用来解析外部引用的算法
  - 按照命令行顺序扫描 **.o** 文件和 **.a** 文件
  - 在扫描过程中，维护一个当前未解析的引用列表
  - 每遇到一个新的 **.o** 或 **.a** 文件，***obj***，就试着用 ***obj*** 里定义的符号去解析该列表中未解析的引用
  - 如果到扫描结束时，这个未解析列表中还有条目，那么就报错

- 问题
  - 命令行顺序很重要！
  - 一般规律：将库放在命令行的最后

```
unix> gcc -L. libtest.o -lmine
unix> gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```

现在的做法：共享库

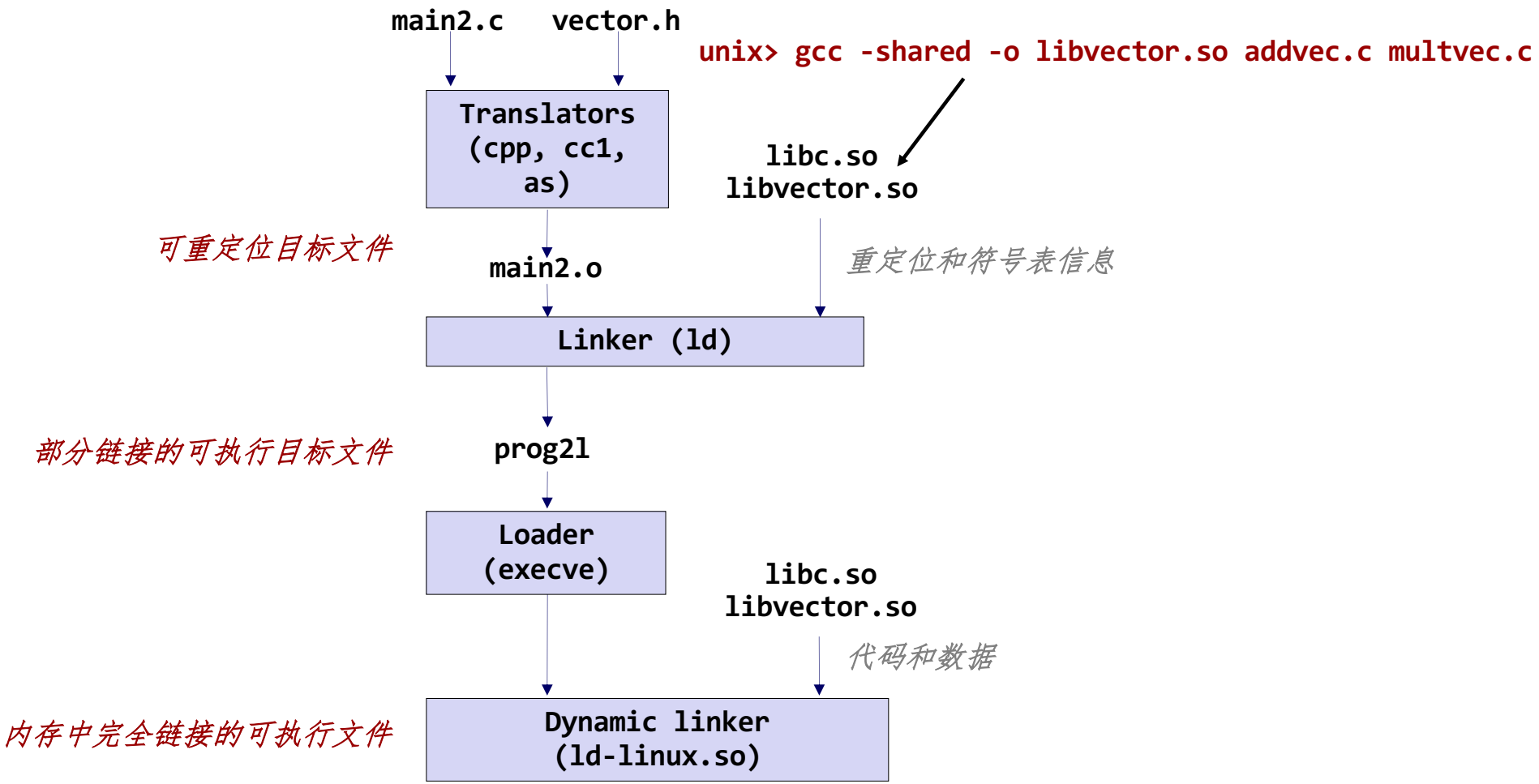
- 静态库的缺点
  - 存储的可执行文件中有大量重复（每个函数都需要 libc）
  - 运行的可执行文件中有大量重复
  - 对系统库的很小的 bug 修复，要求每个应用程序都进行重新链接

- 现在的做法：共享库
  - 包含代码和数据的目标文件，在加载时或运行时，动态地加载和链接到应用程序中
  - 也被称为：动态链接库，DLL，.so 文件

共享库（续）

- 可以在可执行文件初次加载和运行时进行动态链接 (load-time linking)
  - Linux上最常见的情况，由动态链接器（**ld-linux.so**）自动处理
  - 标准 C 库（**libc.so**）通常是动态链接的
- 可以在程序开始执行后进行动态链接 (run-time linking)
  - 在Linux上，是通过调用 **dlopen()** 接口来实现的
    - 分布式软件
    - 高性能 web 服务器
    - 运行时库打桩
- 也可以多个进程共享库函数
  - 这个在学习了虚拟内存后，再深入理解

加载时动态链接


main2.c
vector.h
unix> gcc -shared -o libvector.so addvec.c multvec.c
Translators
(cpp, cc1,
as)
libc.so
libvector.so
可重定位目标文件
main2.o
重定位和符号表信息
Linker (ld)
部分链接的可执行目标文件
prog2l
Loader
(execve)
libc.so
libvector.so
代码和数据
内存中完全链接的可执行文件
Dynamic linker
(ld-linux.so)

运行时动态链接

```c
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main()
{
    void *handle;
    void (*addvec)(int *, int *, int *, int);
    char *error;

    /* Dynamically load the shared library that contains addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
```

dll.c

运行时动态链接

```
    ...

    /* Get a pointer to the addvec() function we just loaded */
    addvec = dlsym(handle, "addvec");
    if ((error = dlerror()) != NULL) {
        fprintf(stderr, "%s\n", error);
        exit(1);
    }

    /* Now we can call addvec() just like any other function */
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n", z[0], z[1]);

    /* Unload the shared library */
    if (dlclose(handle) < 0) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    return 0;
}
```

dll.c

链接小结

- 链接是一项允许从多个目标文件构造程序的技术
- 链接可以发生在一个程序生命周期的各个阶段
  - 编译时（当编译一个程序时）
  - 加载时（当将程序加载进内存时）
  - 运行时（当执行程序时）
- 理解链接能够帮助你避免一些很讨厌的错误，使你成为一个更好的程序员

主要内容

- 链接
- 案例研究：库打桩/插桩（**Library interpositioning**）

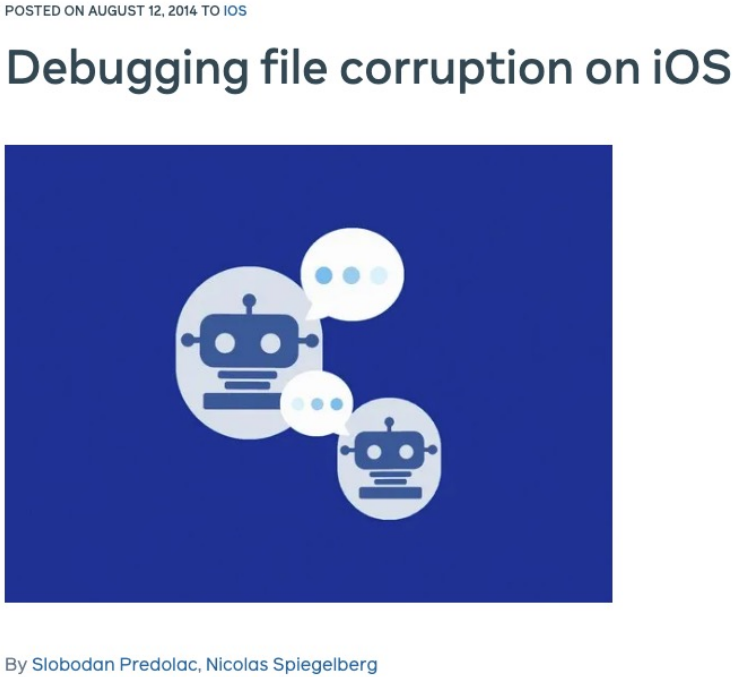
案例研究：库打桩

- 库打桩：一种强大的链接技术，使得程序员可以截获对任意函数的调用
- 打桩可以发生在：
  - 编译时（当编译源码时）
  - 链接时（当将可重定位目标文件静态链接生成可执行目标文件时）
  - 加载/运行时（当可执行目标文件加载进内存、动态链接和执行时）

打桩技术的一些应用

- 安全Security
  - 限制（沙盒）
  - 后台加密
- 调试
  - 2014年两个Facebook (Meta)的工程师用打桩技术调试发现了一个持续一年的bug
  - SPDY网络栈中的代码写入到了错误的位置
  - 是通过拦截对 Posix 写函数(write, writev, pwrite)的调用解决的

Source: Facebook engineering blog post at https://code.facebook.com/posts/313033472212144/debugging-file-corruption-on-ios/



打桩技术的一些应用

- 监控与侧写（profiling）
  - 对函数调用次数计数
  - 描述对函数调用的地点和参数
  - 跟踪 malloc
    - 发现内存泄漏
    - 生成访存地址跟踪序列

程序示例

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

int main()
{
    int *p = malloc(32);
    free(p);
    return(0);
}
```
int.c

- 目标：在不中断程序执行和修改源代码的情况下，跟踪分配和释放块的地址和大小
- 三种解决方法：在编译时、链接时和加载/运行时对库函数 **malloc** 和 **free**打桩

编译时打桩

```c
#ifdef COMPILETIME
#include <stdio.h>
#include <malloc.h>

/* malloc wrapper function */
void *mymalloc(size_t size)
{
    void *ptr = malloc(size);
    printf("malloc(%d)=%p\n",
           (int)size, ptr);
    return ptr;
}

/* free wrapper function */
void myfree(void *ptr)
{
    free(ptr);
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

编译时打桩

```
#define malloc(size) mymalloc(size)
#define free(ptr) myfree(ptr)

void *mymalloc(size_t size);
void myfree(void *ptr);
```

malloc.h

```
linux> make intc
gcc -Wall -DCOMPILETIME -c mymalloc.c
gcc -Wall -I. -o intc int.c mymalloc.o
linux> make runc
./intc
malloc(32)=0x1edc010
free(0x1edc010)
linux>
```

链接时打桩

```c
#ifdef LINKTIME
#include <stdio.h>

void *__real_malloc(size_t size);
void __real_free(void *ptr);

/* malloc wrapper function */
void *__wrap_malloc(size_t size)
{
    void *ptr = __real_malloc(size); /* Call libc malloc */
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}

/* free wrapper function */
void __wrap_free(void *ptr)
{
    __real_free(ptr); /* Call libc free */
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

链接时打桩

```
linux> make intl
gcc -Wall -DLINKTIME -c mymalloc.c
gcc -Wall -c int.c
gcc -Wall -Wl,--wrap,malloc -Wl,--wrap,free -o intl
int.o mymalloc.o
linux> make runl
./intl
malloc(32) = 0x1aa0010
free(0x1aa0010)
linux>
```

- “-Wl”标志是将后面的参数传递给链接器，并用空格替换逗号
- “--wrap,malloc ”参数指示链接器以一种特殊的方式解析引用
  - 对 malloc 的引用应当被解析成 __wrap_malloc
  - 对 __real_malloc 的引用应当被解析成 malloc

加载/运行时打桩

RTLD_NEXT: Find the next occurrence of the desired symbol in the search order after the current object. This allows one to provide a wrapper around a function in another shared object, so that, for example, the definition of a function in a preloaded shared object (see LD_PRELOAD in ld.so(8)) can find and invoke the "real" function provided in another shared object (or for that matter, the "next" definition of the function in cases where there are multiple layers of preloading).

```c
#ifdef RUNTIME
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

/* malloc wrapper function */
void *malloc(size_t size)
{
    void *(*mallocp)(size_t size);
    char *error;

    mallocp = dlsym(RTLD_NEXT, "malloc"); /* Get addr of libc malloc */
    if ((error = dlerror()) != NULL) {
        fputs(error, stderr);
        exit(1);
    }
    char *ptr = mallocp(size); /* Call libc malloc */
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}
```

mymalloc.c

加载/运行时打桩

```c
/* free wrapper function */
void free(void *ptr)
{
    void (*freep)(void *) = NULL;
    char *error;

    if (!ptr)
        return;

    freep = dlsym(RTLD_NEXT, "free"); /* Get address of libc free */
    if ((error = dlerror()) != NULL) {
        fputs(error, stderr);
        exit(1);
    }
    freep(ptr); /* Call libc free */
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

加载/运行时打桩

```
linux> make intr
gcc -Wall -DRUNTIME -shared -fpic -o mymalloc.so mymalloc.c -ldl
gcc -Wall -o intr int.c
linux> make runr
(LD_PRELOAD="./mymalloc.so" ./intr)
malloc(32) = 0xe60010
free(0xe60010)
linux>
```

- **LD_PRELOAD** 环境变量告诉动态链接器在解析未被解析的引用时 (例如，对 **malloc** 的引用)，先在 **mymalloc.so** 中查找

打桩小结

- 编译时
  - 将对 `malloc/free` 的调用宏扩展为对 `mymalloc/myfree` 的调用
- 链接时
  - 使用链接器的技巧做特殊的名字解析
    - `malloc` → `__wrap_malloc`
    - `__real_malloc` → `malloc`
- 加载/运行时
  - 采用动态链接的方法以不同的名字加载库函数 `malloc/free`，实现自定义的 `malloc/free` 版本


坚持就是胜利！
奋斗