# Linking

15-213: Introduction to Computer Systems
13th Lecture, October 10th, 2017

**Instructor:**

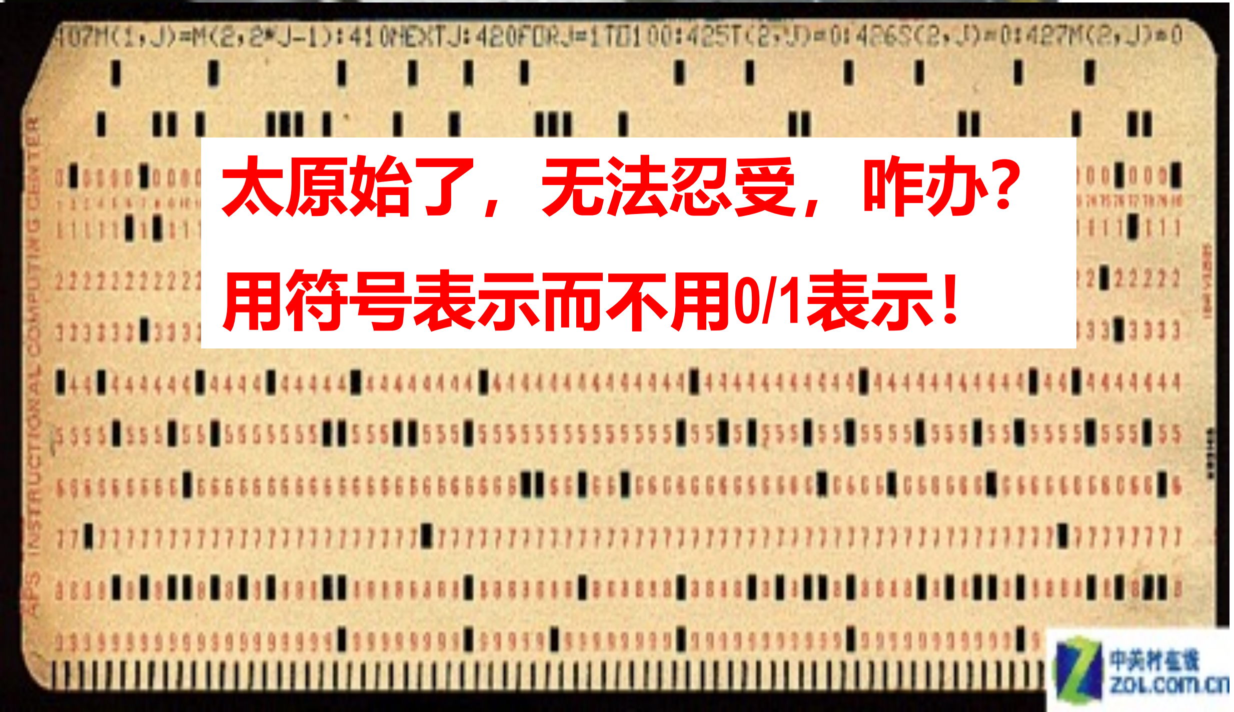
Randy Bryant

## Today

- **Linking**
  - Motivation
  - What it does
  - How it works
  - Dynamic linking

# 链接器的由来

- 原始的链接概念早在高级编程语言出现之前就已存在
- 最早程序员**用机器语言编写程序**，并记录在纸带或卡片上


407M(1,J)=M(2,2*J-1):410NEXTJ:420FORJ=1TO100:425T(2,J)=0:426S(2,J)=0:427M(2,J)=0

太原始了，无法忍受，咋办?

用符号表示而不用0/1表示!

穿孔表示0，未穿孔为1

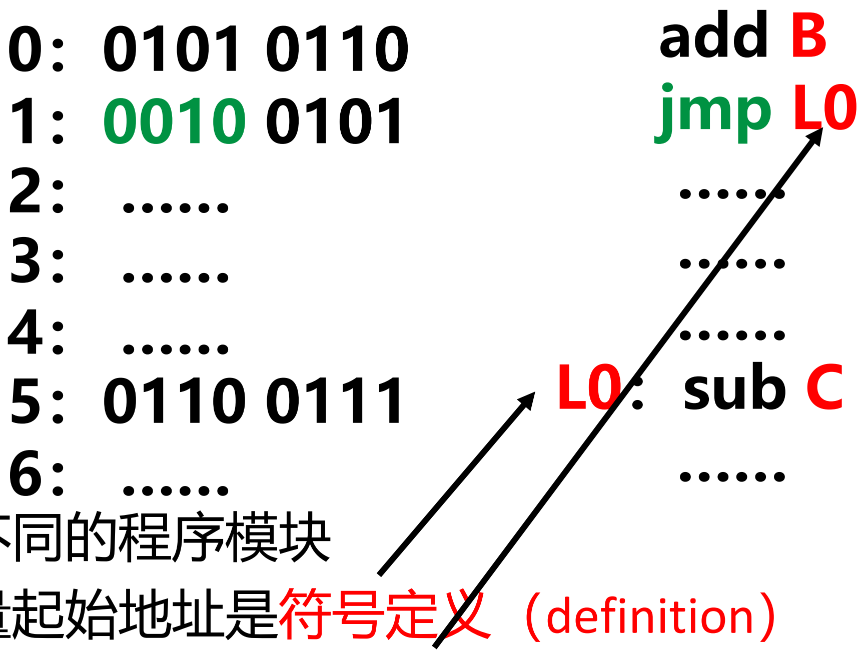
假设：0010-jmp

```
0：  0101 0110
1：  0010 0101
2：   ......
3：   ......
4：   ......
5：  0110 0111
6：   ......
```

若在第5条指令前加入指令，则程序员需重新计算jmp指令的目标地址（重定位），然后重新打孔。

# 链接器的由来

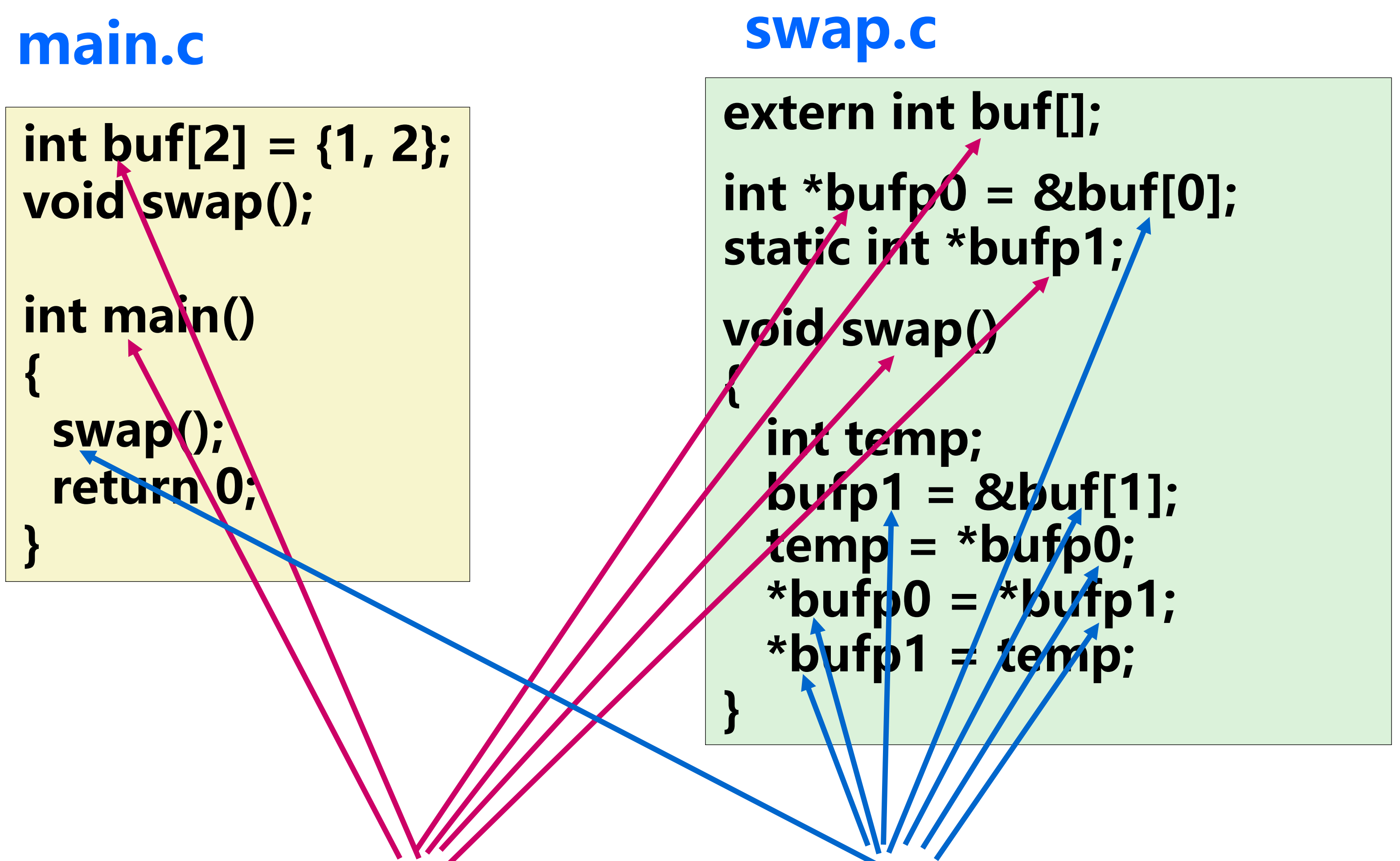
- 用符号表示跳转位置和变量位置，是否简化了问题?
- 汇编语言出现
  - 用助记符表示操作码
  - 用符号表示位置
  - 用助记符表示寄存器
  - ……
- 更高级编程语言出现
  - 程序越来越复杂，需多人开发不同的程序模块
  - 子程序（函数）起始地址和变量起始地址是符号定义（definition）
  - 调用子程序（函数或过程）和使用变量即是符号的引用（reference）
  - 一个模块定义的符号可以被另一个模块引用
  - 最终须链接（即合并），合并时须在符号引用处填入定义处的地址



```
0：0101 0110            add B
1：0010 0101            jmp L0
2：  ……                 ……
3：  ……                 ……
4：  ……                 ……
5：0110 0111        L0:  sub C
6：  ……                 ……
```

如上例，先确定L0的地址，再在jmp指令中填入L0的地址

## 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;
}
```

你能说出哪些是**符号定义**？哪些是**符号的引用**？

**局部变量temp分配在栈中，不会在过程外被引用，因此不是符号定义**

## 关于static的使用

**公司编码规范明确规定：只用于本文件的函数要全部使用static关键字声明，良好的编码风格**

- 变量
  - 关键字 static 声明为“静态变量”，包括静态全局变量或静态局部变量
  - static变量和普通全局变量的存储方式相同、作用域不同：当一个可执行目标文件是由多个源文件编译而成，则普通全局变量在在各个源文件都是有效的；而static变量只能在它定义的源文件中使用
  - 在静态数据区，内存中所有的字节默认值都是 0x00，局部变量呢?
  - 静态局部变量存储于进程的全局数据区，即使函数返回，它的值也会保持不变
- 函数
  - 普通函数的定义和声明默认情况下是extern的，在源程序的所有文件中可见；static函数只是在声明他的文件中可见，不能被其他文件所用
  - static函数优点：多个文件可以定义静态函数，不会发生冲突

```c
#include <stdio.h>

void fn(void)
{
    int n = 10;

    printf("n=%d\n", n);
    n++;
    printf("n++=%d\n", n);
}

void fn_static(void)
{
    static int n = 10;

    printf("static n=%d\n", n);
    n++;
    printf("n++=%d\n", n);
}

int main(void)
{
    fn();
    printf("------------------\n");
    fn_static();
    printf("------------------\n");
    fn();
    printf("------------------\n");
    fn_static();

    return 0;
}
```

```
-> % ./a.out
n=10
n++=11
------------------
static n=10
n++=11
------------------
n=10
n++=11
------------------
static n=11
n++=12
```

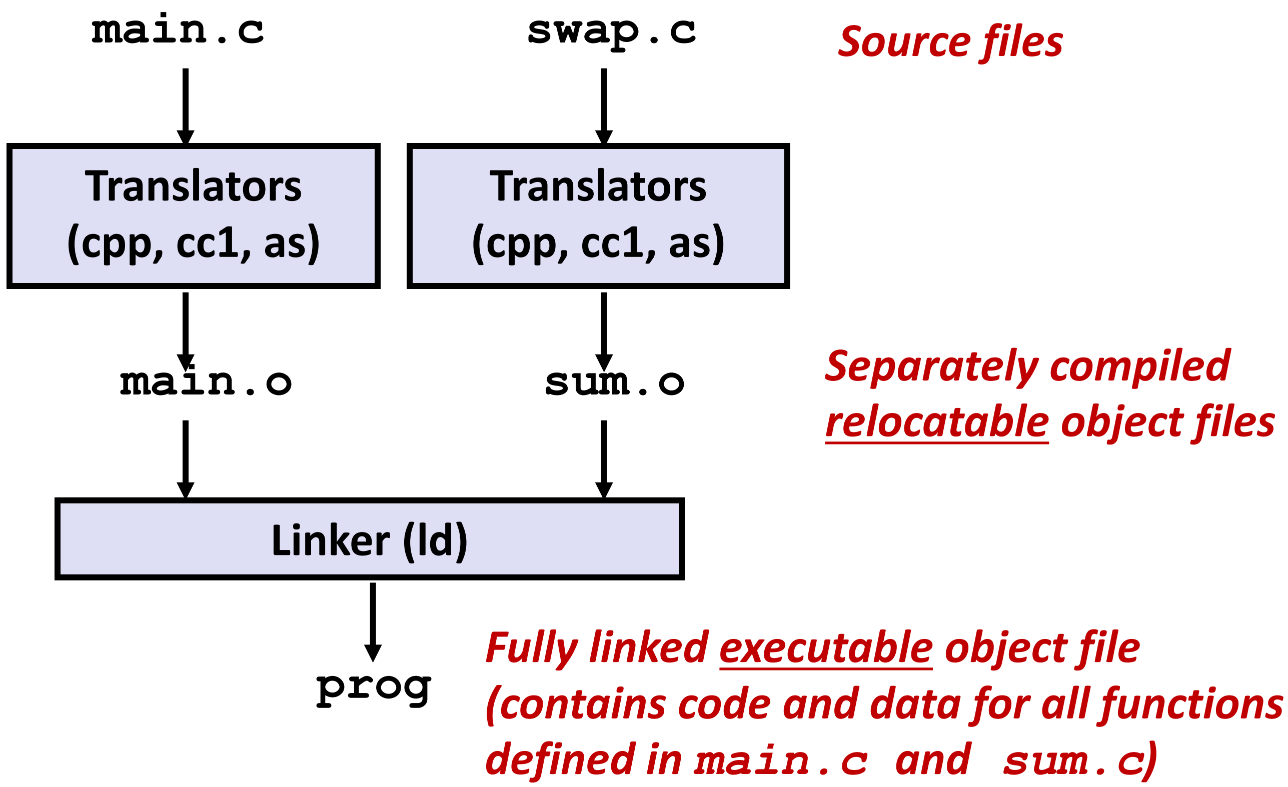
## Linking

- **Programs are translated and linked using a *compiler driver*:**
  - `linux> gcc -Og -o prog main.c swap.c`
  - `linux> ./prog`

-O2：2级优化

-g：生成调试信息

-o：目标文件名


main.c
swap.c
Source files
Translators (cpp, cc1, as)
Translators (cpp, cc1, as)
main.o
sum.o
Separately compiled relocatable object files
Linker (ld)
prog
Fully linked executable object file (contains code and data for all functions defined in main.c and sum.c)

## Why Linkers?

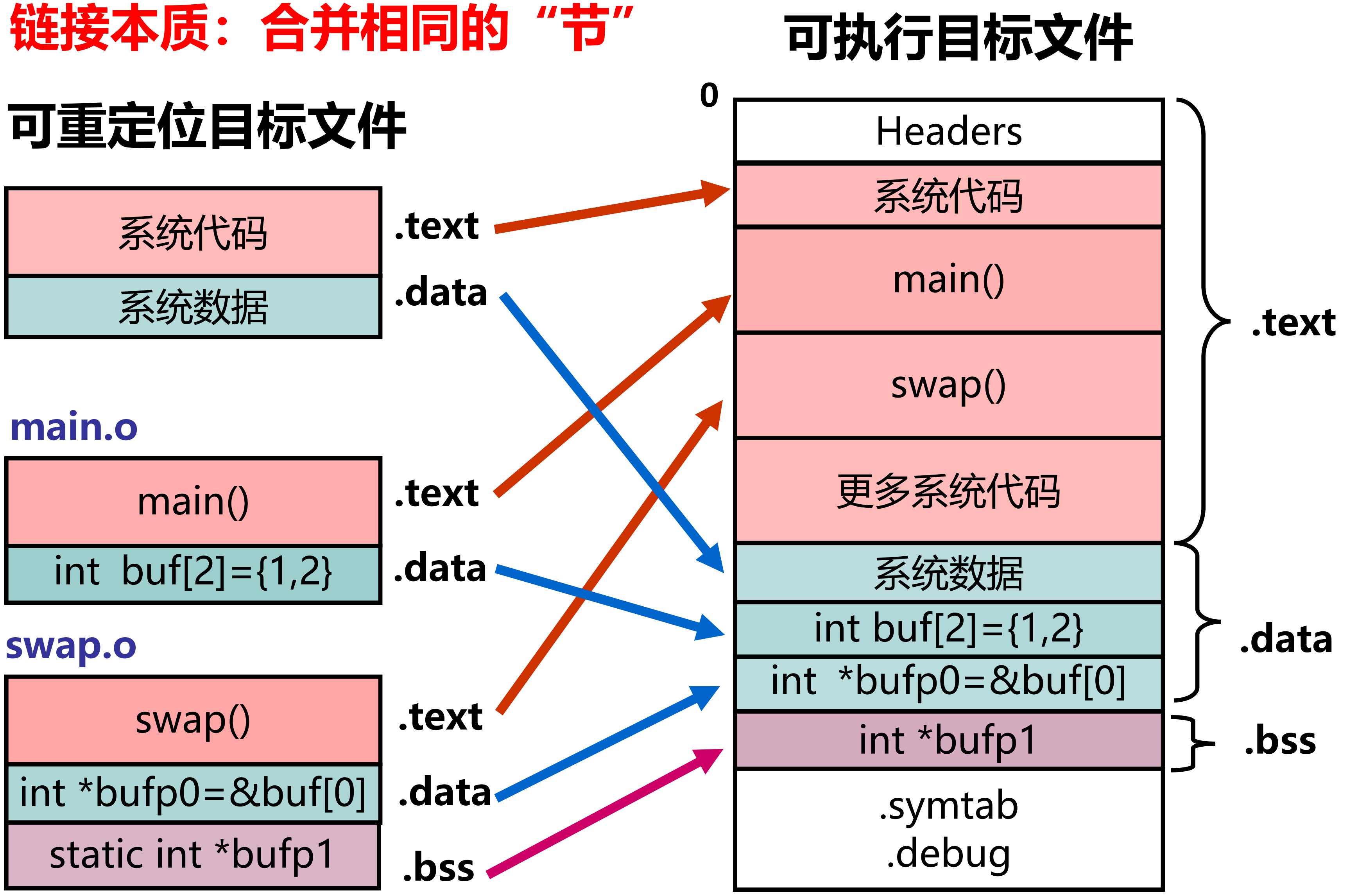
- **Reason 1: Modularity**
  - Program can be written as a collection of smaller source files, rather than one monolithic mass.
  - Can build libraries of common functions (more on this later)
    - e.g., Math library, standard C library

## Why Linkers? (cont)

- **Reason 2: Efficiency**
  - Time: Separate compilation
    - Change one source file, compile, and then relink.
    - No need to recompile other source files.
    - Can compile multiple files concurrently.
  - Space: Libraries
    - Common functions can be aggregated into a single file...
    - **Option 1: *Static Linking***
      - Executable files and running memory images contain only the library code they actually use
    - **Option 2: *Dynamic linking***
      - Executable files contain **no library code**
      - During execution, **single copy of library code** can be shared across all executing processes

# 链接过程的本质

**链接本质：合并相同的“节”**

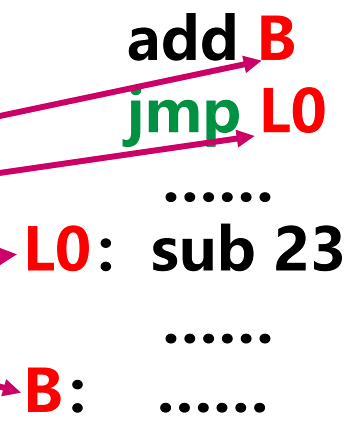


## What Do Linkers Do?

- **Step 1: Symbol resolution**
  - Programs define and reference *symbols* (global variables and functions):
    - `void swap() {…}    /* define symbol swap */`
    - `swap();            /* reference symbol swap */`
    - `int *xp = &x;      /* define symbol xp, reference x */`
  - Symbol definitions are stored in object file (by assembler) in *symbol table*.
    - Symbol table is an array of entries
    - Each entry includes name, size, and location of symbol.
  - **During symbol resolution step, the linker associates each symbol reference with exactly one symbol definition.**

## 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。
- 本地符号在本模块内定义并引用，因此，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



```
      add B
      jmp L0
      ......
L0:   sub 23
      ......
B:    ......
```

确定L0的地址，再在jmp指令中填入L0的地址

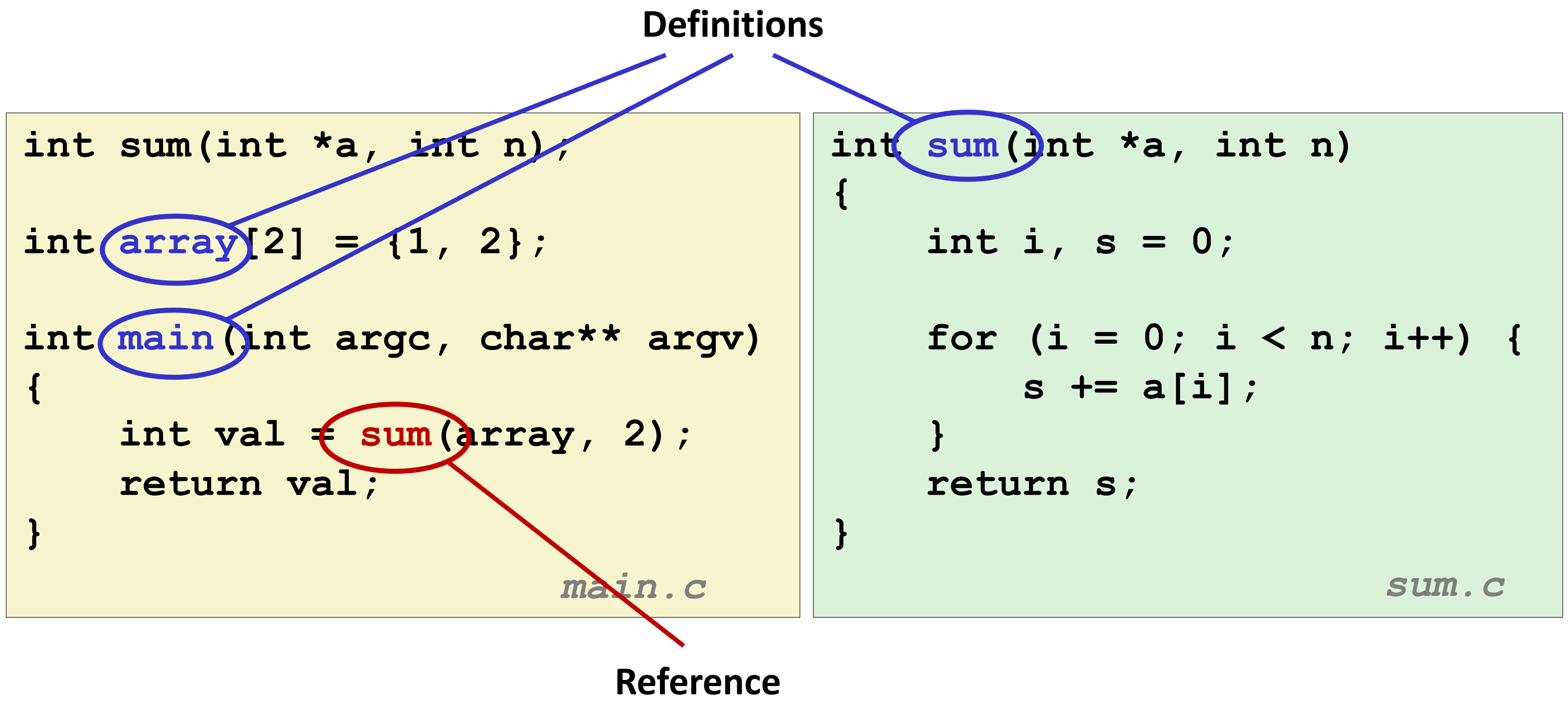
符号解析也称符号绑定

“符号的定义”其实质是什么？

指被分配了存储空间。为函数名指定其代码所在区；为变量名指定其所占的静态数据区。

所有定义符号的值就是其目标所在的首地址！

# Symbols in Example C Program



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

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

int main(int argc, char** argv)
{
    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

# What Do Linkers Do? (cont)

- **Step 2: Relocation**
  - Merges separate code and data sections into single sections
  - Relocates symbols from their **relative locations** in the `.o` files to their final **absolute memory locations** in the executable.
  - Updates all references to these symbols to reflect their new positions.

**Let's look at these two steps in more detail….**

## Step 1: Symbol Resolution

Referencing a global…

…that's defined here

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

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

int main(int argc,char **argv)
{
    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
```

Defining a global

Linker knows nothing of val

Referencing a global…

…that's defined here

Linker knows nothing of i or s

## Linker Symbols

- **Global symbols(自定他用)**
  - Symbols defined by module *m* that can be referenced by other modules.
  - E.g.: non-`static` C functions and non-`static` global variables.
- **External symbols(他定自用)**
  - Global symbols that are referenced by module *m* but defined by some other module.
- **Local symbols(自定自用)**
  - Symbols that are defined and referenced exclusively by module *m*.
  - E.g.: **C functions and global variables defined with the `static` attribute**.
  - **Local linker symbols are *not* local program variables**

## Symbol Identification

***Which*** **of the following names will be in the symbol table of `symbols.o`?**

**symbols.c:**

```
int time;

int foo(int a) {
  int b = a + 1;
  return b;
}

int main(int argc,
         char* argv[]) {
  printf("%d\n", foo(5));
  return 0;
}
```

**Names:**

- `time`
- `foo`
- `a`
- `argc`
- `argv`
- `b`
- `main`
- `printf`
- `"%d\n"`

## Local Symbols

- Local non-static C variables vs. local static C variables
  - local non-static C variables: stored on the stack
  - local static C variables: stored in either `.bss`, or `.data`

```
static int x = 15;

int f() {
    static int x = 17;
    return x++;
}

int g() {
    static int x = 19;
    return x += 14;
}

int h() {
    return x += 27;
}
```

*static-local.c*

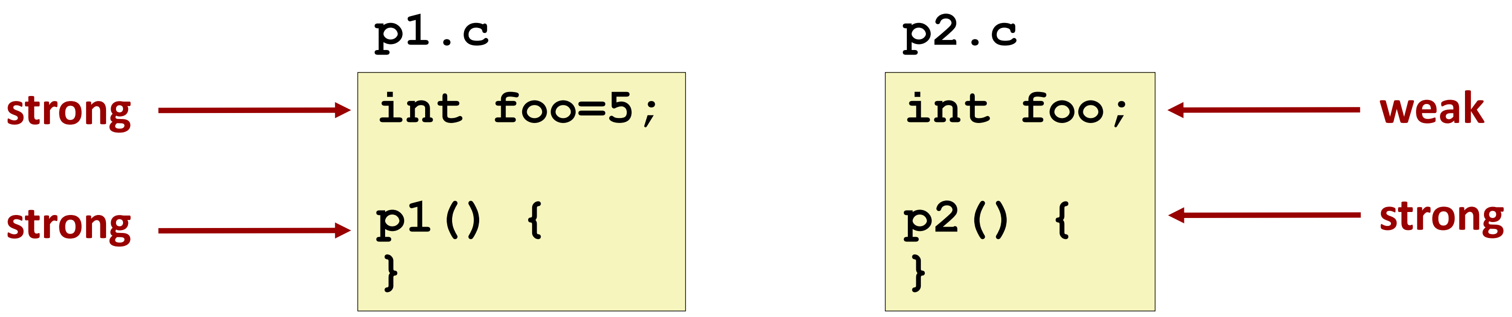
Compiler allocates space in `.data` for each definition of `x`

Creates local symbols in the symbol table with unique names, e.g., `x`, `x.1721` and `x.1724`.

C语言中，源文件扮演模块的角色，任何带有static属性声明的全局变量和函数都是模块私有的。任何不带static属性声明的全局变量都是公共的，可以被其他模块引用

## How Linker Resolves Duplicate Symbol Definitions

- **Program symbols are either *strong* or *weak***
  - ***Strong***: procedures and initialized globals
  - ***Weak***: uninitialized globals
    - Or ones declared with specifier **`extern`**



p1.c

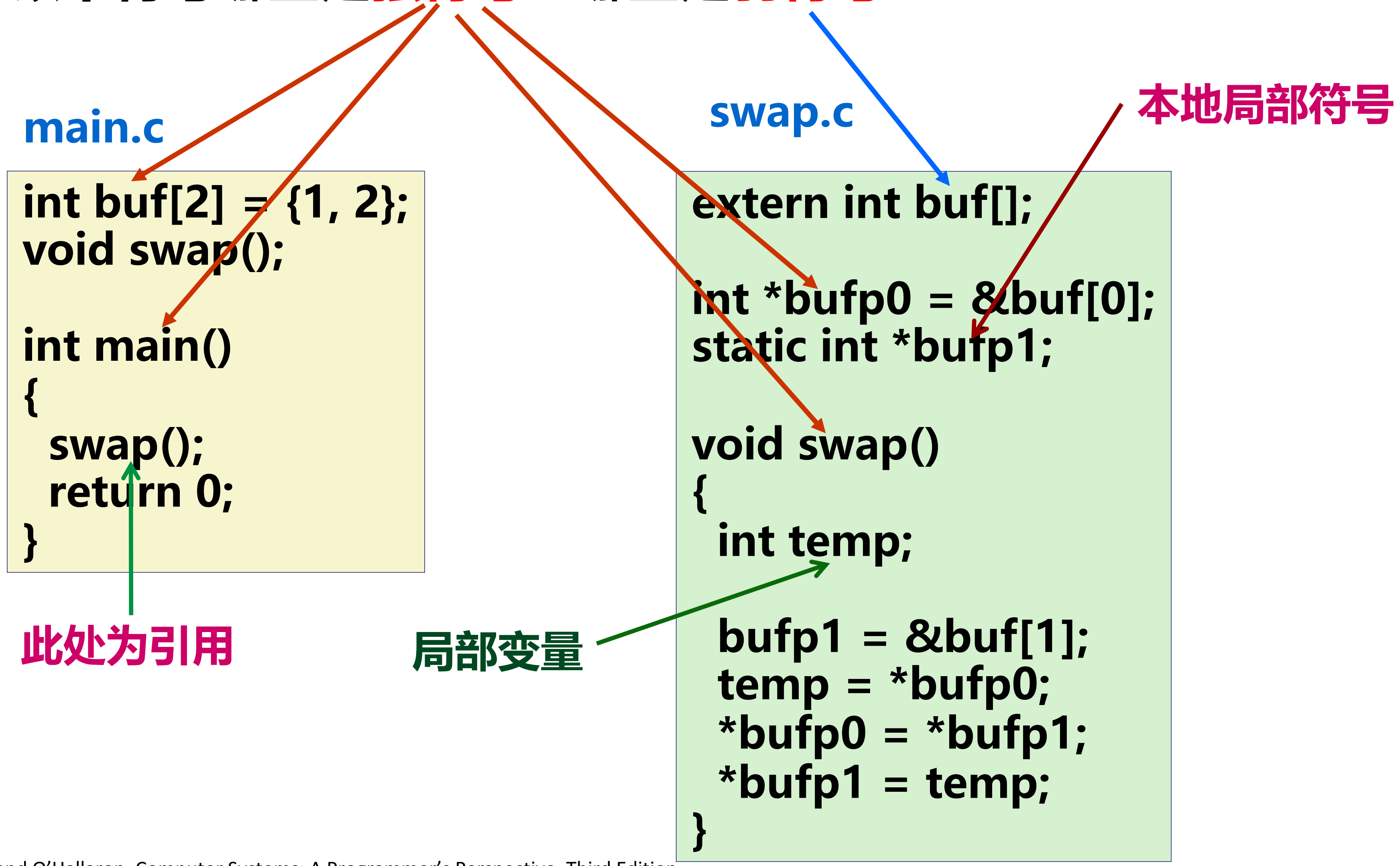
```
int foo=5;

p1() {
}
```

p2.c

```
int foo;

p2() {
}
```

# 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;
}
```

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;
}
```

## Linker's Symbol Rules

- **Rule 1: Multiple strong symbols are not allowed**
  - Each item can be defined only once
  - Otherwise: Linker error
- **Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol**
  - References to the weak symbol resolve to the strong symbol
- **Rule 3: If there are multiple weak symbols, pick an arbitrary one**
  - Can override this with `gcc -fno-common`
- **Puzzles on the next slide**

# Linker Puzzles

```
int x;
p1() {}
```

```
p1() {}
```

Link time error: two strong symbols (**`p1`**)

```
int x;
p1() {}
```

```
int x;
p2() {}
```

References to **`x`** will refer to the same uninitialized int. Is this what you really want?

```
int x;
int y;
p1() {}
```

```
double x;
p2() {}
```

Writes to **`x`** in **`p2`** might overwrite **`y`**! Evil!

```
int x=7;
int y=5;
p1() {}
```

```
double x;
p2() {}
```

Writes to **`x`** in **`p2`** will overwrite **`y`**! Nasty!

```
int x=7;
p1() {}
```

```
int x;
p2() {}
```

References to **`x`** will refer to the same initialized variable.

**Important: Linker does not do type checking.**

## Type Mismatch Example

```
long int x;  /* Weak symbol */

int main(int argc,
         char *argv[]) {
    printf("%ld\n", x);
    return 0;
}
```

*mismatch-main.c*

```
/* Global strong symbol */
double x = 3.14;
```

*mismatch-variable.c*

- Compiles without any errors or warnings
- What gets printed?

# 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有一次强定义，一次弱定义**

**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，一次弱定义**
**main一次强定义**

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

**p1.c**

**问题：打印结果是什么?**

**y=200，z=2000**

**该例说明：在两个不同模块定义相同变量名，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?**

```
#include <stdio.h>
int d=100;
int x=200;
void p1(void);
int main()
{
    p1();
    printf( “d=%d,x=%d\n” ,d,x);
    return 0;
}
```

**main.c**

**p1.c**

```
double d;

void p1()
{
    d=1.0;
}
```

**FLD1**
**FSTPl &d**

**p1执行后d和x处内容是什么?**

**问题：打印结果是什么?**

**d=0,x=1 072 693 248**

**该例说明：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！**

# 多重定义符号的解析举例

main.c

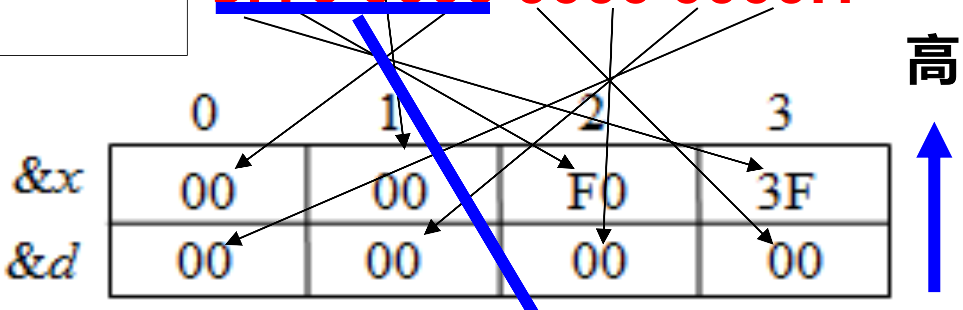
```
.......
int d=100;
int x=200;
int main()
{
  p1( );
  printf ( “d=%d, x=%d\n” , d, x );
  return 0;
}
```

p1.c

```
double d;

void p1( )
{
  d=1.0;
}
```

double型数1.0对应的机器数

3FF0 0000 0000 0000H



IA-32是小端方式

$2^{30}$-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?

## Global Variables

- **Avoid if you can**

- **Otherwise**
  - Use **`static`** if you can
  - Initialize if you define a global variable
  - Use **`extern`** if you reference an external global variable
    - Treated as weak symbol
    - But also causes linker error if not defined in some file

**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，并会在程序执行很久后才能表现出来，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。**

## Use of `extern` in .h Files (#1)

`c1.c`

```
#include "global.h"

int f() {
  return g+1;
}
```

`global.h`

```
extern int g;
int f();
```

`c2.c`

```
#include <stdio.h>
#include "global.h”

int g = 0;

int main(int argc, char argv[]) {
  int t = f();
  printf("Calling f yields %d\n", t);
  return 0;
}
```

# Use of .h Files (#2)

global.h

```
extern int g;
static int init = 0;
```

```
#else
  extern int g;
  static int init = 0;
#endif
```

c1.c

```
#~~include "global.h"~~

int f() {
  return g+1;
}
```

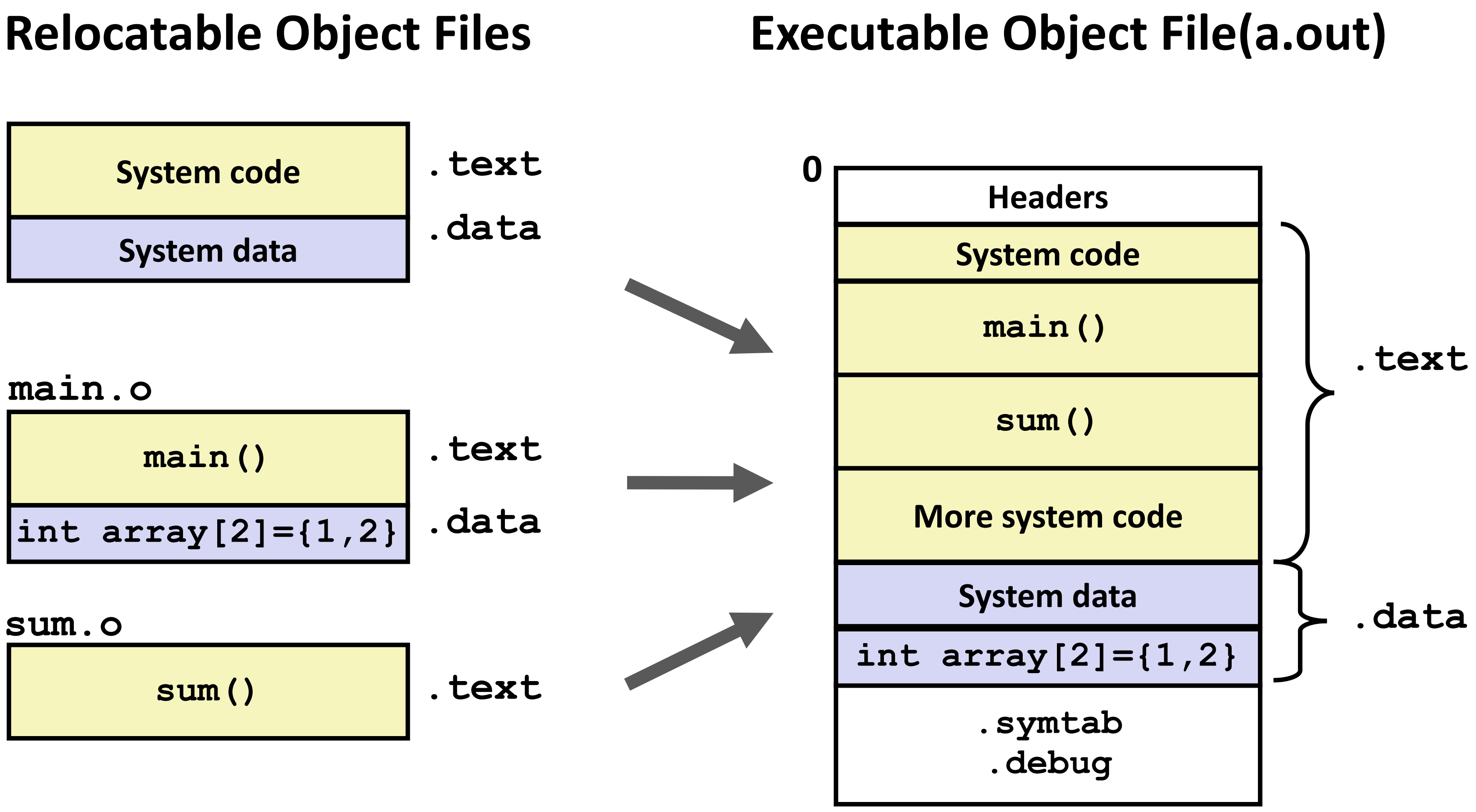
c2.c

```
#define INITIALIZE
#include <stdio.h>
#~~include "global.h"~~
```

```
int g = 23;
static int init = 1;
```

```
int main(int argc, char** argv) {
  if (init)
    // do something, e.g., g=31;
  int t = f();
  printf("Calling f yields %d\n", t);
  return 0;
}
```

## Step 2: Relocation



## 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

## Executable and Linkable Format (ELF) (x86-64Linux/Unix system)

- **Standard binary format for object files**

- **One unified format for**
  - Relocatable object files (`.o`),
  - Executable object files `(a.out)`
  - Shared object files (`.so`)

- **Generic name: ELF binaries**

## Three Kinds of Object Files (Modules)

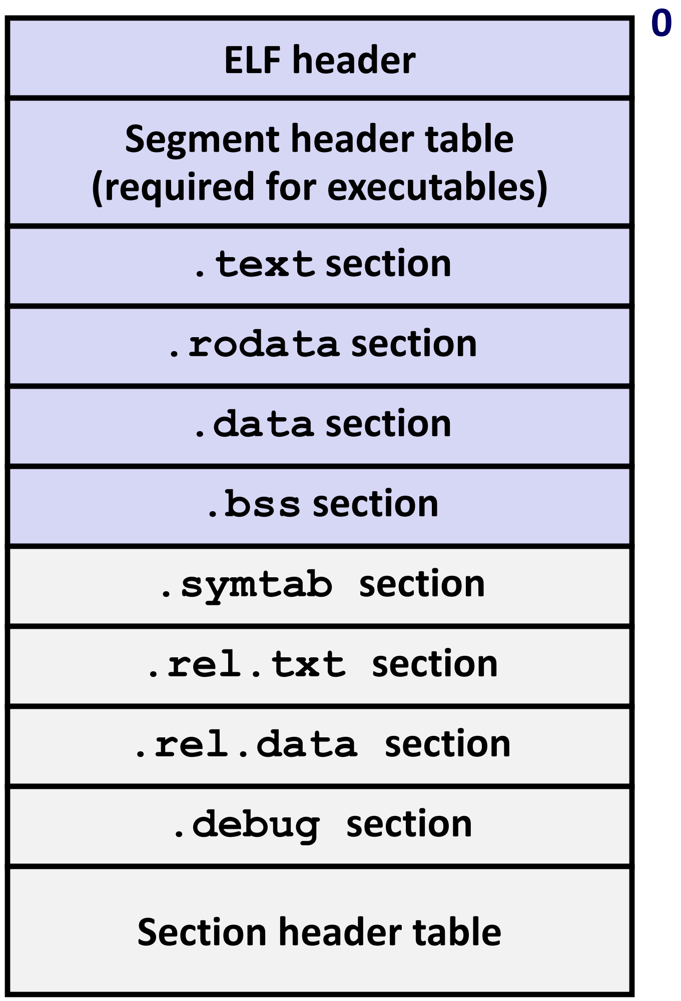
- **Relocatable object file (`.o` file)**
  - Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
    - Each `.o` file is produced from exactly one source (`.c`) file

- **Executable object file (`a.out` file)**
  - Contains code and data in a form that can be copied directly into memory and then executed.

- **Shared object file (`.so` file)**
  - Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
  - Called *Dynamic Link Libraries* (DLLs) by Windows

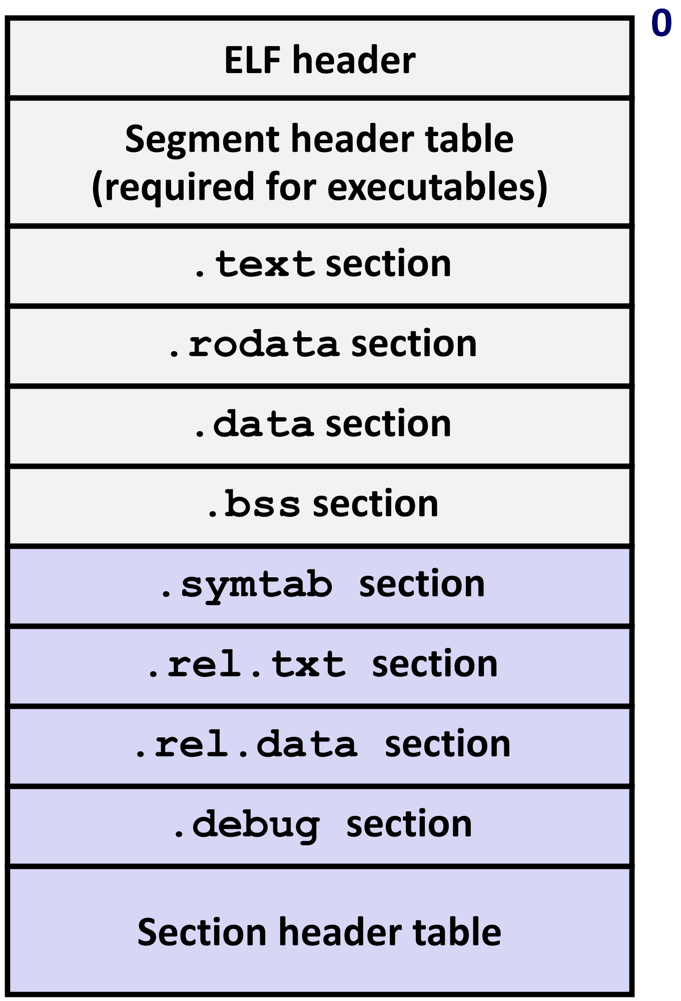
# ELF Object File Format

- **Elf header**
  - Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- **Segment header table**
  - Page size, virtual addresses memory segments (sections), segment sizes.
- **`.text` section**
  - Code
- **`.rodata` section**
  - Read only data: jump tables, string constants, ...
- **`.data` section**
  - Initialized global variables
- **`.bss` section**
  - Uninitialized global variables
  - "Block Started by Symbol"
  - "Better Save Space"
  - Has section header but occupies no space



# ELF Object File Format (cont.)

- **`.symtab` section**
  - Symbol table
  - Procedure and static variable names
  - Section names and locations
- **`.rel.text` section**
  - Relocation info for `.text` section
  - Addresses of instructions that will need to be modified in the executable
  - Instructions for modifying.
- **`.rel.data` section**
  - Relocation info for `.data` section
  - Addresses of pointer data that will need to be modified in the merged executable
- **`.debug` section**
  - Info for symbolic debugging (`gcc -g`)
- **Section header table**
  - Offsets and sizes of each section

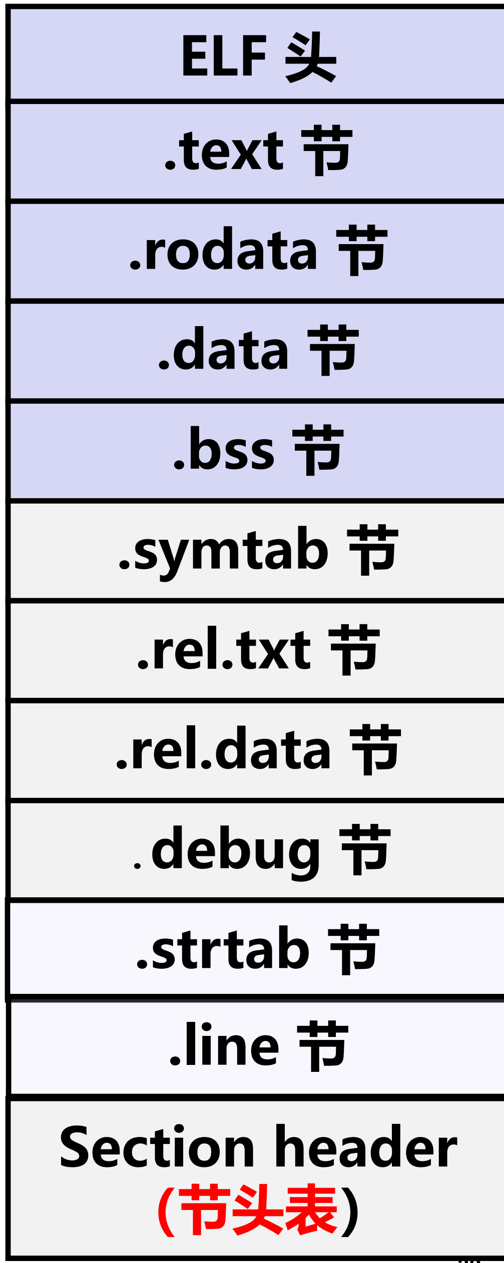


# ELF文件信息举例

```
$ readelf -h main.o
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:    ELF32
  Data:      2's complement, little endian
  Version: 1 (current)
  OS/ABI:  UNIX - System V
  ABI Version:   0
  Type:    REL (Relocatable file)
  Machine:   Intel 80386
  Version:    0x1
  Entry point address:  0x0
  Start of program headers:  0 (bytes into file)
  Start of section headers:   516 (bytes into file)
  Flags:    0x0
  Size of this header:   52 (bytes)
  Size of program headers:   0 (bytes)
  Number of program headers:   0
  Size of section headers:    40 (bytes)
  Number of section headers:  15
  Section header string table index: 12
```

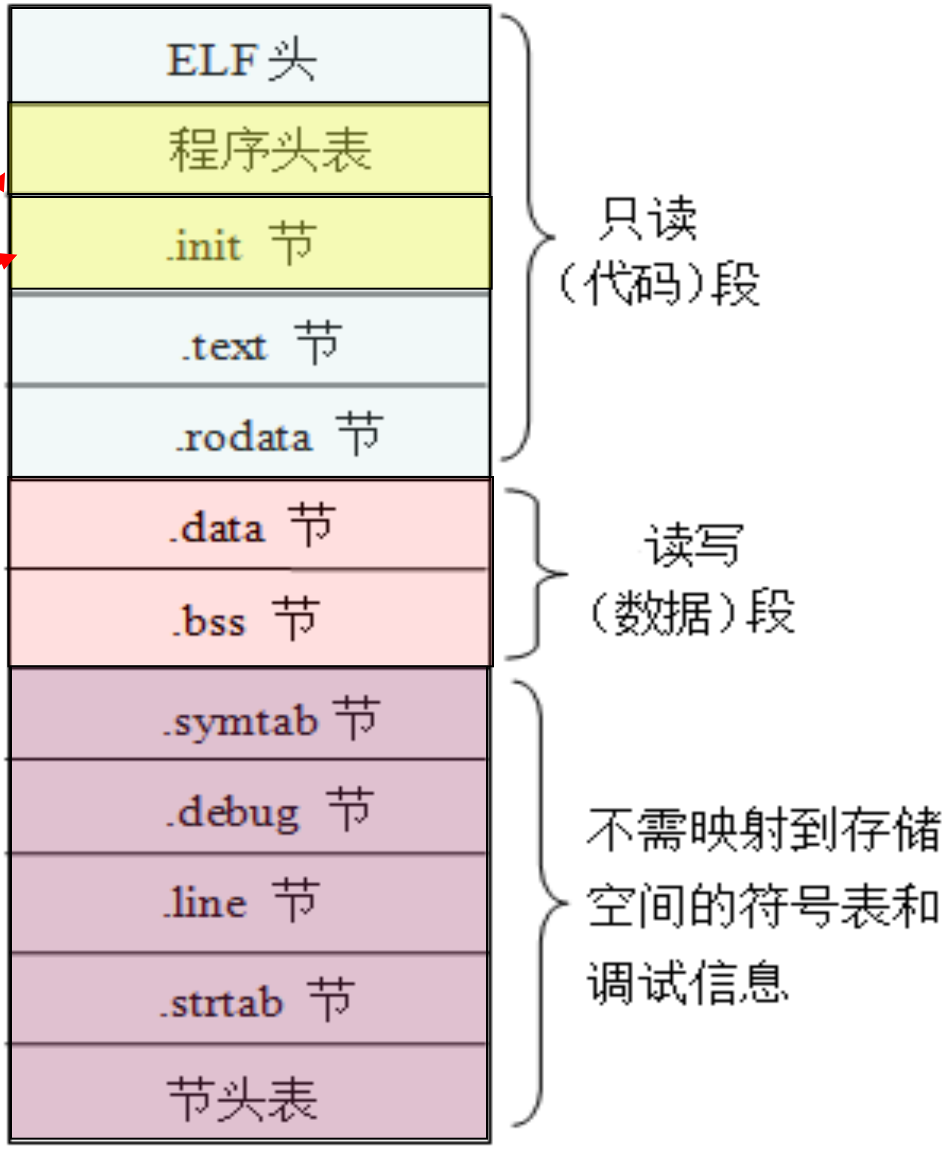
可重定位目标文件的ELF头

没有程序头表



# 可执行目标文件格式

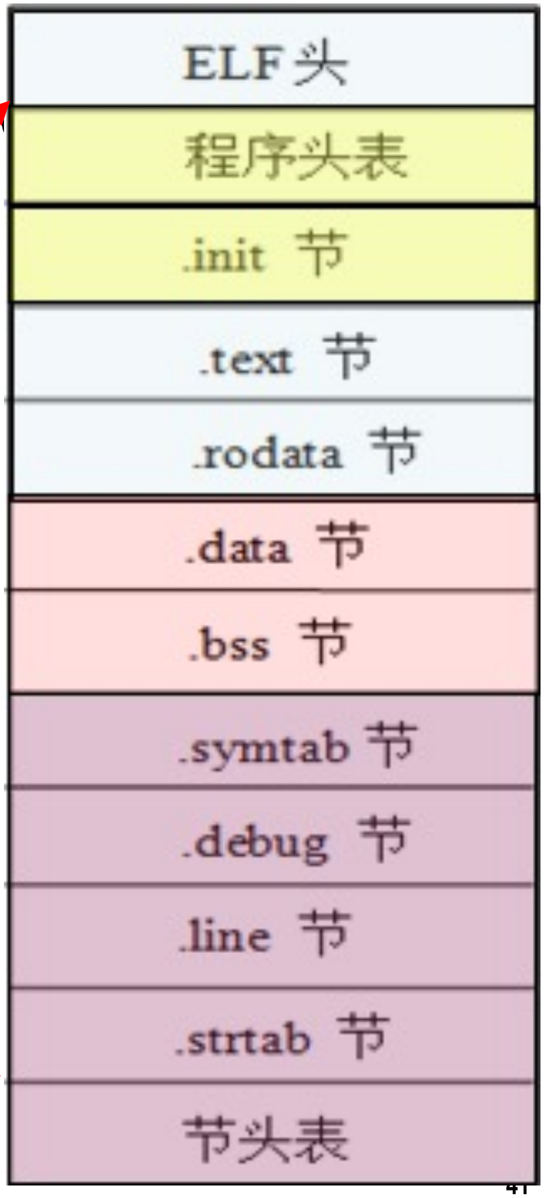
- **与.o文件稍有不同：**
  - **ELF头中字段e_entry给出执行程序时第一条指令的地址，而在可重定位文件中，此字段为0**
  - **多一个.init节，用于定义_init函数，该函数用来进行可执行目标文件开始执行时的初始化工作**
  - **少两.rel节（无需重定位）**
  - **多一个程序头表，也称段头表（segment header table），是一个结构数组**



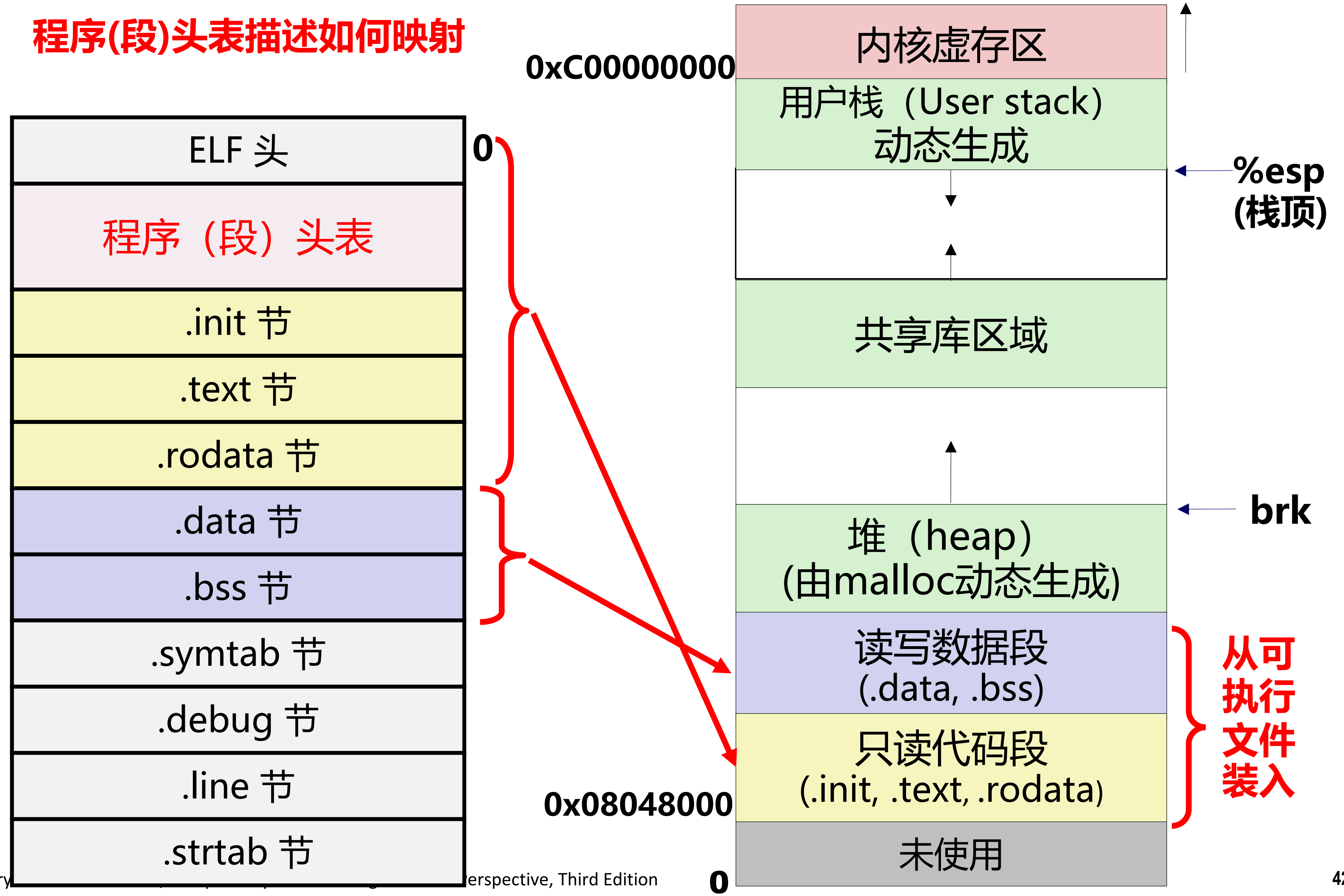
# ELF文件信息举例

**$ readelf -h main** 可执行目标文件的ELF头

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00
  Class:    ELF32
  Data:       2's complement, little endian
  Version:  1 (current)
  OS/ABI:     UNIX - System V
  ABI Version:      0
  Type:    EXEC (Executable file)
  Machine:   Intel 80386
  Version:     0x1
  Entry point address:     x8048580
  Start of program headers:  52 (bytes into file)
  Start of section headers:     3232 (bytes into file)
  Flags:     0x0
  Size of this header:     52 (bytes)
  Size of program headers:     32 (bytes)
  Number of program headers:   8
  Size of section headers:      40 (bytes)
  Number of section headers:    29
  Section header string table index: 26
```



# 可执行文件的存储器映像(32位)



## Relocation Entries

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

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

重定位

```
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          # Relocation entry

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

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

## Relocated .text section

```
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
```

**`callq` instruction uses PC-relative addressing for sum():**

**`0x4004e8` = `0x4004e3` + `0x5`**

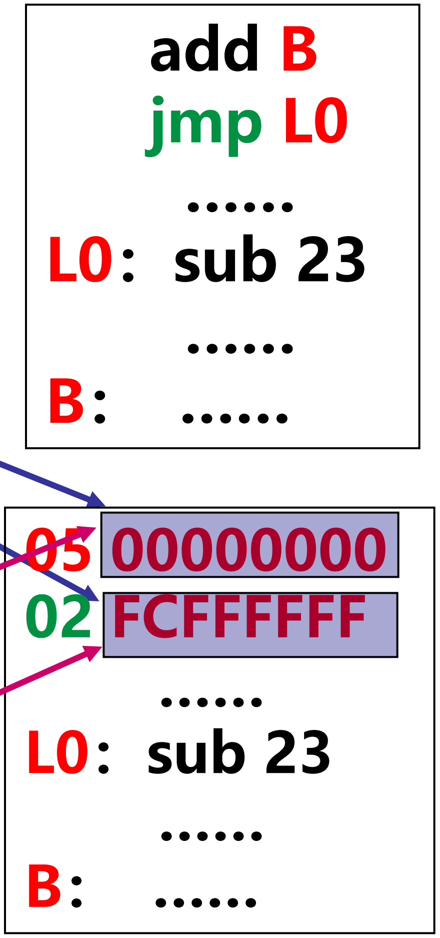
Source: `objdump -d prog`

## 重定位信息

- 汇编器遇到引用时，生成一个重定位条目
- 数据引用的重定位条目在.rel_data节中
- 指令中引用的重定位条目在.rel_text节中
- ELF中重定位条目格式如下：

```
typedef  struct {
            int  offset;         /*节内偏移*/
            int  symbol:24,  /*所绑定符号*/
                  type: 8;       /*重定位类型*/
            } Elf32_Rel;
```

- IA-32有两种最基本的重定位类型
  - R_386_32: 绝对地址
  - R_386_PC32: PC相对地址



```
      add B
      jmp L0
      ……
L0:   sub 23
      ……
B:    ……
```

```
05 00000000
02 FCFFFFFF
      ……
L0:   sub 23
      ……
B:    ……
```

例如，在rel_text节中有重定位条目

| | |
|---|---|
| offset: 0x1 | offset: 0x6 |
| symbol: B | symbol: L0 |
| type:  R_386_32 | type:  R_386_PC32 |

问题：重定位条目和汇编后的机器代码在哪种目标文件中？

在可重定位目标（.o）文件中！

# 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;
}
```

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

**符号解析后的结果是什么?**

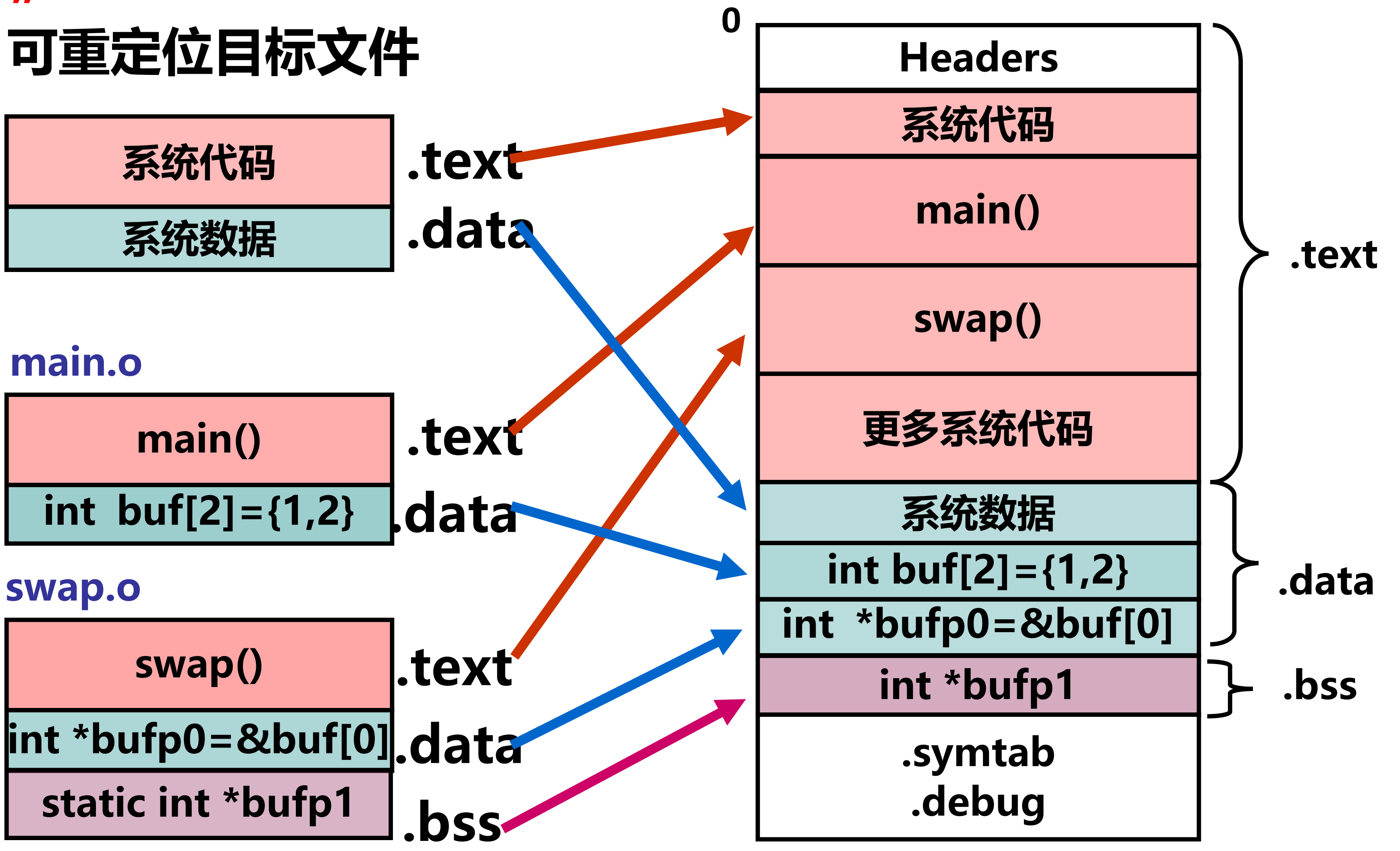
E中有main.o和swap.o两个模块！D中有所有定义的符号！

在main.o和swap.o的重定位条目中有重定位信息，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）**

## 符号引用的地址需要重定位

链接本质：合并相同的“节”



虽然是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0x12B。

main.o

```
Disassembly of section .text:
00000000 <main>:
   0:      55                   push   %ebp
   1:      89 e5                mov   %esp,%ebp
   3:      83 e4 f0             and    $0xfffffff0,%esp
   6:      e8 fc ff ff ff       call    7 <main+0x7>
                               7: R_386_PC32 swap
   b:      b8 00 00 00 00  mov    $0x0,%eax
  10:      c9                   leave
  11:      c3                   ret
```

```
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 | Ot | Ndx | Name |
|---|---|---|---|---|---|---|---|
| 8: | 0 | 8 | Data | Global | 0 | 3 | buf |
| 9: | 0 | 18 | Func | Global | 0 | 1 | main |
| 10: | 0 | 0 | Notype | Global | 0 | UND | swap |

swap是main.o的符号表中第10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。

在rel_text节中的重定位条目为：
r_offset=0x7, r_sym=10,
r_type=R_386_PC32，dump出来后为“7: R_386_PC32 swap”

r_sym=10说明引用的是swap！

## R_386_PC32的重定位方式

```
Disassembly of section .text:
00000000 <main>:
   ……
   6:     e8 fc ff ff ff        call    7 <main+0x7>
                                7: R_386_PC32 swap
   ……
```

- 假定：
  - 可执行文件中mai
  - swap紧跟main后
- 则swap起始地址为
  - 0x8048380+0x12=0x8048392
  - 在4字节边界对齐的情况下，是0x8048394
- 则重定位后call指令的机器代码是什么?
  - 转移目标地址=PC+偏移地址，PC=0x8048380+0x07-init
  - PC=0x8048380+0x07-(-4)=0x804838b
  - 重定位值=转移目标地址-PC=0048394-0x804838b=0x9
  - call指令的机器代码为“e8 09 00 00 00”

重定位值

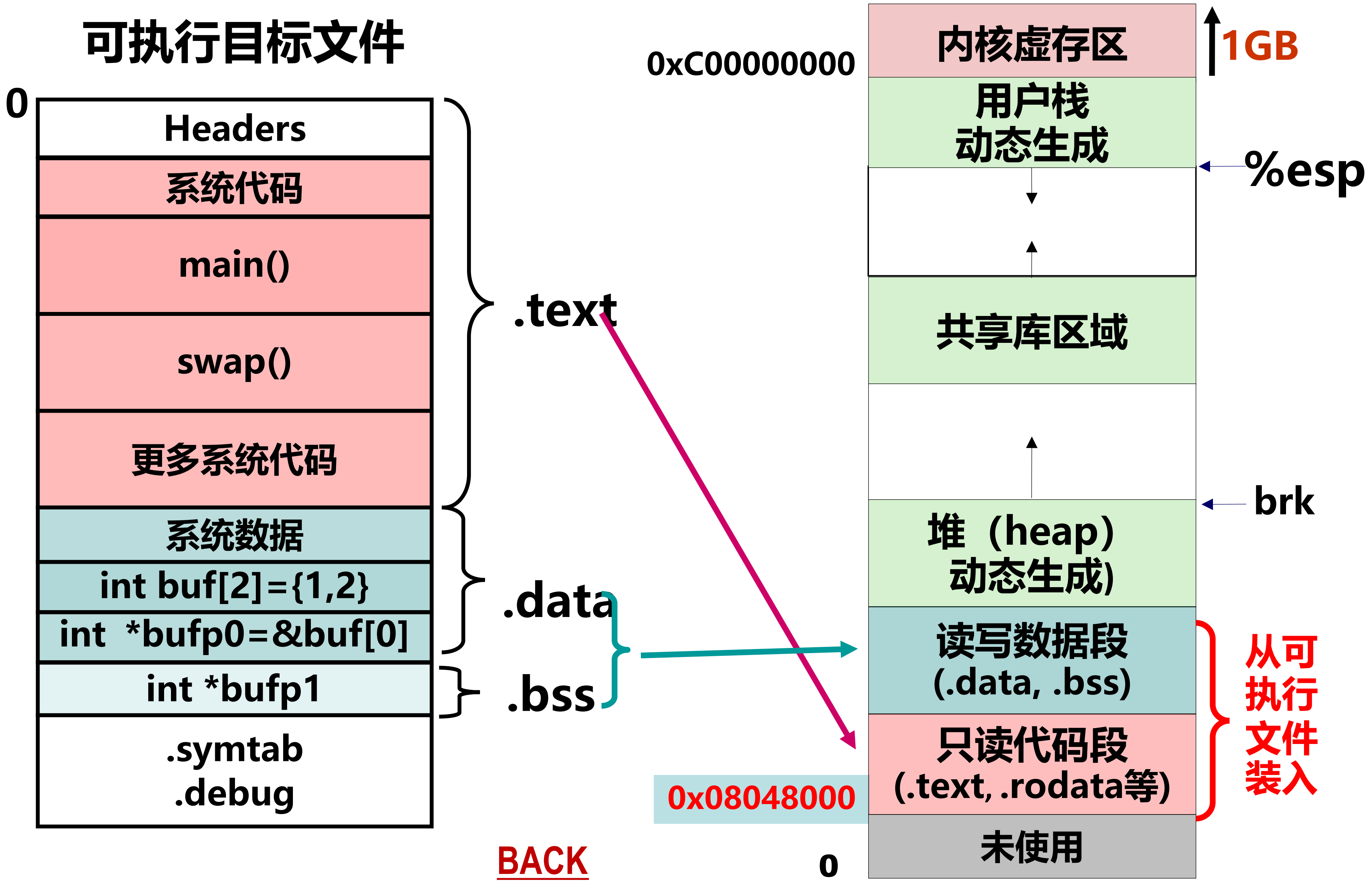
值为-4

PC相对地址方式下，重定位值计算公式为：

$\underbrace{\text{ADDR(r\_sym)}}_{\text{引用目标处}} - \underbrace{(\,(\,\text{ADDR(.text)} + \text{r\_offset}\,) - \text{init}\,)}_{\text{call指令下条指令地址}}$

即当前PC的值

# 确定定义符号的地址



BACK

## R_386_32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_386_32  buf
```

bufp0定义在.data节中偏移为0处，占4B，初值为0x0

swap.c

```
extern int buf[];

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

void swap()
……
```

# swap.o中的符号表

- swap.o中的符号表中最后4个条目

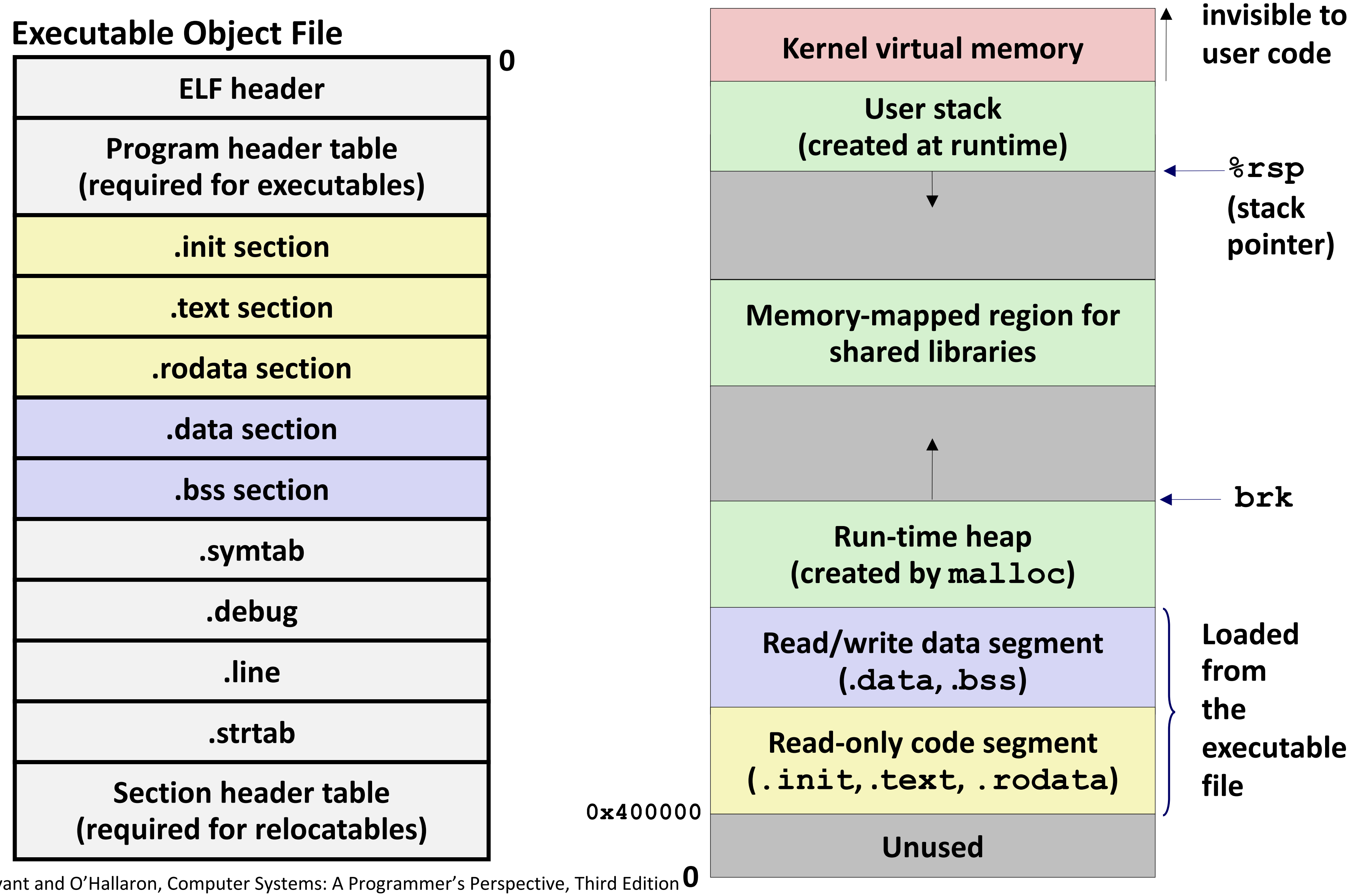
| Num: | value | Size | Type | Bind | Ot | Ndx | Name |
|---|---|---|---|---|---|---|---|
| 8: | 0 | 4 | Data | Global | 0 | 3 | bufp0 |
| 9: | 0 | 0 | Notype | Global | 0 | UND | buf |
| 10: | 0 | 36 | Func | Global | 0 | 1 | swap |
| 11: | 4 | 4 | Data | Local | 0 | COM | bufp1 |

buf是swap.o的符号表中第9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=9, r_type=R_386_32，OBJDUMP工具解释后显示为 “0：R_386_32 buf”

r_sym=9说明引用的是buf!

# Loading Executable Object Files



## Packaging Commonly Used Functions

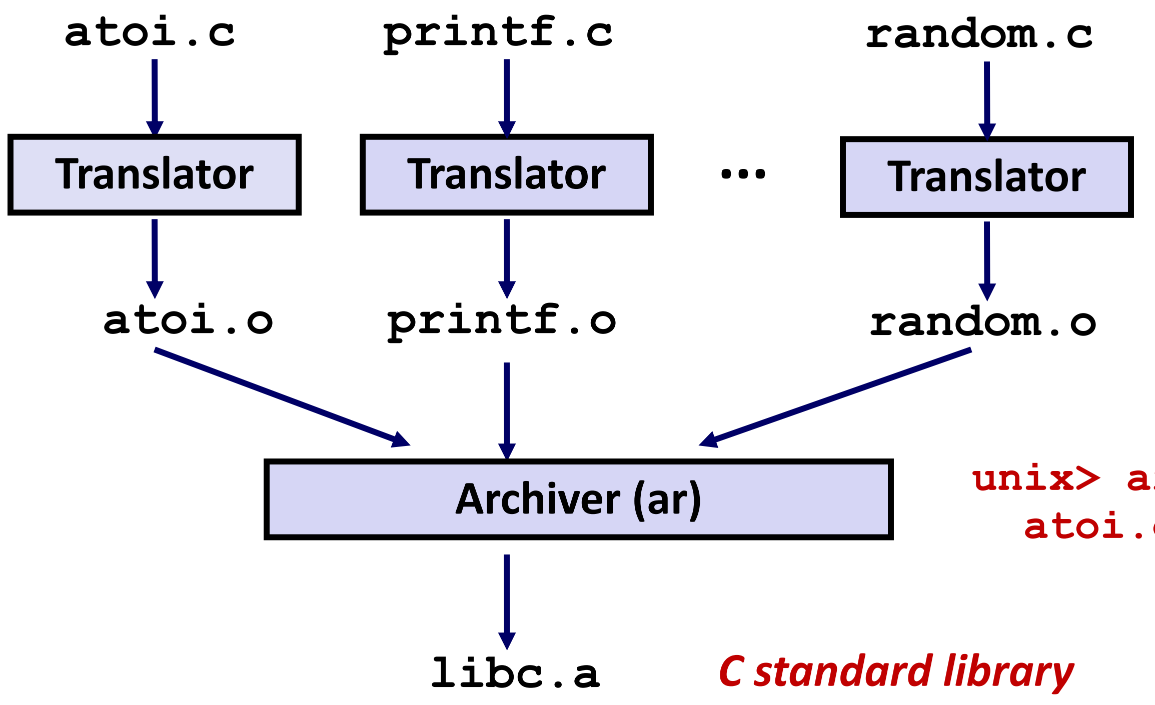
- **How to package functions commonly used by programmers?**
  - Math, I/O, memory management, string manipulation, etc.

- **Awkward, given the linker framework so far(极端做法):**
  - **Option 1:** Put all functions into a single source file
    - Programmers link big object file into their programs
    - Space and time inefficient
  - **Option 2:** Put each function in a separate source file
    - Programmers explicitly link appropriate binaries into their programs
    - More efficient, but burdensome on the programmer

## Old-fashioned Solution: Static Libraries

- **Static libraries (`.a` archive files)**
  - Concatenate related relocatable object files into a single file with an index (called an *archive*).
  - Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
  - If an archive member file resolves reference, link it into the executable.

# Creating Static Libraries



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

*C standard library*

- Archiver allows incremental updates
- Recompile function that changes and replace .o file in archive.

## Commonly Used Libraries

**`libc.a` (the C standard library)**

- 4.6 MB archive of 1496 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

**`libm.a` (the C math library)**

- 2 MB archive of 444 object files.
- floating point math (sin, cos, tan, log, exp, sqrt, …)

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

```
% ar -t /usr/lib/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
…
```

# Linking with Static Libraries

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

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

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

*main2.c*

**libvector.a**

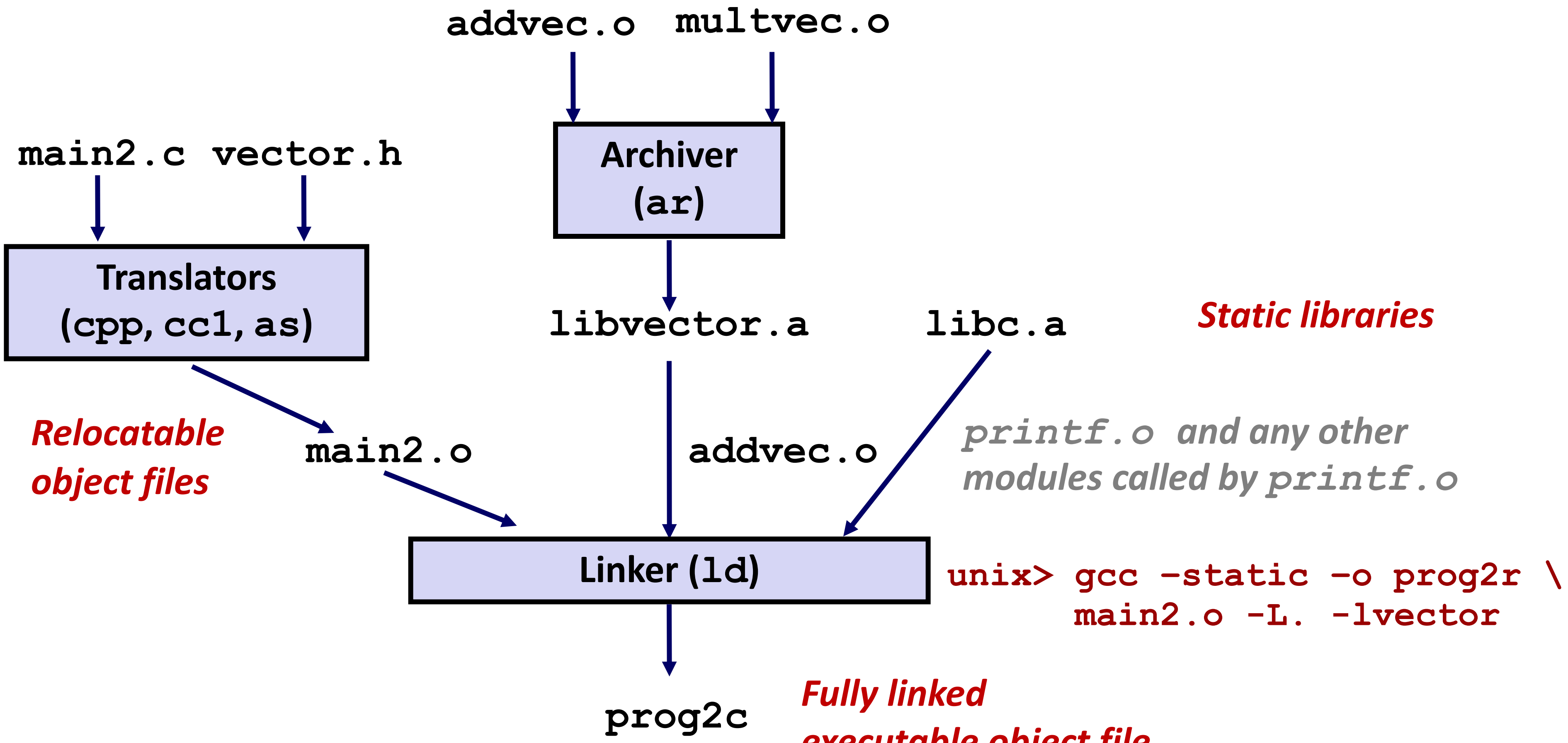
```
void addvec(int *x, int *y,
            int *z, int n) {
    int i;

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

*addvec.c*

```
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*

# Linking with Static Libraries



```
unix> gcc -static -o prog2r \
      main2.o -L. -lvector
```

(892,607 bytes)

*“c” for “compile-time”*

# 自定义一个静态库文件

**举例：将myproc1.o和myproc2.o打包生成mylib.a**

**myproc1.c**

```
# include <stdio.h>
void myfunc1() {
    printf("This is myfunc1!\n");
}
```

**myproc2.c**

```
# include <stdio.h>
void myfunc2() {
    printf("This is myfunc2\n");
}
```

```
$ gcc –c myproc1.c myproc2.c
$ ar rcs mylib.a myproc1.o myproc2.o
```

**main.c**

```
void myfunc1(viod);
int main()
{
   myfunc1();
   return 0;
}
```

```
$ gcc –c main.c
$ gcc –static –o myproc main.o ./mylib.a
```

**libc.a无需明显指出！**

**调用关系：main→myfunc1→printf**

**问题：如何进行符号解析?**

# 链接器中符号解析的全过程

```
$ gcc –c main.c
$ gcc –static –o myproc main.o ./mylib.a
```

libc.a无需明显指出!

调用关系：main→myfunc1→printf

main.c

```
void myfunc1(viod);
int main()
{
   myfunc1();
   return 0;
}
```

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

开始E、U、D为空，首先扫描main.o，把它加入E，同时把myfun1加入U，main加入D。接着扫描到mylib.a，将U中所有符号（本例中为myfunc1）与mylib.a中所有目标模块（myproc1.o和myproc2.o）依次匹配，发现在myproc1.o中定义了myfunc1，故myproc1.o加入E，myfunc1从U转移到D。在myproc1.o中发现还有未解析符号printf，将其加到U。不断在mylib.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。此时U中只有一个未解析符号printf，而D中有main和myfunc1。因为模块myproc2.o没有被加入E中，因而它被丢弃。

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，而所有未解析符号加入U。
处理完libc.a时，U一定是空的。

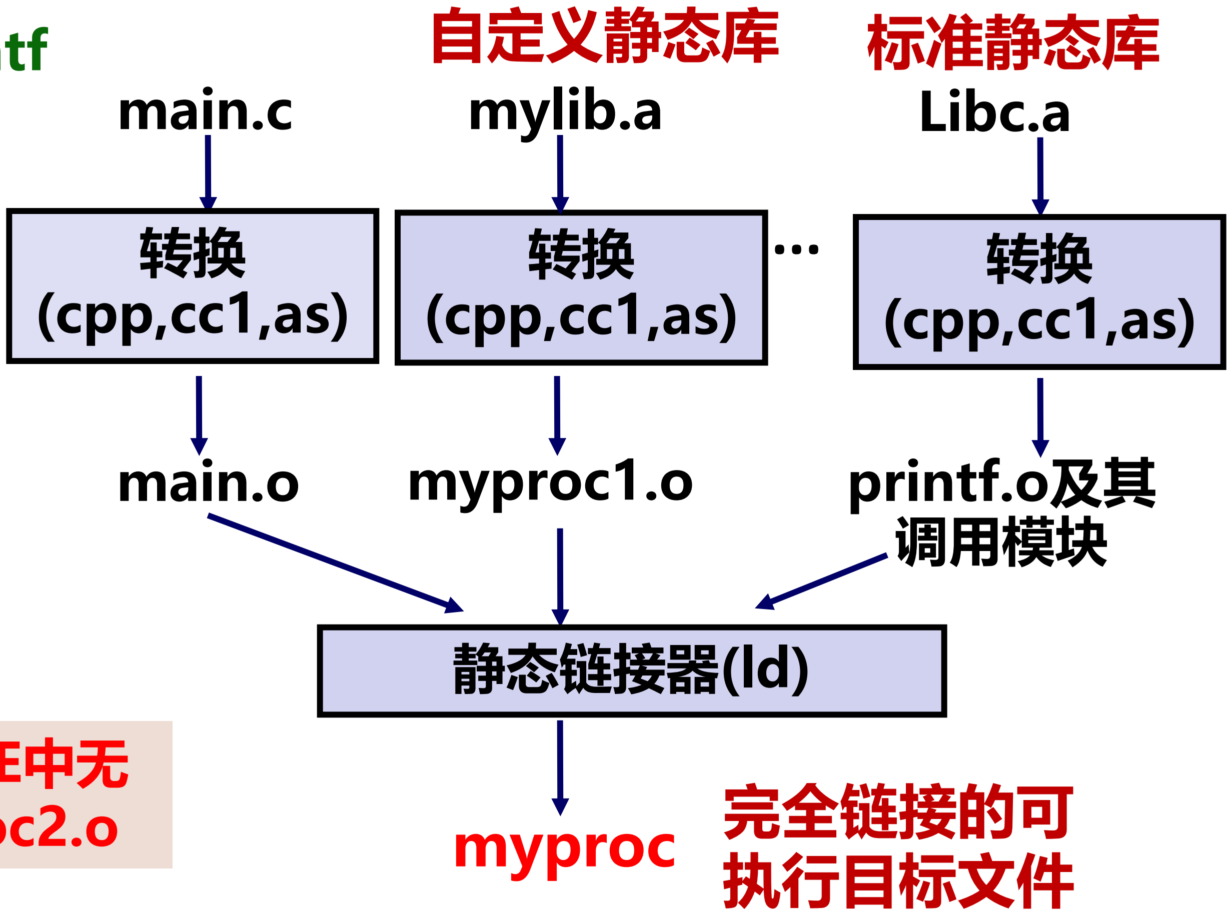
# 链接器中符号解析的全过程

```
$ gcc –static –o myproc main.o ./mylib.a
```

main→myfunc1→printf

main.c

```
void myfunc1(viod);
int main()
{
   myfunc1();
   return 0;
}
```



注意：E中无myproc2.o

解析结果：

E中有main.o、myproc1.o、printf.o及其调用的模块

D中有main、myproc1、printf及其引用的符号

# 链接器中符号解析的全过程

main.c

```
void myfunc1(viod);
int main()
{
   myfunc1();
   return 0;
}
```

$ gcc –static –o myproc main.o ./mylib.a

解析结果：

E中有main.o、myproc1.o、printf.o及其调用的模块

D中有main、myproc1、printf及其引用符号

main→myfunc1→printf

被链接模块应按调用顺序指定！

若命令为：$ gcc –static –o myproc ./mylib.a main.o， 结果怎样?

首先，扫描mylib，因是静态库，应根据其中是否存在U中未解析符号对应的定义符号来确定哪个.o被加入E。因为开始U为空，故其中两个.o模块都不被加入E中而被丢弃。

然后，扫描main.o，将myfunc1加入U，直到最后它都不能被解析。Why?

因此，出现链接错误！

它只能用mylib.a中符号来解析，而mylib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列表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
  - 好的做法：将静态库放在命令行的最后

libmine.a 是静态库

假设调用关系：libtest.o→libfun.o(在libmine.a中)

-lxxx=libxxx.a (main) →(libfun)

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

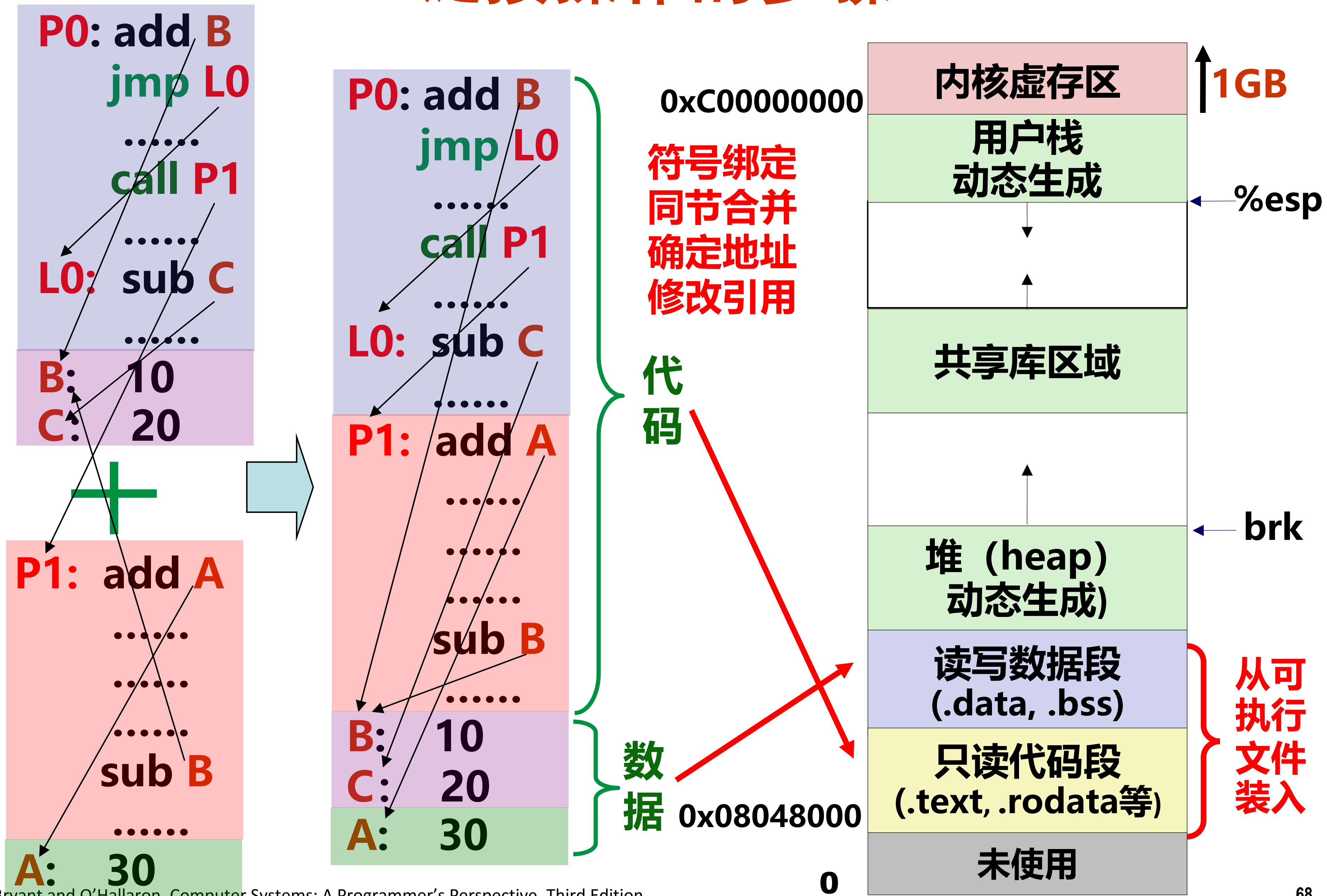
扫描libtest.o，将libfun送U，扫描到libmine.a时，用其定义的libfun来解析

说明在libtest.o中的main调用了libfun这个在库libmine中的函数，所以，在命令行中，应该将libtest.o放在前面，像第一行中那样！

## 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

# 链接操作的步骤


P0: add B
jmp L0
......
call P1
......
L0: sub C
......
B: 10
C: 20
+
P1: add A
......
......
......
sub B
......
A: 30
P0: add B
jmp L0
......
call P1
......
L0: sub C
......
P1: add A
......
......
......
sub B
......
B: 10
C: 20
A: 30
代码
数据
符号绑定
同节合并
确定地址
修改引用
0xC00000000
0x08048000
0
内核虚存区
1GB
用户栈
动态生成
%esp
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
从可
执行
文件
装入

# 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()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，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

## What dynamic libraries are required?

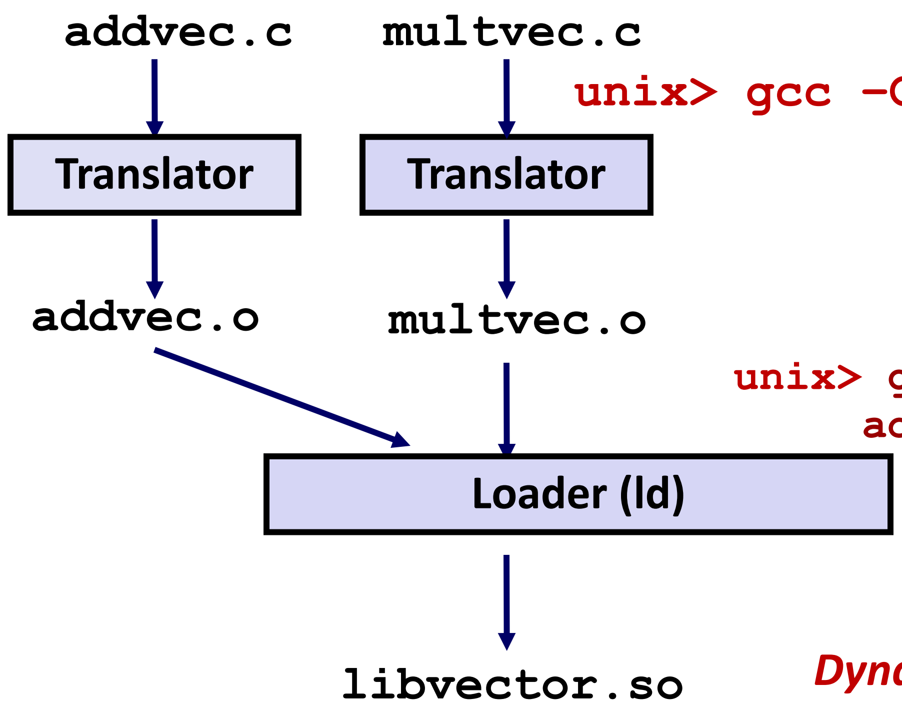
- **.interp section**
  - Specifies the dynamic linker to use (i.e., **`ld-linux.so`**)
- **.dynamic section**
  - Specifies the names, etc of the dynamic libraries to use
  - Follow an example of **`prog`**

```
(NEEDED)             Shared library: [libm.so.6]
```

- **Where are the libraries found?**
  - Use "**`ldd`**" to find out:

```
unix> ldd prog
  linux-vdso.so.1 =>  (0x00007ffcf2998000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f99ad927000)
  /lib64/ld-linux-x86-64.so.2 (0x00007f99adcef000)
```

# Dynamic Library Example

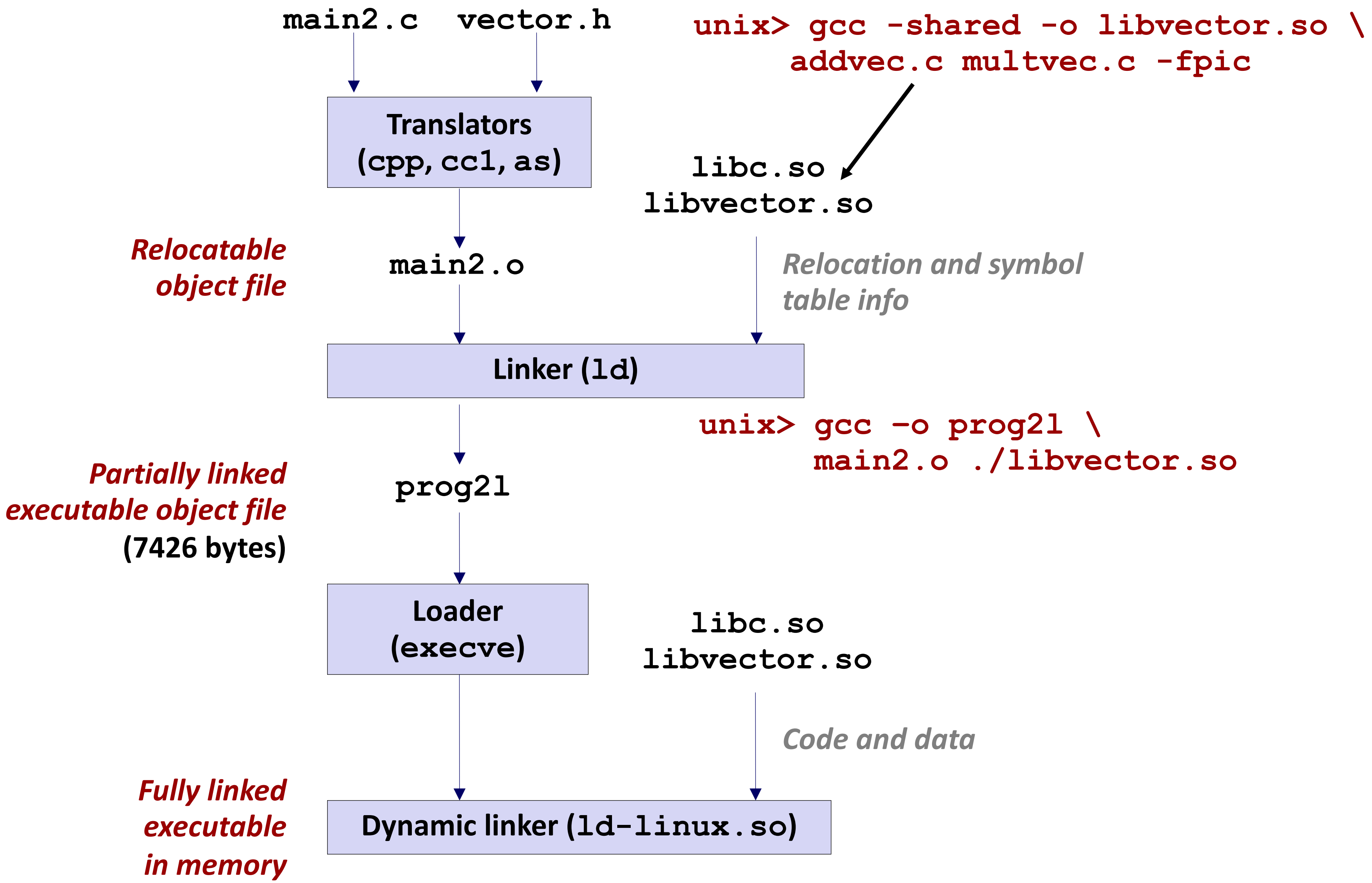


```
unix> gcc -Og -c addvec.c multvec.c -fpic
```

```
unix> gcc -shared -o libvector.so \
    addvec.o multvec.o
```

*Dynamic vector library*

# Dynamic Linking at Load-time



```
unix> gcc -shared -o libvector.so \
      addvec.c multvec.c -fpic
```

```
unix> gcc -o prog21 \
      main2.o ./libvector.so
```

# 自定义一个动态共享库文件

**mypoc1.c**

```
# include <stdio.h>
void myfunc1()
{
    printf("%s","This is myfunc1!\n");
}
```

**myproc2.c**

```
# include <stdio.h>
void myfunc2()
{
     printf("%s","This is myfunc2\n");
}
```

**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，要不会影响调用它的程序**

**位置无关的共享代码库文件**

```
gcc –c myproc1.c myproc2.c
gcc –shared –fPIC –o mylib.so myproc1.o myproc2.o
```

# 加载时动态链接

```
gcc –c main.c
gcc –o myproc main.o ./mylib.so
```

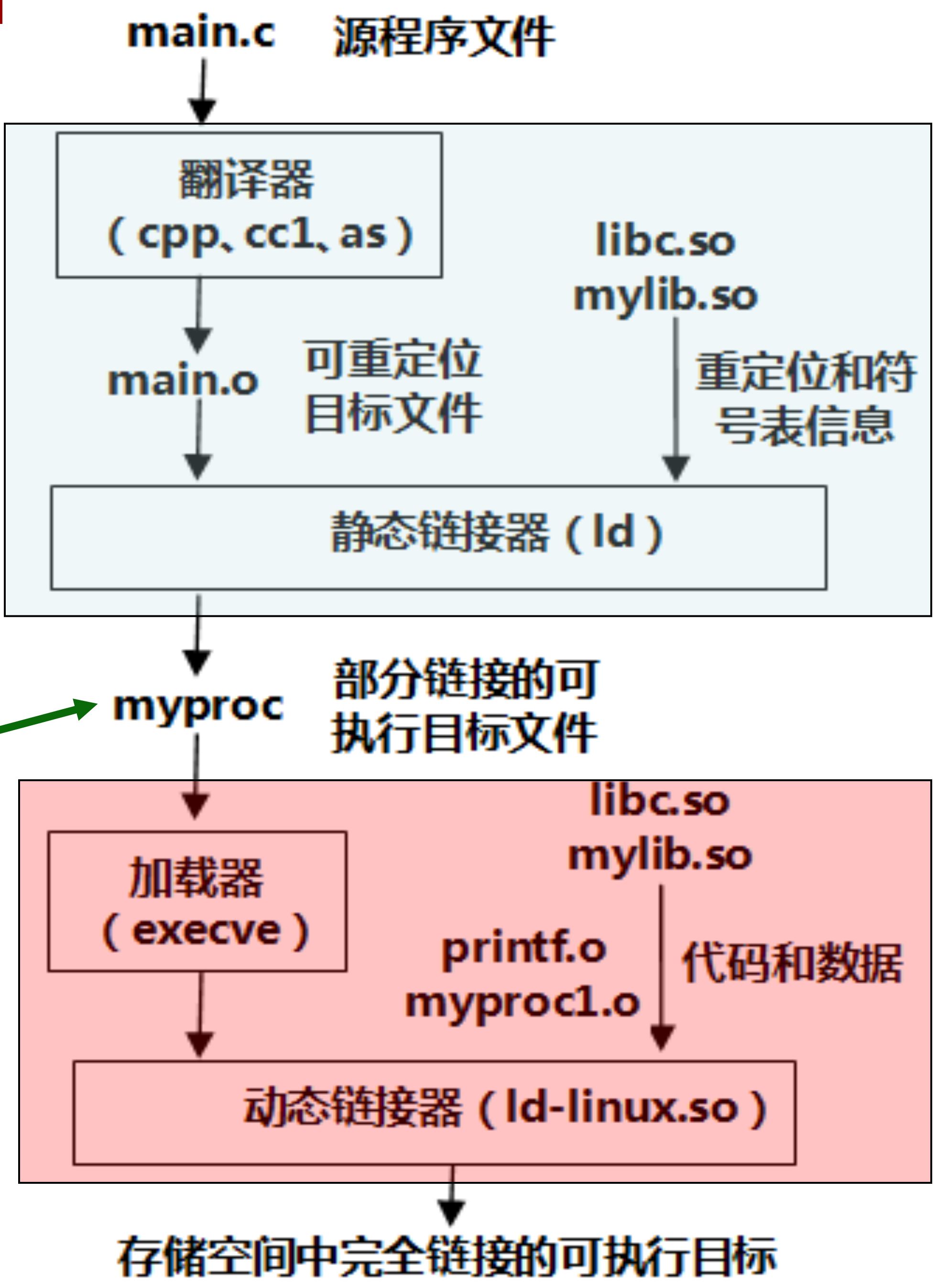
libc.so无需明显指出

调用关系：main→myfunc1→printf

main.c

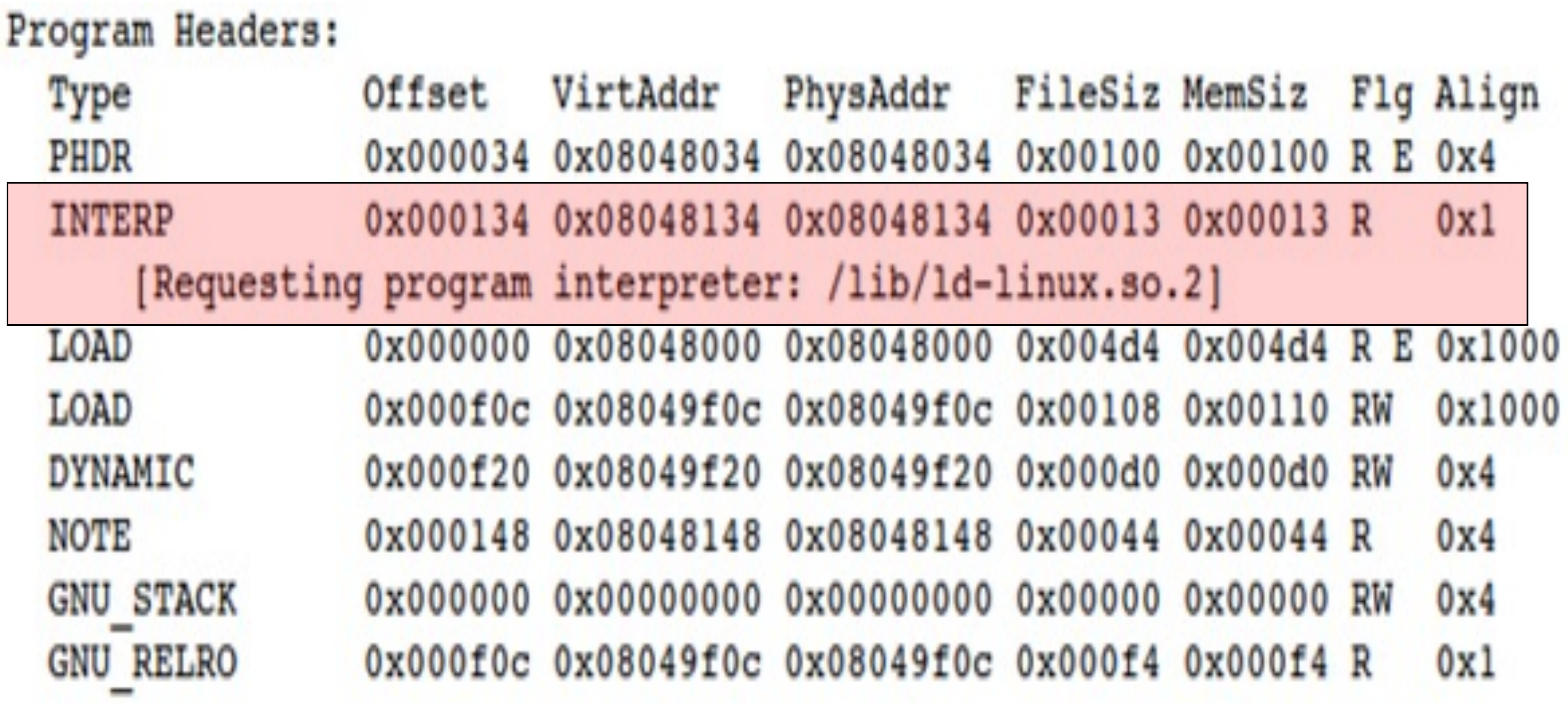
```
void myfunc1(viod);
int main()
{
   myfunc1();
   return 0;
}
```

加载 myproc 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 myproc，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**



```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
  INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x004d4 0x004d4 R E 0x1000
  LOAD           0x000f0c 0x08049f0c 0x08049f0c 0x00108 0x00110 RW  0x1000
  DYNAMIC        0x000f20 0x08049f20 0x08049f20 0x000d0 0x000d0 RW  0x4
  NOTE           0x000148 0x08048148 0x08048148 0x00044 0x00044 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
  GNU_RELRO      0x000f0c 0x08049f0c 0x08049f0c 0x000f4 0x000f4 R   0x1
```

# Dynamic Linking at Run-time

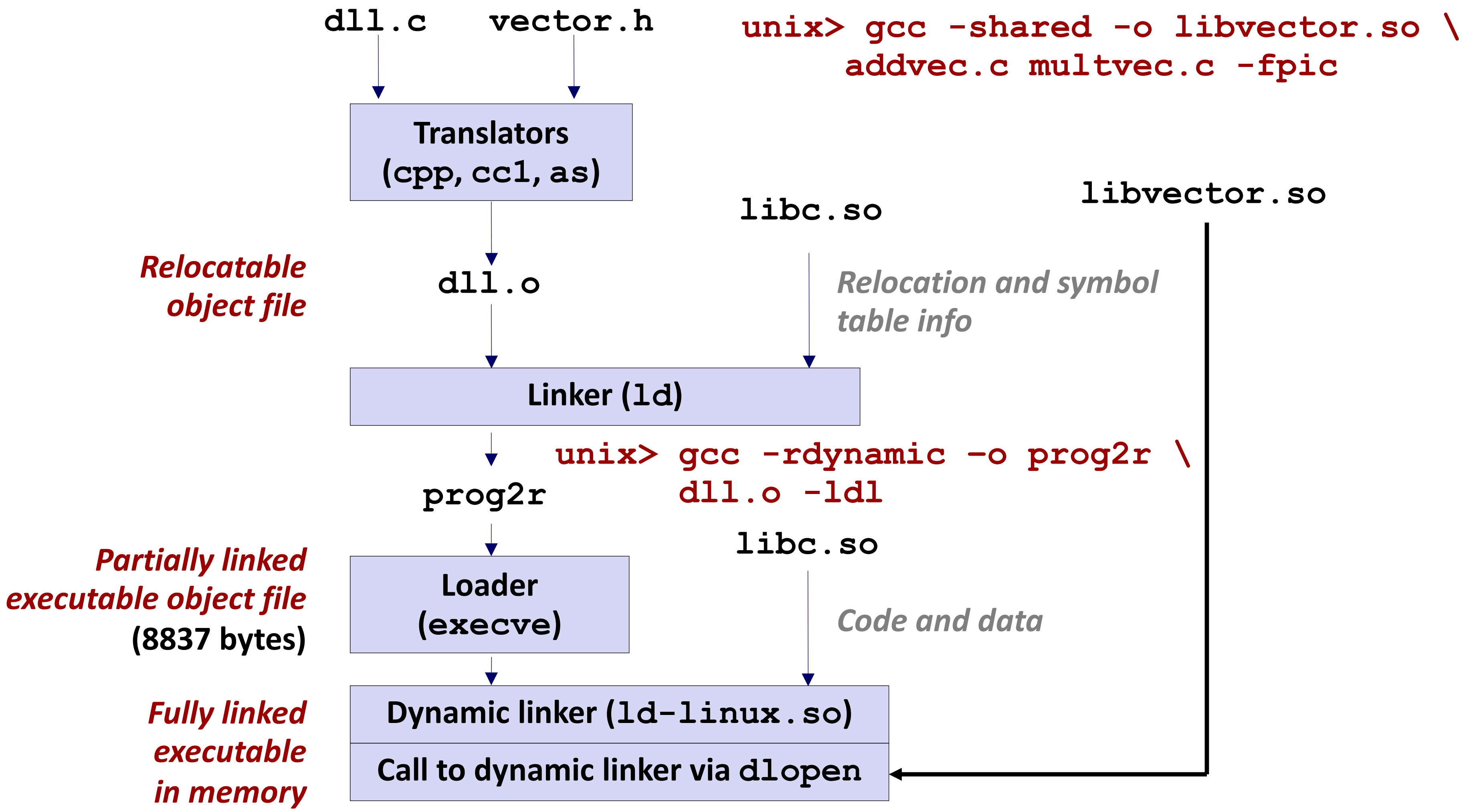
```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

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

int main(int argc, char** argv)
{
    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
```

# Dynamic Linking at Run-time (cont)

```
    ...

    /* 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*

# Dynamic Linking at Run-time


dll.c
vector.h
Translators (cpp, cc1, as)
Relocatable object file
dll.o
libc.so
Relocation and symbol table info
libvector.so
Linker (ld)
prog2r
Partially linked executable object file (8837 bytes)
Loader (execve)
libc.so
Code and data
Fully linked executable in memory
Dynamic linker (ld-linux.so)
Call to dynamic linker via dlopen

```
unix> gcc -shared -o libvector.so \
      addvec.c multvec.c -fpic
```

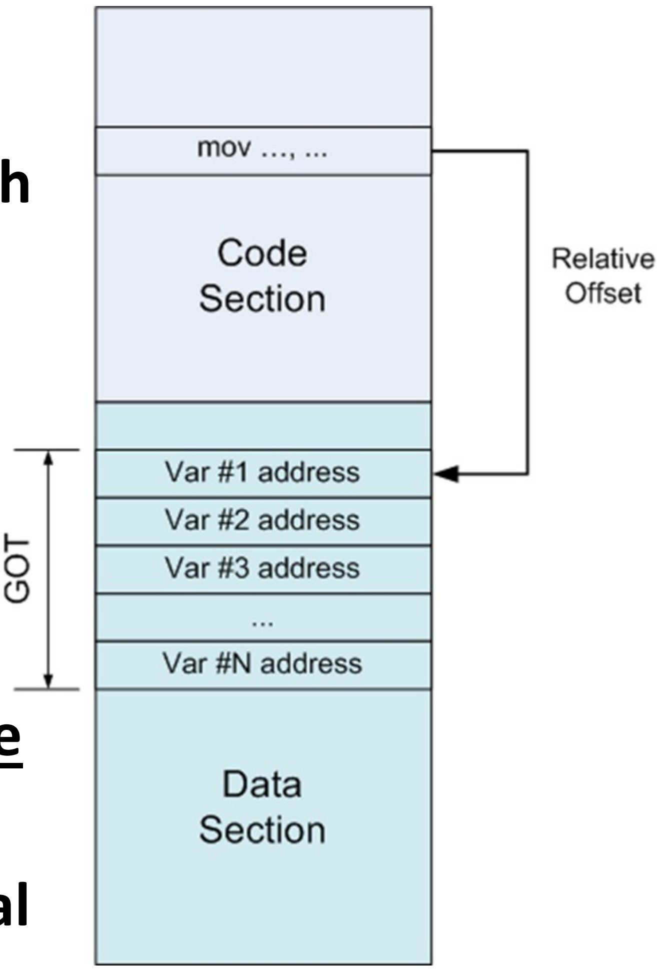
```
unix> gcc -rdynamic -o prog2r \
      dll.o -ldl
```

## Lazy Binding

- Lazy binding, or dynamic binding, defers the binding of each procedure address until the first time the procedure is called.
- The motivation for lazy binding is that a typical application program will call only a handful of the hundreds or thousands of functions exported by a shared library such as libc.so.
- By deferring the resolution of a function's address until it is actually called, the dynamic linker can avoid hundreds or thousands of unnecessary relocations at load time.
- There is a nontrivial run-time overhead the first time the function is called, but each call thereafter takes only little time
- Lazy binding is implemented with a compact yet somewhat complex interaction between two data structures: the GOT and the PLT.

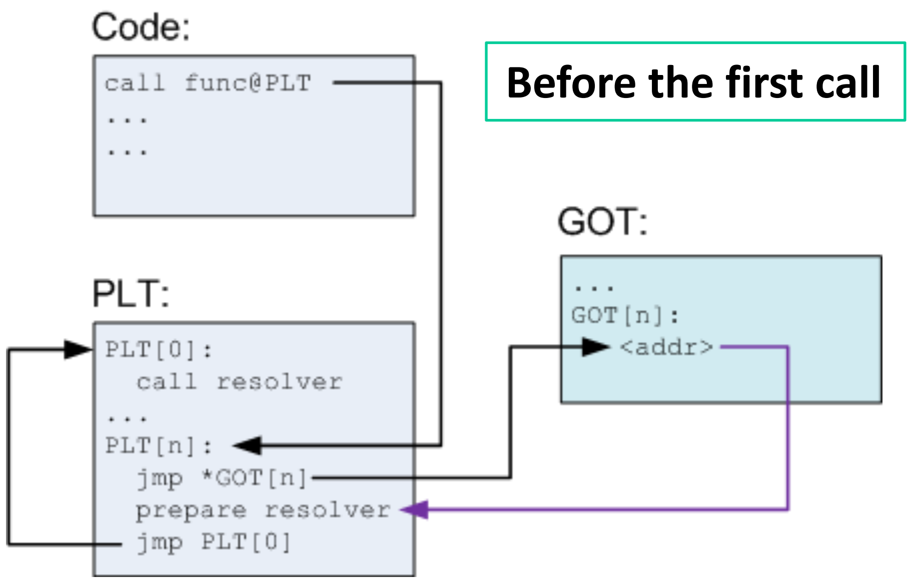
## The Global Offset Table (GOT)

- A GOT is simply a table of addresses, residing in the data section
- The GOT contains an 8-byte entry for each global data object (procedure or global variable) that is referenced by the object module
- The compiler also generates a relocation record for each entry in the GOT. At load time, the dynamic linker relocates each GOT entry so that it contains the absolute address of the object.
- Each object module that references global objects has its own GOT.



## The Procedure Linkage Table (PLT)

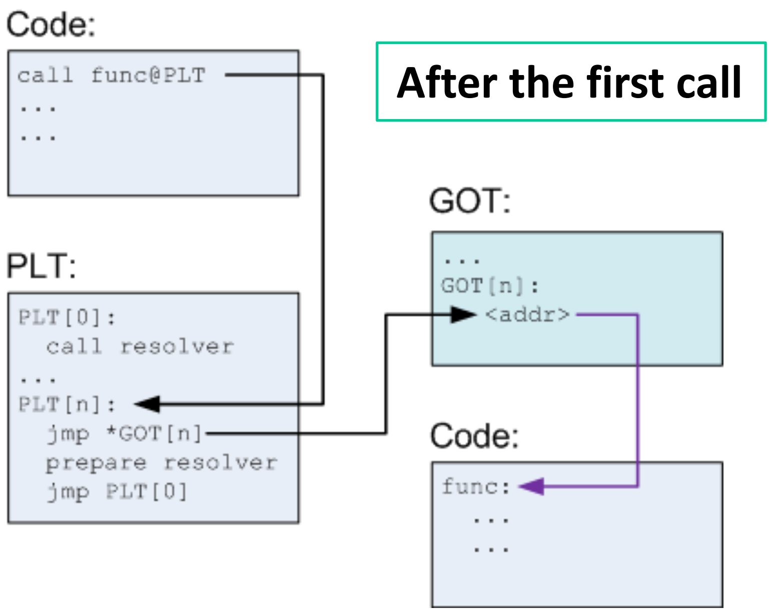
- Each PLT entry is 16 bytes of executable code. Instead of calling the function directly, the code calls an entry in the PLT, which then takes care to call the actual function
- Each PLT entry also has a corresponding entry in the GOT which contains the actual offset to the function, but only when the dynamic linker resolves it



```
call func@PLT
...
...
```

```
...
GOT[n]:
  <addr>
```

```
PLT[0]:
  call resolver
...
PLT[n]:
  jmp *GOT[n]
  prepare resolver
  jmp PLT[0]
```



```
call func@PLT
...
...
```

```
PLT[0]:
  call resolver
...
PLT[n]:
  jmp *GOT[n]
  prepare resolver
  jmp PLT[0]
```

```
...
GOT[n]:
  <addr>
```

```
func:
  ...
  ...
```

## Example program

```
/* fopen.c
   Open a file, write "Hello World!" to it */

#include <stdio.h>
int main() {
    FILE *out;
    char buf[16] = "Hello World!\n";

    out = fopen("hello.txt", "w+");
    fprintf(out, "%s", buf);
    fclose(out);
    return 0;
}
```

## PLT

```
/* section .plt */
# PLT[0] <fclose@plt-0x10>: call dynamic linker
  400410:  pushq  0x200552(%rip)          # GOT[1]
  400416:  jmpq   *0x200554(%rip)         # GOT[2]
  40041c:  nopl   0x0(%rax)
# PLT[1] <fclose@plt>:
  400420:  jmpq   *0x200552(%rip)         # GOT[3]
  400426:  pushq  $0x0
  40042b:  jmpq   400410 <_init+0x10>
# PLT[2] <fputs@plt>:
  400430:  jmpq   *0x20054a(%rip)         # GOT[4]
  400436:  pushq  $0x1
  40043b:  jmpq   400410 <_init+0x10>
# PLT[3] <__libc_start_main@plt>:
  400440:  jmpq   *0x200542(%rip)         # GOT[5]
  400446:  pushq  $0x2
  40044b:  jmpq   400410 <_init+0x10>
# PLT[4] <fopen@plt>:
  400450:  jmpq   *0x20053a(%rip)         # GOT[6]
  400456:  pushq  $0x3
  40045b:  jmpq   400410 <_init+0x10>
```

## GOT

Before the first call

```
/* (gdb) x /8xg 0x600960
   <_GLOBAL_OFFSET_TABLE_> */
0x600960 0x0000000000600788  # GOT[0] addr of .dynamic
0x600968 0x0000003852e22190  # GOT[1] addr of reloc entries
0x600970 0x0000003852c14c20  # GOT[2] addr of dynamic linker
0x600978 0x0000000000400426  # GOT[3] fclose()
0x600980 0x0000000000400436  # GOT[4] fputs()
0x600988 0x000000385301ec20  # GOT[5] sys startup
0x600990 0x0000000000400456  # GOT[6] fopen()
0x600998 0x0000000000000000
```

After the first call

```
/* (gdb) x /8xg 0x600960
   <_GLOBAL_OFFSET_TABLE_> */
0x600960 0x0000000000600788  # GOT[0] addr of .dynamic
0x600968 0x0000003852e22190  # GOT[1] addr of reloc entries
0x600970 0x0000003852c14c20  # GOT[2] addr of dynamic linker
0x600978 0x0000003853066260  # GOT[3] fclose()
0x600980 0x0000003853067100  # GOT[4] fputs()
0x600988 0x000000385301ec20  # GOT[5] sys startup
0x600990 0x0000003853066e60  # GOT[6] fopen()
0x600998 0x0000000000000000
```

# 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、程序头表（段头表）以及各种节组成的段
- 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

# 本章小结

- 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。
- 在不同目标模块中可能会定义相同符号，因为相同的多个符号只能分配一个地址，因而链接器需要确定以哪个符号为准。
- 编译器通过对定义符号标识其为强符号还是弱符号，由链接器根据一套规则来确定多重定义符号中哪个是唯一的定义符号，如果不了解这些规则，则可能无法理解程序执行的有些结果。
- 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。

## 练习

- 7.1、7.2、7.3、7.4、7.5