第四章 程序的链接

目标文件格式

符号解析与重定位

共享库与动态链接

可执行文件的链接生成

- 主要教学目标
  - 了解链接器是如何工作的，从而能够养成良好的程序设计习惯，并增强程序调试能力。
  - 通过了解可执行文件的存储器映像来进一步深入理解进程的虚拟地址空间的概念。
- 包括以下内容
  - 链接和静态链接概念
  - 三种目标文件格式
  - 符号及符号表、符号解析
  - 使用静态库链接
  - 重定位信息及重定位过程
  - 可执行文件的存储器映像
  - 可执行文件的加载
  - 共享（动态）库链接

使用链接的好处

链接带来的好处1：模块化

（1）一个程序可以分成很多源程序文件

（2）可构建公共函数库，如数学库，标准I/O库等

链接带来的好处2：效率高

（1）时间上，可分开编译

只需重新编译被修改的源程序文件，然后重新链接

（2）空间上，无需包含共享库所有代码

源文件中无需包含共享库函数的源码，只要直接调用即可，

可执行文件和运行时的内存中只需包含所调用函数的代码

而不需要包含整个共享库

程序的链接

- 第一讲：目标文件格式
  - 程序的链接概述、链接的意义与过程
  - ELF目标文件、重定位目标文件格式
- 第二讲：符号解析与重定位
  - 符号和符号表、符号解析
  - 与静态库的链接
  - 重定位信息、重定位过程
  - 可执行目标文件格式
  - 可执行文件的加载
- 第三讲：动态链接
  - 动态链接的特性、程序加载时的动态链接、程序运行时的动态链接、动态链接举例

一个C语言程序举例

main.c

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

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

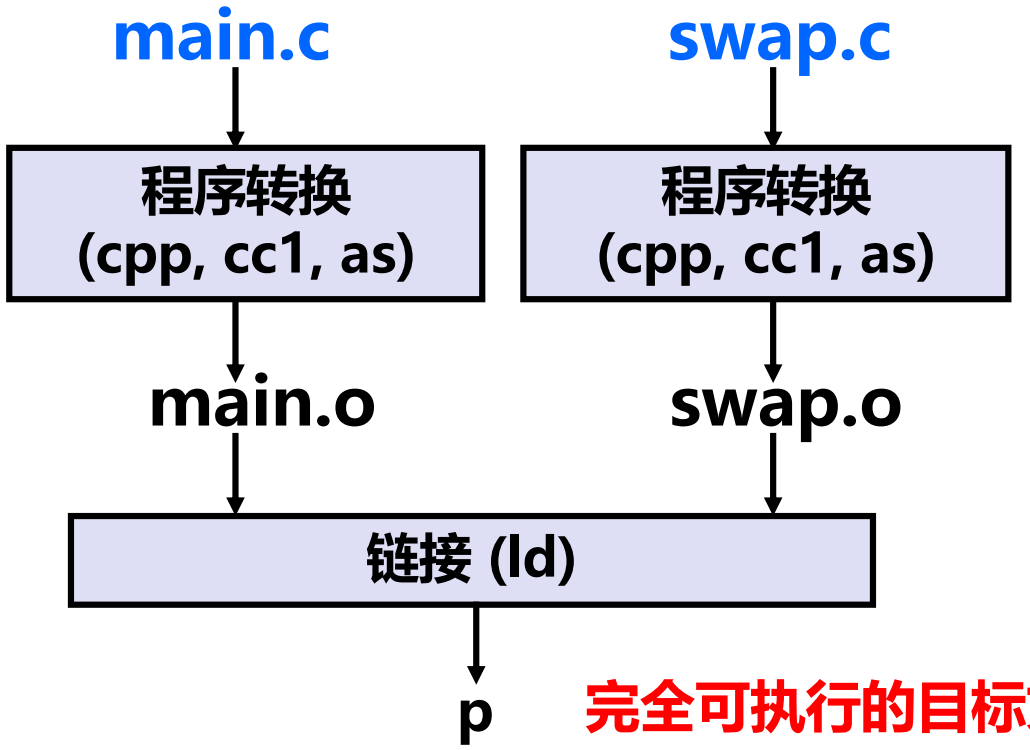
swap.c

```
extern int buf[];

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

void swap()
{
   int temp;
   bufp1 = &buf[1];
   temp = *bufp0;
   *bufp0 = *bufp1;
   *bufp1 = temp;
}
```

可执行文件的生成

- 使用GCC编译器编译并链接生成可执行程序P:
  - $ gcc -o p main.c swap.c
  - $ ./p

GCC编译器的静态链接过程



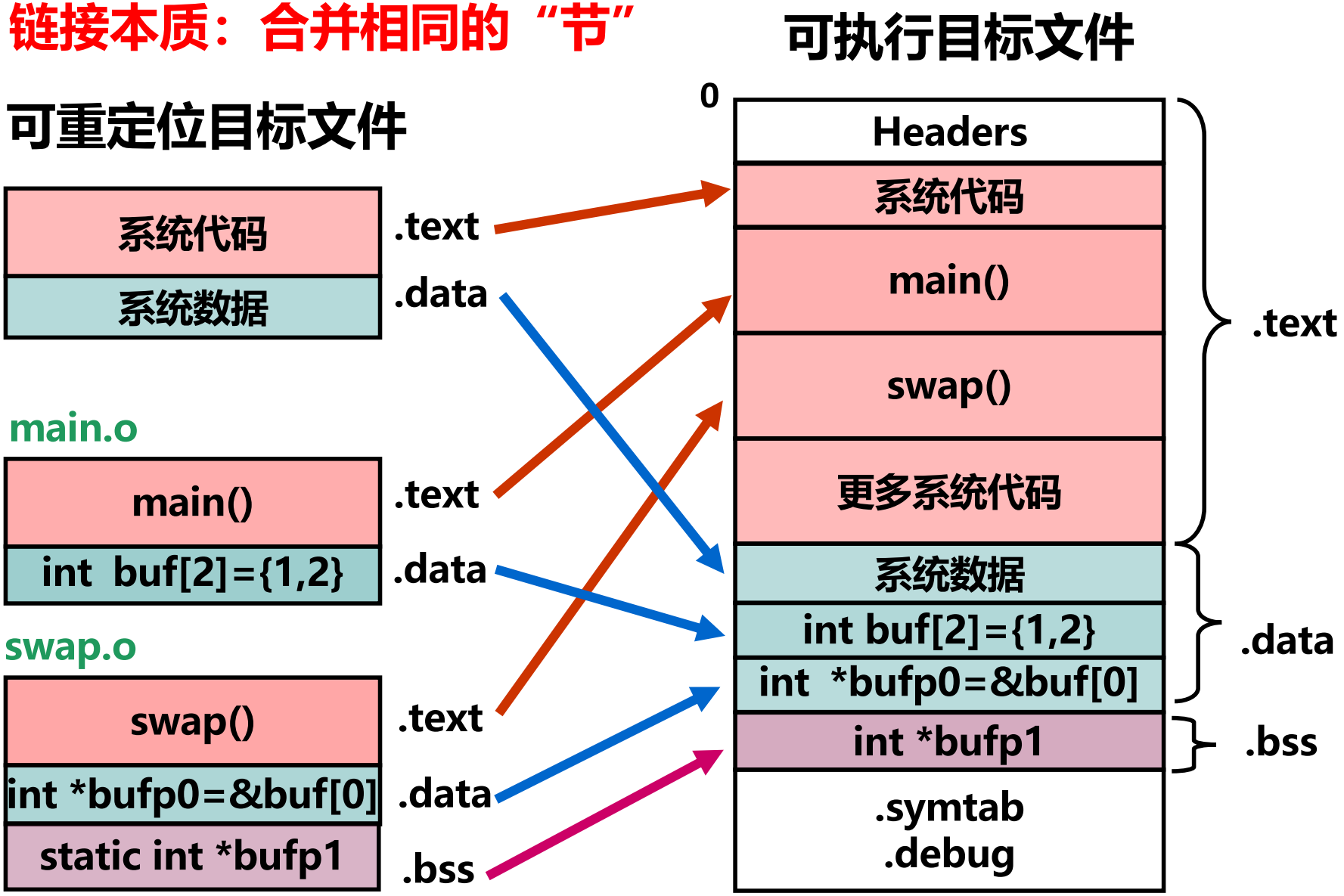
源程序文件

分别转换（预处理、编译、汇编）为可重定位目标文件

完全可执行的目标文件

链接过程的本质

链接本质：合并相同的“节”



三类目标文件

- 可重定位目标文件 (.o)
  - 其代码和数据可和其他可重定位文件合并为可执行文件
    - 每个.o 文件由对应的.c文件生成
    - 每个.o文件代码和数据地址都从0开始
- 可执行目标文件 (默认为a.out)
  - 包含的代码和数据可以被直接复制到内存并被执行
  - 代码和数据地址为虚拟地址空间中的地址
- 共享的目标文件 (.so)
  - 特殊的可重定位目标文件，能在装入或运行时被装入到内存并自动被链接，称为共享库文件
  - Windows 中称其为 *Dynamic Link Libraries* (DLLs)

目标文件的格式

- 目标代码（Object Code）指编译器和汇编器处理源代码后所生成的机器语言目标代码
- 目标文件（Object File）指包含目标代码的文件
- 最早的目标文件格式是自有格式，非标准的
- 标准的几种目标文件格式
  - DOS操作系统（最简单）：COM格式和EXE格式，文件中仅包含代码和数据，且被加载到固定位置
  - System V UNIX早期版本：COFF格式，文件中不仅包含代码和数据，还包含重定位信息、调试信息、符号表等其他信息，由一组严格定义的数据结构序列组成
  - Windows：PE格式（COFF的变种），称为可移植可执行（Portable Executable，简称PE）
  - Linux等类UNIX：ELF格式（COFF的变种），称为可执行可链接（Executable and Linkable Format，简称ELF）

链接操作的步骤

- Step 1. 符号解析（Symbol resolution）
  - 程序中有定义和引用的符号 (包括变量和函数等)
    - void swap() {…} /* 定义符号swap */
    - swap(); /* 引用符号swap */
    - int *xp = &x; /* 定义符号 xp, 引用符号 x */
  - 编译器将定义和引用的符号存放在一个符号表（ symbol table）
  - 链接器将每个符号的引用都与一个确定的符号定义建立关联
- Step 2. 重定位
  - 将多个代码节与数据节分别合并为一个单独的代码段和数据段
  - 计算每个定义的符号在虚拟地址空间中的绝对地址
  - 将可执行文件中符号引用处的地址修改为重定位后的地址信息

可重定位目标文件格式

ELF 头

- ✓ 定义了ELF魔数、版本、小端/大端、操作系统平台、目标文件的类型、机器结构类型、节头表的起始位置和长度等

节头表：不同节的位置和大小

.text 节

- ✓ 编译后的代码部分

.rodata 节

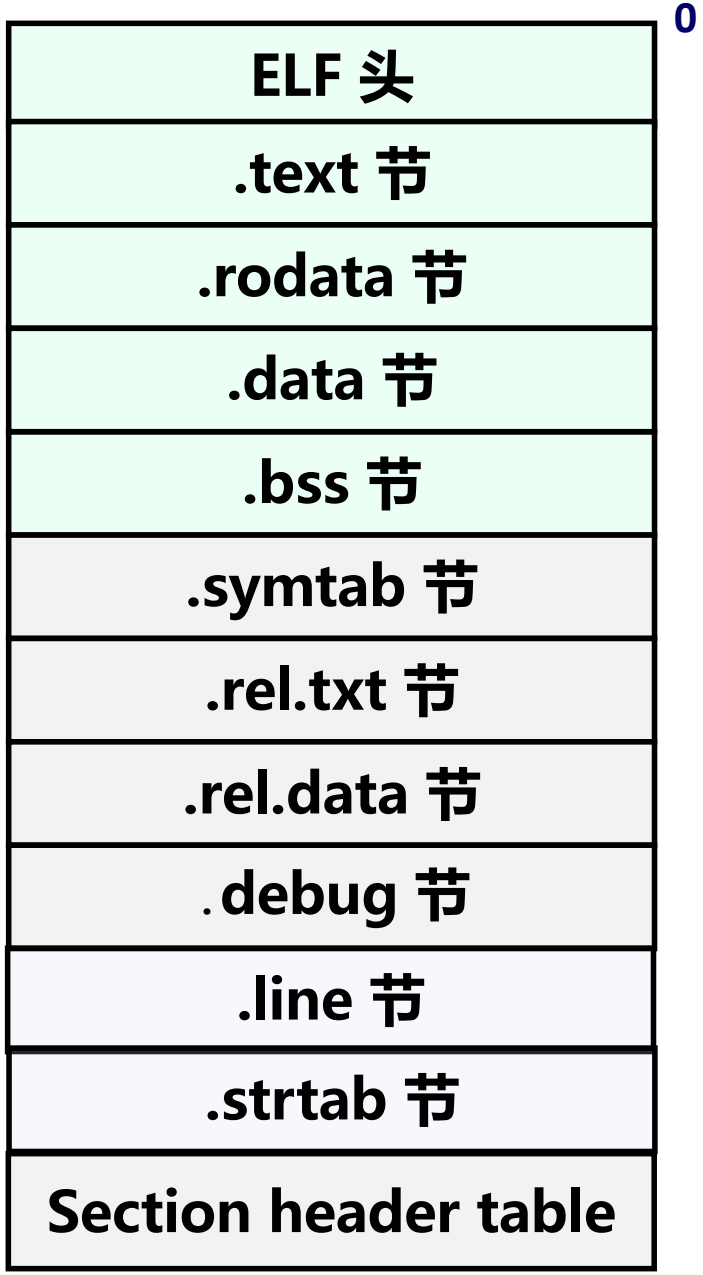
- ✓ 只读数据，如 printf 格式串、switch 跳转表等

.data 节

- ✓ 已初始化的全局变量

.bss 节

- ✓ 未初始化全局变量，仅是占位符，不占据任何实际磁盘空间。区分初始化和非初始化是为了空间效率



```c
#include <stdio.h>
int main() {
  printf("Hello, %s!\n", "World"); // 格式字符串是 "Hello, %s!\n"
  return 0;
}
```

```asm
.section .rodata          # 只读数据节
.LC0:
  .string "Hello, %s!\n" # 格式字符串存储在这里
.LC1:
  .string "World"         # 另一个字符串常量
.section .text            # 代码节
main:
  pushq %rbp              # 保存基址指针
  movq %rsp, %rbp         # 设置新的栈帧
  # 调用 printf(格式字符串, "World")
  movq $.LC0, %rdi        # 第一个参数：格式字符串地址 -> %rdi
  movq $.LC1, %rsi        # 第二个参数："World" 地址 -> %rsi
  call printf             # 调用 printf
  movl $0, %eax           # 返回值 0
  popq %rbp               # 恢复基址指针
  ret                     # 返回
```

```
long switch_eg
(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

跳转表属于只读数据

Jump table

```
.section .rodata
  .align 8
.L4:
  .quad   .L8 # x = 0
  .quad   .L3 # x = 1
  .quad   .L5 # x = 2
  .quad   .L9 # x = 3
  .quad   .L8 # x = 4
  .quad   .L7 # x = 5
  .quad   .L7 # x = 6
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # Use default
    jmp     *.L4(,%rdi,8)   # goto *JTab[x]
```

可重定位目标文件格式

.symtab

- 符号表：定义和引用的函数和全局变量的信息

.rel.text

- .text节中位置的列表：链接器组合多个可重定位目标文件时需要修改

.rel.data

- 全局变量的重定位信息

.debug (gcc -g)

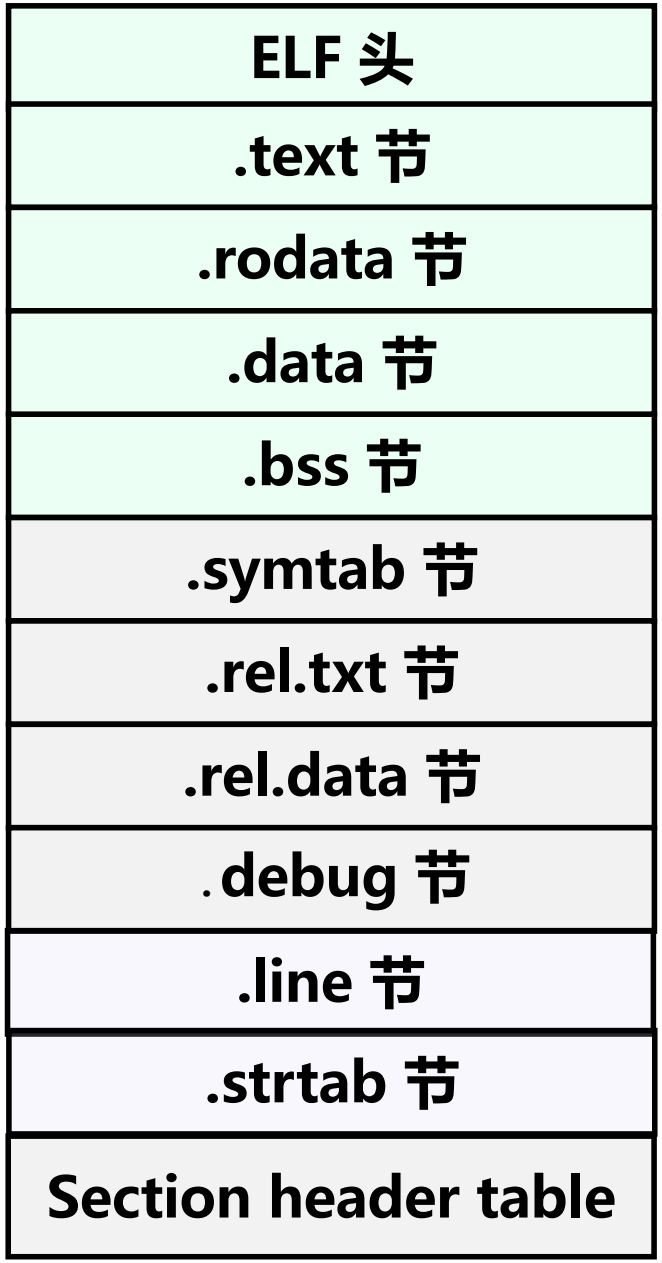
- 调试符号表：程序中定义的局部变量和类型、全局变量、C源文件

.line

- C源程序的行号和.text节中机器指令的映射

.strtab

- 字符串表：包括所有符号和节名



ELF头（ELF Header）

- ELF头位于ELF文件开始，包含文件结构说明信息。以下是64位系统对应的数据结构

```
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: 560 (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: 12
```

定义了ELF魔数、版本、小端/大端、操作系统平台、目标文件的类型、机器结构类型、程序执行的入口地址、程序头表（段头表）的起始位置和长度、节头表的起始位置和长度等

ELF头（ELF Header）

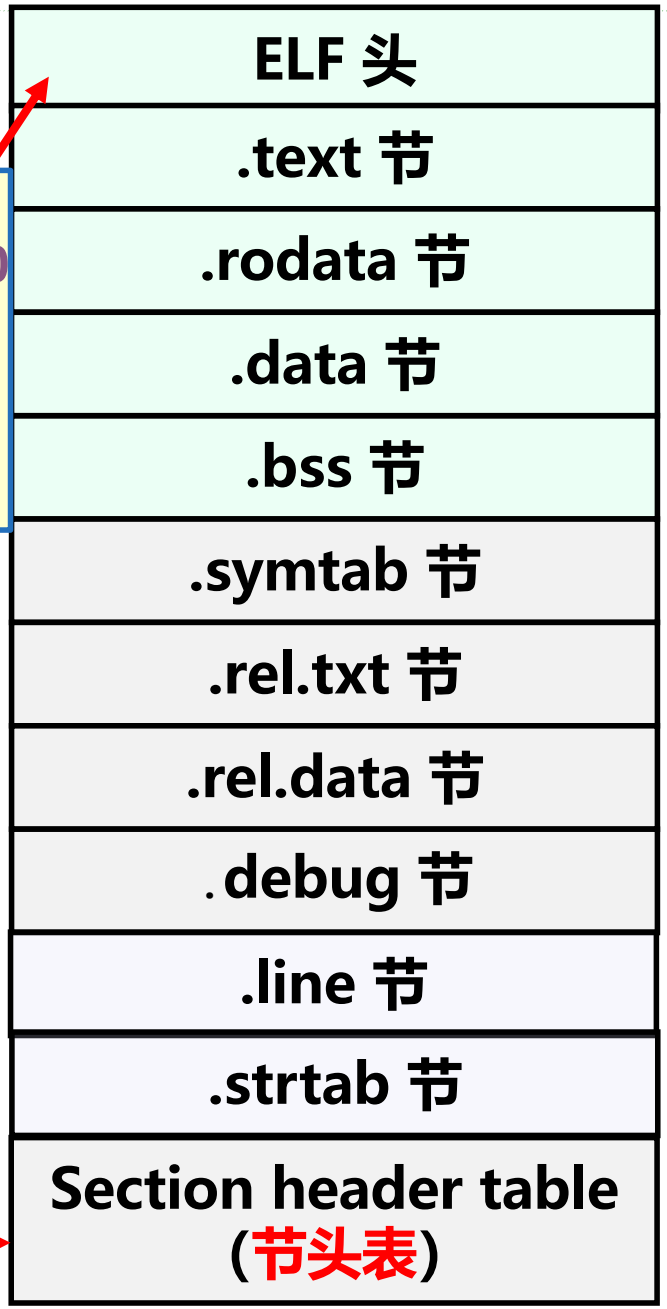
$ readelf -h main.o　　可重定位目标文件的ELF头

```
ELF Header:  ELF文件的魔数
  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: 560 (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: 12
```

没有程序头表

13x64B

.strtab在节头表中的索引



节头表信息举例

$ readelf -S main.o

13 section headers, starting at offset 0x230:

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
       000000000000001c  0000000000000000  AX       0     0     1
  [ 2] .rela.text        RELA             0000000000000000  00000190
       0000000000000018  0000000000000018   I      10     1     8
  [ 3] .data             PROGBITS         0000000000000000  00000060
       0000000000000008  0000000000000000  WA       0     0     8
  [ 4] .bss              NOBITS           0000000000000000  00000068
       0000000000000000  0000000000000000  WA       0     0     1
  [ 5] .comment          PROGBITS         0000000000000000  00000068
       000000000000002c  0000000000000001  MS       0     0     1
  [ 6] .note.GNU-stack   PROGBITS         0000000000000000  00000094
       0000000000000000  0000000000000000           0     0     1
  [ 7] .note.gnu.pr[...] NOTE             0000000000000000  00000098
       0000000000000020  0000000000000000   A       0     0     8
  [ 8] .eh_frame         PROGBITS         0000000000000000  000000b8
       0000000000000030  0000000000000000   A       0     0     8
  [ 9] .rela.eh_frame    RELA             0000000000000000  000001a8
       0000000000000018  0000000000000018   I      10     8     8
  [10] .symtab           SYMTAB           0000000000000000  000000e8
       0000000000000090  0000000000000018          11     3     8
  [11] .strtab           STRTAB           0000000000000000  00000178
       0000000000000016  0000000000000000           0     0     1
  [12] .shstrtab         STRTAB           0000000000000000  000001c0
       000000000000006c  0000000000000000           0     0     1
```

程序的链接

- 第一讲：目标文件格式
  - 程序的链接概述、链接的意义与过程
  - ELF目标文件、重定位目标文件格式
- 第二讲：符号解析与重定位
  - 符号和符号表、符号解析
  - 与静态库的链接
  - 重定位信息、重定位过程
  - 可执行目标文件格式
  - 可执行文件的加载
- 第三讲：动态链接
  - 动态链接的特性、程序加载时的动态链接、程序运行时的动态链接、动态链接举例

链接操作的步骤

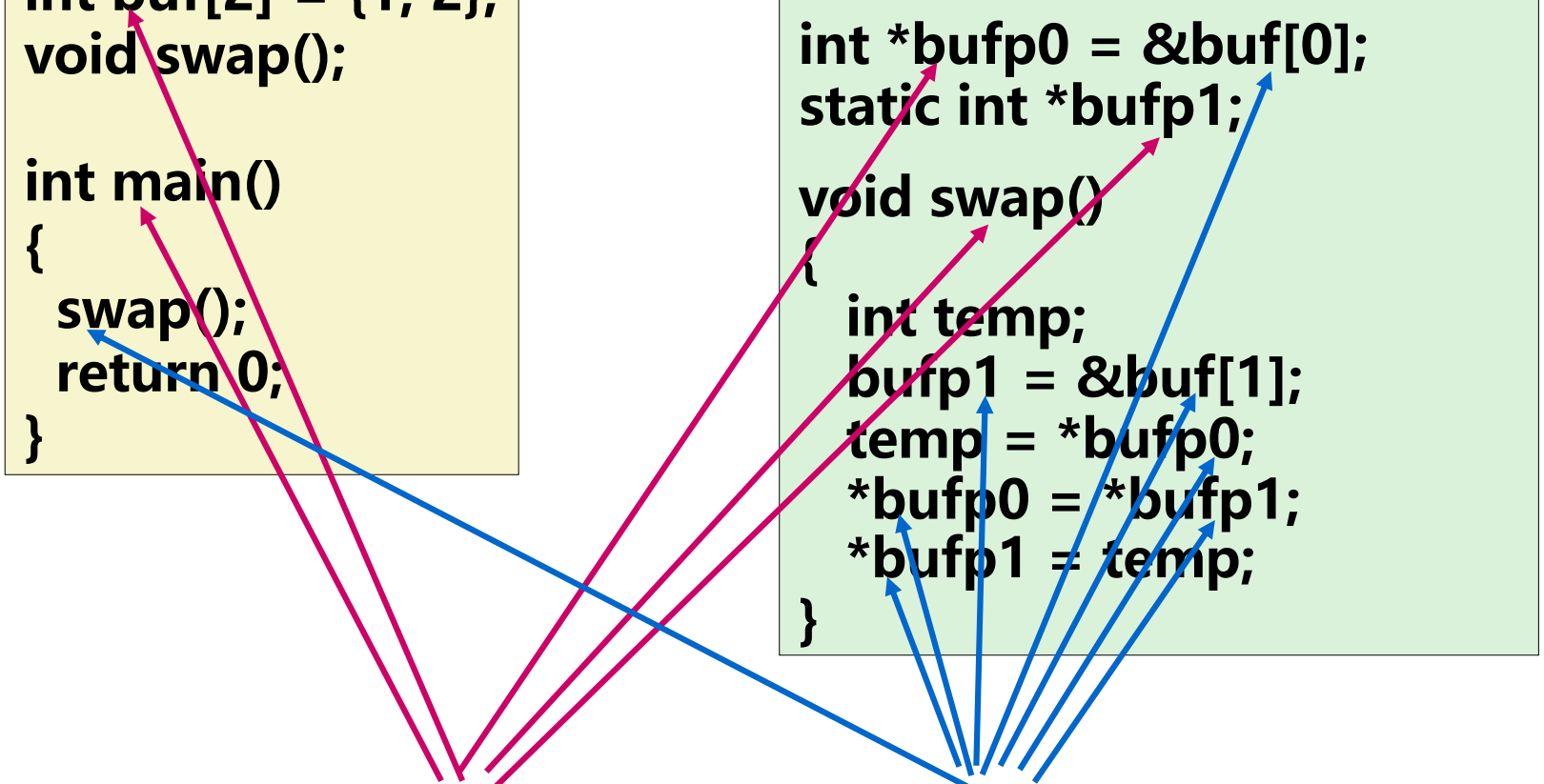
- Step 1. 符号解析（Symbol resolution）
  - 程序中有定义和引用的符号 (包括变量和函数等)
  - 编译器将定义和引用的符号存放在一个符号表（ symbol table ）中.
  - 链接器将每个符号的引用都与一个确定的符号定义建立关联
- Step 2. 重定位
  - 合并相关.o文件
  - 确定每个定义符号地址
  - 将可执行文件中符号引用处的地址修改为重定位后的地址信息

main.c

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

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

swap.c

```
extern int buf[];

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

void swap()
{
    int temp;
    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```



哪些是符号定义？哪些是符号的引用？

局部变量temp分配在栈/寄存器中，不会在过程外被引用，因此不是符号定义

符号和符号解析

每个可重定位目标模块m都有一个符号表，它包含了在m中定义的符号。

有三种链接器符号：

- Global symbols（模块内部定义的全局符号）
  - 由模块m定义并能被其他模块引用的符号。例如，非static函数和非static的全局变量

    如，main.c中的全局变量名buf
- External symbols（外部符号，外部定义的全局符号）
  - 由其他模块定义并被模块m引用的全局符号

    如，main.c中的函数名swap
- Local symbols（本模块的局部符号）
  - 仅由模块m定义和引用的本地符号。例如，在模块m中定义的带static的函数和全局变量

    如，swap.c中的static变量名bufp1

    链接器局部符号不是指程序中的局部变量（分配在寄存器/栈中的临时性变量），链接器不关心局部变量

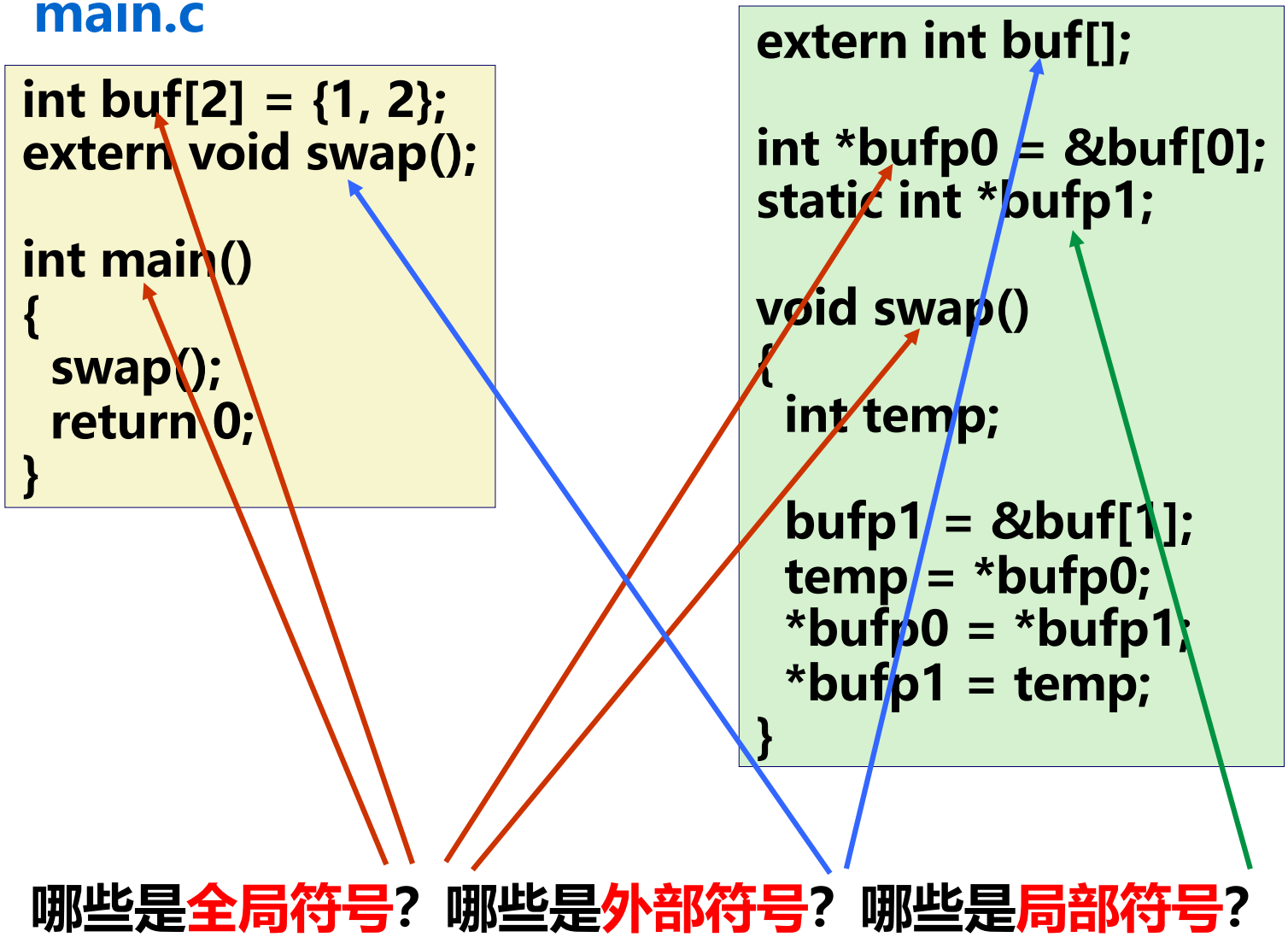
符号和符号解析

main.c

```
int buf[2] = {1, 2};
extern void swap();

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

swap.c

```
extern int buf[];

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

void swap()
{
  int temp;

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



哪些是**全局符号**？ 哪些是**外部符号**？ 哪些是**局部符号**？

目标文件中的符号表

.symtab 节记录符号表信息，是一个结构数组

函数名在text节中

变量名在data节或bss节中

- 符号表（symtab）中每个条目的结构如下：

```
typedef  struct {
        int    name;          /*符号对应字符串在strtab节中的偏移量*/
        char  type: 4,        /*符号对应目标的类型：数据、函数、源文件、节*/
              binding: 4;     /*符号类别：全局符号、局部符号*/
        char  reserved;
        short  section;       /*符号对应目标所在的节，或其他情况*/
        long   value;         /*在对应节中的偏移量，可执行文件中是虚拟地址*/
        long    size;         /*符号对应目标所占字节数*/
} Elf64_Symbol;
```

函数大小或变量长度

其他情况：

- **ABS 表示不该被重定位的符号；**
- **UNDEF 表示未定义（本模块引用，其他地方定义）；**
- **COMMON 表示未初始化数据（.bss），value表示对齐要求，size给出最小大小**

目标文件中的符号表

- **main.o中的符号表中三个条目** $ readelf -s main.o

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
3:	0000000000000000	28	FUNC	GLOBAL	DEFAULT	1	main
4:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	swap
5:	0000000000000000	8	OBJECT	GLOBAL	DEFAULT	3	buf

main是第1节（.text）偏移为0的符号，是全局函数，占28B

swap是main.o中未定义全局（在其他模块定义）符号，类型和大小未知

buf是main.o中第3节（.data）偏移为0的符号，是全局变量，占8B

- **swap.o中的符号表中4个条目**

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
4:	0000000000000000	8	OBJECT	LOCAL	DEFAULT	4	bufp1
5:	0000000000000000	42	FUNC	GLOBAL	DEFAULT	1	swap
6:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	buf
7:	0000000000000000	8	OBJECT	GLOBAL	DEFAULT	5	bufp0

bufp1是未分配地址且未初始化的局部符号(ndx=COM/.bss)，占8B

符号解析（Symbol Resolution）

- 目的：将每个模块中引用的符号与某个目标模块中的定义符号建立关联。
- 每个定义符号在代码段或数据段中都被分配了存储空间，将引用符号与定义符号建立关联后，就可在重定位时将引用符号的地址重定位为相关联的定义符号的地址。
- 局部（本地）符号在本模块定义并引用，其解析较简单，只要与本模块内唯一的定义符号关联即可。
- 全局符号（外部定义的、内部定义的）的解析涉及多个模块，故较复杂

“符号的定义”其实质是什么？

符号解析也称符号绑定

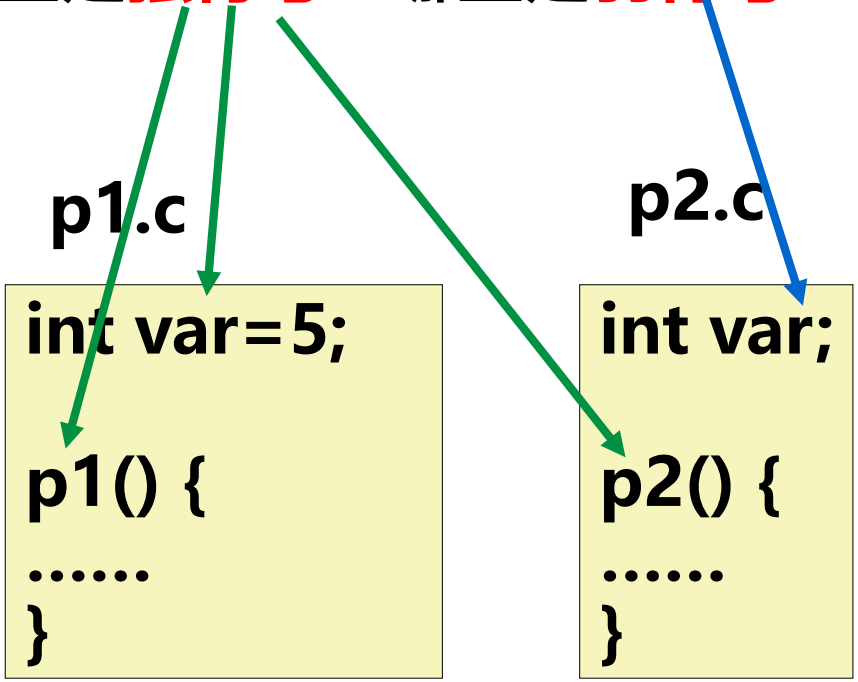
指被分配了存储空间。为函数名时，指代码所在区；为变量名时，指所占的静态数据区。

所有定义符号的值就是其目标所在的首地址

全局符号的符号解析

- 全局符号（定义）的强/弱特性
  - 函数名和已初始化的全局变量名是强符号
  - 未初始化的全局变量名是弱符号

以下符号哪些是强符号？哪些是弱符号？



全局符号的符号解析

以下符号哪些是强符号？哪些是弱符号？

main.c

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

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

此处为引用

swap.c

本地局部符号

```
extern int buf[];

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

void swap()
{
  int temp;

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

局部变量

链接器对符号的解析规则

符号解析时只能有一个确定的定义（即每个符号仅占一处存储空间）

- 多重定义符号的处理规则

  Rule 1: 强符号不能多次定义

  – 强符号只能被定义一次，否则链接错误

  Rule 2: 若一个符号被定义为一次强符号和多次弱符号，则按强定义为准

  – 对弱符号的引用被解析为其强定义符号

  Rule 3: 若有多个弱符号定义，则任选其中一个

  – 使用命令 gcc –fno-common链接时，会告诉链接器在遇到多个弱定义的全局符号时输出一条警告信息。

多重定义符号的解析举例

以下程序会发生链接出错吗?

```
int  x=10;
int  p1(void);
int main()
{
     x=p1();
     return x;
}
```

main.c

```
int  x=20;
int p1()
{
     return x;
}
```

p1.c

main只有一次强定义

p1只有一次强定义

x有两次强定义，所以，链接器将输出一条出错信息

多重定义符号的解析举例

以下程序会发生链接出错吗?

```
# include <stdio.h>
int  y=100;
int  z;
void  p1(void);
int main()
{
    z=1000;
    p1( );
    printf( "y=%d, z=%d\n" , y, z);
    return 0;
}
```

main.c

y一次强定义，一次弱定义
z两次弱定义
p1一次强定义
main一次强定义

```
int  y;
int  z;
void p1( )
{
    y=200;
    z=2000;
}
```

p1.c

问：打印结果是什么?

y=200，z=2000

在两个模块中定义相同的全局变量名，很可能发生意想不到的结果！

多重定义符号的解析举例

以下程序会发生链接出错吗?

```
1  #include <stdio.h>
2  int d=100;
3  int x=200;
4  void p1(void);
5  int main()
6  {
7      p1();
8      printf( "d=%d,x=%d\n" ,d,x);
9      return 0;
10 }
```

main.c

问：打印结果是什么?

d=0,x=1072693248

两个重复定义的变量具有不同类型时，更容易出现难以理解的结果！

p1.c

```
1  double d;
2
3  void p1()
4  {
5      d=1.0;
6  }
```

p1执行后d和x处内容是什么?

	0	1	2	3
&x	00	00	F0	3F
&d	00	00	00	00

1.0：0 01111111111 0...0B

=3FF0 0000 0000 0000H

多重定义符号的解析举例

main.c

```
.......
1  int d=100;
2  int x=200;
3  int main()
4  {
5    p1( );
6    printf ( “d=%d, x=%d\n” , d, x );
7    return 0;
8  }
```

p1.c

```
1  double d;
2
3  void p1( )
4  {
5    d=1.0;
6  }
```

double型数1.0对应的机器数
3FF0 0000 0000 0000H

X86-64是小端方式

	0	1	2	3
&x	00	00	F0	3F
&d	00	00	00	00

高

低

$2^{30}-1-(2^{20}-1)=2^{30}-2^{20}$

$=1024*1024*1023$

$=1\ 072\ 693\ 248$

打印结果：
d=0，x=1 072 693 248
Why?

多重定义全局符号的问题

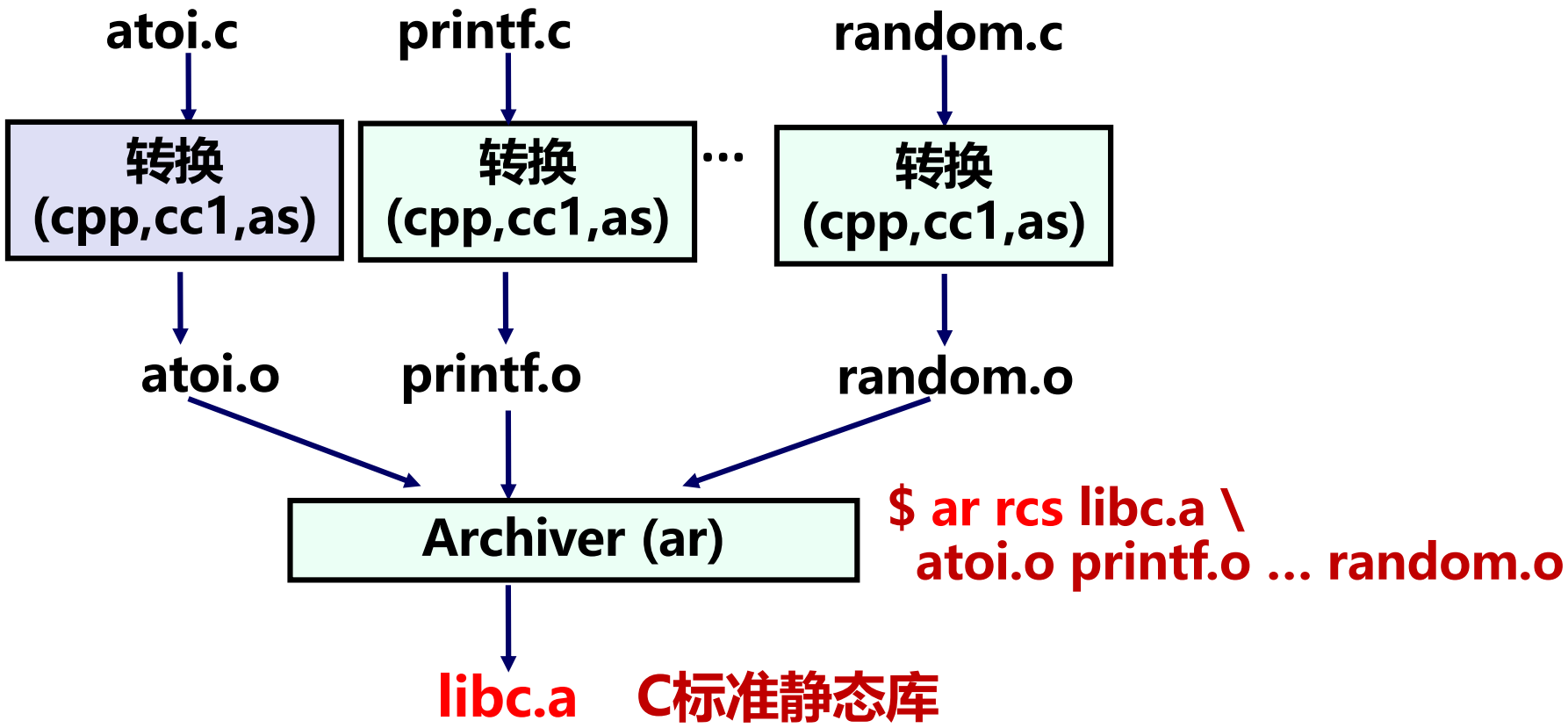
- 尽量避免使用全局变量
- 一定需要用的话，就按以下规则使用
  - 尽量使用本地变量（static）
  - 全局变量要赋初值
  - 外部全局变量要使用extern

多重定义全局变量会造成一些意想不到的错误，而且是默默发生的，编译系统不会警告，并会在程序执行（一段时间）后才能表现出来，且远离错误引发处。特别是在一个具有几百个模块的大型软件中，这类错误很难修正。

大部分程序员并不了解链接器如何工作，因而养成良好的编程习惯是非常重要的。

- 静态库 (.a archive files)
  - 将所有相关的目标模块（.o）打包为一个单独的库文件（.a），称为静态库文件，也称存档文件（archive）
  - 增强了链接器功能，使其能通过查找一个或多个库文件中的符号来解析符号
  - 在构建可执行文件时只需指定库文件名，链接器会自动到库中寻找那些应用程序用到的目标模块，并且只把用到的模块从库中拷贝出来
  - 在gcc命令行中无需显式指定C标准库libc.a(默认库)

静态库的创建



- Archiver（归档器）允许增量更新，只要重新编译需修改的源码并将其.o文件替换到静态库中。

在gcc命令行中无需显式指定C标准库libc.a(默认库)

自定义一个静态库文件

举例：将addvec.o和multvec.o打包生成libvector.a

addvec.c

```
int addcnt= 0
void addvec(int *x , int *y ,
int *z , int n)
{
  int i;
  addcnt++;
  for (i=0;i<n;i++)
        z[i]=x[i]+y[i];
}
```

multvec.c

```
int multcnt= 0
void multvec(int *x , int *y ,
int *z , int n)
{
  int i;
  multcnt++;
  for (i=0;i<n;i++)
        z[i]=x[i]*y[i];
}
```

```
$ gcc –c addvec.c multvec.c
$ ar rcs libvector.a addvec.o multvec.o
```

自定义一个静态库文件

main2.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;
}
```

```
$ gcc –c main2.c
$ gcc -static -o prog2c main2.o ./libvector.a
```

问：如何进行符号解析?

链接器中符号解析的全过程

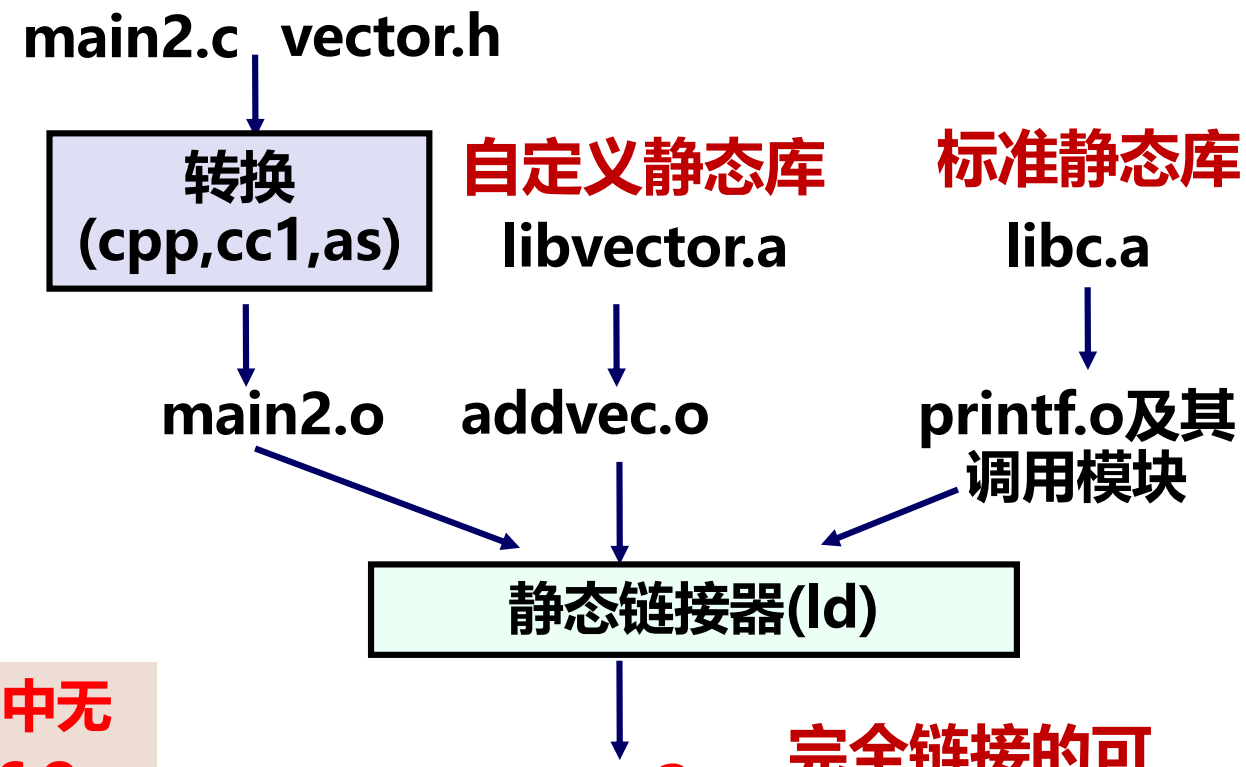
$ gcc –c main2.c
$ gcc -static -o prog2c main2.o ./libvector.a

维护三个集合：
E 将被合并以组成可执行文件的所有目标文件集合
U 当前所有未解析的引用符号的集合
D 当前所有定义符号的集合

1. 开始E、U、D为空，首先扫描main2.o，把它加入E，同时把addvec, printf加入U，main加入D。
2. 接着扫描到libvector.a，将U中所有符号（addvec）与libvectoa中所有目标模块（addvec.o和multvec.o）依次匹配，发现在addvec.o中定义了addvec，故addvec.o加入E，addvec从U转移到D，同时addvec中的符号定义addcnt也加入D。
3. 不断在libvector.a的各模块上进行迭代以匹配U中的符号，直到U、D都不再变化。
4. 此时U中只有一个未解析符号printf，而D中有main、addvec、addcnt。因为模块multvec.o没有被加入E中，因而它被丢弃。
5. 接着，扫描默认的库文件libc.a，发现其目标模块printf.o定义了printf，于是printf也从U移到D，并将printf.o加入E，同时把它定义的所有符号加入D，其中未解析符号加入U。
6. 重复直到处理完时，U一定是空的，且D中符号唯一。

链接器中符号解析的全过程

```
$ ar rcs libvector.a addvec.o multvec.o
$ gcc -static -o prog2c main2.o ./libvector.a
```



注意：E中无multvec.o

解析结果：

E中有main2.o、addvec.o、printf.o及其调用的模块

D中有main、addvec、addcnt、printf及其引用的符号

链接器中符号解析的全过程

若命令为：$ gcc –static –o progc ./libvector.a main2.o， 结果怎样?

首先扫描libvector.a，根据其中是否存在U中未解析符号对应的定义符号来确定哪个.o被加入E。因为U中一开始为空，所以libvector.a中的addvec.o和multvec.o都被丢弃。

然后，扫描main2.o，将addvec加入U，直到最后它都不能被解析。

链接器报错!

Why?

它只能用libvector.a中符号来解析，

而libvector.a中两个.o模块都已被丢弃!

使用静态库

- 链接器对外部引用的解析算法要点如下:
  - 按照命令行给出的顺序扫描.o和.a文件
  - 扫描期间将当前未解析的引用记录到一个集合U中
  - 每遇到一个新的.o或.a中的模块，都试图用其来解析U中的符号
  - 如果扫描到最后，U中还有未被解析的符号，则发生错误
- 问题和对策
  - 能否正确解析与命令行给出的顺序有关
  - 一般将静态库放在命令行的最后

链接顺序问题

- 假设调用关系如下：
  func.o → libx.a 和 liby.a 中的函数
  libx.a → libz.a 中的函数
  libx.a 和 liby.a 之间、liby.a 和 libz.a 相互独立
  则以下几个命令行都是可行的：
  - gcc -static –o myfunc func.o libx.a liby.a libz.a
  - gcc -static –o myfunc func.o liby.a libx.a libz.a
  - gcc -static –o myfunc func.o libx.a libz.a liby.a
- 假设调用关系如下：
  func.o → libx.a 和 liby.a 中的函数
  libx.a → liby.a 同时 liby.a → libx.a
  则以下命令行可行：
  - gcc -static –o myfunc func.o libx.a liby.a libx.a
  - gcc -static –o myfunc func.o liby.a libx.a liby.a

重定位

符号解析完成后，可进行重定位工作，分三步

- 合并相同的节
  - 将集合E的所有目标模块中相同的节合并成新节

    例如，所有.text节合并作为可执行文件中的.text节
- 对定义符号进行重定位（确定地址）
  - 确定新节中所有定义符号在虚拟地址空间中的地址

    例如，为函数确定首地址，进而确定每条指令的地址，为变量确定首地址
  - 完成这一步后，每条指令和每个全局变量都可确定地址
- 对引用符号进行重定位（确定地址）
  - 修改.text节和.data节中对每个符号的引用（地址）

    需要用到在.rel.data和.rel.text节中保存的重定位信息

重定位操作举例

main.c

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

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

swap.c

```
extern int buf[];

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

void swap()
{
   int temp;
   bufp1 = &buf[1];
   temp = *bufp0;
   *bufp0 = *bufp1;
   *bufp1 = temp;
}
```

符号解析后的结果是什么?

E中有main.o和swap.o两个模块！D中有所有符号定义！

在main.o和swap.o的重定位节(.rel.text、.rel.data)中有重定位信息，反映符号引用的位置、绑定的定义符号名、重定位类型

用命令readelf -r main.o可显示main.o中的重定位条目（表项）

```
readelf -r main.o 可重定位信息在可重定位目标（.o）文件中

Relocation section '.rela.text' at offset 0x190 contains 1 entry:
Offset          Info           Type             Sym.Value    Sym.Name + Addend
00000000000e  000400000004  R_X86_64_PLT32  00000…0000   swap      - 4
```

.rel.text存储.text中需要重定位的指令地址（call printf，printf地址未知）
.rel.data存储.data或.bss中需要重定位的变量地址（extern int var，var地址未知）

ELF重定位条目的结构如下：

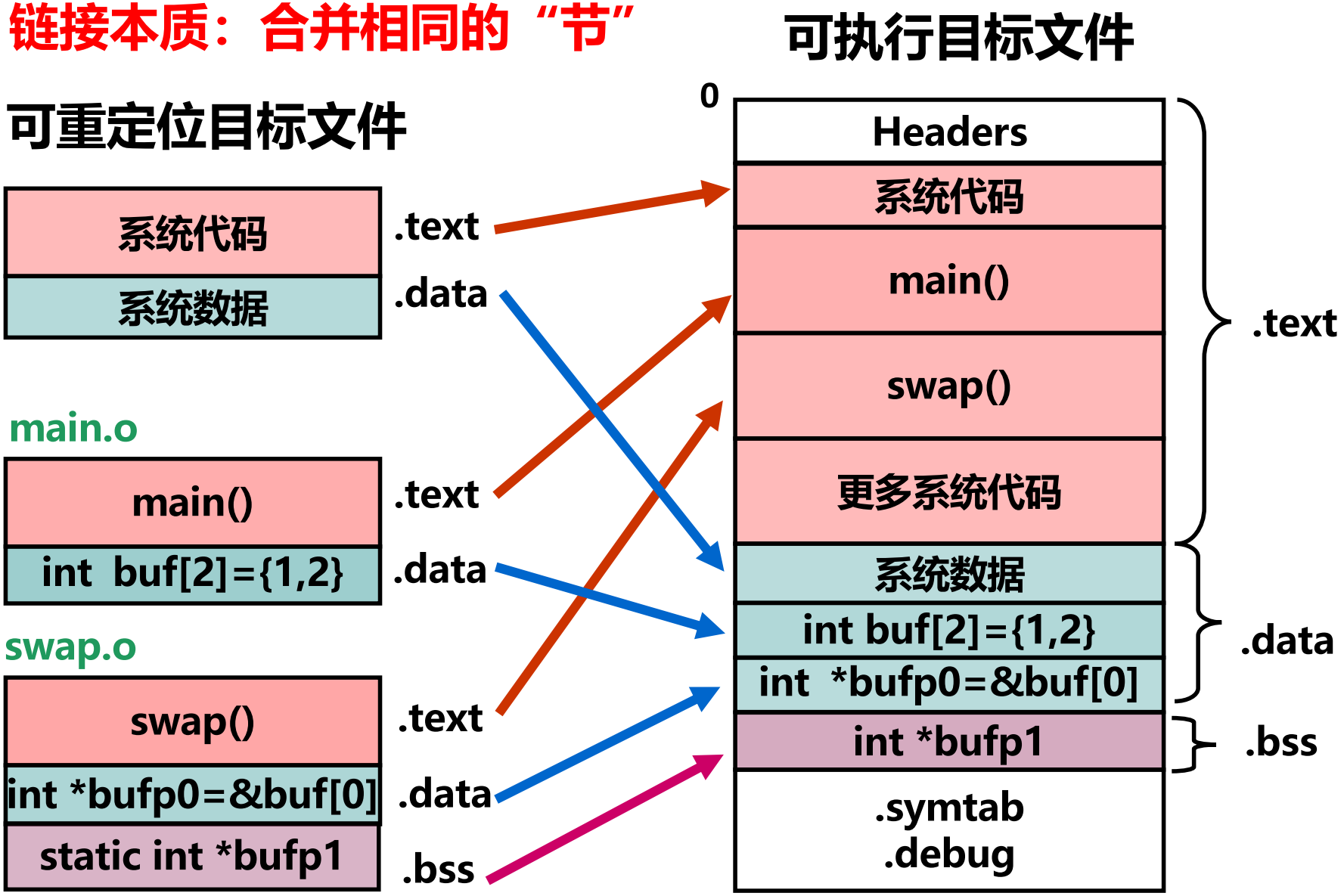
```
typedef struct {
  long offset; // 需要被修改的节偏移
  int  type;   // 重定位类型
  int  symbol; // 被修改引用指向的符号
  long addend; // 重定位使用的调整偏移值
} Elf64_Rela;
```

重定位类型：

- **R_X86_64_64(32)：** 重定位一个使用64(32)位绝对地址的引用
- **R_X86_64_PC32：** 重定位一个使用32位PC相对地址的引用
- **R_X86_64_PLT32：** 过程链接 延迟绑定

符号引用的地址需要重定位

链接本质：合并相同的“节”



虽然bufp1是swap的本地符号，也需在.bss节重定位

main.o重定位前

main.c

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

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

main的定义在.text节中偏移为0处开始，占0x1C字节。

main.o

```
Disassembly of section .text:
00000000 <main>:
 0:   f3 0f 1e fa          endbr64
 4:   48 83 ec 08          sub    $0x8,%rsp
 8:   b8 00 00 00 00       mov    $0x0,%eax
 d:   e8 00 00 00 00       call   12 <main+0x12>
                e: R_X86_64_PLT32      swap-0x4
12: b8 00 00 00 00         mov    $0x0,%eax
17: 48 83 c4 08            add    $0x8,%rsp
1b: c3                     ret
```

在rel_text节中的重定位条目为：
r_offset=0xe, r_sym=4, r_type=R_X86_64_PLT32, swap-4

```
Disassembly of section .data:

00000000 <buf>:
  0:   01 00 00 00 02 00 00 00
```

buf的定义在.data节中偏移为0处开始，占8B。

main.o中的符号表

- main.o中的符号表中最后三个条目

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
3:	0	28	FUNC	GLOBAL	DEFAULT	1	main
4:	0	0	NOTYPE	GLOBAL	DEFAULT	UND	swap
5:	0	8	OBJECT	GLOBAL	DEFAULT	3	buf

swap是main.o的符号表中第4项，是未定义符号，类型和大小未知，并是全局符号，故在其他模块中定义。

相对地址重定位方式

- 假定：
  - 可执行文件中ma
  - swap紧跟main后
- 则swap起始地址
  - 0x401106+0x1C=0x401122
- 则重定位后call指令的机器代码是什么？
  - 转移目标地址=PC(当前指令地址)+偏移量(重定位值)
  - PC= **ADDR(s) +r.offset – r.addend**
    = 0x401106+0x0e-(-4) =0x401118
  - 重定位值=转移目标地址-PC=0x4x1122-0x401118=0xa
  - call指令的机器代码为“e8 0a 00 00 00”

```
Disassembly of section .text:
00000000 <main>:
   ......
d:  e8 00 00 00 00   call   12 <main+0x12>
            e: R_X86_64_PLT32 swap-0x4
12: b8 00 00 00 00       mov    $0x0,%eax
```

0x401118 (call-PC)
- 0x401106 (main)=12

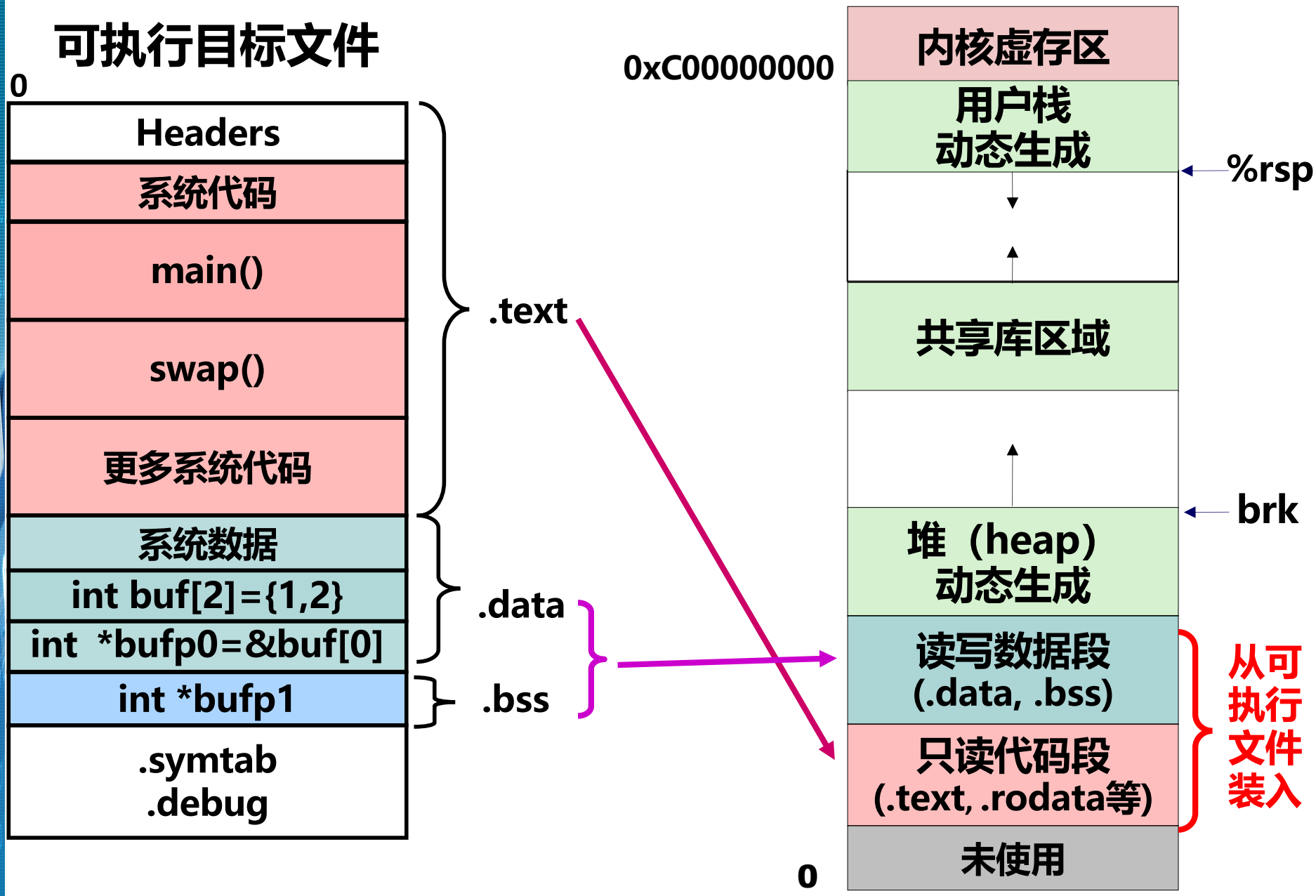
main.o中text节的地址

PC相对地址方式下，重定位值计算公式为：

ADDR(r.sym) – ((ADDR(.text) + r.offset) – r.addend)

引用目标处地址　　**call指令下条指令地址**　　**即当前PC的值**

确定定义符号的地址


可执行目标文件
0
Headers
系统代码
main()
swap()
更多系统代码
系统数据
int buf[2]={1,2}
int *bufp0=&buf[0]
int *bufp1
.symtab
.debug
.text
.data
.bss
0xC0000000
内核虚存区
用户栈
动态生成
%rsp
共享库区域
brk
堆（heap）
动态生成
读写数据段
(.data, .bss)
只读代码段
(.text, .rodata等)
未使用
0
从可执行文件装入

绝对地址的重定位方式

main.o中.data和.rel.data节内容

```
Disassembly of section .data:

00000000 <buf>:
  0:   01 00 00 00 02 00 00 00
```

buf定义在.data节中偏移为0处，占8B，没有需重定位的符号。

main.c

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

int main()
……
```

swap.o中.data和.rel.data节内容

```
Disassembly of section .data:

00000000 <bufp0>:
  0:  00 00 00 00
        0: R_x86_64_64  buf
```

bufp0定义在.data节中偏移为0处，占8B，初值为0x0

swap.c

```
extern int buf[];

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

void swap()
……
```

重定位节.rel.data中有一个重定位表项：

r_offset=0x0, r_sym=6, r_type=R_x86_64_64 buf+0

绝对地址的重定位方式

- **swap.o中的符号表中最后4个条目** readelf –s swap.o

```
Num: Value              Size  Type    Bind   Vis      Ndx Name
4:   0000000000000000   8     OBJECT  LOCAL  DEFAULT    4 bufp1
5:   0000000000000000   42    FUNC    GLOBAL DEFAULT    1 swap
6:   0000000000000000   0     NOTYPE  GLOBAL DEFAULT  UND buf
7:   0000000000000000   8     OBJECT  GLOBAL DEFAULT    5 bufp0
```

buf是swap.o的符号表中第6项，是未定义符号，类型和大小未知，并是全局符号，故在其他模块中定义。

重定位节.rel.data中有一个重定位表项：
r_offset=0x0, r_sym=6, r_type=R_x86_64_64 buf+0

绝对地址的重定位方式

- 假定：合并后buf的存储地址ADDR(buf)=0x404028
- 则重定位后，bufp0的地址及内容变为什么？
  - buf和bufp0同属于.data节，故在可执行文件中它们被合并
  - bufp0紧接在buf后，故地址为0x404028+8=0x404030
  - 因是R_x86_64_64方式，故bufp0内容为buf的绝对地址0x404028，即“28 40 40 00 00 00 00 00”

PC绝对地址方式下，重定位值计算公式为：ADDR(r.sym) + r.addend

可执行目标文件中.data节的内容

```
Disassembly of section .data:

404028 <buf>:
  404028:                01 00 00 00 02 00 00 00

404030 <bufp0>:
  404030:                28 40 40 00 00 00 00 00
```

swap.o重定位

objdump –dx swap.o

共有5处需要重定位

划红线处：7、e、15、1d、25

swap.o中.text节内容

```
000000000000000 <swap>:
   0:    f3 0f 1e fa          endbr64
   4:    48 8d 05 00 00 00 00     lea    0x0(%rip),%rax        //&buf[1]
                  7: R_X86_64_PC32        buf
   b:    48 89 05 00 00 00 00     mov    %rax,0x0(%rip)    //bufp1 ←&buf[1]
                        e: R_X86_64_PC32        .bss -0x4
  12:    48 8b 05 00 00 00 00     mov    0x0(%rip),%rax        // bufp0
                        15: R_X86_64_PC32       bufp0 -0x4
  19:    8b 10                    mov    (%rax),%edx           //temp ← *bufp0
  1b:    8b 0d 00 00 00 00        mov    0x0(%rip),%ecx        // *bufp1
                  1d: R_X86_64_PC32       buf
  21:    89 08                    mov    %ecx,(%rax)           //*bufp0 = *bufp1
  23:    89 15 00 00 00 00        mov    %edx,0x0(%rip)        //*bufp1 = temp
                        25: R_X86_64_PC32       buf
  29:    c3                       ret
```

swap.c

```
extern int buf[];
int *bufp0 = &buf[0];
static int *bufp1;
void swap(){
   int temp;
   bufp1 = &buf[1];
   temp = *bufp0;
   *bufp0 = *bufp1;
   *bufp1 = temp;}
```

swap.o中.data和.rel.data节内容　　readelf –r swap.o

```
Relocation section '.rela.text' at offset 0x1d8 contains 5 entries:
Offset          Info           Type            Sym.Value        Sym.Name + Addend
000000000007 000600000002 R_X86_64_PC32  0000000000000000 buf     + 0
00000000000e 000300000002 R_X86_64_PC32  0000000000000000 .bss    - 4
000000000015 000700000002 R_X86_64_PC32  0000000000000000 bufp0   - 4
00000000001d 000600000002 R_X86_64_PC32  0000000000000000 buf     + 0
000000000025 000600000002 R_X86_64_PC32  0000000000000000 buf     + 0

Relocation section '.rela.data' at offset 0x250 contains 1 entry:
Offset          Info           Type            Sym.Value        Sym.Name + Addend
000000000000 000600000001 R_X86_64_64 0000000000000000 buf      + 0
```

buf,bufp0,bufp1地址分别是0x404028,0x404030,0x404040

PC相对地址方式下，重定位值计算公式为：
ADDR(r.sym) – ((ADDR(.text) + r.offset) – r.addend)

&buf[1](7处重定位值) =0x404028-(0x401122+7-0)=0x2eff　　ff 2e 00 00
bufp1(e处重定位值)=0x404040-(0x401122+0xe-(-4))=0x2f0c　　0c 2f 00 00

swap.o中.text节的地址

```
0000000000000000 <swap>:
   0:    f3 0f 1e fa          endbr64
   4:    48 8d 05 ff 2e 00 00        lea    0x0(%rip),%rax        //&buf[1]
                  7: R_X86_64_PC32          buf
   b:    48 89 05 0c 2f 00 00        mov    %rax,0x0(%rip)    //bufp1 ←&buf[1]
                        e: R_X86_64_PC32          .bss-0x4
  12:    48 8b 05 00 00 00 00        mov    0x0(%rip),%rax        // bufp0
                        15: R_X86_64_PC32         bufp0-0x4
  19:    8b 10                       mov    (%rax),%edx           //temp ← *bufp0
  1b:    8b 0d 00 00 00 00           mov    0x0(%rip),%ecx        // *bufp1
                  1d: R_X86_64_PC32         buf
  21:    89 08                       mov    %ecx,(%rax)           //*bufp0 = *bufp1
  23:    89 15 00 00 00 00           mov    %edx,0x0(%rip)        //*bufp1 = temp
                        25: R_X86_64_PC32         buf
  29:    c3                          ret
```

buf,bufp0,bufp1地址分别是0x404028,0x404030,0x404040

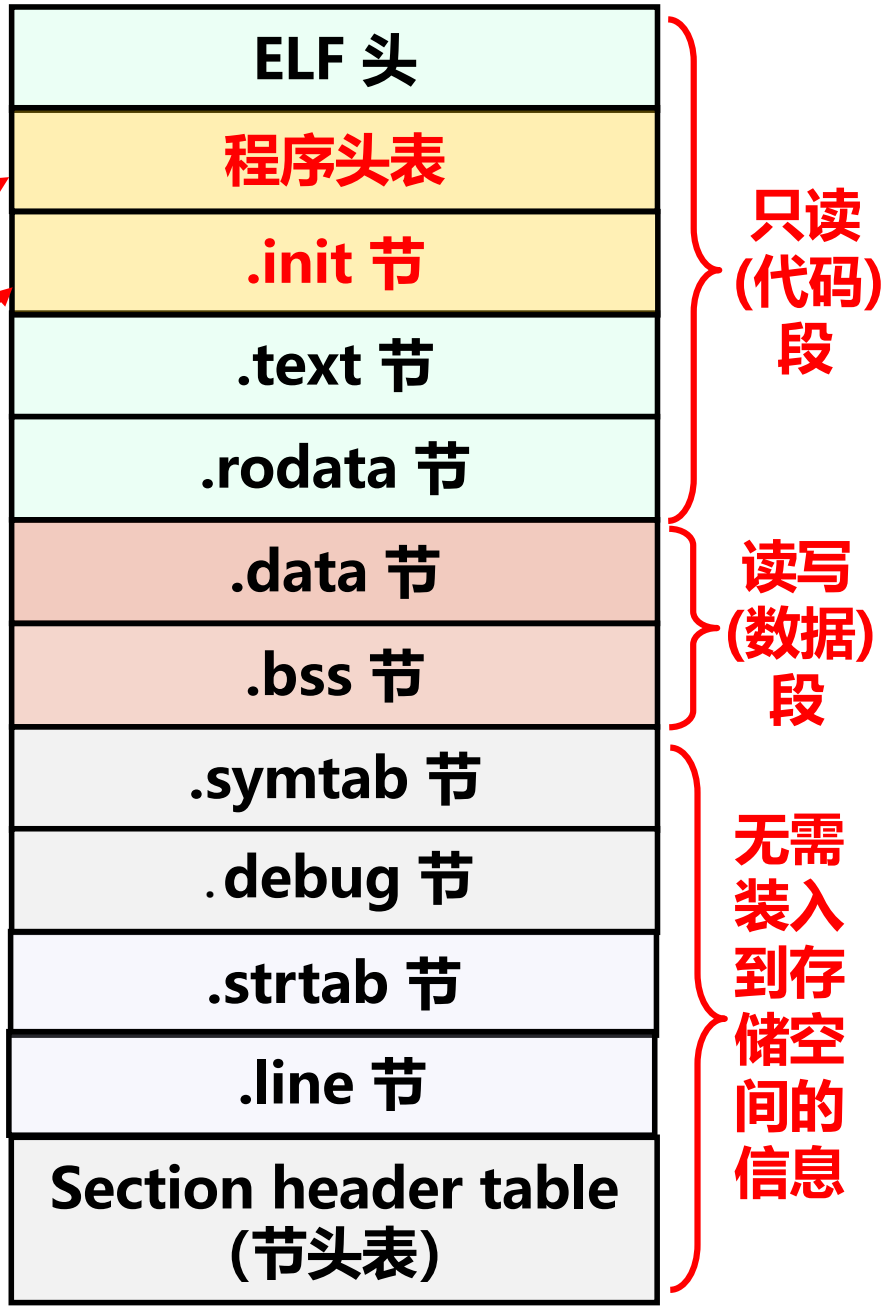
PC相对地址方式下，重定位值计算公式为：
ADDR(r.sym) – ((ADDR(.text) + r.offset) – r.addend)

bufp0 (15处重定位值)=0x404030-(0x401122+0x15-(-4))=2ef5
bufp1 (1d处重定位值)=0x404040-(0x401122+0x1d-0)=2f01
bufp1 (25处重定位值)=0x404040-(0x401122+0x25-0)=2ef9

```
0000000000000000 <swap>:
   0:   f3 0f 1e fa             endbr64
   4:   48 8d 05 ff 2e 00 00    lea    0x0(%rip),%rax        //&buf[1]
                7: R_X86_64_PC32        buf
   b:   48 89 05 0c 2f 00 00    mov    %rax,0x0(%rip)    //bufp1 ←&buf[1]
                e: R_X86_64_PC32        .bss-0x4
  12:   48 8b 05 f5 2e 00 00    mov    0x0(%rip),%rax        // bufp0
                15: R_X86_64_PC32       bufp0-0x4
  19:   8b 10                   mov    (%rax),%edx           //temp ← *bufp0
  1b:   8b 0d 2f 01 00 00       mov    0x0(%rip),%ecx        // *bufp1
                1d: R_X86_64_PC32       buf
  21:   89 08                   mov    %ecx,(%rax)           //*bufp0 = *bufp1
  23:   89 15 f9 2e 00 00       mov    %edx,0x0(%rip)        //*bufp1 = temp
                25: R_X86_64_PC32       buf
  29:   c3                      ret
```

可执行目标文件格式

- 与可重定位文件稍有不同：
  - ELF头中字段e_entry给出执行程序时第一条指令的地址，而在可重定位文件中，此字段为0
  - 多一个程序头表，也称段头表（segment header table），是一个结构数组
  - 多一个.init节，用于定义_init函数，该函数用来进行可执行目标文件开始执行时的初始化工作
  - 少两个.rel节（无需重定位）



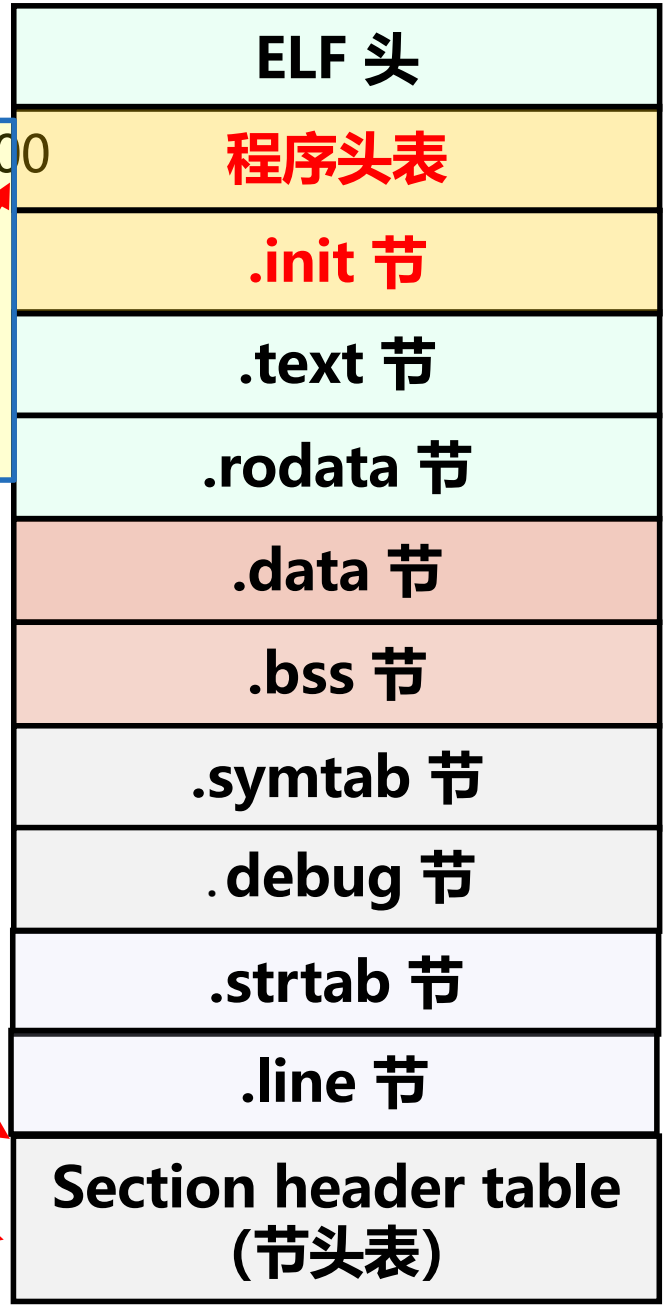
ELF头信息举例

$ readelf -h main　　可执行目标文件的ELF头

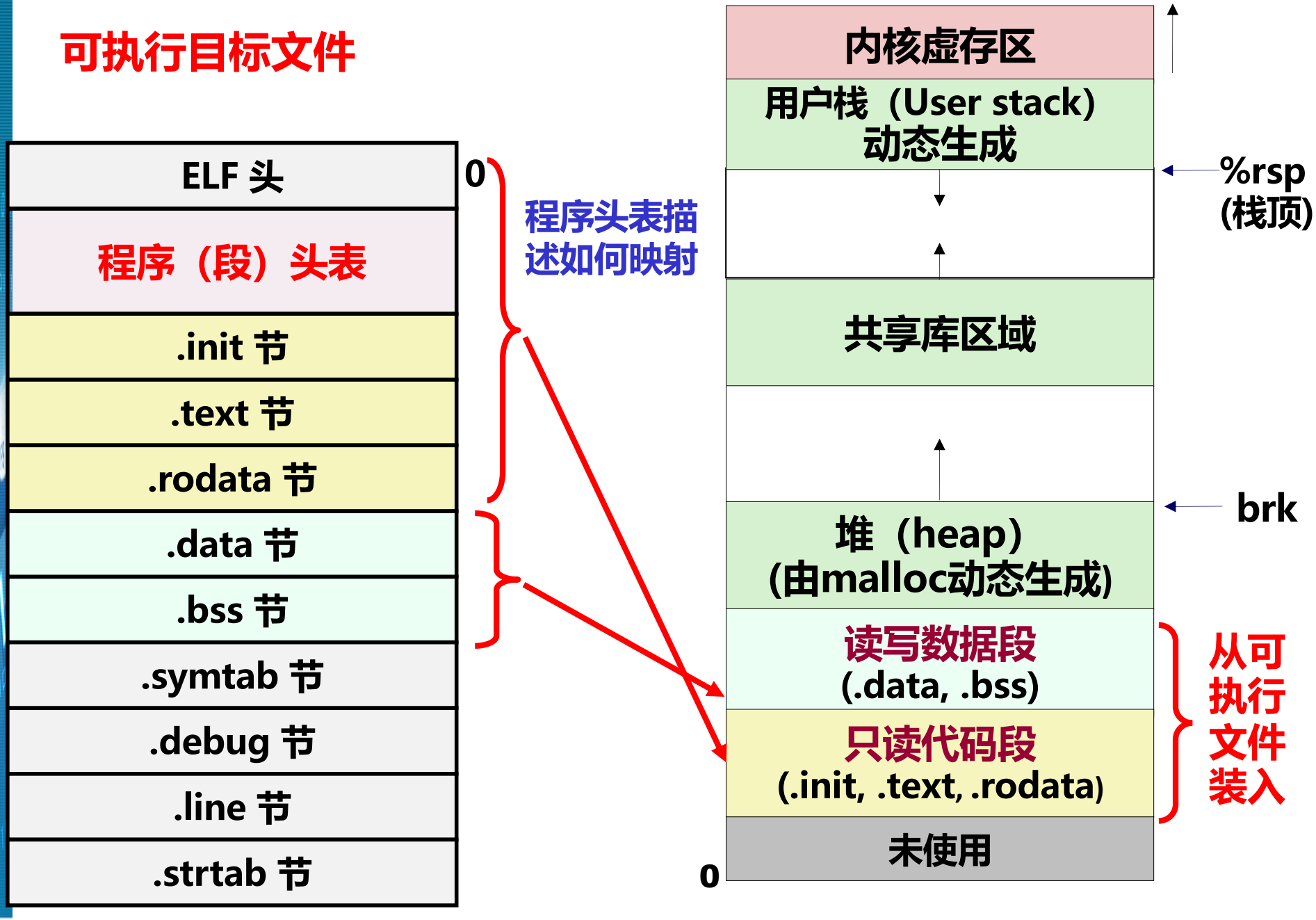
```
ELF Header:
  Magic:7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x401020
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14000 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         28
  Section header string table index: 27
```

13x56B

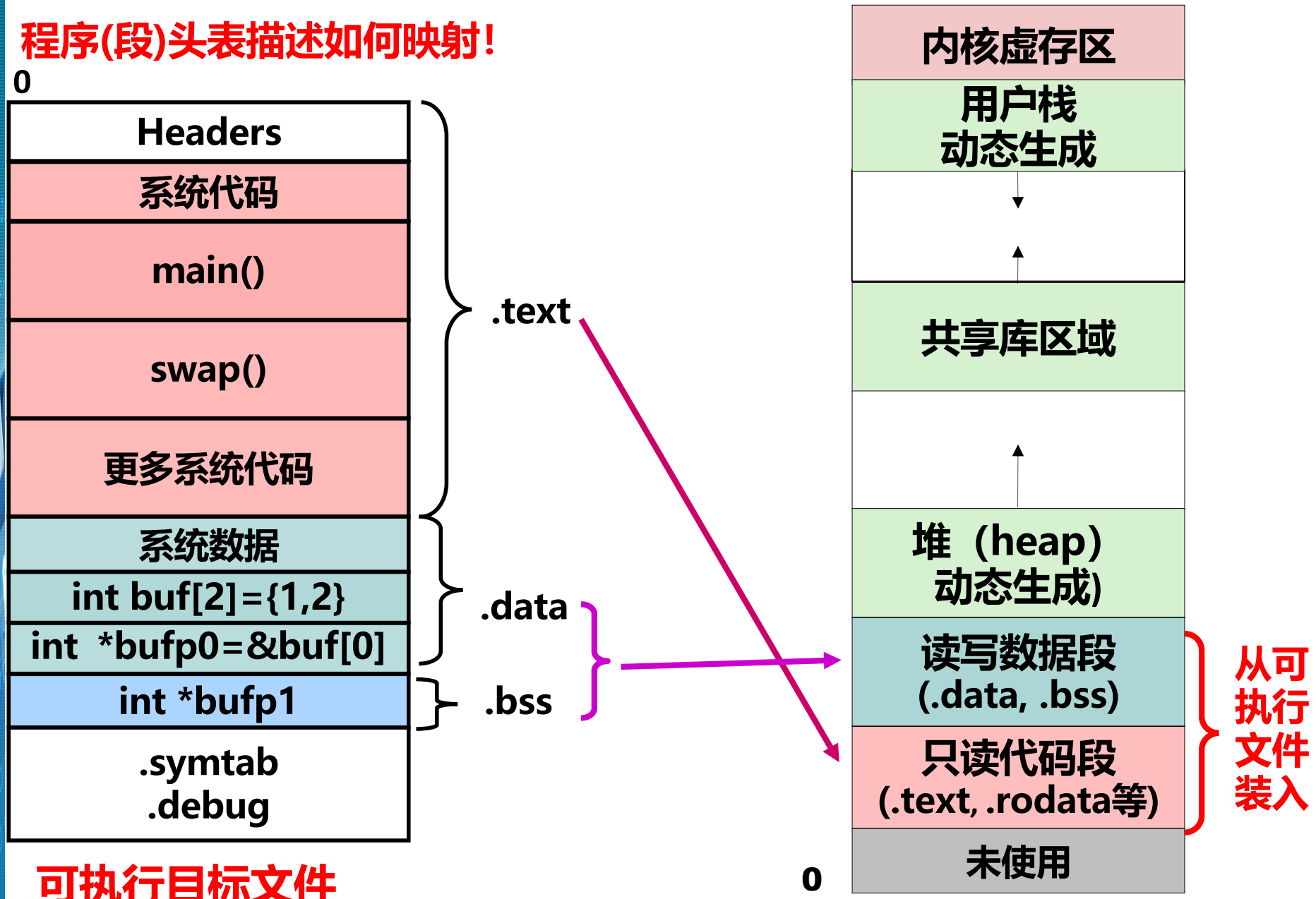
28x64B



可执行目标文件


ELF 头
程序（段）头表
.init 节
.text 节
.rodata 节
.data 节
.bss 节
.symtab 节
.debug 节
.line 节
.strtab 节
0
程序头表描述如何映射
内核虚存区
用户栈（User stack）
动态生成
%rsp
(栈顶)
共享库区域
brk
堆（heap）
(由malloc动态生成)
读写数据段
(.data, .bss)
只读代码段
(.init, .text, .rodata)
从可执行文件装入
未使用
0

程序(段)头表描述如何映射!



可执行目标文件

可执行文件中的程序头表

程序头表描述可执行文件中的节与虚拟空间中的存储段之间的映射关系

一个表项（56B）说明虚拟地址空间中一个连续的段或一个特殊的节

以下是某可执行目标文件程序头表信息

有13个表项，其中4个为可装入段（即Type=LOAD）

```
Type           Offset             VirtAddr           PhysAddr
               FileSiz            MemSiz              Flags  Align
  PHDR         0x0000000000000040 0x0000000000400040 0x0000000000400040
               0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP       0x0000000000000318 0x0000000000400318 0x0000000000400318
               0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD         0x0000000000000000 0x0000000000400000 0x0000000000400000
               0x00000000000004a8 0x00000000000004a8  R      0x1000
  LOAD         0x0000000000001000 0x0000000000401000 0x0000000000401000
               0x0000000000000159 0x0000000000000159  R E    0x1000
  LOAD         0x0000000000002000 0x0000000000402000 0x0000000000402000
               0x00000000000000a0 0x00000000000000a0  R      0x1000
  LOAD         0x0000000000002e50 0x0000000000403e50 0x0000000000403e50
               0x00000000000001e8 0x00000000000001f8  RW     0x1000
```

- 两种视图
  - 链接视图（被链接）：Relocatable object files
  - 执行视图（被执行）：Executable object files

链接视图

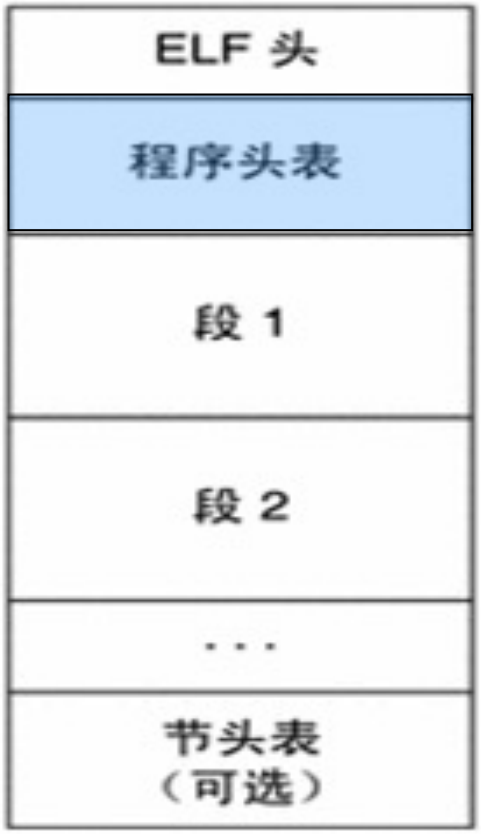
节（section）是ELF 文件中具有相同特征的最小可处理单位

.text节: 代码

.data节: 数据

.rodata: 只读数据

.bss: 未初始化数据

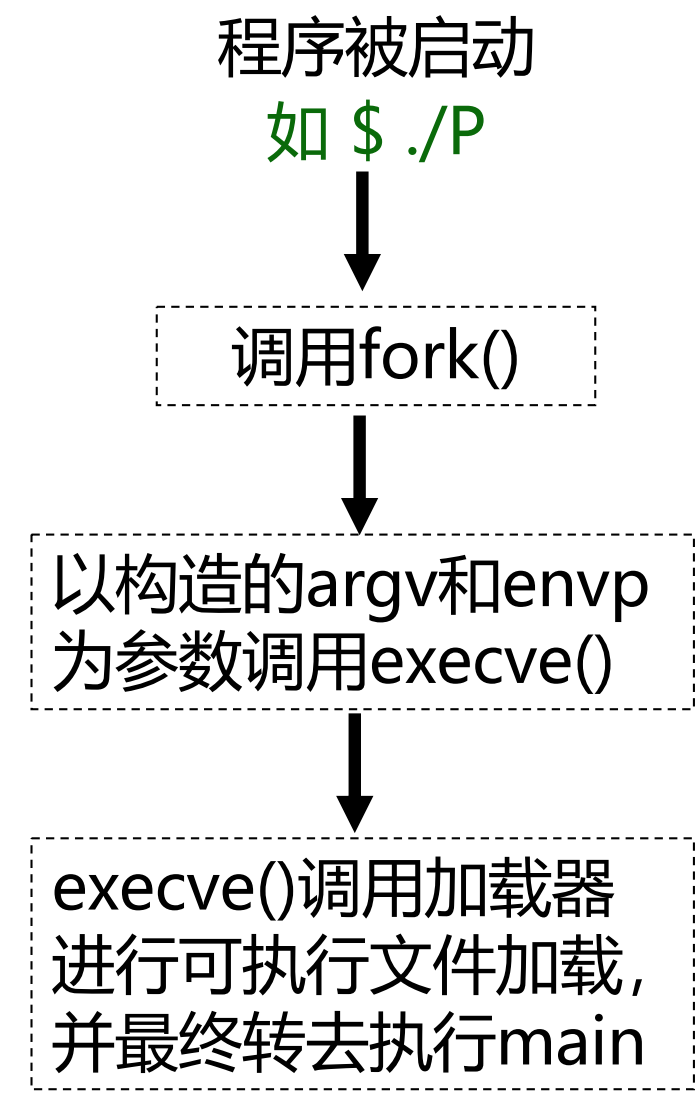


执行视图

由不同的段（segment）组成，描述节如何映射到存储段中，可多个节映射到同一段，如：可合并.data节和.bss节,并映射到一个可读可写数据段中

可执行文件的加载

- 通过调用execve系统调用函数来调用加载器
- 加载器（loader）根据可执行文件的程序（段）头表中的信息，将可执行文件的代码和数据从磁盘“拷贝”到存储器中（实际上不会真正拷贝，仅建立一种映像，后续课程）
- 加载后，将PC（RIP）设定指向Entry Point (即符号_start处)，最终执行main函数，以启动程序执行。



程序的链接

- 分以下三个部分介绍
  - 第一讲：目标文件格式
    - 程序的链接概述、链接的意义与过程
    - ELF目标文件、重定位目标文件格式、可执行目标文件格式
  - 第二讲：符号解析与重定位
    - 符号和符号表、符号解析
    - 与静态库的链接
    - 重定位信息、重定位过程
    - 可执行文件的加载
  - 第三讲：动态链接
    - 动态链接的特性、程序加载时的动态链接、程序运行时的动态链接、动态链接举例

动态链接的共享库（Shared Libraries）

- 静态库有一些缺点：
  - 库函数（如printf）被包含在每个运行进程的代码段中，对于并发运行上百个进程的系统，造成极大的主存资源浪费
  - 库函数（如printf）被合并在可执行目标中，磁盘上存放着数千个可执行文件，造成磁盘空间的极大浪费
  - 程序员需关注是否有函数库的新版本出现，并须定期下载、重新编译和链接，更新困难、使用不便

- 解决方案: Shared Libraries （共享库）
  - 是一个目标文件，包含有代码和数据
  - 从程序中分离出来，磁盘和内存中都只有一个备份
  - 可以动态地在装入时或运行时被加载并链接
  - Window称其为动态链接库（Dynamic Link Libraries，.dll文件）
  - Linux称其为动态共享对象（Dynamic Shared Objects, .so文件）

共享库（Shared Libraries）

动态链接可以按以下两种方式进行：

- 在第一次加载并运行时进行 (load-time linking)
  - Linux通常由动态链接器(ld-linux.so)自动处理
  - 标准C库 (libc.so) 通常按这种方式动态被链接
- 在已经开始运行后进行(run-time linking).
  - 在Linux中，通过调用dlopen()等接口来实现
    - 分发软件包、构建高性能Web服务器等

在内存中只有一个备份，被所有进程共享（调用），节省内存空间

一个共享库目标文件被所有程序共享链接，节省磁盘空间

共享库升级时，被自动加载到内存和程序动态链接，使用方便

共享库可分模块、独立、用不同编程语言进行开发，效率高

第三方开发的共享库可作为程序插件，使程序功能易于扩展

自定义一个动态共享库文件

PIC：Position Independent Code 位置无关代码

1）保证共享库代码的位置可以是不确定的

2）即使共享库代码的长度发生变化，也不会影响调用它的程序

addvec.c

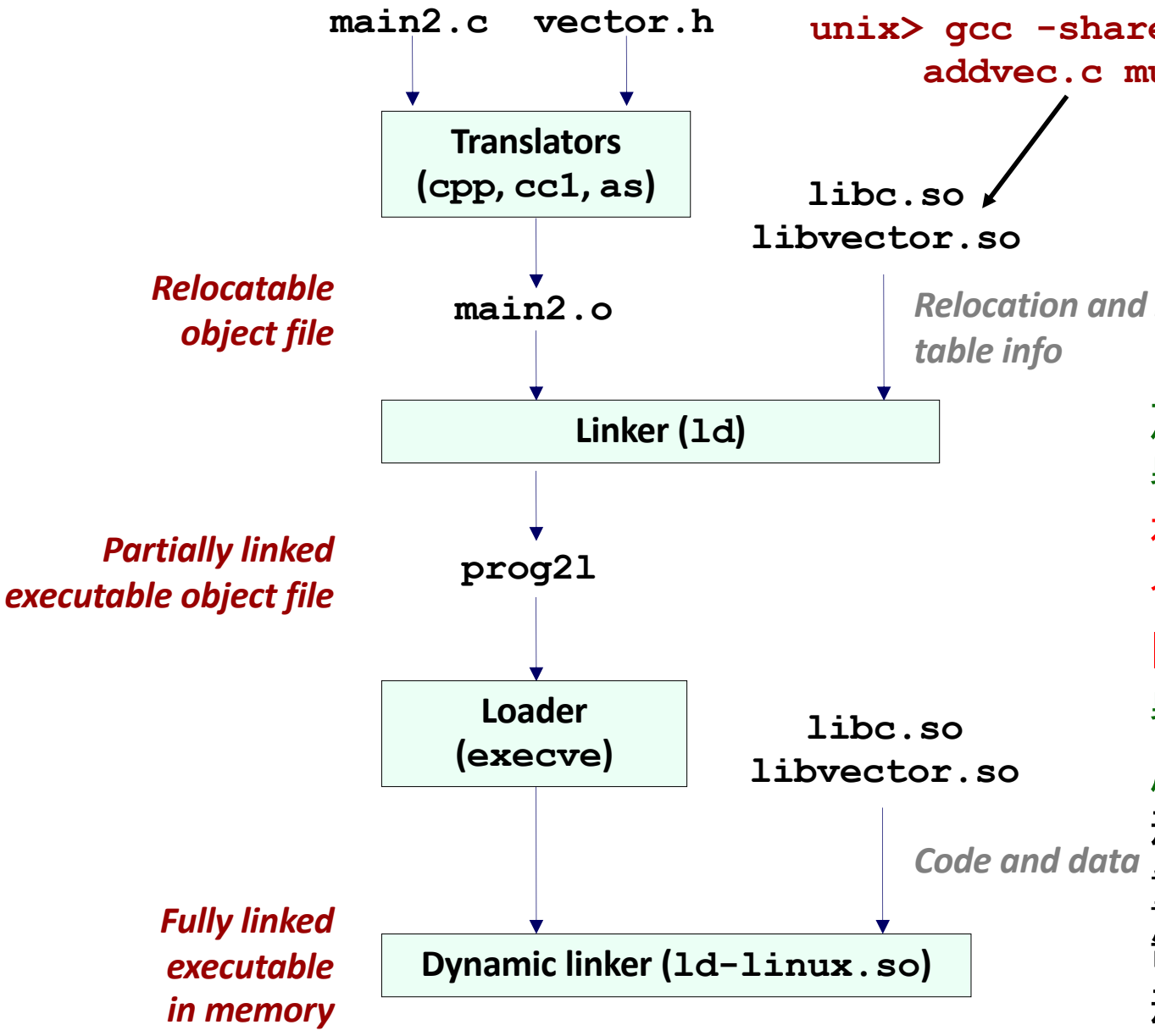
```
int addcnt= 0
void addvec(int *x , int *y ,
int *z , int n)
{
  int i;
  addcnt++;
  for (i=0;i<n;i++)
        z[i]=x[i]+y[i];
}
```

multvec.c

```
int multcnt= 0
void multvec(int *x , int *y ,
int *z , int n)
{
  int i;
  multcnt++;
  for (i=0;i<n;i++)
        z[i]=x[i]*y[i];
}
```

```
gcc –shared –fPIC –o libvector.so addvec.c multvec.c
gcc –o prog2l main2.c ./libvector.so
```

加载时动态链接



加载prog2l时，加载器发现在其程序头表中有.interp段，其中包含了动态链接器路径名ld-linux.so，因而加载器根据指定路径加载并启动动态链接器运行。动态链接器完成相应的重定位工作后，再把控制权交给prog2l，启动其第一条指令执行。

加载时动态链接

- 程序头表中有一个特殊的段：INTERP
- 其中记录了动态链接器目录及文件名ld-linux.so

```
Type           Offset             VirtAddr           PhysAddr
               FileSiz            MemSiz             Flags  Align
  PHDR         0x0000000000000040 0x0000000000400040 0x0000000000400040
               0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP       0x0000000000000318 0x0000000000400318 0x0000000000400318
               0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD         0x0000000000000000 0x0000000000400000 0x0000000000400000
               0x00000000000004a8 0x00000000000004a8  R      0x1000
  LOAD         0x0000000000001000 0x0000000000401000 0x0000000000401000
               0x0000000000000159 0x0000000000000159  R E    0x1000
  LOAD         0x0000000000002000 0x0000000000402000 0x0000000000402000
               0x00000000000000a0 0x00000000000000a0  R      0x1000
  LOAD         0x0000000000002e50 0x0000000000403e50 0x0000000000403e50
               0x00000000000001e8 0x00000000000001f8  RW     0x1000
```

运行时动态链接

可通过**动态链接器接口**提供的函数在运行时进行动态链接

类UNIX系统中的动态链接器接口定义了相应的函数，如dlopen, dlsym, dlerror, dlclose等，其头文件为dlfcn.h

```
#include <stdio.h>
#include <dlfcn.h>
int main()
{
    void *handle;
    void (*myfunc1)();
    char *error;
    /* 动态装入包含函数myfunc1()的共享库文件 */
    handle = dlopen("./mylib.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    /* 获得一个指向函数myfunc1()的指针myfunc1*/
    myfunc1 = dlsym(handle, "myfunc1");
    if ((error = dlerror()) != NULL) {
        fprintf(stderr, "%s\n", error);
        exit(1);
    }
    /* 现在可以像调用其他函数一样调用函数myfunc1(
*/
    myfunc1();
    /* 关闭（卸载）共享库文件 */
    if (dlclose(handle) < 0) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    return 0;
}
```

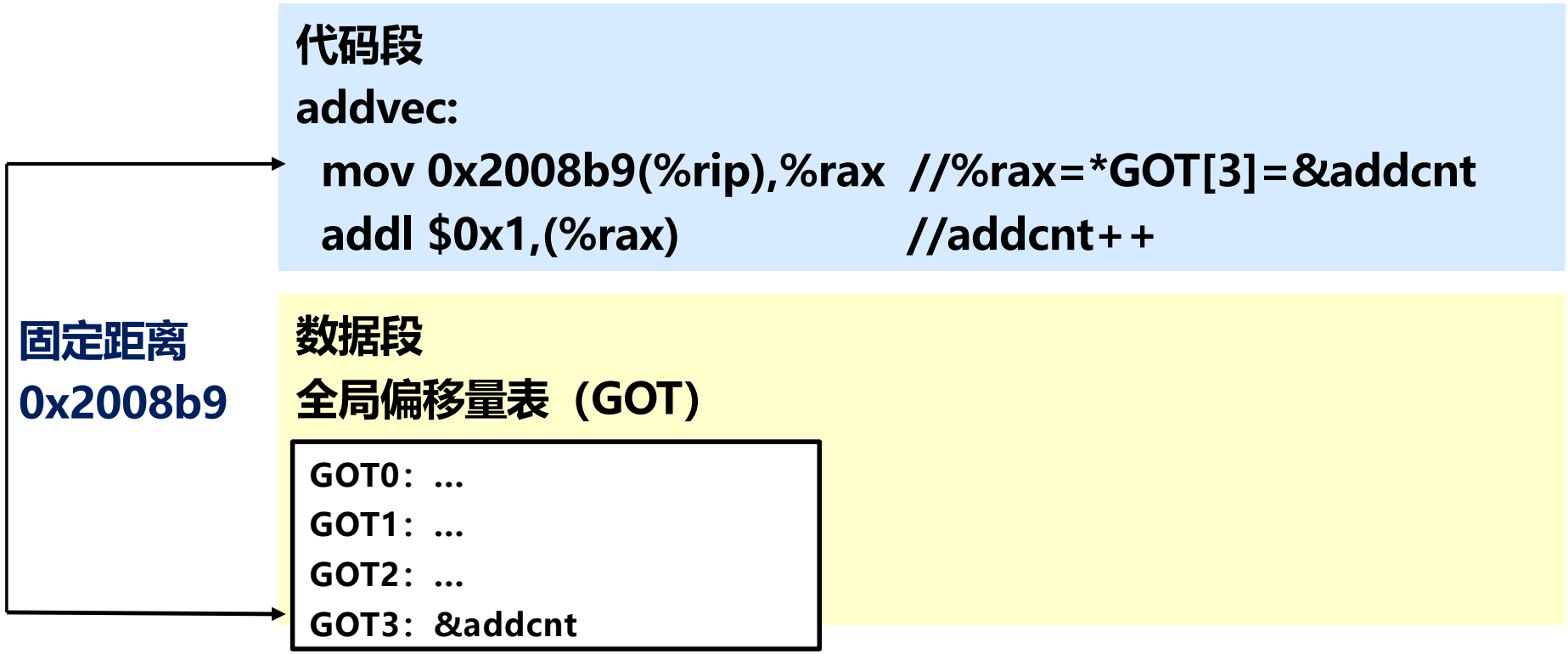
位置无关代码（PIC）

- 动态链接用到一个重要概念：
  - 位置无关代码（Position-Independent Code，PIC）
  - GCC选项-fPIC指示生成PIC代码
- 共享库代码是一种PIC
  - 共享库代码的位置可以是不确定的
  - 即使共享库代码的长度发生变化，也不影响调用它的程序
- 引入PIC的目的
  - 链接器无需修改代码即可将共享库加载到任意地址运行

要实现动态链接，必须生成PIC代码

PIC数据引用

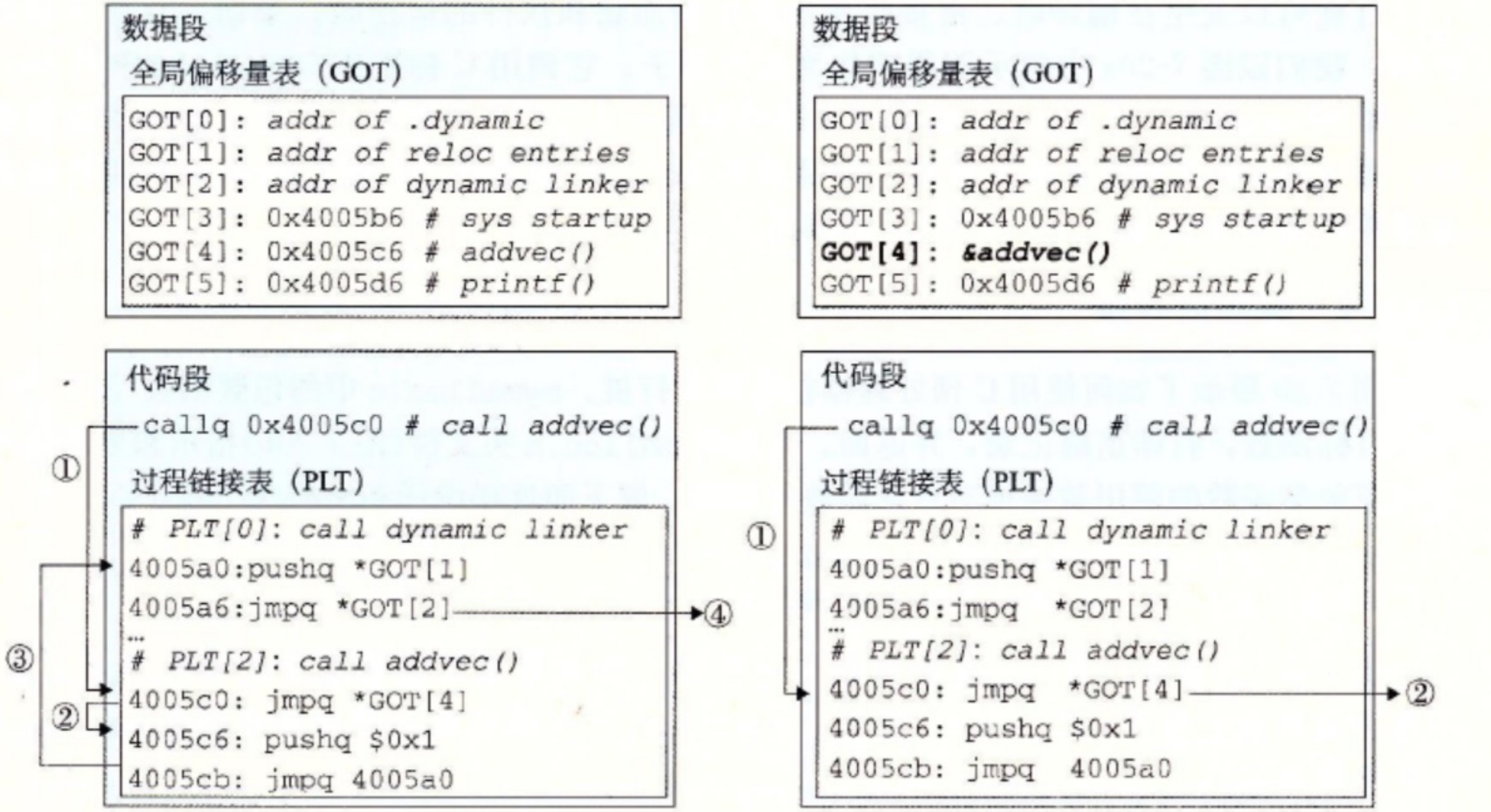
- 引用其他模块的全局变量，无法确定相对距离
- 在.data节开始处设置一个指针数组（全局偏移表，GOT），指针可指向一个全局变量
- GOT与引用数据的指令之间相对距离固定（运行时常量）



- 编译器为GOT每一项生成一个重定位项（如.rel节...）
- 加载时，动态链接器对GOT中各项进行重定位，填入所引用的地址

PIC函数调用

延迟绑定（lazy binding）技术：不在加载时重定位，而是延迟到第一次函数调用时。需要用GOT和PLT（Procedure linkage Table, 过程链接表）



本章小结

- 链接处理涉及到三种目标文件格式：可重定位目标文件、可执行目标文件和共享目标文件。共享库文件是一种特殊的可重定位目标。
- ELF目标文件格式有链接视图和执行视图两种，前者是可重定位目标格式，后者是可执行目标格式。
  - 链接视图中包含ELF头、各个节以及节头表
  - 执行视图中包含ELF头、各种节组成的段以及程序头表（段头表）
- 链接过程需要完成符号解析和重定位两方面的工作
  - 符号解析的目的就是将符号的引用与符号的定义关联起来
  - 重定位的目的是分别合并代码和数据，并根据代码和数据在虚拟地址空间中的位置，确定每个符号的最终存储地址，然后根据符号的确切地址来修改符号的引用处的地址。
- 加载器在加载可执行目标文件时，实际上只是把可执行目标文件中的只读代码段和可读写数据段通过页表映射到了虚拟地址空间中确定的位置，并没有真正把代码和数据从磁盘装入主存。

本章小结

链接分为静态链接和动态链接两种:

静态链接将多个可重定位目标模块中相同类型的节合并起来，以生成完全链接的可执行目标文件，其中所有符号的引用都是在虚拟地址空间中确定的最终地址，因而可以直接被加载执行。

动态链接的可执行目标文件是部分链接的，还有一部分符号的引用地址没有确定，需要利用共享库中定义的符号进行重定位，因而需要由动态链接器来加载共享库并重定位可执行文件中部分符号的引用。

加载时进行共享库的动态链接
执行时进行共享库的动态链接